

HDL Verifier™

Reference



MATLAB® & SIMULINK®

R2023a



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™ Reference

© COPYRIGHT 2003–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

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

Revision History

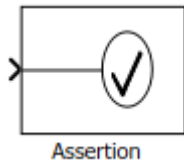
August 2003	Online only	New for Version 1 (Release 13SP1)
February 2004	Online only	Revised for Version 1.1 (Release 13SP1)
June 2004	Online only	Revised for Version 1.1.1 (Release 14)
October 2004	Online only	Revised for Version 1.2 (Release 14SP1)
December 2004	Online only	Revised for Version 1.3 (Release 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)
March 2015	Online only	Revised for Version 4.6 (Release 2015a)
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)
March 2023	Online only	Revised for Version 7.1 (Release 2023a)

1	<hr/>	Blocks
2	<hr/>	System Objects
3	<hr/>	Objects
4	<hr/>	Functions
5	<hr/>	Apps

Blocks

Assertion

Generate SystemVerilog assertions from Simulink assertion



Libraries:
HDL Verifier / For Use with DPI-C SystemVerilog

Description

The Assertion block asserts that its input signal is nonzero. If its input is zero, the block halts the simulation by default and displays an error message. When you generate a DPI-C SystemVerilog component - the block creates an immediate SystemVerilog assertion. Using the block parameters, you can:

- Enable or disable the assertion.
- Specify a MATLAB® expression for Simulink® to evaluate when the assertion fails.
- Select for Simulink to either stop simulation or continue but display a warning when assertion fails.

Use the DPI-C parameters to control runtime options:

- Specify the severity of the generated assertion.
- Specify a custom message or action when the assertion fails.

Ports

Input

Port_1 — Signal to check for nonzero value
scalar | vector | matrix

The Assertion block accepts input signals of any dimensions and numeric data type that Simulink supports.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Enable assertion — Enable or disable assertion

on (default) | off

Selecting this check box enables the block to display a simulation warning or error. It also enables the block to create a SystemVerilog assertion in your generated code. Clearing this check box disables the assertion in simulation, and it does not generate a SystemVerilog assertion.

Simulation callback when assertion fails — Expression to evaluate when assertion fails

MATLAB expression

Specify a MATLAB expression for Simulink to evaluate when the assertion fails. The block ignores this parameter in the generated DPI-C assertion.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Stop simulation when assertion fails — Stop Simulink simulation when assertion fails

off (default) | on

Selecting this check box causes Simulink to stop the simulation and display an error when the block input is zero. Clearing this check box enables Simulink to continue the simulation, displaying a warning when the block input is zero. The block ignores this parameter in the generated DPI-C assertion.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

DPI-C Assertion Options

Use these parameters to control the behavior of a generated DPI-C assertion, in a SystemVerilog simulation environment. To enable generation of DPI-C assertion, select **Enable assertion**.

Severity — Severity of assertion failure

error (default) | warning | custom

Select error or warning for the DPI-C assertion to issue a SystemVerilog error or warning message. Set to custom to execute a custom command.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Assertion fail message — Custom message when assertion fails

no default

Specify a custom SystemVerilog message to be emitted when the SystemVerilog assertion fails. This feature supports only ASCII characters.

Example: RX fail

Dependencies

To enable this parameter, set **Severity** to error or warning.

Assertion custom command — Custom command to execute when assertion fails

SystemVerilog command

Specify a custom SystemVerilog command to execute when the assertion fails. You can set this parameter to be a display statement, command, or script. This feature supports only ASCII characters

Example: `$display("RX fail at %0t", $time);`

Dependencies

To enable this parameter, set **Severity** to custom.

Version History

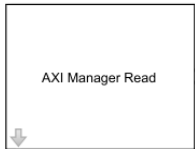
Introduced in R2018a

See Also**Topics**

“Generate SystemVerilog Assertions from Simulink Test Bench”

AXI Manager Read

Read memory locations on FPGA board from Simulink



Libraries:

HDL Verifier Support Package for Intel Boards
 HDL Verifier Support Package for Xilinx Boards

Description

The AXI Manager Read block communicates with the AXI manager IP when it is running on an FPGA board. The block forwards read commands to the IP to access memory-mapped locations on the FPGA board.

Note The AXI Master Read block has been renamed to AXI Manager Read block. For more information, see “Version History” on page 1-10.

Before using this block, you must create an AXI manager IP and integrate it in your FPGA design. For more information, see “Set Up AXI Manager”.

Ports

Output

data — Data read from FPGA board
 scalar | vector

Data read from the FPGA board, returned as a scalar or vector. The output is of size 1-by- N , where N is the **Output vector size** parameter value. The **Output data type** parameter sets the data type of this output. The read data from the FPGA is of type `uint32`, `int32`, `uint64`, or `int64` depending on the data width of the AXI manager IP on your FPGA. The block converts the data type to the value specified by the **Output data type** parameter.

Data Types: `uint8` | `int8` | `uint16` | `int16` | `half` | `uint32` | `int32` | `single` | `uint64` | `int64` | `double` | `fixed point`

Parameters

Main

Address — Starting address for read operation
 0 (default) | nonnegative integer multiple of 4 or 8 | nonnegative hexadecimal value multiple of 4 or 8

Specify the starting address for the read operation as a nonnegative integer or hexadecimal value. The block supports the address width of 32, 40, and 64 bits. The block converts the address data type

to `uint32` or `uint64` according to the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

Memory Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and each address is a 4-byte increment (0x0, 0x4, 0x8). For example the address 0x1 returns an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and each address is an 8-byte increment (0x0, 0x8, 0x10). For example, specifying the address 0x1 or 0x4 are both invalid and return an error.
- If the AXI manager IP data width is 32 bits and the **Burst type** parameter is set to `Increment`, the block increments the address by 4 bytes.
- If the AXI manager IP data width is 64 bits and the **Burst type** parameter is set to `Increment`, the block increments the address by 8 bytes.
- If the AXI manager IP data width is 32 bits and the **Output data type** parameter is set to `half`, the block reads the lower 2 bytes and ignores the higher 2 bytes.
- If the AXI manager IP data width is 64 bits and the **Output data type** parameter is set to `half`, the block reads the lower 2 bytes and ignores the higher 6 bytes.
- Do not use a 64-bit AXI manager IP for accessing 32-bit registers.

Example: 0xa4

Burst type — AXI4 burst type

`Increment` (default) | `Fixed`

In `Increment` mode, the AXI manager reads a vector of data from contiguous memory spaces starting with the specified address. In `Fixed` mode, the AXI manager reads all data from the same address.

Note The `Fixed` burst type is not supported for the PCI Express® interface. Use the `Increment` burst type instead.

Output data type — Data type of the output data

`uint32` (default) | `double` | `single` | `half` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `int64` | `uint64` | `fixdt(1,16,0)` | `<data type expression>`

The block converts the data read out of the FPGA to the specified data type.

Output vector size — Number of memory locations to read

`1` (default) | positive integer

Specify the number of memory locations for the block to read. By default, the block reads from a contiguous address block, incrementing the address for each operation. To turn off address increment mode and read repeatedly from the same location, set the **Burst type** parameter to `Fixed`.

When you specify a large operation size, such as reading a block of double data rate (DDR) memory, the block automatically breaks the operation into multiple bursts, using the maximum supported burst size. The maximum supported burst size is 256 words.

Sample time — Read sample time

`-1` (default) | positive scalar

Specify the simulation sample time for the block. When you specify -1 (default), the block inherits the sample time from other blocks in the system.

Vector register data with strobe synchronization — Read data from registers with strobe synchronization
off (default) | on

To enable reading data from a set of registers with strobe synchronization, select this parameter. Enable this parameter when your FPGA design includes strobe synchronization generated by HDL Coder™. For more information about strobe synchronization, see the "Vector Data Read/Write with Strobe Synchronization" section in "IP Core User Guide" (HDL Coder).

Strobe address — Strobe address used for strobe synchronization
0 (default) | nonnegative integer multiple of 4 or 8 | nonnegative hexadecimal value multiple of 4 or 8

Set the absolute address for the strobe generated with HDL Coder. The absolute address is the sum of the base address and the strobe offset provided by the IP core report.

Example: If the base address is 0x41000000 and offset is 0x110, the absolute address is 0x41000110.

Dependencies

To enable this parameter, select **Vector register data with strobe synchronization**.

Interface

Type — Type of interface used for communication with FPGA board
JTAG (default) | UDP | PCIe

Specify the interface type for communicating between the host and the FPGA.

AXI Manager Interface Configuration

To view these parameters, open the AXI Manager Interface Configuration dialog box by clicking **Configure global parameters**. The visible parameters depend on the **Type** parameter value.

Global parameters apply to the entire Simulink model.

Vendor — FPGA brand name
Intel | Xilinx

Specify the manufacturer of your FPGA board. The AXI manager IP varies depending on the FPGA board type.

Dependencies

To enable this parameter, click **Configure global parameters**.

AXI data width — Data width of AXI manager IP on FPGA
32 (default) | 64

Select the data width, in bits, of the AXI manager IP on the FPGA.

For PCI Express, set this value to 32. For JTAG or Ethernet connections, set this value to 32 or 64.

Dependencies

To enable this parameter, click **Configure global parameters**.

Cable type — Type of JTAG cable used for communication with FPGA board (Xilinx® only)
auto (default) | FTDI

Specify the type of JTAG cable used for communication with the FPGA board. Use this parameter when more than one cable is connected to the host computer.

When you set this parameter to `auto` (default), the block automatically detects the JTAG cable type. The block prioritizes searching for Digilent® cables and uses this process to detect the cable type.

- 1 The AXI Manager Write block searches for a Digilent cable. If the block finds:
 - Exactly one Digilent cable, it uses that cable for communication with the FPGA board.
 - More than one Digilent cable - it returns an error. To resolve this error, specify the desired cable using the **Cable name** parameter.
 - No Digilent cables, it searches for an FTDI cable.
- 2 If no Digilent cable is found, the AXI Manager Write block searches for an FTDI cable. If the block finds:
 - Exactly one FTDI cable, it uses that cable for communication with the FPGA board.
 - More than one FTDI cable, it returns an error - To resolve this error, specify the desired cable using the **Cable name** parameter.
 - No FTDI cables, it returns an error - To resolve this error, connect a Digilent or FTDI cable.
- 3 If it finds two cables of different types, it prioritizes the Digilent cable. To use an FTDI cable, set this parameter to FTDI.

When you set this parameter to FTDI, the block searches for FTDI cables. If the object finds:

- Exactly one FTDI cable, it uses that cable for communication with the FPGA board.
- More than one FTDI cable, it returns an error - To resolve this error, specify the desired cable using the **Cable name** parameter.
- No FTDI cables, it returns an error - To resolve this error, connect a Digilent or FTDI cable.

For more details, for Intel® boards, see “Select from Multiple JTAG Cables”. For Xilinx boards, see “Select from Multiple JTAG Cables”.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Cable name — Name of JTAG cable used for communication with FPGA board
auto (default) | name of connected JTAG cable

Specify this parameter if more than one JTAG cable of the same type are connected to the host computer. If more than one JTAG cable is connected to the host computer, and you do not specify this parameter, the block returns an error. The error message contains the names of the available JTAG cables. For more details, for Intel boards, see “Select from Multiple JTAG Cables”. For Xilinx boards, see “Select from Multiple JTAG Cables”.

Dependencies

To enable this parameter, set **Type** to JTAG.

Clock frequency in MHz — JTAG clock frequency

15 (default) | positive scalar

Specify the JTAG clock frequency in MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board. Check the board documentation for the supported frequency range.

Dependencies

To enable this parameter, set **Type** to JTAG.

Chain position — Position of FPGA in JTAG chain (Xilinx only)

auto (default) | nonnegative integer

Specify this parameter value as a nonnegative integer if more than one FPGA or Zynq® device is on the JTAG chain. Otherwise, select **auto** (default) for automatic detection of chain position.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Instruction registers before FPGA — Sum of instruction register lengths for all devices before target FPGA (Xilinx only)

0 (default) | nonnegative integer

Specify this parameter value as a nonnegative integer if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Instruction registers after FPGA — Sum of instruction register length for all devices after target FPGA (Xilinx only)

0 (default) | nonnegative integer

Specify this parameter value as a nonnegative integer if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Device address — IP address of FPGA board

192.168.0.2 (default) | IP address

Specify the IP address of the Ethernet port on the FPGA board.

Example: 192.168.0.10

Dependencies

To enable this parameter, set **Type** to UDP.

Port — UDP port number of FPGA board
50101 (default) | integer from 255 to 65,535

Specify the user datagram protocol (UDP) port number of the target FPGA as an integer from 255 to 65,535.

Dependencies

To enable this parameter, set **Type** to UDP.

Version History

Introduced in R2019b

R2022a: AXI Master Read renamed to AXI Manager Read

Warns starting in R2022a

The AXI Master Read block has been renamed to the AXI Manager Read block. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

In R2022a, you cannot use a Simulink model that contains the AXI Master Read block. Recreate your model in R2022a by using the AXI Manager Read block.

R2023a: Support for half data type

The block reads half data from the memory locations on the FPGA board. The address for the read operation must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, the block reads the lower 2 bytes and ignores the higher 2 bytes.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, the block reads the lower 2 bytes and ignores the higher 6 bytes.

See Also

Blocks

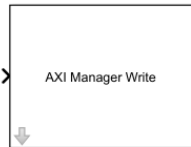
AXI Manager Write

Topics

“Set Up AXI Manager”

AXI Manager Write

Write memory locations on FPGA board from Simulink



Libraries:

HDL Verifier Support Package for Intel Boards
 HDL Verifier Support Package for Xilinx Boards

Description

The AXI Manager Write block communicates with the AXI manager IP when it is running on an FPGA board. The block forwards write commands to the IP to access memory-mapped locations on the FPGA board.

Note The AXI Master Write block has been renamed to AXI Manager Write block. For more information, see “Version History” on page 1-16.

Before using this block, you must create an AXI manager IP and integrate it in your FPGA design. For more information, see “Set Up AXI Manager”.

Ports

Input

data — Data words to write on the FPGA board
 scalar | vector

Input data to write on the FPGA board, specified as a scalar or vector. Before sending the write request to the FPGA, the block converts the input data to `uint32`, `int32`, `uint64`, or `int64`. The data type conversion follows these rules.

- If the input data is of type `double`, the block converts the data to type `int32` or `int64` depending on the data-width of the AXI manager IP.
- If the input data is of type `single`, the block converts the data to type `uint32` or `uint64` depending on the AXI manager IP data width.
- If the input data is of type `half`, the block converts the data to type `uint16` and then packs the data to type `uint32` or `uint64` depending on the AXI manager IP data width.
- If the bit width of the input data type is less than the AXI manager IP data width, the data is extended to the width of the AXI manager IP data width.
- If the bit width of the input data type is greater than the AXI manager IP data width, the block converts the data to type `int32`, `uint32`, `int64`, `uint64`, to match the data width of the AXI manager IP and signedness of the original data type.
- If the input data is a fixed-point data type, the block writes the stored integer value of the data.

When you specify a large operation size, such as writing a block of double data rate (DDR) memory, the block automatically breaks the operation into multiple bursts, using the maximum supported burst size. The maximum supported burst size is 256 words.

Data Types: `uint8` | `int8` | `uint16` | `int16` | `half` | `uint32` | `int32` | `single` | `uint64` | `int64` | `double` | `fixed point`

Parameters

Main

Address — Starting address for write operation

`0` (default) | nonnegative integer multiple of 4 or 8 | nonnegative hexadecimal value multiple of 4 or 8

Specify the starting address for the write operation as a nonnegative integer or hexadecimal value. The block supports the address width of 32, 40, and 64 bits. The block converts the address data type to `uint32` or `uint64` according to the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

Memory Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and each address is a 4-byte increment (`0x0`, `0x4`, `0x8`). For example the address `0x1` returns an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and each address is an 8-byte increment (`0x0`, `0x8`, `0x10`). For example, specifying the address `0x1` or `0x4` are both invalid and return an error.
- If the AXI manager IP data width is 32 bits and the **Burst type** parameter is set to **Increment**, the block increments the address by 4 bytes.
- If the AXI manager IP data width is 64 bits and the **Burst type** parameter is set to **Increment**, the block increments the address by 8 bytes.
- If the AXI manager IP data width is 32 bits and the input data is `half`, the block writes data to the lower 2 bytes and pads the higher 2 bytes with zeros.
- If the AXI manager IP data width is 64 bits and the input data is `half`, the block writes data to the lower 2 bytes and pads the higher 6 bytes with zeros.
- Do not use a 64-bit AXI manager IP for accessing 32-bit registers.

Example: `0xa4`

Burst type — AXI4 burst type

`Increment` (default) | `Fixed`

In **Increment** mode, the AXI manager writes a vector of data to contiguous memory spaces, starting with the specified address. In **Fixed** mode, the AXI manager writes all data to the same address.

Note The **Fixed** burst type is not supported for the PCI Express interface. Use the **Increment** burst type instead.

Vector register data with strobe synchronization — Write data to registers with strobe synchronization

`off` (default) | `on`

To enable writing data to a set of registers with strobe synchronization, select this parameter. Enable this parameter when your FPGA design includes strobe synchronization generated by HDL Coder. For more information about strobe synchronization, see the "Vector Data Read/Write with Strobe Synchronization" section in "IP Core User Guide" (HDL Coder).

Strobe address — Strobe address used for strobe synchronization

0 (default) | nonnegative integer multiple of 4 or 8 | nonnegative hexadecimal value multiple of 4 or 8

Set the absolute address for the strobe generated with HDL Coder. The absolute address is the sum of the base address and the strobe offset provided by the IP core report.

Example: If the base address is 0x41000000 and offset is 0x110, the absolute address is 0x41000110.

Dependencies

To enable this parameter, select **Vector register data with strobe synchronization**.

Interface

Type — Type of interface used for communication with FPGA board

JTAG (default) | UDP | PCIe

Specify the interface type for communicating between the host and the FPGA.

AXI Manager Interface Configuration

To view these parameters, open the AXI Manager Interface Configuration dialog box by clicking **Configure global parameters**. The visible parameters depend on the **Type** parameter value.

Global parameters apply to the entire Simulink model.

Vendor — FPGA brand name

Intel | Xilinx

Specify the manufacturer of your FPGA board. The AXI manager IP varies depending on the FPGA board type.

Dependencies

To enable this parameter, click **Configure global parameters**.

AXI data width — Data width of AXI manager IP on FPGA

32 (default) | 64

Select the data width, in bits, of the AXI manager IP on the FPGA.

For PCI Express, set this value to 32. For JTAG or Ethernet connections, set this value to 32 or 64.

Dependencies

To enable this parameter, click **Configure global parameters**.

Cable type — Type of JTAG cable used for communication with FPGA board (Xilinx only)

auto (default) | FTDI

Specify the type of JTAG cable used for communication with the FPGA board. Use this parameter when more than one cable is connected to the host computer.

When you set this parameter to `auto` (default), the block automatically detects the JTAG cable type. The block prioritizes searching for Digilent cables and uses this process to detect the cable type.

- 1** The AXI Manager Write block searches for a Digilent cable. If the block finds:
 - Exactly one Digilent cable, it uses that cable for communication with the FPGA board.
 - More than one Digilent cable - it returns an error. To resolve this error, specify the desired cable using the **Cable name** parameter.
 - No Digilent cables, it searches for an FTDI cable.
- 2** If no Digilent cable is found, the AXI Manager Write block searches for an FTDI cable. If the block finds:
 - Exactly one FTDI cable, it uses that cable for communication with the FPGA board.
 - More than one FTDI cable, it returns an error - To resolve this error, specify the desired cable using the **Cable name** parameter.
 - No FTDI cables, it returns an error - To resolve this error, connect a Digilent or FTDI cable.
- 3** If it finds two cables of different types, it prioritizes the Digilent cable. To use an FTDI cable, set this parameter to FTDI.

When you set this parameter to FTDI, the block searches for FTDI cables. If the object finds:

- Exactly one FTDI cable, it uses that cable for communication with the FPGA board.
- More than one FTDI cable, it returns an error - To resolve this error, specify the desired cable using the **Cable name** parameter.
- No FTDI cables, it returns an error - To resolve this error, connect a Digilent or FTDI cable.

For more details, for Intel boards, see “Select from Multiple JTAG Cables”. For Xilinx boards, see “Select from Multiple JTAG Cables”.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Cable name — Name of JTAG cable used for communication with FPGA board
`auto` (default) | name of connected JTAG cable

Specify this parameter if more than one JTAG cable of the same type are connected to the host computer. If more than one JTAG cable is connected to the host computer, and you do not specify this parameter, the block returns an error. The error message contains the names of the available JTAG cables. For more details, for Intel boards, see “Select from Multiple JTAG Cables”. For Xilinx boards, see “Select from Multiple JTAG Cables”.

Dependencies

To enable this parameter, set **Type** to JTAG.

Clock frequency in MHz — JTAG clock frequency
`15` (default) | positive scalar

Specify the JTAG clock frequency in MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board. Check the board documentation for the supported frequency range.

Dependencies

To enable this parameter, set **Type** to JTAG.

Chain position — Position of FPGA in JTAG chain (Xilinx only)

auto (default) | nonnegative integer

Specify this parameter value as a nonnegative integer if more than one FPGA or Zynq device is on the JTAG chain. Otherwise, select **auto** (default) for automatic detection of chain position.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Instruction registers before FPGA — Sum of instruction register lengths for all devices before target FPGA (Xilinx only)

0 (default) | nonnegative integer

Specify this parameter value as a nonnegative integer if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Instruction registers after FPGA — Sum of instruction register length for all devices after target FPGA (Xilinx only)

0 (default) | nonnegative integer

Specify this parameter value as a nonnegative integer if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this parameter, set **Type** to JTAG and **Vendor** to Xilinx.

Device address — IP address of FPGA board

192.168.0.2 (default) | IP address

Specify the IP address of the Ethernet port on the FPGA board.

Example: 192.168.0.10

Dependencies

To enable this parameter, set **Type** to UDP.

Port — UDP port number of FPGA board

50101 (default) | integer from 255 to 65,535

Specify the user datagram protocol (UDP) port number of the target FPGA as an integer from 255 to 65,535.

Dependencies

To enable this parameter, set **Type** to UDP.

Version History

Introduced in R2019b**R2022a: AXI Master Write renamed to AXI Manager Write**

Warns starting in R2022a

The AXI Master Write block has been renamed to the AXI Manager Write block. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

In R2022a, you cannot use a Simulink model that contains the AXI Master Write block. Recreate your model in R2022a by using the AXI Manager Write block.

R2023a: Support for half data type

The block writes `half` data to the memory locations on the FPGA board. Before sending the write request to the FPGA, the function typecasts the `half` data to the `uint16` and then packs the data to `uint32` or `uint64`, depending on the AXI manager IP data width.

The address for the write operation must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, the block writes data to the lower 2 bytes and pads the higher 2 bytes with zeros.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, the block writes data to the lower 2 bytes and pads the higher 6 bytes with zeros.

See Also**Blocks**

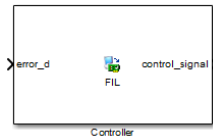
AXI Manager Read

Topics

"Set Up AXI Manager"

FIL Simulation

Simulate HDL code on FPGA hardware from Simulink



Libraries:
Generated

Description

The generated FPGA-in-the-loop (FIL) simulation block is the communication interface between the FPGA and your Simulink model. It integrates the hardware into the simulation loop and allows it to participate in simulation as any other block.

You can generate a FIL Simulation block from existing HDL code using the **FPGA-in-the-Loop Wizard**, or, generate HDL code and an accompanying FIL Simulation block using HDL Workflow Advisor. Generating HDL code requires an HDL Coder license.

For the generation and simulation workflow, see “Block Generation with the FIL Wizard”. If you encounter any issues during FIL simulation, refer to “Troubleshooting FIL” for help in diagnosing the problem.

You can use the FIL Simulation block in models running in Normal, Accelerator, or Rapid Accelerator simulation modes. The FIL Simulation parameters are not tunable in any of the simulation modes. For more information about these modes, see “How Acceleration Modes Work” (Simulink).

Ports

The ports of the block correspond to the interface of your HDL design running on your FPGA. You can configure the data types of the signals that the FIL Simulation block returns to Simulink.

Input

HDL_input_port_name — Signal passed from Simulink to FPGA
scalar | vector

The ports on the block correspond with ports on your HDL design. You can configure the **Sample time** and **Data type**

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

Output

HDL_output_port_name — Signal passed from the FPGA to Simulink
scalar | vector

The ports on the block correspond with ports on your HDL design. You can configure the **Sample time** and **Data type**

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

Parameters

The parameters displayed in the **Hardware Information** section reflect your selections when you generated the FIL Simulation block from a subsystem. These parameters are informational only.

- **Connection:** Either Ethernet or PCI Express. Some boards can use only one connection type or the other; with other boards, you may have the option of using either connection. You configure the **MAC address** and **IP address** of the board when you generate the block.
- **Board:** The make and model of FPGA board. For supported boards, see “Supported FPGA Devices for FPGA Verification”.
- **FPGA part:** Chip identification number.
- **FPGA project file:** The location of the FPGA project file generated for your design.

To download the generated FPGA programming file onto the FPGA, set the parameters in **FPGA Programming File**. This step is required before you can run a FIL simulation. See “Load Programming File onto FPGA”.

To configure data rate parameters, set options in the **Runtime Options** group.

On the **Signal Attributes** pane, you can configure **Sample time** and **Data type** for each output port. The direction and bit width of the signals, and the sample time and data type of the input ports, are informational only.

FPGA Programming File

File name — Location of programming file
string

Location of the FPGA programming file generated for your design. To load this design to the FPGA for simulation, click **Load**.

Runtime Options

Overclocking factor — FPGA sample rate relative to Simulink clock
1 (default) | integer

Ratio of FPGA clock rate to the Simulink clock rate. The FPGA clock samples inputs to the FPGA this many times for each Simulink timestep.

Output frame size — Amount of data returned to Simulink
Inherit: auto (default)

Output signals are returned as **Output frame size**-by-1 column vectors. Increasing the frame size can speed up your simulation by reducing the communication time between Simulink and the FPGA board.

Note these limitations on the frame size :

- The input frame size must be an integer multiple of the output frame size.
- The output frame size must be less than the input frame size.

- The input frame size and output frame size cannot vary during simulation.

Signal Attributes

Sample Time — Sample time of each port

Inherit: `Inherit via internal rule` (default)

Explicitly set sample times for the output signals, or use `Inherit: Inherit via internal rule`. The internal rule is to set the output sample times to the input base sample time divided by the scaling factor.

Data type — Data type of each port

`fixdt(0,N,0)` (default) | data type expression

How Simulink interprets the bits in the output signal from the FPGA. You can explicitly set output data types, use the default unscaled and unsigned type, or specify `Inherit: auto` to inherit a data type from context.

Version History

Introduced in R2012b

See Also

Topics

“FPGA-in-the-Loop Simulation”

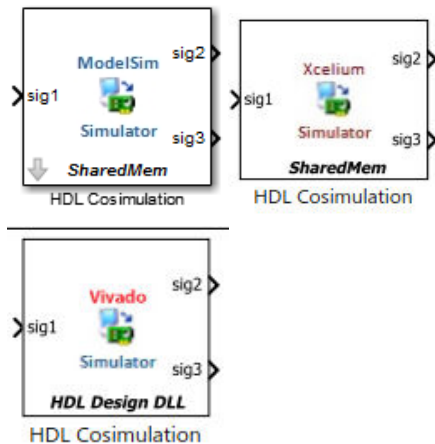
“FPGA-in-the-Loop Simulation Workflows”

“FIL Simulation with HDL Workflow Advisor for Simulink”

“Block Generation with the FIL Wizard”

HDL Cosimulation

Cosimulate HDL design by connecting Simulink with HDL simulator



Libraries:

HDL Verifier / For Use with Cadence Xcelium
 HDL Verifier / For Use with Mentor Graphics
 ModelSim
 HDL Verifier / For Use with Xilinx Vivado
 Simulator

Description

The 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. Configure and generate this block using the **Cosimulation Wizard**. When cosimulating with the Vivado® simulator, you must first generate this block using the **Cosimulation Wizard**. When cosimulating with ModelSim® or Xcelium™, you can optionally bypass the wizard and configure the block directly.

You can configure these options on the block:

- Mapping of the input and output ports of the block to correspond with signals (including internal signals) of an HDL module. You must specify a sample time for each output port. You can optionally specify a data type for each output port. Vivado cosimulation does not support mapping of internal signals.
- Type of communication and communication settings used to exchange data between simulators.
- The timing relationship between units of simulation time in Simulink and the HDL simulator.
- Rising-edge or falling-edge clocks to apply to your model. You can specify the period for each clock signal.
- Tcl commands to run before and after the simulation.

You can use this block to model a source or sink device by configuring the block with input or output ports only.

Compatibility with Simulink Code Generation

- This block participates in HDL code generation with HDL Coder. The coder generates an interface to your manually written or legacy HDL code. It does not participate in C code generation with Simulink Coder™.

Ports

The ports shown on the block correspond with signals from your HDL design running in the HDL simulator. The **Ports** tab displays the HDL signals that correspond to the ports.

When using Xcelium or ModelSim simulators, You can add and remove ports, and configure their data types and sample times, by changing the block parameters. Use the **Auto Fill** button to fill the table via a port information request to the HDL simulator. This request returns port names and information from your HDL design running in the HDL simulator (this step is not necessary if you used the **Cosimulation Wizard** to generate the block). See “Get Signal Information from HDL Simulator”.

All signals that you specify when you configure the HDL Cosimulation block must have read/write access in the HDL simulator. Refer to the HDL simulator product documentation for details.

Input

HDL_input_port_name — Signal passed from Simulink to HDL simulator
scalar | vector

The ports on the block correspond with ports on your HDL design. Add or remove ports on the **Ports** tab.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

Output

HDL_output_port_name — Signal passed from HDL simulator to Simulink
scalar | vector

The ports on the block correspond with ports on your HDL design. Add or remove ports on the **Ports** tab.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

Parameters

Block Info — Block information for Vivado cosimulation
block generation settings

This parameter is read-only.

This section displays information after you generate this block using the **Cosimulation Wizard**. It displays the following items:

- HDL Simulator - Vivado simulator
- HDL Design Library - Location of HDL library files
- HDL Language - VHDL or Verilog
- HDL Time Precision - Time precision of HDL design
- HDL Waveform File - Name and path of waveform file

When generating a cosimulation block for Vivado simulator, you cannot change the names and directions of ports, clocks, resets, or time precision using the block mask. To change these items, open the **Cosimulation Wizard** and regenerate the block.

Dependencies

This parameter is visible only when you use Vivado cosimulation.

Ports**Enable direct feedthrough** — Work around algebraic loop warnings

`true` (default) | `false`

Eliminates the one output-sample delay difference between the cosimulation and Simulink that occurs when your model contains purely combinational paths. Clear this check box if the HDL Cosimulation block is in a feedback loop and generates algebraic loop warnings or errors. When you simulate a sequential circuit that has a register on the data path, specifying direct feedthrough does not affect the timing of that data path.

Full HDL Name — Signal path name

`string`

Specify the signal path name using the HDL simulator path name syntax. For example, `manchester.samp` for Xcelium HDL simulators. The signal can be at any level of the HDL design hierarchy. The HDL Cosimulation block port corresponding to the signal is labeled with this name.

For rules on specifying port and module path names in Simulink, see “Specify HDL Signal/Port and Module Paths for Cosimulation”.

You can copy signal path names directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field. Use the *Path.Name* view and not *Db::Path.Name* view. After pasting a signal path name into **Full HDL Name**, click **Apply** to complete the paste operation and update the signal list.

Dependencies

If the block was generated from the **Cosimulation Wizard**, this value should not be changed. For the Vivado HDL Cosimulation block the parameter is read-only.

I/O Mode — Port direction

`Input` | `Output`

To add a bidirectional port, add the port to the list twice, as both input and output.

Input — HDL signals that Simulink drives. Simulink deposits values on the specified HDL simulator signal at the specified sample rate.

Note When you define a block input port, make sure that only one source is set up to drive input to that signal. For example, avoid defining an input port that has multiple instances. If multiple sources drive input to a single signal, your simulation model produces unexpected results.

Output — HDL signals that Simulink reads. For output signals, you must specify an explicit sample time. You can also specify the data type, but the width must match the width of the signal in HDL. For details on specifying a data type, see the **Data Type** and **Fraction Length** parameters.

Simulink signals do not have a tristate semantic because there is no 'Z' value. To interface with bidirectional signals, connect to the input and enable signals of both the output driver and the output signal of the input driver. This approach leaves the actual tristate buffer in HDL, where resolution functions can handle interfacing with other tristate buffers.

Dependencies

If the block was generated from the **Cosimulation Wizard**, this value should not be changed. For the Vivado HDL Cosimulation block the parameter is read-only.

HDL Type — Type of HDL port signal

Logic (default) | Vector

This parameter is read-only.

Displays the type of the HDL port signal, as configured by the **Cosimulation Wizard**.

Dependencies

This parameter is visible only if the block was generated for use with Vivado cosimulation.

HDL Dims — Dimension of HDL port signal

scalar | vector

This parameter is read-only.

Displays the dimensions of the HDL port signal, as configured by the **Cosimulation Wizard**.

Dependencies

This parameter is visible only if the block was generated for use with Vivado cosimulation.

Sample Time — Time between reading samples on an output port

1 (default) | integer

Time interval between consecutive samples applied to an output port.

Simulink reads a value from the associated HDL simulator signal at the sample rate specified here.

In general, Simulink handles port sample periods as follows:

- If you connect an input port to a signal that has an explicit sample period, based on forward propagation, Simulink applies that rate to the port.
- If you connect an input port to a signal that does not have an explicit sample period, Simulink assigns a sample period that is equal to the least common multiple (LCM) of all identified input port sample periods in the model.
- You must specify an explicit sample time for each output port.

The HDL time corresponding to the Simulink sample time hits depends on the **Timescales** setting. See “Simulation Timescales”.

Dependencies

To enable this parameter, set **I/O Mode** to Output.

Data Type — Simulink Data type for output signal

Inherit (default) | Fixedpoint | Double | Single | Half

Select **Inherit** to automatically determine the data type. The block checks that the inherited word length matches the word length queried from the HDL simulator. If they do not match, Simulink generates an error message. For example, if you connect a Signal Specification block to an output, **Inherit** forces the data type specified by the Signal Specification block onto the output port.

If Simulink cannot determine the data type of the signal connected to the output port, it queries the HDL simulator for the data type of the port. As an example, if the HDL simulator returns the VHDL® data type `STD_LOGIC_VECTOR` for a signal of size N bits, the data type `ufixN` is forced on the output port. The implicit fraction length is 0.

You can also assign an explicit data type, with optional **Fraction Length**. By explicitly assigning a data type, you can force fixed-point data types on output ports of the HDL Cosimulation block. For example, for an 8-bit output port, setting the **Sign** to **Signed** and setting the **Fraction Length** to 5 forces the data type to `sfix8_En5`. You cannot force width. The width is always inherited from the HDL simulator.

The **Data Type** and **Fraction Length** properties apply only to the following types of HDL signals:

- VHDL signals of any logic type, such as `STD_LOGIC` or `STD_LOGIC_VECTOR`
- Verilog® signals of `wire` or `reg` type

Dependencies

To enable this parameter, set **I/O Mode** to Output.

During cosimulation with the Vivado simulator, this parameter is displayed as **Simulink Data Type**.

Sign — Sign component of output data type

Unsigned (default) | Signed

Sign designation for explicit output port data type.

Dependencies

To enable this parameter, set **I/O Mode** to Output, and set **Data Type** to Fixedpoint.

Fraction Length — Number of fractional bits in output data type

integer

Size, in bits, of the fractional part of a fixed-point output signal. For example, for an 8-bit output port, setting the **Sign** to **Signed** and setting the **Fraction Length** to 5 forces the data type to `sfix8_En5`. You cannot force width; the width is always inherited from the HDL simulator.

Dependencies

To enable this parameter, set **I/O Mode** to Output, and **Data Type** property to Fixedpoint.

The **Data Type** and **Fraction Length** properties apply only to the following types of HDL signals:

- VHDL signals of any logic type, such as `STD_LOGIC` or `STD_LOGIC_VECTOR`
- Verilog signals of `wire` or `reg` type

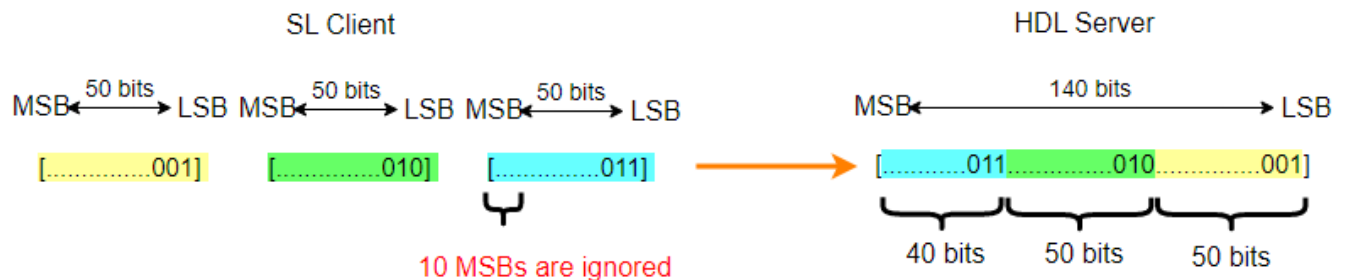
Simulink Word Length — Bit width of Simulink port
`integer <129 | inherit`

Size of the Simulink port, specified as a positive integer smaller than 129.

For input ports — this value is set to `inherit` (read only).

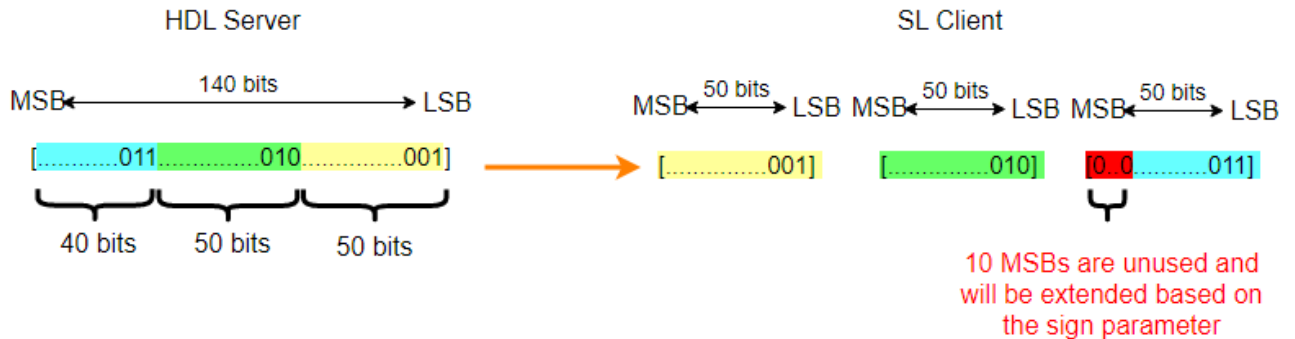
When the **HDL Word Length** parameter is greater than 128 bits the Simulink port dimensions are determined at compile time by the data type of the driving signal. For example:

- When **HDL Word Length** = 150 and **Simulink Word Length** = 50, HDL Verifier allows a Simulink port with data width of 50 bits, and dimensions of size 3 such as `sfix50(3)` or `ufix50(3)`.
- When **HDL Word Length** = 140 and **Simulink Word Length** = 50, HDL Verifier packs 150 bits of Simulink into 140 bits of HDL. HDL Verifier ignores the 10 most significant bits (MSB) of the last word.



For output ports — there are 2 cases.

- When the **HDL Word Length** parameter is less than 129 bits, this parameter matches the width of the HDL port, and it is read-only.
- When the **HDL Word Length** parameter is greater than 128 bits, HDL Verifier creates a vector of ports to represent this port. For example:
 - When **HDL Word Length** = 150 and **Simulink Word Length** = 50, HDL Verifier creates a Simulink port with data width of 50 bit. For example `sfix50(3)` or `ufix50(3)`.
 - When **HDL Word Length** = 150 and **Simulink Word Length** = 60, HDL Verifier creates a Simulink port with data width of 60, such as `sfix60(3)` or `ufix60(3)`. Since the HDL word has only 150 bits, and the Simulink port requires 180 bits, 30 bits are padded or sign extended.
 - When **HDL Word Length** = 140 and **Simulink Word Length** = 50, every 50 bits of the HDL output are represented as a Simulink word. The 10 MSB of the last Simulink word are unused and extended according to the **Sign** parameter.



Dependencies

When **HDL Word Length** is smaller than 129, this parameter is read-only.

HDL Word Length — Bit width of HDL port
integer

This parameter is read-only.

Size, in bits, of the HDL port.

Verilog or SystemVerilog example: In the code below, the input In1 has an HDL word length of 140 bits, and the output Out1 has an HDL word length of 160 bits.

```
input [1:0][69:0] In1;
output [159:0] Out1;
```

VHDL example: In the code below, In1 has an HDL word length of 120 bits, and In2 has an HDL word length of 1200 bits.

```
TYPE matrix_of_std_logic_vector120 IS ARRAY (NATURAL RANGE <>, NATURAL RANGE <>) OF std_logic_vector(119 DOWNT0 0);
In1: IN matrix_of_std_logic_vector120(0 TO 4, 0 TO 1);
In2: IN std_logic_vector(1199 DOWNT0 0);
```

Clocks

Note During cosimulation with the Vivado simulator, this tab is named **Clocks, Resets, Enables**.

Create optional rising-edge and falling-edge clocks that apply stimuli to your cosimulation model. The scrolling list displays HDL clocks that drive values to the HDL signals that you are modeling, using the deposit method. The clock signals must be single-bit signals. Vector signals are not supported. For instructions on adding and editing clock signals, see “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block”.

Time to run HDL simulator before cosimulation starts — time required for HDL simulation before cosimulation starts

0 (default) | nonnegative integer

Specify the time required for the HDL simulation to run before starting the cosimulation. Specify a nonnegative integer, and select time units from the menu.

- fs - Femtoseconds
- ps - Picoseconds
- ns - Nanoseconds
- us - Microseconds
- ms - Milliseconds
- s - Seconds

Dependencies

To enable this parameter, generate this block for Vivado cosimulation.

Full HDL Name — Signal path name

string

Specify each clock as a signal path name, using the HDL simulator path name syntax. For example: /manchester/clock or manchester.clk.

For information about and requirements for path specifications in Simulink, see “Specify HDL Signal/Port and Module Paths for Simulink Test Bench Cosimulation”.

You can copy signal path names directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field. Use the *Path.Name* view and not *Db::Path.Name* view. After pasting a signal path name into **Full HDL Name**, click **Apply** to complete the paste operation and update the signal list.

Dependencies

If the block was generated from the **Cosimulation Wizard**, this value should not be changed. For the Vivado HDL Cosimulation block the parameter is read-only.

Active Clock Edge — HDL clock edge used to sample signals

Rising (default) | Falling

Select one of the following options.

- Rising - Specify a rising-edge clock.
- Falling - Specify a falling-edge clock.

The periods and durations are Simulink times. To relate Simulink times to HDL times, go to the **Timescales** tab and click **Determine Timescale Now**.

Dependencies

This parameter is visible only for ModelSim or Xcelium cosimulation.

Waveform Type — Generate waveforms to drive clocks, resets, or enables to HDL design

Rising (default) | Falling | Step 1 to 0 | Step 0 to 1

For Vivado cosimulation, select one of the following options.

- Active Rising Edge Clock - Create a periodic signal with 50% duty cycle where the rising edge is offset from when Simulink drives the inputs.

- **Active Falling Edge Clock** - Create a periodic signal with 50% duty cycle where the falling edge is offset from when Simulink drives the inputs.
- **Step 0 to 1** - Create a step function that starts by driving a 0 for the specified duration, then transitions to 1.
- **Step 1 to 0** - Create a step function that starts by driving a 1 for the specified duration, then transitions to 0.

The periods and durations are Simulink times. To relate Simulink times to HDL times, go to the **Timescales** tab and click **Show Times and Suggest Timescale**.

Dependencies

This parameter is visible only for Vivado cosimulation.

Period/Duration — Clock period

2 (default) | integer

To specify an explicit clock period, enter a sample time equal to or greater than two resolution units (ticks).

If the clock period is not an even integer, Simulink cannot create a 50% duty cycle. Instead, the HDL Verifier software creates the falling edge at `clockperiod/2` (rounded down to the nearest integer).

For ModelSim or Xcelium periods and durations are specified as Simulink time, and for Vivado they are specified as HDL time.

Timescales

Choose a timing relationship between Simulink and the HDL simulator, either manually or automatically. These parameters specify 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:

- *Relative* timing relationship (Simulink seconds correspond to an HDL simulator precision, or "tick").
- *Absolute* timing relationship (Simulink seconds correspond to an absolute unit of HDL simulator time)

For more information on calculating relative and absolute timing modes, see "Defining the Simulink and HDL Simulator Timing Relationship".

For detailed information on the relationship between Simulink and the HDL simulator during cosimulation, and on the operation of relative and absolute timing modes, see "Simulation Timescales".

To see the relationship between the Simulink times and HDL times of all of the ports, clocks, resets, and enables click **Show Times and Suggest Timescale**. This action also automatically determines a usable timescale if needed.

Automatically determine timescale at start of simulation — When to calculate automatic timescale

true (default) | false

If you select this option, HDL Verifier calculates the timescale when you start the Simulink simulation. If this option is not selected, click **Determine Timescale Now** to calculate the timescale immediately without starting a simulation. For Vivado cosimulation, this button shows as **Show Times and Suggest Timescale**. Alternatively, you can manually select a timescale. For guidance through the automatic timescale calculation, see “Specify Timing Relationship Automatically”.

1 second in Simulink corresponds to {} in the HDL simulator — Timing relationship between Simulink and HDL simulator
integer and time units

This parameter consists of a *Time* value and a *TimeUnit* value.

To configure relative timing mode for a cosimulation:

- 1 Verify that *Tick*, the default setting for *TimeUnit*, is selected. If it is not, then select it from the list on the right.
- 2 Enter a scale factor in the *Time* text box on the left. The default scale factor is 1.

To configure absolute timing mode for a cosimulation:

- 1 Set *TimeUnit* to a unit of absolute time: *fs* (femtoseconds), *ps* (picoseconds), *ns* (nanoseconds), *us* (microseconds), *ms* (milliseconds), or *s* (seconds).
- 2 Enter a scale factor in the *Time* text box on the left. The default scale factor is 1.

Connection

This tab is not visible during cosimulation with the Vivado simulator.

Connection mode — Connection between Simulink and HDL simulator
Full Simulation (default) | Confirm Interface Only | No Connection

Type of connection between Simulink and the HDL simulator.

- **Full Simulation:** Confirm interface and run HDL simulation.
- **Confirm Interface Only:** Connect to the HDL simulator and check for signal names, dimensions, and data types, but do not run HDL simulation. During Simulink simulation, there is no contact with the HDL simulator.
- **No Connection:** Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

HDL simulator is running on this computer — Same host for HDL simulator and Simulink
true (default) | false

When both applications run on the same computer, you can choose shared memory or TCP sockets for the communication channel between the applications. If you do not select this option, only TCP/IP socket mode is available, and the **Connection method** list becomes unavailable.

Connection method — Connection between HDL simulator and Simulink
Socket (default) | Shared memory

- **Socket:** Simulink and the HDL simulator communicate via a designated TCP/IP socket. TCP/IP socket mode is more versatile. You can use it for single-system and network configurations. This

option offers the greatest scalability. For more on TCP/IP socket communication, see “TCP/IP Socket Ports”.

- **Shared memory:** Simulink and the HDL simulator communicate via shared memory. Shared memory communication provides optimal performance and is the default mode of communication.

Dependencies

This parameter shows when you select **HDL Simulator is running on this computer**.

Host name — HDL simulator host machine

string

This parameter applies if you run Simulink and the HDL simulator on different computers.

Port number or service — Socket port number

string

Indicate a valid TCP socket port number or service for your computer system, if you are not using shared memory. For information on choosing TCP socket ports, see “TCP/IP Socket Ports”.

Show connection info on icon — Add connection parameters on block icon

true (default) | false

When you select this option, the HDL Cosimulation block icon displays the current communication parameter settings. If you select shared memory, the icon displays `SharedMem`. If you select TCP socket communication, the icon displays `Socket` and displays the host name and port number in the format `hostname:port`.

This information can help you distinguish between multiple HDL Cosimulation blocks, where each block is communicating to a different instance of the HDL simulator.

Simulation

This tab is not visible during cosimulation with the Vivado simulator.

Time to run HDL simulator before cosimulation starts — Offset that aligns Simulink with HDL simulator

integer and time unit

Specifies the amount of time to run the HDL simulator before beginning simulation in Simulink. Specifying this time properly aligns the signal of the Simulink block and the HDL signal so that they can be compared and verified directly without additional delays.

This setting consists of a *PreRunTime* value and a *PreRunTimeUnit* value.

- *PreRunTime*: Any valid time value. The default is 0.
- *PreRunTimeUnit*: Specifies the units of time for *PreRunTime*.
 - Tick
 - s
 - ms
 - us

- ns
- ps
- fs

Pre-simulation Tcl commands — Commands to run in HDL simulator before cosimulation
string

The cosimulation tool executes these commands in the HDL simulator, before simulating the HDL component of your Simulink model. If you enter multiple commands on one line, append each command with a semicolon (;), the standard Tcl concatenation operator.

For example, use this parameter to generate a one-line echo command to confirm that a simulation is running, or a complex script that performs an extensive simulation initialization and startup sequence. You cannot use these commands to change simulation state.

You can specify any valid Tcl command. The Tcl command you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the character vector cannot include commands such as `start`, `stop`, or `restart` (for ModelSim) or `run`, `stop`, or `reset` (for Xcelium).

Post-simulation Tcl commands — Commands to run in HDL simulator after cosimulation
string

The cosimulation tool executes these commands in the HDL simulator, after simulating the HDL component of your Simulink model.

You can specify any valid Tcl command. The Tcl command you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the string cannot include commands such as `start`, `stop`, or `restart` (for ModelSim) or `run`, `stop`, or `reset` (for Xcelium).

Note After each ModelSim simulation, the simulator takes 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.

Version History

Introduced in R2008a

Extended Capabilities

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. You can generate HDL code for cosimulation blocks used with Mentor Graphics® ModelSim, Cadence® Xcelium, or Xilinx Vivado simulators.

Each of the HDL Cosimulation blocks cosimulates a hardware component by applying input signals to, and reading output signals from, an HDL model that executes under an HDL simulator. See “Generate a Cosimulation Model” (HDL Coder).

For information about timing, latency, data typing, frame-based processing, and other issues when setting up an HDL cosimulation, see “Define HDL Cosimulation Block Interface”.

You can use an HDL Cosimulation block with HDL Coder to generate an interface to your manually written or legacy HDL code. When an HDL Cosimulation block is included in a model, the coder generates a VHDL or Verilog interface, depending on the selected target language.

When the target language is VHDL, the generated interface includes:

- An entity definition. The entity defines ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. Clock enable and reset ports are also declared.
- An RTL architecture including a component declaration, a component configuration declaring signals corresponding to signals connected to the HDL Cosimulation ports, and a component instantiation.
- Port assignment statements as required by the model.

When the target language is Verilog, the generated interface includes:

- A module defining ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. The module also defines clock enable and reset ports, and wire declarations corresponding to signals connected to the HDL Cosimulation ports.
- A module instance.
- Port assignment statements as required by the model.

Before initiating code generation, check that the model meets the requirements for code generation. To check the requirements for code generation, select the **Debug** tab, and then click **Update Model**.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

For implementation parameter descriptions, see “Customize Black Box or HDL Cosimulation Interface” (HDL Coder).

See Also

`hdlverifier.HDLCosimulation`

Topics

“Import HDL Code for HDL Cosimulation Block”
“Create Simulink Model for Component Cosimulation”
“Create a Simulink Cosimulation Test Bench”
“Run a Simulink Cosimulation Session”
“Simulation Timescales”
“Clock, Reset, and Enable Signals”

Sequence Feedback

Connect between scoreboard and sequence in UVM test bench model



Libraries:

HDL Verifier / For Use with DPI-C SystemVerilog

Description

The Sequence Feedback block promotes a feedback signal from the scoreboard to the sequence in a UVM test bench model. The block implements a Unit Delay block with the **Initial condition** parameter set to 0.

Ports

Input

Port_1 — Feedback signal from scoreboard subsystem
scalar

The Sequence Feedback block accepts an input signal directly from the scoreboard subsystem.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Output

Port_1 — Feedback signal to sequence subsystem
scalar

The Sequence Feedback block outputs a signal directly to the sequence subsystem.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Version History

Introduced in R2023a

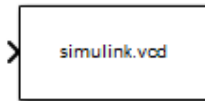
See Also

Topics

“UVM Component Generation Overview”

To VCD File

Generate value change dump (VCD) file



Libraries:

HDL Verifier / For Use with Cadence Xcelium
 HDL Verifier / For Use with Mentor Graphics ModelSim
 HDL Verifier / For Use with Xilinx Vivado Simulator

Description

The To VCD File block generates a VCD file that logs changes to its input ports. You can use VCD files during design verification in these ways:

- Compare results of multiple simulation runs, using the same or different simulator environments.
- Provide input to post-simulation analysis tools.
- Porting areas of an existing design to a new design.

You can specify the following parameters:

- Name of the generated VCD file
- Number of block input ports
- Timescale, that relates Simulink sample times with HDL simulator ticks

VCD files can grow large for large designs or small designs with long simulation runs. The maximum number of signals supported in a generated VCD file is 94^3 (830,584).

You can use the To VCD File block in models running in normal, accelerator, or rapid accelerator simulation modes. The To VCD File parameters are not tunable in any of the simulation modes. For more information about these modes, see “How Acceleration Modes Work” (Simulink).

The To VCD File block is integrated into the Simulink Viewers and Generators Manager. When you add a VCD block to a model using the manager, the signal name that appears in the VCD file may not be the one you specified. After simulation, open the VCD file and check the signal name. If you cannot find the signal name you specified, look for an automatic signal name such as *In_1*. When you use the VCD block directly from the HDL Verifier library, the signal names match correctly.

Note The To VCD File block does not support framed signals.

VCD File Format

The format of generated VCD files adheres to IEEE® Std 1364-2001. The table describes the format.

VCD File Content	Description
\$date 23-Sep-2003 14:38:11 \$end	Date and time the file was generated.

VCD File Content	Description
<code>\$version HDL Verifier version 1.0 \$ end</code>	Version of the To VCD File block that generated the file.
<code>\$timescale 1 ns \$ end</code>	Timescale used during the simulation.
<code>\$scope module manchestermodel \$end</code>	Scope of module being dumped.
<code>\$var wire 1 ! Original Data [0] \$end \$var wire 1 " Recovered Clock [0] \$end \$var wire 1 # Recovered Data [0] \$end \$var wire 1 \$ Data Validity [0] \$end</code>	Variable definitions. Each definition associates a signal with a character identification code (symbol). The symbols are derived from printable characters in the ASCII character set from ! to ~. Variable definitions also include the variable type (wire) and size in bits.
<code>\$upscope \$end</code>	Marks a change to the next highest level in the HDL design hierarchy.
<code>\$enddefinitions \$end</code>	Marks the end of the header and definitions section.
<code>#0</code>	Simulation start time.
<code>\$dumpvars 0! 0" 0# 0\$ \$end</code>	Lists the values of all defined variables at time 0.
<code>#630 1!</code>	Starting point of logged value changes from checks of variable values made at each simulation time increment. This entry indicates that at 63 nanoseconds, the value of signal <code>Original Data</code> changed from 0 to 1.
<code>. . . . #1160 1# 1\$</code>	At 116 nanoseconds, the values of signals <code>Recovered Data</code> and <code>Data Validity</code> changed from 0 to 1.
<code>\$dumpoff x! x" x# x\$ \$end</code>	Marks the end of the file by dumping the values of all variables as the value x.

Display VCD File Data

You can display VCD file data graphically or analyze the data with postprocessing tools. For example, the ModelSim `vcd2wlf` tool converts a VCD file to a WLF file, which you can view in a ModelSim **wave** window. Other examples of postprocessing include the extraction of data pertaining to a particular section of a design hierarchy or data generated during a specific time interval.

Ports

Specify the number of signals to log using **Number of input ports**. The block has no output ports.

Input

Port_1, Port_2, ..., Port_N — Signal to log to VCD file
scalar | vector | matrix

Multi-dimensional signals are flattened to 1-D vectors in the VCD file.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

Parameters

VCD file name — Name of generated VCD file
string

Name of the generated VCD file. If you specify a file name only, Simulink places the file in your current MATLAB folder. To place the generated file in a different location, specify a complete path name. If you specify the same name for multiple To VCD File blocks, Simulink automatically adds a numeric postfix to identify each instance uniquely.

Note To save the generated file with the `.vcd` file extension, you must specify it explicitly.

Number of input ports — Number of input signals to log
integer

Number of input signals to log data from. The block can log up to 94^3 (830,584) signals, each of which maps to a unique symbol in the VCD file.

In some cases, a single input port maps to multiple symbols. This multiple mapping occurs when the input port receives a multidimensional signal. Because the VCD specification does not include multidimensional signals, Simulink flattens them to a 1-D vector in the file.

Timescale — Timing relationship between Simulink and the HDL simulator
integer and time units

Timing relationship, defined as the correspondence between one second of Simulink time and some quantity of HDL simulator time. You can express this quantity of HDL simulator time in one of the following ways:

- In *relative* terms, that is, as some number of HDL simulator ticks. In this case, the cosimulation operates in *relative timing mode*, which is the timing mode default.

To use relative mode, in the **1 second in Simulink corresponds to {value} {unit} in the HDL simulator** parameter, set the unit to `Tick`, and the value to the number of ticks you want. The default value is 1 tick.

- In *absolute* units, such as milliseconds or nanoseconds. In this case, the cosimulation operates in *absolute timing mode*.

To use absolute mode, in the **1 second in Simulink corresponds to {value} {unit} in the HDL simulator** parameter, set the number of resolution units and the type of unit (`fs`, `ps`, `ns`, `us`, `ms`, `s`). Then, in the **1 HDL Tick is defined as** parameter, set the value of the HDL simulator tick to 1, 10, or 100, and choose a resolution unit.

Version History

Introduced in R2008a

Extended Capabilities

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Architecture

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

See Also

Topics

“Add a Value Change Dump (VCD) File”

“Visually Compare Simulink Signals with HDL Signals”

“Simulation Timescales”

System Objects

hdlverifier.FILSimulation

Package: hdlverifier

FIL simulation with MATLAB

Description

The `FILSimulation` System object™ connects an FPGA execution to a MATLAB test bench. It does so by applying input signals to and reading output signals from an HDL model running on an FPGA. You can use this object to model a source or sink device by configuring the object with input or output ports only.

To run a simulation consisting of a MATLAB test bench communicating with an FPGA execution:

- 1 Customize the `hdlverifier.FILSimulation` object using **FPGA-in-the-Loop Wizard**.
- 2 Create the object in your design and set its properties.
- 3 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Creation

To create an `hdlverifier.FILSimulation` System object, use the **FPGA-in-the-Loop Wizard** to customize the `FILSimulation` System object. The output of the `FILWizard` is a file called `toplevel_fil`, where `toplevel` is the name of the top level HDL module. You can then create the System object by assigning it to a local variable.

`filobj = toplevel_fil` creates the System object customized by the `FPGA-in-the-Loop Wizard`. `toplevel` is the name of the top-level module in your HDL code.

You can create the System object and set its properties:

```
filobj = toplevel_fil('InputSignals', {'/top/in1','/top/in2'}, ...  
                    'OutputSignals', {'/top/out1','/top/out2'}, ...  
                    'OutputDataTypes', {'double','fixedpoint'}, ...  
                    'OutputSigned', [true,false]);
```

You can also adjust writable properties after creating the System object:

```
filobj = toplevel_fil;  
filobj.OutputDataTypes = char('fixedpoint', 'integer', 'fixedpoint');  
filobj.OutputSigned = [false, true, true];
```

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

Connection — Parameters for connection with FPGA board

`char('UDP', '192.168.0.2', '00-0A-35-02-21-8A')` (default) | character vector | string scalar

This property is read-only.

Parameters for the connection with the FPGA board, specified as a character vector or string scalar. The vector consists of three parts:

- Connection type
- Board IP address
- Board MAC address (optional)

Example: `char('UDP', '192.168.0.2', '00-0A-35-02-21-8A')` specifies a UDP connection to IP address 192.168.0.2, where the board's MAC address is 00-0A-35-02-21-8A.

DUTName — DUT top-level name

`' '` (default) | character vector | string scalar

This property is read-only.

Design under test (DUT) top-level name, specified as a character vector or string scalar.

Example: `'inverter_top'`

FPGABoard — FPGA board name

`' '` (default) | character vector | string scalar

This property is read-only.

FPGA board name, specified as a character vector or string scalar.

FPGAProgrammingFile — Path to FPGA programming file

`' '` (default) | character vector | string scalar

Path to the FPGA programming file, specified as a character vector or string scalar.

Example: `'c:\work\filename'`

FPGAVendor — Name of FPGA chip vendor

`'Xilinx'` (default) | `'Altera'` | `'Microsemi'`

This property is read-only.

Name of the FPGA chip vendor, specified as `'Xilinx'`, `'Microsemi'`, or `'Altera'`.

Example: `'Altera'`

InputBitWidths — Input widths in bits

`0` (default) | integer | vector of integers

This property is read-only.

Input widths in bits, specified as an integer or a vector of integers. When this property is an integer, all inputs have the same bit width. When this property is a vector of integers, the vector must be the same size as the number of inputs, where each value specifies a different input width.

Example: `10` - All inputs are ten bits wide.

Example: [12, 6, 1] - The design has three inputs: One is 12 bits wide, one is 6 bits wide, and one is 1 bit wide.

InputSignals — Input paths in HDL code

' ' (default) | character vector | cell array of character vectors | string scalar | string array

This property is read-only.

Input paths in the HDL code, specified as a character vector, cell array of character vectors, string scalar, or string array.

Example: '/top/in1'

Example: char('in1','in2')

OutputBitWidths — Output widths, in bits

0 (default) | integer | vector of integers

This property is read-only.

Output widths in bits, specified as an integer or a vector of integers.

If you specify a scalar, the outputs each have the same bit width. If you specify a vector, the vector must be the same size as the number of outputs.

Example: 10 - All outputs are 10 bits wide.

Example: [12, 6, 1] - The design has three outputs: one is 12 bits wide, one is 6 bits wide, and one is 1 bit wide.

OutputDataTypes — Output data types

'fixedpoint' (default) | character vector | cell array of character vectors | string scalar | string array

Output data types, specified as a character vector, cell array of character vectors, string scalar, or string array.

If you specify only one data type, all outputs have the same type. Otherwise, specify a cell array of the same size as the number of outputs.

Example: 'integer'

Example: char('integer','fixedpoint','integer')

OutputDownsampling — Downsampling factor and phase of outputs

[1, 0] (default) | vector of two integers

Downsampling factor and phase of the outputs, specified as a vector of two integers. The first integer specifies the downsampling factor and is positive. The second integer specifies the phase and is either zero or positive but less than the downsampling factor.

Example: [3, 1]

OutputFractionLengths — Output fraction lengths

0 (default) | integer | vector of integers

Output fraction lengths, specified as an integer or as a vector of integers.

If you only specify a scalar, each output has the same fraction length. Otherwise specify a vector of the same size as the number of outputs.

Example: 10 — All output fraction lengths are 10 bits.

Example: [16, 8] — One output fraction length is 16 bits, and the other one has a fraction length of 8 bits.

OutputSignals — Output port name in HDL top level

' ' (default) | character vector | cell array of character vectors | string scalar | string array

This property is read-only.

Output port names in the HDL top-level module, specified as a character vector, cell array of character vectors, string scalar, or string array.

Example: 'out1',

Example: char('out1', 'out2')

OutputSigned — Sign of outputs

false (default) | true | logical vector

Sign of the outputs, specified as false (unsigned), true (signed), or as a logical vector.

If you provide only a scalar, each output has the same sign. Otherwise, you should provide a vector of the same size as the number of outputs.

Example: true

Example: [true, true, false] — Three outputs consisting of a signed value, an unsigned value, and a signed value.

OverclockingFactor — Hardware overclocking factor

1 (default) | integer

Hardware overclocking factor, specified as an integer.

Example: 3

ScanChainPosition — Position of FPGA in JTAG scan chain

1 (default) | positive integer

This property is read-only.

Position of the FPGA in the JTAG scan chain, specified as a positive integer.

Example: 1

SourceFrameSize — Frame size of source (only for HDL source block)

1 (default) | integer

Frame size of the source, specified as an integer. This property is relevant only for HDL source blocks, that is, HDL blocks that have no inputs.

Example: 1

Usage

Syntax

```
[hdloutputs] = filobj([hdlinputs])
```

Description

[hdloutputs] = filobj([hdlinputs]) connects to the FPGA, writes hdlinputs to the FPGA and reads hdloutputs from the FPGA.

Input Arguments

hdlinputs — Inputs to run on FPGA

types are as specified by InputBitWidths property

Inputs to run on the FPGA, specified as an array of values. The size of the array must match the number of inputs of the module executed on the FPGA.

Example: [RealFft, ImagFft] = fft_obj(3,12); the values 3 and 12 are driven into the FPGA.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | Fixed-point

Output Arguments

hdloutputs — Outputs returned from FPGA

'' (default) | character vector | cell array of character vectors | string scalar | string array

Outputs returned from the FPGA, specified as an array of values. The size of the array matches the number of outputs of the module executed on the FPGA.

Example: [RealFft, ImagFft] = fft_obj(real_in,imaginary_in); returns a complex number from the FPGA with two values: RealFft and ImagFft.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | Fixed-point

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

Specific to hdlverifier.FILSimulation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

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.

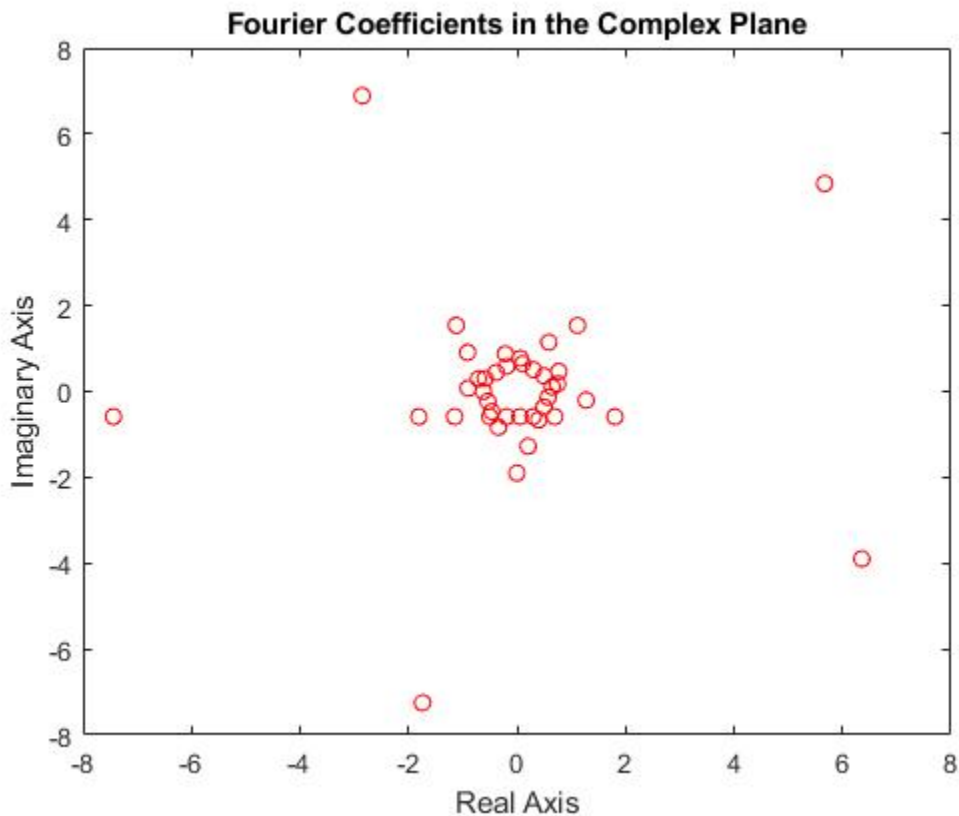
```
for ii=1:1000
    % Read 1 sample from the sine wave generator
    ComplexSinus = step(SinGenerator);
    % Send/receive 1 sample to/from the HDL FFT on the FPGA
    [RealFft, ImagFft] = step(Fft,real(ComplexSinus),imag(ComplexSinus));
    % Store the FFT sample in a vector
    ComplexFft(ii) = RealFft + 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.

Version History

Introduced in R2012b

See Also

FIL Simulation | FPGA-in-the-Loop Wizard

Topics

"FPGA-in-the-Loop Simulation Workflows"

hdlverifier.HDLCosimulation

Package: hdlverifier

Create a System object for HDL cosimulation with MATLAB

Description

The `hdlverifier.HDLCosimulation` System object cosimulates MATLAB and a hardware component. The System object writes input signals to and reads output signals from an HDL model under simulation in the HDL simulator. You can use this System object to model a source or sink device by configuring the System object with only output or input ports, respectively.

To create a System object for HDL cosimulation with MATLAB:

- 1 Customize the `hdlverifier.HDLCosimulation` object using **Cosimulation Wizard**.
- 2 Create the object in your design and set its properties.
- 3 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

Creation

To create an `hdlverifier.HDLCosimulation` System object, use the **Cosimulation Wizard** to customize the `HDLCosimulation` System object. The output of the Cosim Wizard is a file called `hdlcosim_toplevel.m`, where *toplevel* is the name of the top level HDL module. You can then create the System object by assigning it to a local variable.

Syntax

```
hdlc = hdlverifier.HDLCosimulation
hdlc = hdlverifier.HDLCosimulation(Name,Value)
hdlc = hdlcosim
hdlc = hdlcosim(Name,Value)
```

Description

`hdlc = hdlverifier.HDLCosimulation` creates an `hdlverifier.HDLCosimulation` System object with default property values. This System object provides an interface to your HDL simulation in your MATLAB workspace.

`hdlc = hdlverifier.HDLCosimulation(Name,Value)` specifies properties by one or more *Name,Value* pairs. Enclose each property name in single quotes. For example,

```
hdlc = hdlverifier.HDLCosimulation('InputSignals','/top/in1', ... ,
'OutputFractionLengths',10);
```

`hdlc = hdlcosim` creates an `hdlverifier.HDLCosimulation` System object with default property values. This syntax is equivalent to the `hdlverifier.HDLCosimulation` syntax.

`hdlc = hdlcosim(Name,Value)` is equivalent to the `hdlverifier.HDLCosimulation(Name,Value)` syntax.

The **Cosimulation Wizard** creates an `hdlverifier.HDLCosimulation System` object using existing HDL code, and an HDL launch script. Use the **Cosimulation Wizard** for easier startup.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

InputSignals — Input paths in HDL code

' ' (default) | character vector | cell array of character vectors

Input paths in the HDL code, specified as a character vector or cell array of character vectors. The paths are specified relative to the top level of the HDL hierarchy.

Example: 'data_in'

Example: {'/top/in1','/top/in2'}

Data Types: char | cell

OutputSignals — Output paths in HDL code

' ' (default) | character vector | cell array of character vectors

Output paths in the HDL code, specified as a character vector or cell array of character vectors. The paths are specified relative to the top level of the HDL hierarchy.

Example: 'out1'

Example: {'out1','out2'}

Data Types: char | cell

OutputDataTypes — Data types of output signals

' ' (default) | 'fixedpoint' | 'double' | 'single'

Data types of the output signals, specified as a cell array of character vectors. Valid data types are 'fixedpoint', 'double', or 'single'.

If you specify only one data type, each output has that same data type. To assign different data types to each output, specify a cell array of the same size as the number of outputs. Each element in the `OutputDataTypes` cell array specifies the data type of the corresponding element in the System object output (`hdloutputs`).

Example: {'fixedpoint'} - All output data types are fixedpoint.

Example: {'double','single'} - The data type of the first output is double and the second is single.

Note When `OutputDataTypes` is `{'fixedpoint'}`, the bit-width matches the size of a built-in data type (8,16,32, or 64), and `OutputFractionLengths` is set to 0, the data type of the output signal is returned as that built-in data type.

Data Types: cell

OutputSigned — Sign of outputs

false (default) | true | logical vector

Sign of the outputs, specified as false (unsigned), true (signed), or a logical vector.

If you provide only true or false, each output has that corresponding sign. To apply different signs to each output, specify a logical vector of the same size as the number of outputs. Each element in the `OutputSigned` vector specifies the sign of the corresponding element in the System object output (`hdloutputs`).

Example: true - All outputs have a signed value.

Example: [true, true, false] — The first output is a signed value, the second output is a signed value, and the third (and final) output is an unsigned value.

OutputFractionLengths — Output fraction lengths

0 (default) | integer | vector of integers

Output fraction lengths, in bits, specified as an integer or vector of integers.

If you specify only a scalar, each output has that same fraction length. To apply different fraction lengths to each output, specify a vector of the same size as the number of outputs. Each element in the `OutputFractionLengths` vector specifies the fraction length of the corresponding element in the System object output (`hdloutputs`).

Example: 10 — All outputs have a fraction length of 10 bits.

Example: [16, 8] — The first output has a fraction length of 16 bits, and the second (and final) output has a fraction length of 8 bits.

TCLPreSimulationCommand — Tool Command Language (Tcl) presimulation command executed by HDL simulator

' ' (default) | character vector

Tcl pre simulation command executed by the HDL simulator during the first call to the System object, specified as a character vector. This Tcl presimulation command is also executed during the first call to the System object after it is released.

Example: 'force /top/rst 1 0, 0 2 ns; force /top/clk 0 0, 1 1 ns -repeat 2 ns'

Data Types: char

TCLPostSimulationCommand — Tcl post-simulation command executed by HDL simulator

' ' (default) | character vector

Tcl post simulation command executed by the HDL simulator during a call to release the System object, specified as a character vector.

Example: 'echo "done"'

Data Types: char

PreRunTime — Delay in HDL simulator before cosimulation`{0, 'ns'}` (default) | cell array

Delay in HDL simulator before the cosimulation starts, specified as a cell array with two elements.

- The first element is the HDL presimulation delay, specified as a nonnegative integer.
- The second element is the time unit, specified as one of these character vectors: 'fs', 'ps', 'ns', 'us', 'ms', or 's'.

Example: `{10, 'fs'}`

Data Types: cell

Connection — Parameters for connection to HDL simulator`{'SharedMemory'}` (default) | cell array

Parameters for the connection to the HDL simulator, specified as a cell array with one, two, or three elements.

- The first element is the connection type, specified as 'SharedMemory' or 'Socket'. If specifying shared memory, then the port number and host name (the second and third elements in this cell array) are not applicable.
- The second element is the port number, which must be a positive integer. This value is set to 4449 if not specified.
- The third element is the host name of the HDL session. This value is set to localhost if not specified.

Example: `{'SharedMemory'}`

Example: `{'Socket', 1234}`

Example: `{'Socket', 1234, 'hostname'}`

Data Types: cell

FrameBasedProcessing — Enable frame-based processing`false` (default) | true

Note The `FrameBasedProcessing` property will be removed in a future release.

Sample mode or frame mode is automatically detected based on the size of the inputs during the System object execution.

SampleTime — Elapsed simulator time between calls to the System object`{10, 'ns'}` (default) | cell array

Elapsed time in the HDL simulator between each call to the System object, specified as a cell array with two elements.

- The first element is the time between two calls to the System object, specified as a positive integer.
- The second element is the time unit, specified as a character vector: 'fs', 'ps', 'ns', 'us', 'ms', 's'.

Example: {10, 'ns'}

Data Types: cell

Usage

Syntax

```
hdloutputs = hdlc(hdlinputs)
```

Description

`hdloutputs = hdlc(hdlinputs)` connects to the HDL simulator, writes `hdlinputs` to the HDL simulator, and reads `hdloutputs` from the HDL simulator. The elapsed simulation time between each call to the System object is defined by the `SampleTime` property.

Input Arguments

hdlinputs — Inputs to HDL simulator

comma-separated list of values for HDL input ports

Inputs to the HDL simulator, specified as a comma-separated list of values that are driven to your HDL input ports. The HDL input ports are set by the `InputSignals` property. The number of elements in this comma-separated pair must equal the number of HDL input ports. Each input argument value is driven to its corresponding HDL input port.

For example, if `InputSignals` is set as {'in1', 'in2'}, specify `out = hdlc(input1, input2)` to drive the value `input1` to `in1` and `input2` to `in2`.

Example: `[RealFft, ImagFft] = hdlc(3, 12)`; the values 3 and 12 are driven as inputs to the HDL simulator, which has two input ports.

Output Arguments

hdloutputs — Outputs from the HDL simulator

scalar | vector

Outputs from the HDL simulator, returned as a scalar or vector. Each returned element is the output from its corresponding HDL output port. The HDL output ports are specified in the `OutputSignals` property. The number of elements returned is the same as the number of HDL output ports specified. For example, if `OutputSignals` is set as {'out1', 'out2'}, specify `[o1, o2] = hdlc(i1, i2)` to assign the value from `out1` to `o1` and `out2` to `o2`.

Example: `out1 = hdlc(3, 12)`; assigns the output value from an HDL simulator with one output port.

Example: `[RealFft, ImagFft] = hdlc(3, 12)`; assigns output values from an HDL simulator with two output ports. In this example, `RealFft` is the output from the first port and `ImagFft` is the output from the second port.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

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/QuestaSim, 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
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 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/clock' '/viterbi_block/reset' '/viterbi_block/clock_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.

```
fprintf('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

Version History

Introduced in R2012b

See Also**Functions**

`hdlverifier.VivadoHDLCosimulation`

Blocks

HDL Cosimulation

Tools

Cosimulation Wizard

Topics

“Create a MATLAB System Object”

hdlverifier.VivadoHDLCosimulation

Package: hdlverifier

Create a System object for HDL cosimulation with the Vivado simulator and MATLAB

Description

The `hdlverifier.VivadoHDLCosimulation` System object cosimulates MATLAB and a hardware component using the Vivado simulator. The system object writes input signals to and reads output signals from an HDL model under simulation in the HDL simulator. You can use this System object to model a source or sink device by configuring the System object with only output or input ports, respectively.

To create a System object for HDL cosimulation with MATLAB:

- 1 Create a customized `hdlverifier.VivadoHDLCosimulation` object using **Cosimulation Wizard**.
- 2 Assign the object to a variable in your design.
- 3 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#).

Creation

To create an `hdlverifier.VivadoHDLCosimulation` System object, you must first use the **Cosimulation Wizard** to generate a customized `VivadoHDLCosimulation` System object. The output of the **Cosimulation Wizard** is a file called `hdlcosim_topLevel.m`, where *topLevel* is the name of the top level HDL module. You can then create the System object by assigning it to a local variable.

Syntax

```
hdlc = hdlcosim_topLevel
```

Description

`hdlc = hdlcosim_topLevel` creates an `hdlverifier.VivadoHDLCosimulation` System object, where *topLevel* is the name of your top level HDL module. The properties of this System object are configured by the **Cosimulation Wizard**. This System object provides an interface to your HDL simulation in your MATLAB workspace.

`hdlcosim_topLevel` is created and configured with the **Cosimulation Wizard**, and that is the recommended syntax to use.

After assigning the object to a variable, you can change properties by assigning a value to it. For example, to change the fraction length value:

```
hdlc = hdlcosim_myTopLevel;  
hdlc.OutputFractionLengths = 10;
```


The **Cosimulation Wizard** creates an `hdlverifier.VivadoHDLCosimulation System` object using existing HDL code, and an HDL launch script. Use the **Cosimulation Wizard** for easy startup.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see *System Design in MATLAB Using System Objects*.

Note that only the following properties can be changed: `OutputSigned`, `OutputDataTypes`, `OutputFractionLengths`, `ClockResetTypes`, `ClockResetTimes`, `PreRunTime`, `SampleTime`. Other properties should only be configured with the **Cosimulation Wizard**.

InputSignals — Input paths in HDL code

' ' (default) | string | character vector | cell array of character vectors

This property is read-only.

Input paths in the HDL code, specified as a string, character vector, or cell array of character vectors. The paths are specified relative to the top level of the HDL hierarchy.

Example: `'data_in'`

Example: `{ '/top/in1', '/top/in2' }`

Data Types: `char` | `cell` | `string`

OutputSignals — Output paths in HDL code

' ' (default) | string | character vector | cell array of character vectors

This property is read-only.

Output paths in the HDL code, specified as a string, character vector, or cell array of character vectors. The paths are specified relative to the top level of the HDL hierarchy.

Example: `'out1'`

Example: `{ 'out1', 'out2' }`

Data Types: `char` | `cell` | `string`

OutputDataTypes — Data types of output signals

' ' (default) | `'fixedpoint'` | `'double'` | `'single'`

Data types of the output signals, specified as a cell array of character vectors. Valid data types are `'fixedpoint'`, `'double'`, or `'single'`.

If you specify only one data type, each output has that same data type. To assign different data types to each output, specify a cell array of the same size as the number of outputs. Each element in the `OutputDataTypes` cell array specifies the data type of the corresponding element in the System object output (`hdloutputs`).

Example: `{ 'fixedpoint' }` - All output data types are `fixedpoint`.

Example: `{'double', 'single'}` - The data type of the first output is double and the second is single.

Note When `OutputDataTypes` is `{'fixedpoint'}`, the bit-width matches the size of a built-in data type (8,16,32, or 64), and `OutputFractionLengths` is set to 0, the data type of the output signal is returned as that built-in data type.

Data Types: cell

OutputSigned – Sign of outputs

false (default) | true | logical vector

Sign of the outputs, specified as false (unsigned), true (signed), or a logical vector.

If you provide only true or false, each output has that corresponding sign. To apply different signs to each output, specify a logical vector of the same size as the number of outputs. Each element in the `OutputSigned` vector specifies the sign of the corresponding element in the System object output (`hdloutputs`).

Example: true - All outputs have a signed value.

Example: [true, true, false] — The first output is a signed value, the second output is a signed value, and the third (and final) output is an unsigned value.

OutputFractionLengths – Output fraction lengths

0 (default) | integer | vector of integers

Output fraction lengths, in bits, specified as an integer or vector of integers.

If you specify only a scalar, each output has that same fraction length. To apply different fraction lengths to each output, specify a vector of the same size as the number of outputs. Each element in the `OutputFractionLengths` vector specifies the fraction length of the corresponding element in the System object output (`hdloutputs`).

Example: 10 — All outputs have a fraction length of 10 bits.

Example: [16, 8] — The first output has a fraction length of 16 bits, and the second (and final) output has a fraction length of 8 bits.

ClockResetSignals – Clock and reset signals to drive in HDL code

' ' (default) | string | character vector | cell array of strings | cell array of character vectors

This property is read-only.

Clock and reset signals to drive in the HDL code, specified as a string or cell array of N strings. Each string contains a path to a clock or reset port in the HDL module.

Example: /inverter/clk

Data Types: char | cell | string

ClockResetTypes – Clock and reset waveform types to generate

' ' (default) | string | character vector | cell array of strings | cell array of character vectors

Clock and reset waveform types to generate, specified as a string or cell array of strings. Each string contains a clock or reset type, corresponding to the list specified in the `ClockResetSignals` property. The following values are valid clock and reset types:

- 'Active Rising Edge Clock'
- 'Active Falling Edge Clock'
- 'Step 0 to 1'
- 'Step 1 to 0'

Example: Active Rising Edge Clock

Data Types: char | cell | string

ClockResetTimes — HDL times for clock period or step function duration

{ } (default) | cell array of positive integer and time unit | cell array of cell arrays

HDL times for clock period or step function duration, specified as a cell array of a positive integer and a time unit. Valid values for time units are:

- 'fs' — Femtoseconds
- 'ps' — Picoseconds
- 'ns' — Nanoseconds
- 'us' — Microseconds
- 'ms' — Milliseconds
- 's' — Seconds

To specify multiple clocks or step functions, use a cell array of cell arrays corresponding to the list specified in the `ClockResetSignals` property.

Example: {10, 'ps'} specifies a single clock or step function with a 10 picosecond duration.

Example: {{10, 'ns'}, {8, 'ps'}} specifies two clocks, one with a 10 nanosecond duration and one with an 8 picosecond duration.

Data Types: cell

PreRunTime — Delay in HDL simulator before cosimulation

{0, 'ns'} (default) | cell array

Delay in HDL simulator before the cosimulation starts, specified as a cell array with two elements.

- The first element is the HDL presimulation delay, specified as a nonnegative integer.
- The second element is the time unit, specified as one of these character vectors: 'fs', 'ps', 'ns', 'us', 'ms', or 's'.

Example: {10, 'fs'}

Data Types: cell

SampleTime — Elapsed simulator time between calls to the System object

{10, 'ns'} (default) | cell array

Elapsed time in the HDL simulator between each call to the System object, specified as a cell array with two elements.

- The first element is the time between two calls to the System object, specified as a positive integer.
- The second element is the time unit, specified as a character vector: 'fs', 'ps', 'ns', 'us', 'ms', 's'.

Example: {10, 'ns'}

Data Types: cell

XSiData — Data structure that matches the cosimulation interface to vivadosimlib.slx library struct

This property is read-only.

Data structure matching the cosimulation interface to the vivadosimlib.slx library, specified as an XsiData struct. Create this struct by invoking the **Cosimulation Wizard** and customize your design for Vivado cosimulation. XSiData includes the following fields:

- **ProductName** — 'EDA Simulator Link VS'
- **DesignLib** — Path to the dynamic link library (DLL) file.
- **Language** — HDL language, where 0 indicates Verilog and 1 indicates VHDL
- **TimePrecision** — HDL time precision, in seconds, specified as the exponent. For example, a time precision of one picosecond is equivalent to 10^{-12} seconds, is specified as -12
- **HdlSigInfo** — A struct that contains the dimensions and type of all inputs and outputs
- **ResetInfo** — A struct that contains the name, initial value, and duration of the reset signal

Note The information in this struct is read-only. To change any of the fields in this struct, rerun the **Cosimulation Wizard** tool.

Example: xsiData = struct with fields: ProductName: 'EDA Simulator Link VS' DesignLib: 'xsim.dir/design/xsimk' Language: 1 TimePrecision: -12 HdlSigInfo: [1x2 struct] ResetInfo: [0x0 struct]

Data Types: struct

Usage

Syntax

```
hdloutputs = hdlc(hdlinputs)
```

Description

hdloutputs = hdlc(hdlinputs) connects to the HDL simulator, writes hdlinputs to the HDL simulator, and reads hdloutputs from the HDL simulator. The elapsed simulation time between each call to the System object is defined by the SampleTime property.

Input Arguments

hdlinputs — Inputs to HDL simulator

comma-separated list of values for HDL input ports

Inputs to the HDL simulator, specified as a comma-separated list of values that are driven to your HDL input ports. The HDL input ports are set by the `InputSignals` property. The number of elements in this comma-separated pair must equal the number of HDL input ports. Each input argument value is driven to its corresponding HDL input port.

For example, if `InputSignals` is set as `{'in1','in2'}`, specify `out = hdlc(input1,input2)` to drive the value `input1` to `in1` and `input2` to `in2`.

Example: `[RealFft, ImagFft] = hdlc(3,12)`; the values 3 and 12 are driven as inputs to the HDL simulator, which has two input ports.

Output Arguments

hdloutputs — Outputs from the HDL simulator

scalar | vector

Outputs from the HDL simulator, returned as a scalar or vector. Each returned element is the output from its corresponding HDL output port. The HDL output ports are specified in the `OutputSignals` property. The number of elements returned is the same as the number of HDL output ports specified. For example, if `OutputSignals` is set as `{'out1','out2'}`, specify `[o1, o2] = hdlc(i1,i2)` to assign the value from `out1` to `o1` and `out2` to `o2`.

Example: `out1 = hdlc(3,12)`; assigns the output value from an HDL simulator with one output port.

Example: `[RealFft, ImagFft] = hdlc(3,12)`; assigns output values from an HDL simulator with two output ports. In this example, `RealFft` is the output from the first port and `ImagFft` is the output from the second port.

Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

<code>step</code>	Run <code>System</code> object algorithm
<code>release</code>	Release resources and allow changes to <code>System</code> object property values and input characteristics
<code>reset</code>	Reset internal states of <code>System</code> object

Examples

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

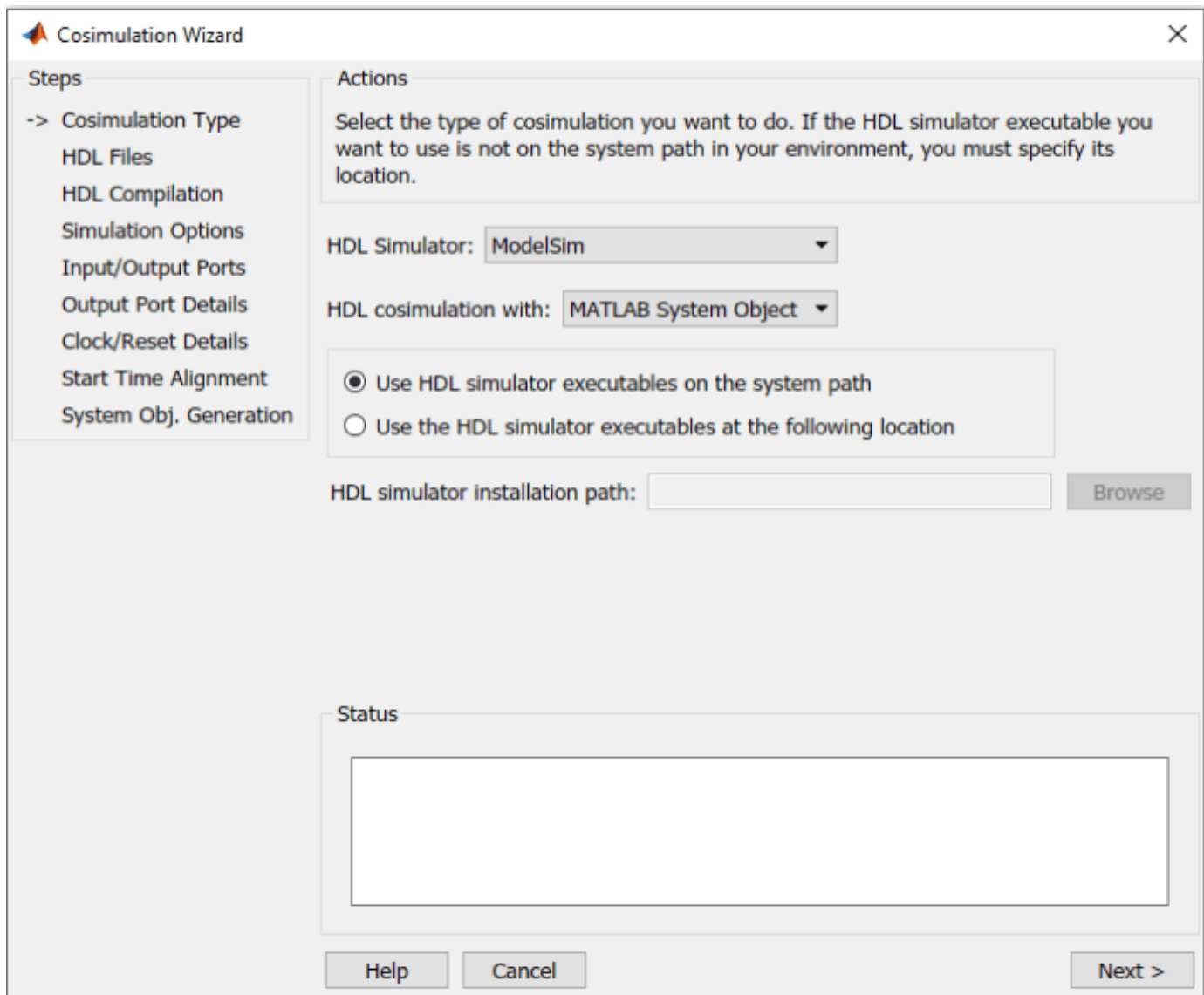
Launch the Cosimulation Wizard tool by executing this command in MATLAB.

```
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**.

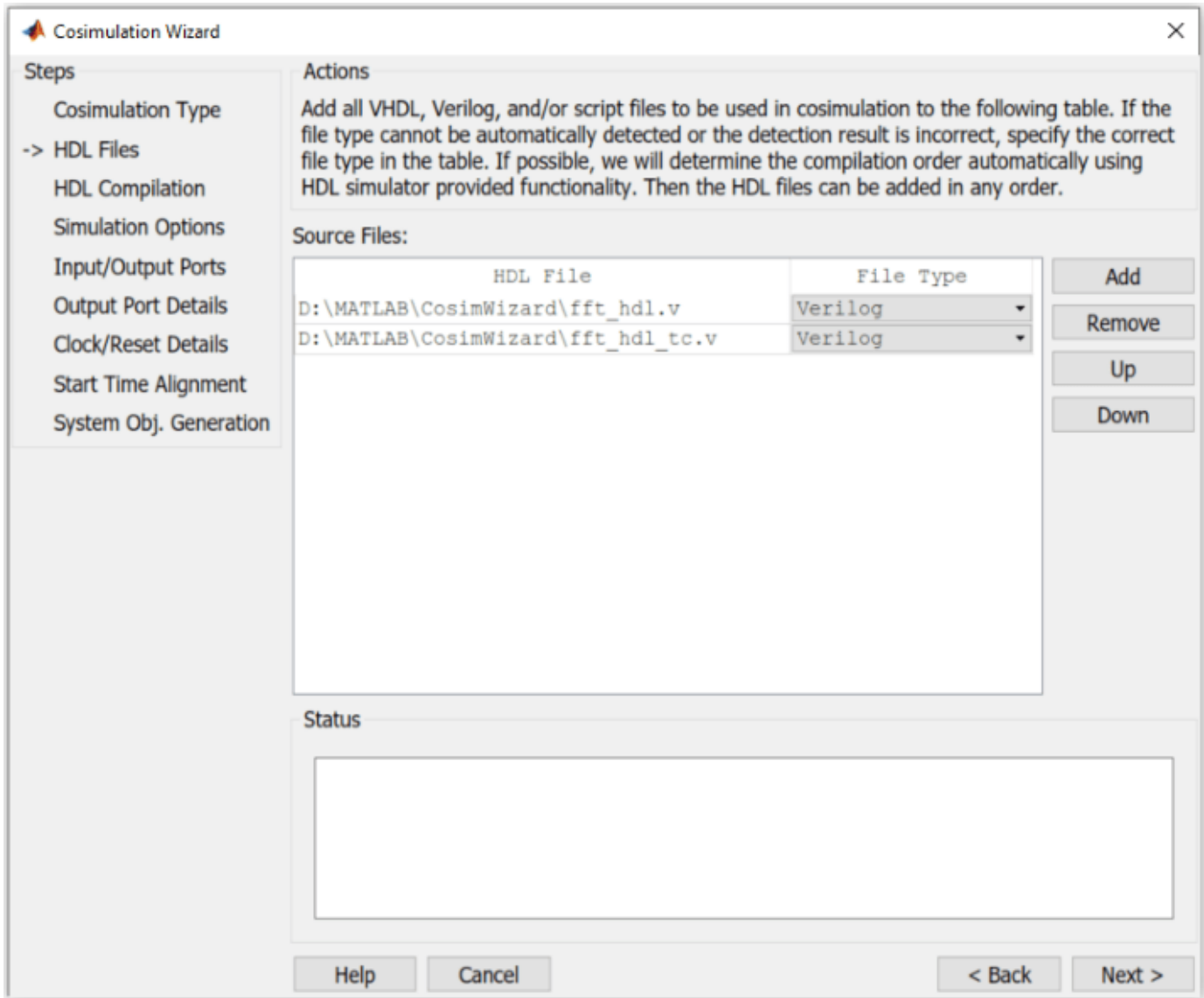
3. Select HDL Files

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.

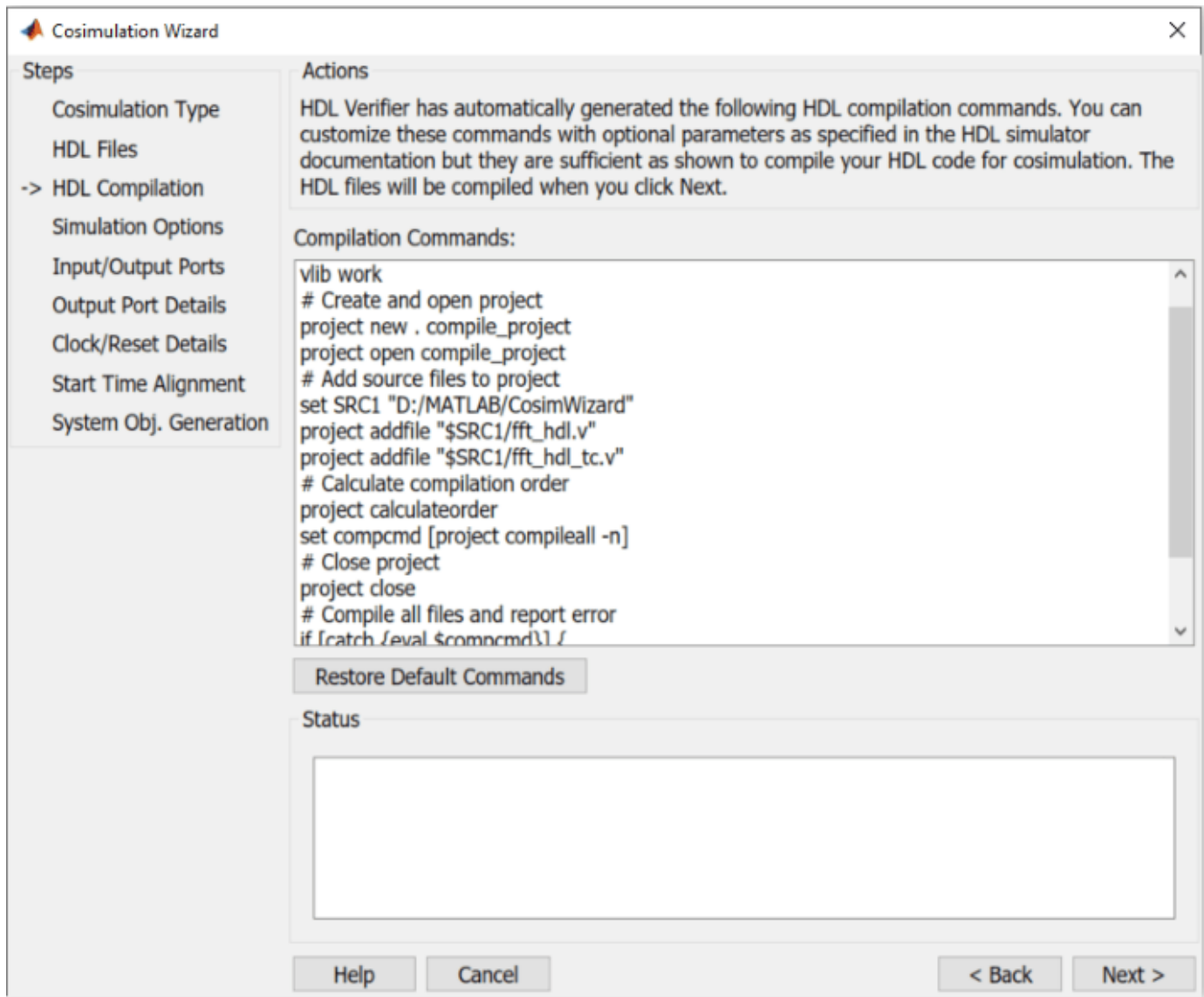
b. Click **Next**.



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

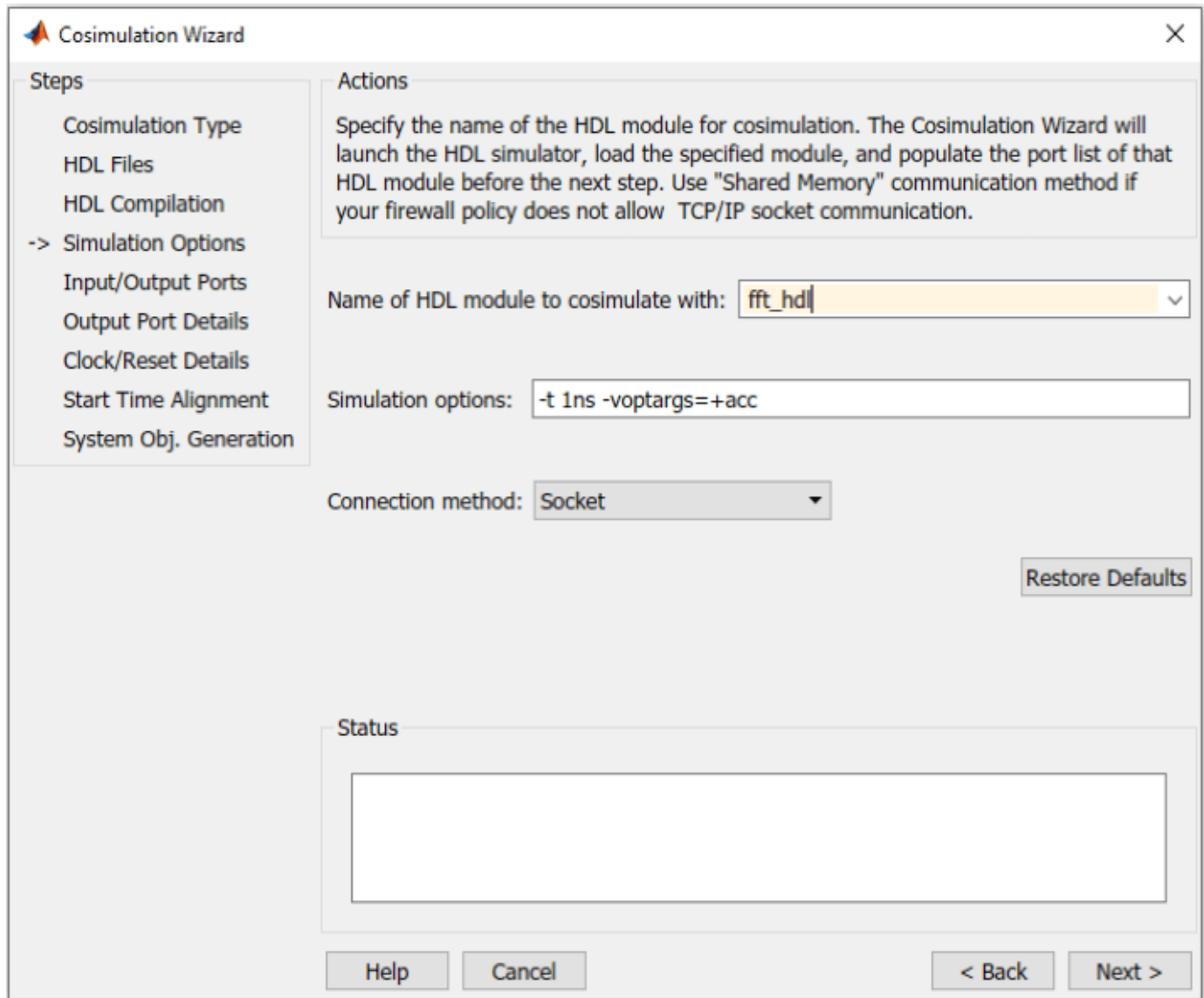
On the Simulation Options page, perform the following steps:

- a. Specify the name of the HDL module or entity for cosimulation.

For ModelSim or Xcelium

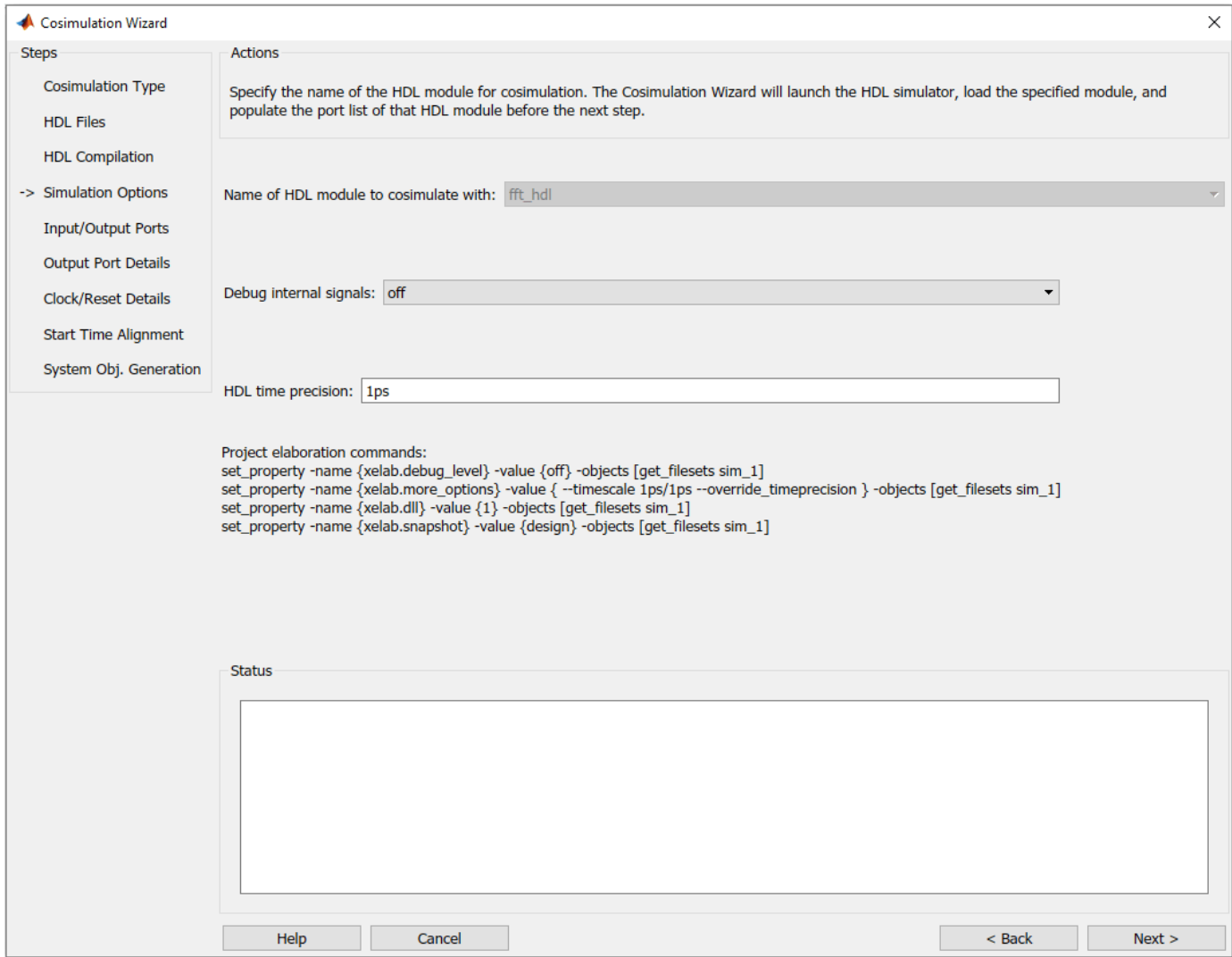
From the list, select `fft_hdl`. This module is the Verilog module you use for cosimulation. If you do not see `fft_hdl` in the list, enter the file name manually.

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

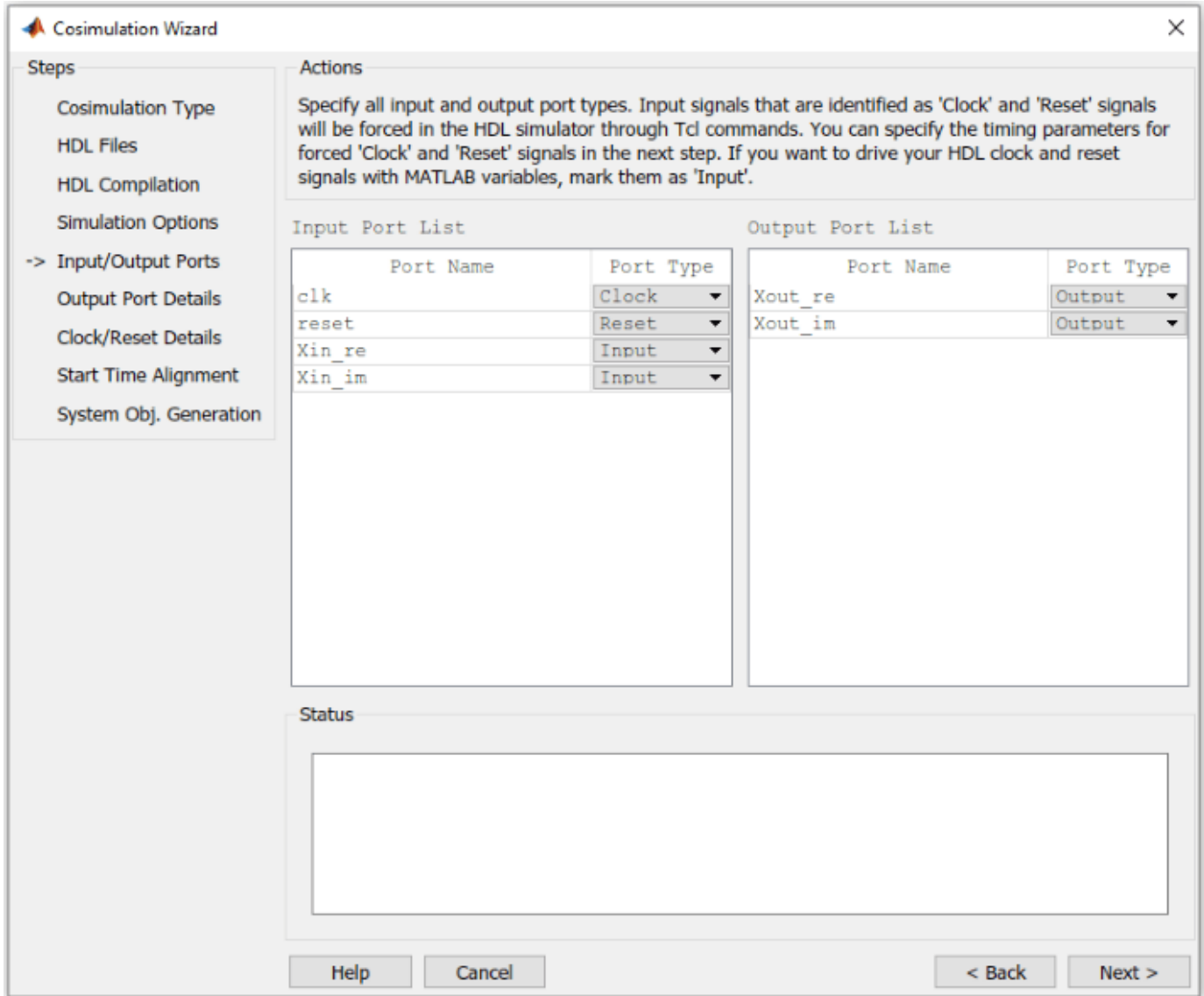
In this step, the Cosimulation Wizard displays two tables containing the input and output ports of **fft_hdl**, respectively.

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**.

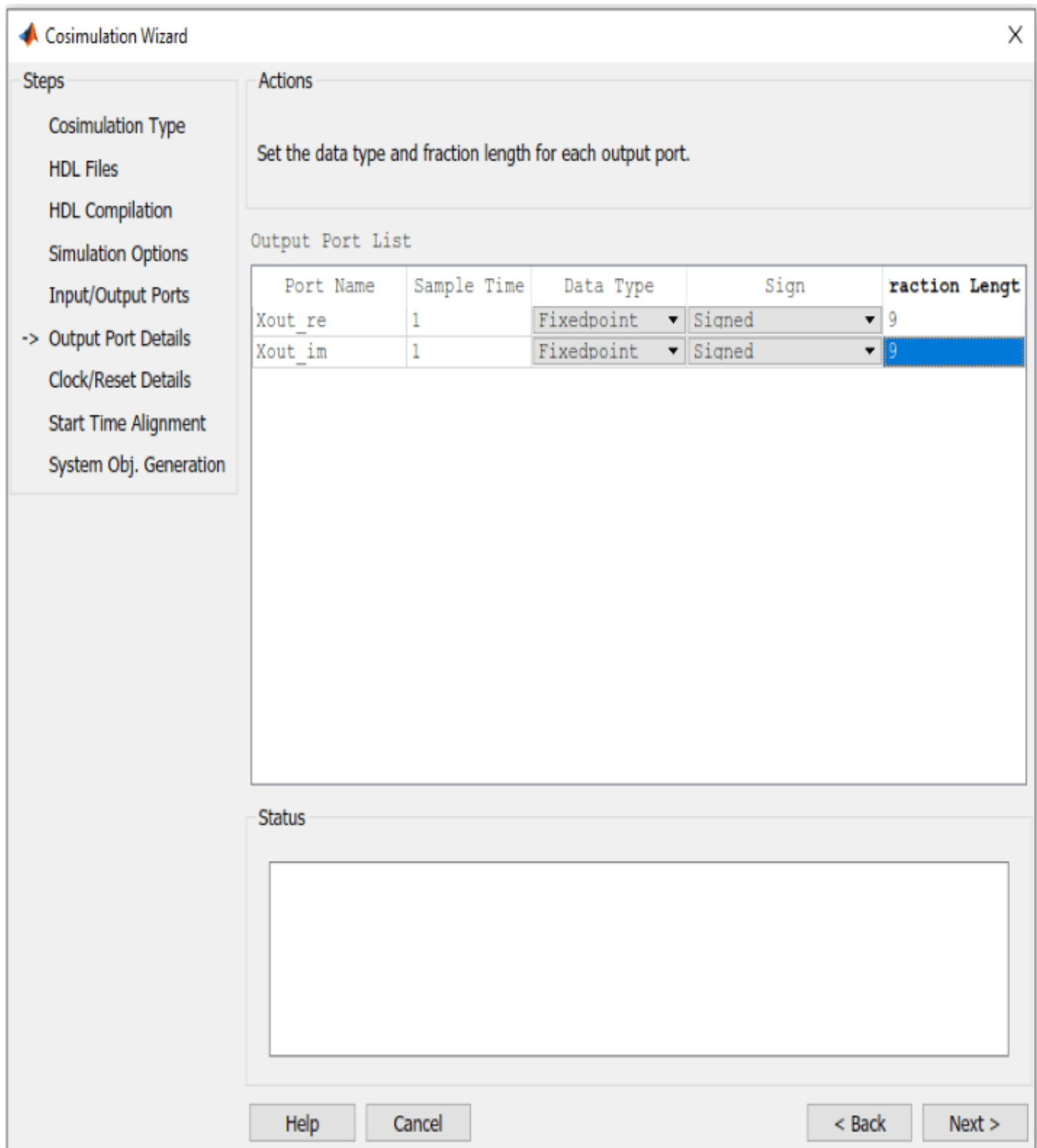


7. Specify Output Port Details

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

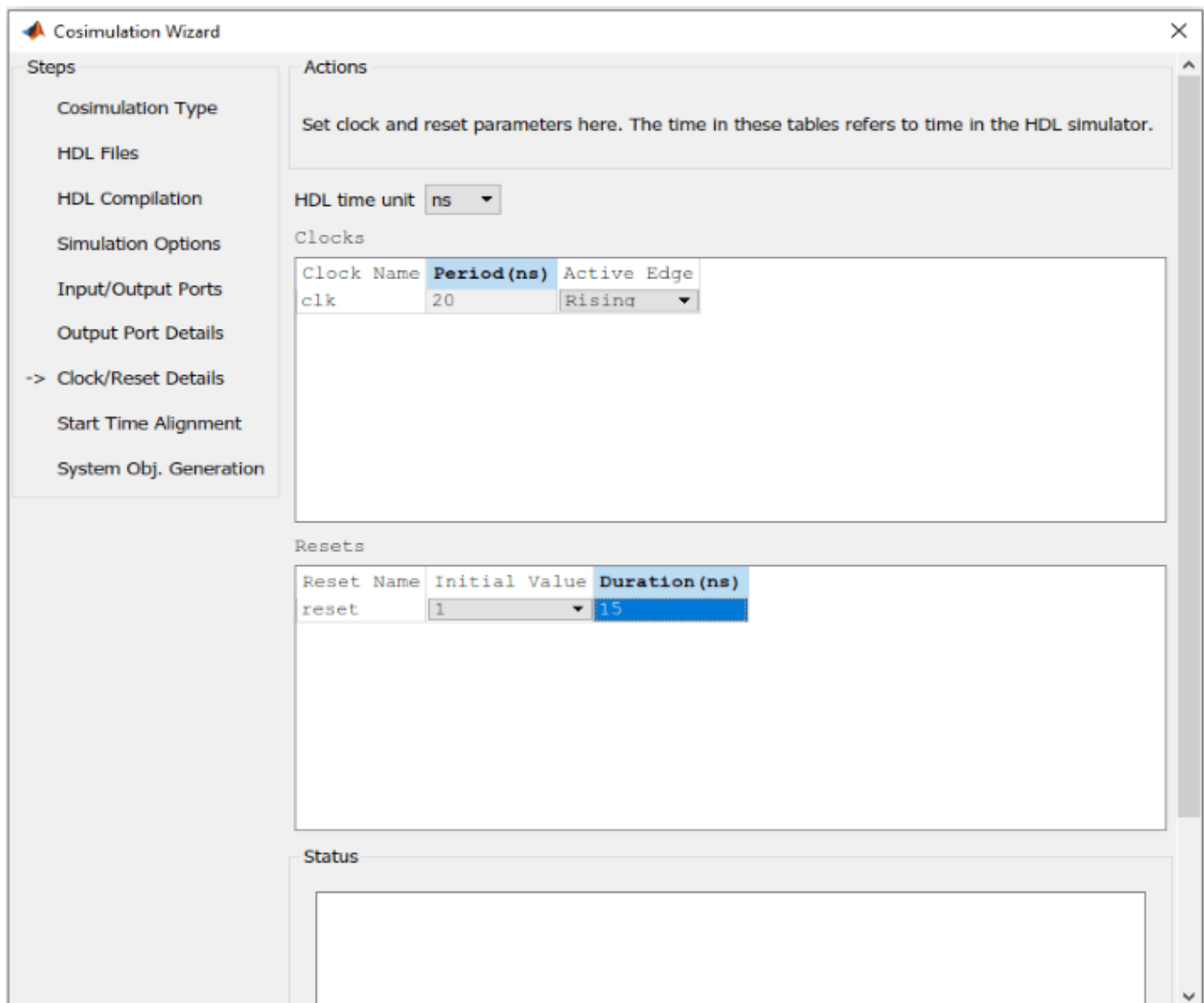
Help Cancel < Back Next >

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

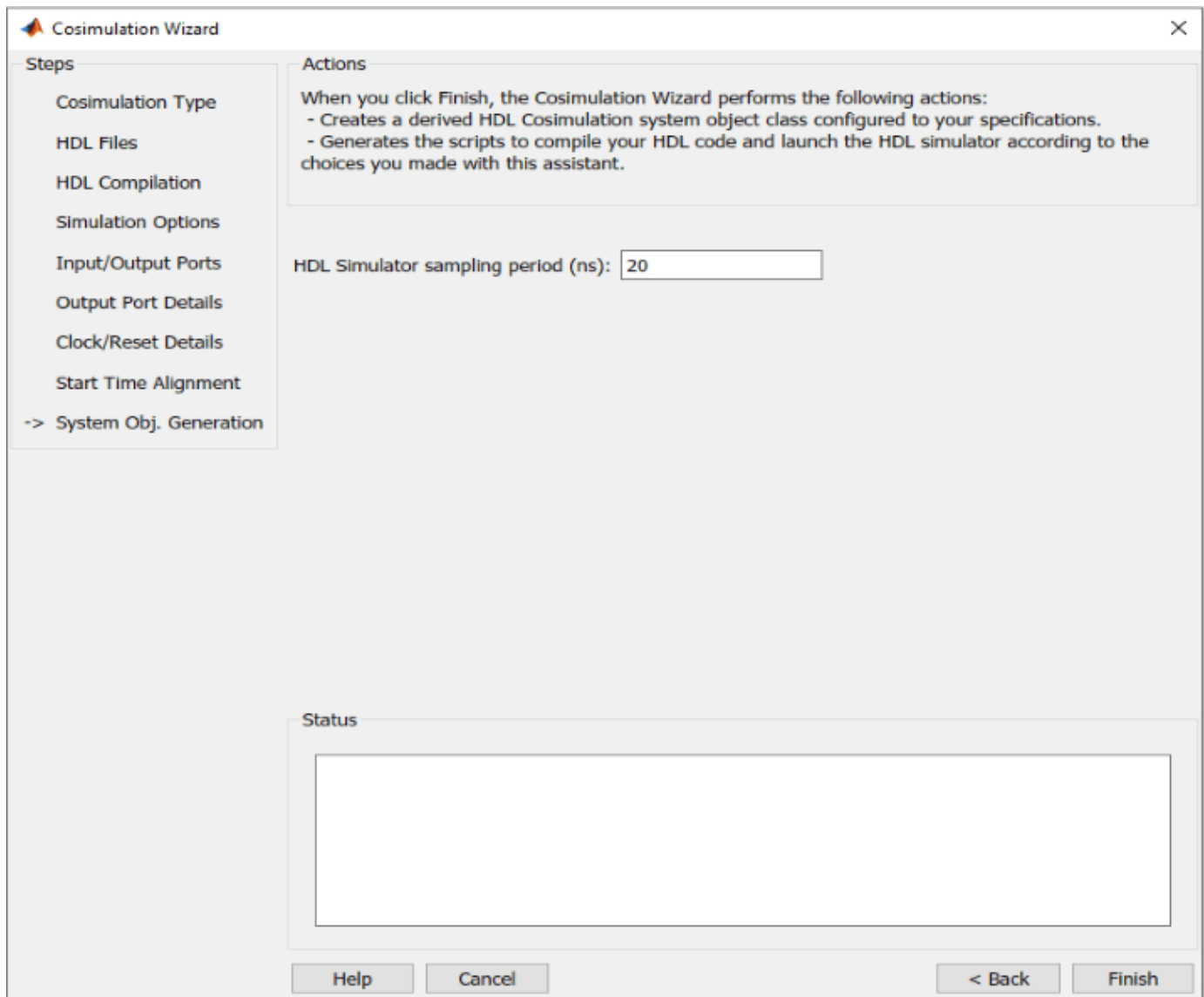
Status

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

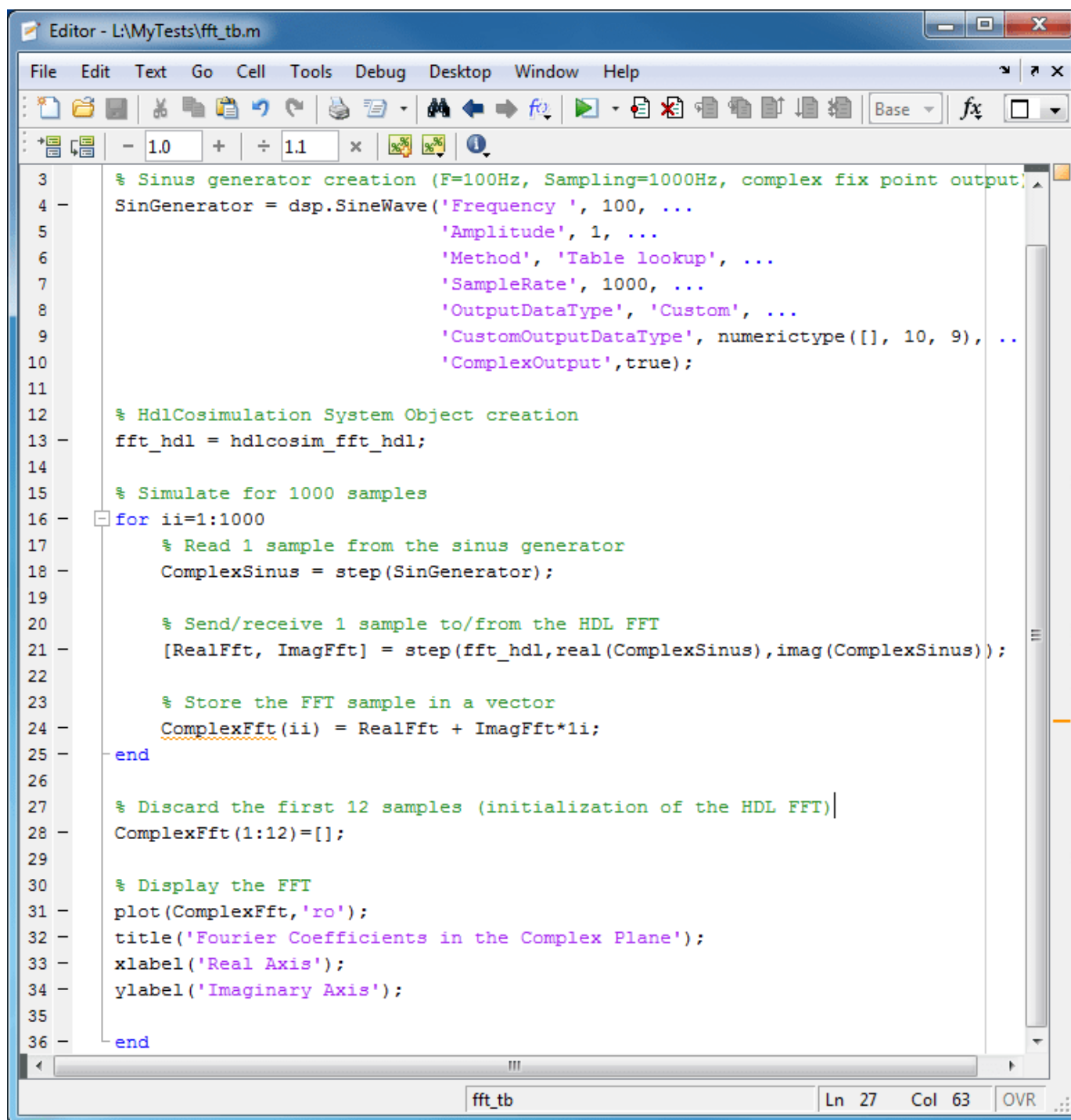
- **compile_hdl_design_fft_hdl.m**: To recompile the HDL design.

- **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**.

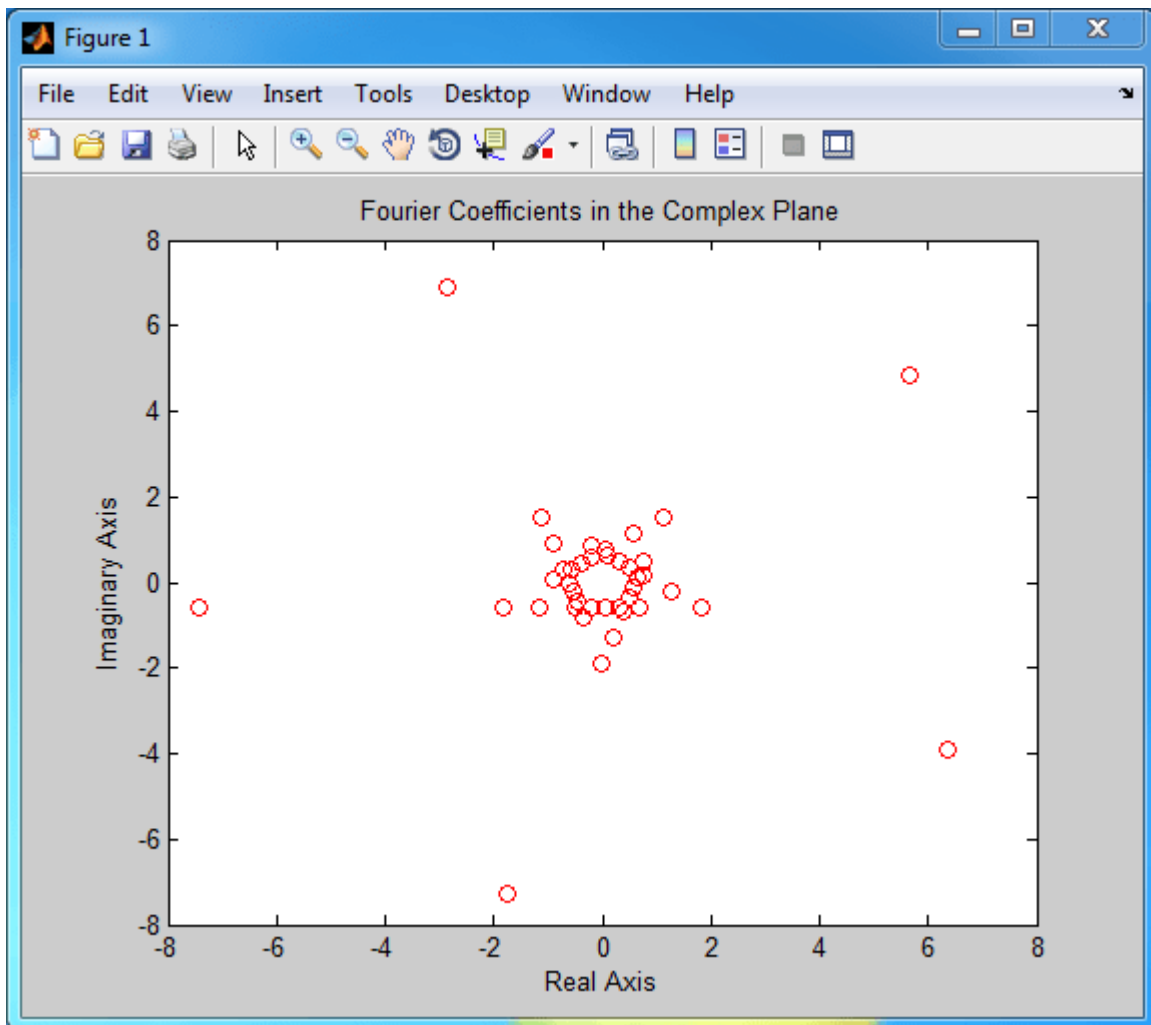
```
fft_tb
```

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

Version History

Introduced in R2022a

See Also

Functions

`hdlverifier.HDLCosimulation`

Tools

Cosimulation Wizard

Topics

“Create a MATLAB System Object”

“Cosimulation Wizard for MATLAB System Object”

Objects

cosimulationConfiguration

Configure HDL cosimulation workflow

Description

The `cosimulationConfiguration` object controls the creation of an HDL Cosimulation block or System object for a specified HDL module.

Creation

Syntax

```
cosimObj = cosimulationConfiguration(HDLSimulator,SubWorkflow,  
HDLTopLevelName)  
cosimObj = cosimulationConfiguration(MATfile)
```

Description

`cosimObj = cosimulationConfiguration(HDLSimulator,SubWorkflow,HDLTopLevelName)` creates a `cosimulationConfiguration` object with the specified `HDLSimulator`, `SubWorkflow`, and `HDLTopLevelName` values.

This workflow is a command-line alternative to using the **Cosimulation Wizard** for creating an HDL Cosimulation block or an `hdlverifier.HDLCosimulation` System object.

`cosimObj = cosimulationConfiguration(MATfile)` creates a `cosimulationConfiguration` object based on the results of a previous run of the **Cosimulation Wizard** or command-line workflows.

Properties

Workflow

HDLSimulator — HDL simulator to use

'ModelSim' | 'Xcelium' | 'Vivado Simulator'

HDL simulator to use for cosimulation, specified as 'ModelSim', 'Xcelium', or 'Vivado Simulator'.

SubWorkflow — Cosimulation kind

'Simulink' | 'MATLAB System Object'

Kind of cosimulation to perform:

- 'Simulink' — Configures the workflow to create an HDL Cosimulation block for a Simulink cosimulation.
- 'MATLAB System Object' — Configures the workflow to create an `hdlverifier.HDLCosimulation` System object for a MATLAB cosimulation.

HDLTopLevelName — Name of top level HDL module or entity

string

Name of the top level HDL module or entity, specified as a string or character vector.

Data Types: char | string

HDL Compilation**HDLFiles — HDL files for cosimulation**

string | character vector | cell array

HDL directory, file, or files required for cosimulation, specified as a string or cell array. If one of the files does not have a known HDL file extension, the tool ignores it.

Example: `'./hdlsrc'` includes all files from the `hdlsrc` directory

Example: `{ './hdlsrc', 'myModule.v' }` includes all files from the `hdlsrc` directory and a local file named `myModule.v`.

Another option, is to specify a list where each file is followed by a file type. Available file types depend on the simulator:

- ModelSim: Verilog (*.v, *.sv), VHDL (*.vhd), ModelSim macro file (*.do)
- Xcelium: Verilog (*.v, *.sv), VHDL (*.vhd), Shell script (*.sh)
- Vivado: Verilog (*.v, *.sv), VHDL (*.vhd), Vivado XCI file (*.xci)

Example: `{ 'mytop.v', 'Verilog', 'submod.v', 'Verilog' }`

Data Types: char | string | cell array

HDL Simulator Path — HDL simulator location

string | character vector

Location of the HDL simulator, specified as a string or character vector. Use this option when you want to use a simulator in a different location than the one specified on the system PATH. When empty, the system path is used.

Data Types: char | string

HDLCompilationCommand — Compilation commands to use

string | character vector

Compilation commands to use, specified as a string or character vector. The compilation commands are automatically generated based on the selected simulator and design file list. You can supply different compilation commands if required, for example when you have to compile into a specific library.

Tip It is useful to run the workflow once to generate a `cosimWizard_<TOP>.mat` file. Then you can modify the compilation commands and set this property.

Data Types: char | string

HDL Elaboration Options — Options for the HDL elaboration tool

string | character vector

Options for the HDL elaboration tool, specified as a string or character vector.

This property is valid for cosimulation with Xcelium.

Data Types: `char` | `string`

HDLSimulationOptions — Options for the HDL simulation tool

`string` | character vector

Options for the HDL simulation tool, specified as a string or character vector.

The default value for Xcelium forces use of the 64-bit app.

The default value for ModelSim includes the necessary flags to define the HDL time precision and to give read/write access of the port for cosimulation.

This property is only valid for cosimulation with Xcelium or ModelSim.

Data Types: `char` | `string`

HDL Timing

HDLTimeUnit — Time unit for HDL simulation

`fs` | `ps` | `ns` | `us` | `ms` | `s`

Time unit for HDL simulation. This unit affects the following configurations:

- `PreCosimulationRunTime` property
- `SampleTime` property (MATLAB System object sub-workflow)
- Period attribute of clock ports
- Duration attribute of reset ports

The default unit is determined dynamically by a query of the time precision to the HDL simulator during compilation (ModelSim, Xcelium) or by the `HDLTimePrecision` property (Vivado).

Note The `SampleTime` attribute of output ports is in seconds because it specifies a Simulink time.

PreCosimulationRunTime — Time to run HDL simulator before cosimulation starts

`integer`

The amount of time to run the HDL simulator before beginning simulation in Simulink or MATLAB, in `HDLTimeUnit` units. This time allows getting through any necessary resets and asserting clock enables.

A visual timing diagram of clocks, resets, and the cosimulation start time is created in file `hdlverifier_rstclk_waveform.jpg` after you run the workflow.

AutoTimeScale — Automatically determine timescale

`true` (default) | `false`

Automatically determine the ratio between Simulink time to HDL time at the first cosimulation.

The Simulink sample times are related to the HDL simulation time using a timing scale. Often the best choice is for the fundamental Simulink sample time to equal the fastest clock period in the HDL.

Because all of the input sample times are not known at the time that this workflow executes, you can choose to have the timescale determined automatically on the first simulation when all of the sample times are known.

To enable this property, set `Subworkflow` to `Simulink`.

TimeScale — Explicitly set timescale

{1, 's'} (default) | scale factor and time unit

Manually set the timescale, specified as a scale factor and a time unit: {factor, unit}.

The scale is relative to 1 second in Simulink: 1 second in Simulink is equal to {factor, unit} in HDL. The factor is a double and the unit is one of 'fs', 'ps', 'ns', 'us', 'ms', or 's'. The default value is {1, 's'}, which means that Simulink times are equivalent to HDL

times.

Example: With a Simulink sample time of 1 second and an HDL clock of 100 MHz, create a timescale where 1 second in Simulink is equal to 10 ns in HDL: `c.AutoTimeScale = false; c.TimeScale = {10, 'ns'};`

Dependencies

To enable this property, set `Subworkflow` to `Simulink`.

SampleTime — HDL time for each System object evaluation

nonnegative numeric

Set the MATLAB sample time for the generated System object in HDLTimeUnit units. All inputs and outputs share the same sample time in MATLAB and each call to the `step` method elapses this amount of time.

The default value is determined by the specified clock periods and output sample times.

Dependencies

To enable this property, set `Subworkflow` to `MATLAB System Object`.

HDL Simulator Connection

Connection — Type of connectivity to the HDL simulator

Socket (default) | Shared Memory

Channel type for connecting to MATLAB or Simulink to the HDL simulator, specified as `Socket` or `Shared Memory`.

- **Socket:** MATLAB or Simulink and the HDL simulator communicate through a designated TCP/IP socket. You can use it for single-system and network configurations. This option offers the greatest scalability. For more information about TCP/IP socket communication, see “TCP/IP Socket Ports”.
- **Shared memory:** MATLAB or Simulink and the HDL simulator communicate through shared memory. Shared memory communication provides optimal performance.

Port Interface Properties

ClockPortRegularExpression — Expression for auto-assigning ports to clocks

'clk\$|clock\$' (default) | regular expression

Expression for auto-assigning ports to clocks, specified as a regular expression. For more information about regular expressions in MATLAB, see “Regular Expressions”.

Example: `'clk_in$'` assigns all the ports that end with "clk_in" as clock ports.

ResetPortRegularExpression — Expression for auto-assigning ports to resets

`'rst$|reset$|rst_n$|reset_n$|reset_x$'` (default) | regular expression

Expression for auto-assigning ports to resets, specified as a regular expression. For more information about regular expressions in MATLAB, see “Regular Expressions”.

Example: `c.ResetPortRegularExpression = [c.ResetPortRegularExpress 'rst[0-9]$']` assigns all the ports that end with "rst" and a numerical value as reset ports, in addition to the default expression for reset ports.

UnusedPortRegularExpression — Expression for auto-assigning ports as unused

`''` (default) | regular expression

Expression for labelling ports as unused, specified as a regular expression. Ports assigned as unused are not part of the cosimulation. Unused ports do not show up in the cosimulation interface and are not driven or sampled.

To clear the specifications, provide an empty argument after the signal names.

For more information about regular expressions in MATLAB, see “Regular Expressions”.

Example: `'clk_out$'` indicates that any ports that end with "clk_out" are unused.

InputDataPorts — Display input ports

table

This property is read-only.

Input ports, displayed as a table. See the full table contents by calling the `portInterface` object function.

The default is an empty table and the workflow determines input data ports. Inputs that do not match the clock, reset, and unused regular expressions are considered input data ports.

You can explicitly assign input ports by using the `specifyInput` object function.

OutputDataPorts — Display output ports

table

This property is read-only.

Output ports, displayed as a table. Each row in the table contains the following information per output:

- Name of output port
- SampleTime
- DataType
- Signed
- FractionLength

In addition to the DUT outputs, the table includes a row named `default_output_definition`, which contains the default values assigned for the outputs.

You can change the default output values or manually assign output ports by using the `specifyOutput` object function.

See the full table contents by calling the `portInterface` object function.

ClockPorts — Display clock ports

table

This property is read-only.

Clock ports, displayed as a table. Each row in the table contains the following information per clock:

- Name of clock port
- Edge
- Period

In addition to the DUT clocks, the table includes an extra row named `default_clock_definition` for the default clock definitions.

You can change the default clock values or manually assign clocks by using the `specifyClock` object function.

See the full table contents by calling the `portInterface` object function.

ResetPorts — Display reset ports

table

This property is read-only.

Reset ports, displayed as a table. Each row in the table contains the following information per reset:

- Name — Name of reset port
- `InitialValue` — Initial value of the reset
- `Duration` — Duration of the reset signal

In addition to the DUT resets, the table includes an extra row named `default_reset_definition` for the default reset definitions.

You can change the default reset values or manually assign resets by using the `specifyReset` object function.

See the full table contents by calling the `portInterface` object function.

UnusedPorts — Display unused ports

table

This property is read-only.

Unused ports, displayed as a table. See the full table contents by calling the `portInterface` object function.

The default is an empty table and the workflow determines the unused ports. Ports that match the unused regular expression are considered unused.

You can explicitly label ports as unused by using the `specifyUnused` object function.

Object Functions

<code>portInterface</code>	Display port specifications
<code>runWorkflow</code>	Execute cosimulation workflow and generate required artifacts
<code>specifyClock</code>	Assign clock ports to cosimulation block or System object
<code>specifyInput</code>	Assign HDL input ports to cosimulation block or System object
<code>specifyOutput</code>	Assign HDL output ports to cosimulation block or System object
<code>specifyReset</code>	Assign reset ports to cosimulation block or System object
<code>specifyUnused</code>	Label HDL ports as unused ports

Examples

Command-Line Workflow for Verifying Raised Cosine Filter in Simulink

This example shows how to cosimulate a raised cosine filter in Simulink® using the command-line interface. It follows the same workflow to generate cosimulation artifacts as in “Get Started with Simulink HDL Cosimulation”, but it uses the command line instead of the Cosimulation Wizard.

Configure Cosimulation Workflow

Create a cosimulation configuration object.

```
c = cosimulationConfiguration('ModelSim','Simulink','rcosflt_rtl');
```

Set up the HDL file.

```
c.HDLFiles = {'./rcosflt_rtl.v','Verilog'};
```

Set the `filter_out` port as an output with a signed fixed-point data-type, and set the fraction length to 29.

```
specifyOutput(c,'filter_out',Datatype='Fixedpoint',Signed=true,FractionLength=29)
```

Set the clock to a period of 20 ns, and set the reset duration to 15 ns.

```
specifyClock(c,'clk',Period=20)
specifyReset(c,'reset',Duration=15)
```

Display the port table. It reflects the settings just made for output, clock, and reset attributes. The other design ports will take on default attributes.

```
portInterface(c);
```

```
----- Input Data Ports -----
      0x1 empty table
```

```
----- Output Data Ports -----
      2x5 table
```

Name	SampleTime	DataType	Signed	FractionLength
{'default_output_definition'}	1	{'Inherit' }	true	0
{'filter_out' }	1	{'Fixedpoint'}	true	29

----- Clock Ports -----
2x3 table

Name	Edge	Period
{'default_clock_definition'}	{'Rising'}	10
{'clk' }	{'Rising'}	20

----- Reset Ports -----
2x3 table

Name	InitialValue	Duration
{'default_reset_definition'}	1	8
{'reset' }	1	15

----- Unused Ports -----
0x1 empty table

Generate HDL Cosimulation Block

Run the workflow to generate an HDL Cosimulation block and the accompanying files.

```
runWorkflow(c);
```

```
----- Step 1-----
Select the type of cosimulation you want to do. If the HDL simulator executable you want to use :
----- Step 2-----
Add all VHDL, Verilog, and/or script files to be used in cosimulation to the following table. If
----- Step 3-----
HDL Verifier has automatically generated the following HDL compilation commands. You can customize
Compiling HDL files. Please wait ...
### Compiling HDL design
Reading pref.tcl

# 2021.4

# Create design library
vlib work
# Create and open project
project new . compile_project
# Loading project compile_project
project open compile_project
# Add source files to project
```

```

set SRC1 "."
# .
project addfile "$SRC1/rcosflt_rtl.v"
# Calculate compilation order
project calculateorder
# QuestaSim-64 vlog 2021.4 Compiler 2021.10 Oct 13 2021
# Start time: 10:47:39 on Jul 21,2022
# vlog -work work -vopt C:/Users/pmishra/OneDrive - MathWorks/Documents/MATLAB/ExampleManager/pm
# -- Compiling module rcosflt_rtl
#
# Top level modules:
#   rcosflt_rtl
# End time: 10:47:42 on Jul 21,2022, Elapsed time: 0:00:03
# Errors: 0, Warnings: 0
# Compile of rcosflt_rtl.v was successful.
# All compile dependencies have been resolved.
set compcmd [project compileall -n]
# vlog -work work -vopt -stats=none {C:/Users/pmishra/OneDrive - MathWorks/Documents/MATLAB/Exampl
# Close project
project close
# reading modelsim.ini
# Compile all files and report error
if [catch {eval $compcmd}] {
    exit -code 1
}
# QuestaSim-64 vlog 2021.4 Compiler 2021.10 Oct 13 2021
# -- Compiling module rcosflt_rtl
#
# Top level modules:
#   rcosflt_rtl
#
# <EOF>
...done
----- Step 4-----
Specify the name of the HDL module for cosimulation. The Cosimulation Wizard will launch the HDL
Elaborating and Loading HDL simulation image. Please wait ...
Waiting for HDL Simulator to startup ...
120 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
119 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
118 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
117 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
116 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
...done
----- Step 5-----
Specify all input and output port types. Input signals that are identified as 'Clock' and 'Reset'
----- Step 6-----
Set the sample time and data type for each output port. You can specify the sample time as -1, wh
----- Step 7-----
Set clock and reset parameters here. The time in these tables refers to time in the HDL simulator

```

Please wait while generating waveforms.

...done

----- Step 8-----

The diagram below shows the current settings for forced 'Clock' and 'Reset' signals. The red line

To change the MATLAB/Simulink start time relative to the HDL simulation time, enter the new start

----- Step 9-----

When you click Finish, the Cosimulation Wizard performs the following actions:

- Creates and opens a new Simulink model containing an HDL Cosimulation block configured to your
- Generates the scripts to compile your HDL code and launch the HDL simulator according to the c
- (If you check the box below) Configures the HDL Cosimulation block to assist you in setting th

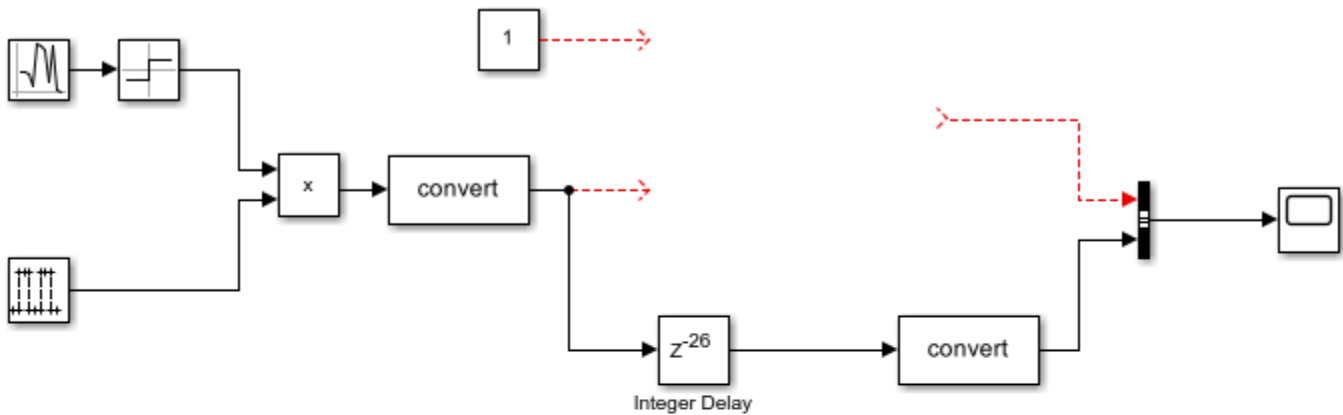
Generating blocks ... Please wait.

...done

The workflow executes the steps and generates a Simulink model named `hdlverifier_wizard_rcosflt_rtl.slx`, which includes an HDL Cosimulation block and two additional blocks for compilation and communication with the HDL simulator.

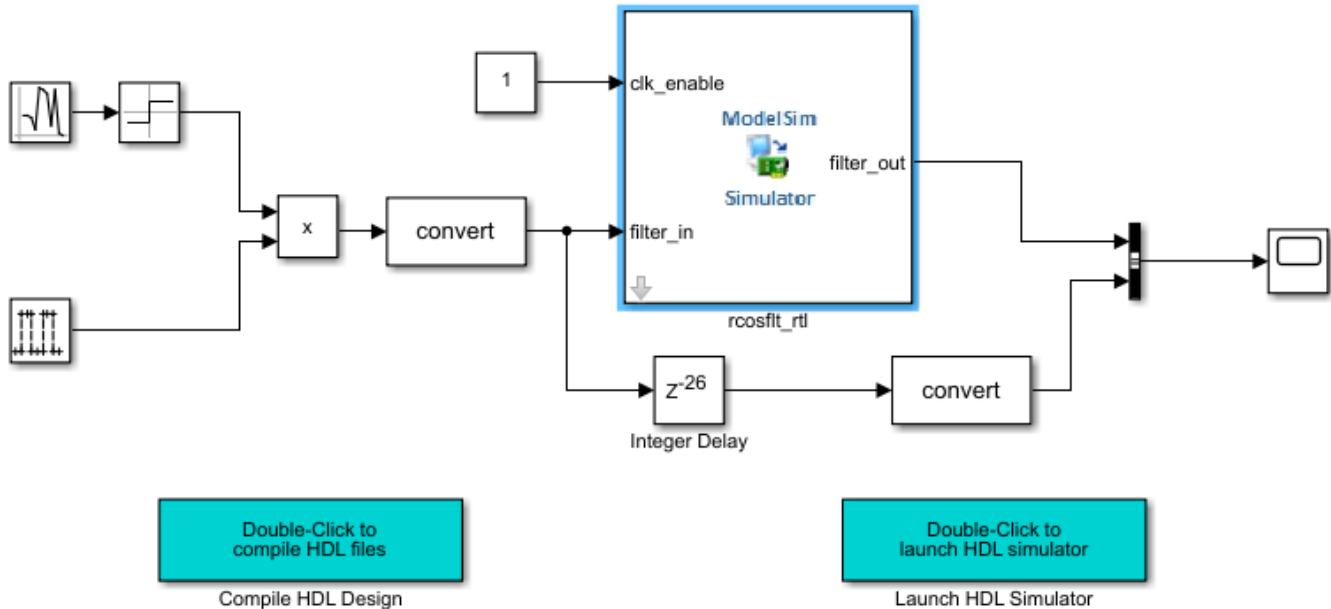
Open the provided testbench model.

```
open_system('rcosflt_tb.slx')
```



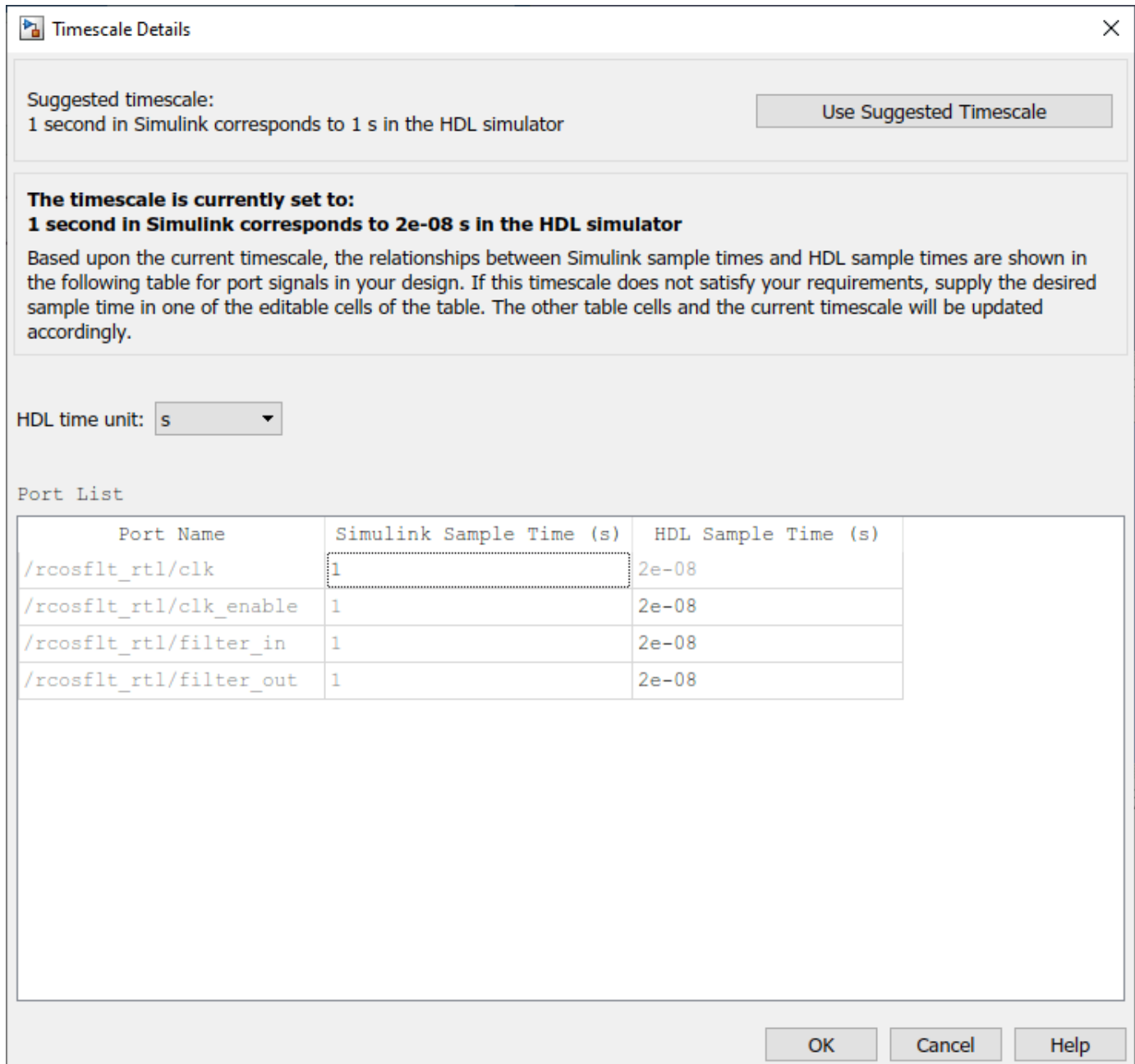
Copyright 2010 The MathWorks, Inc.

Drag the generated HDL Cosimulation block to the canvas, and connect its inputs and outputs to the testbench. Your model now looks similar to the following figure.



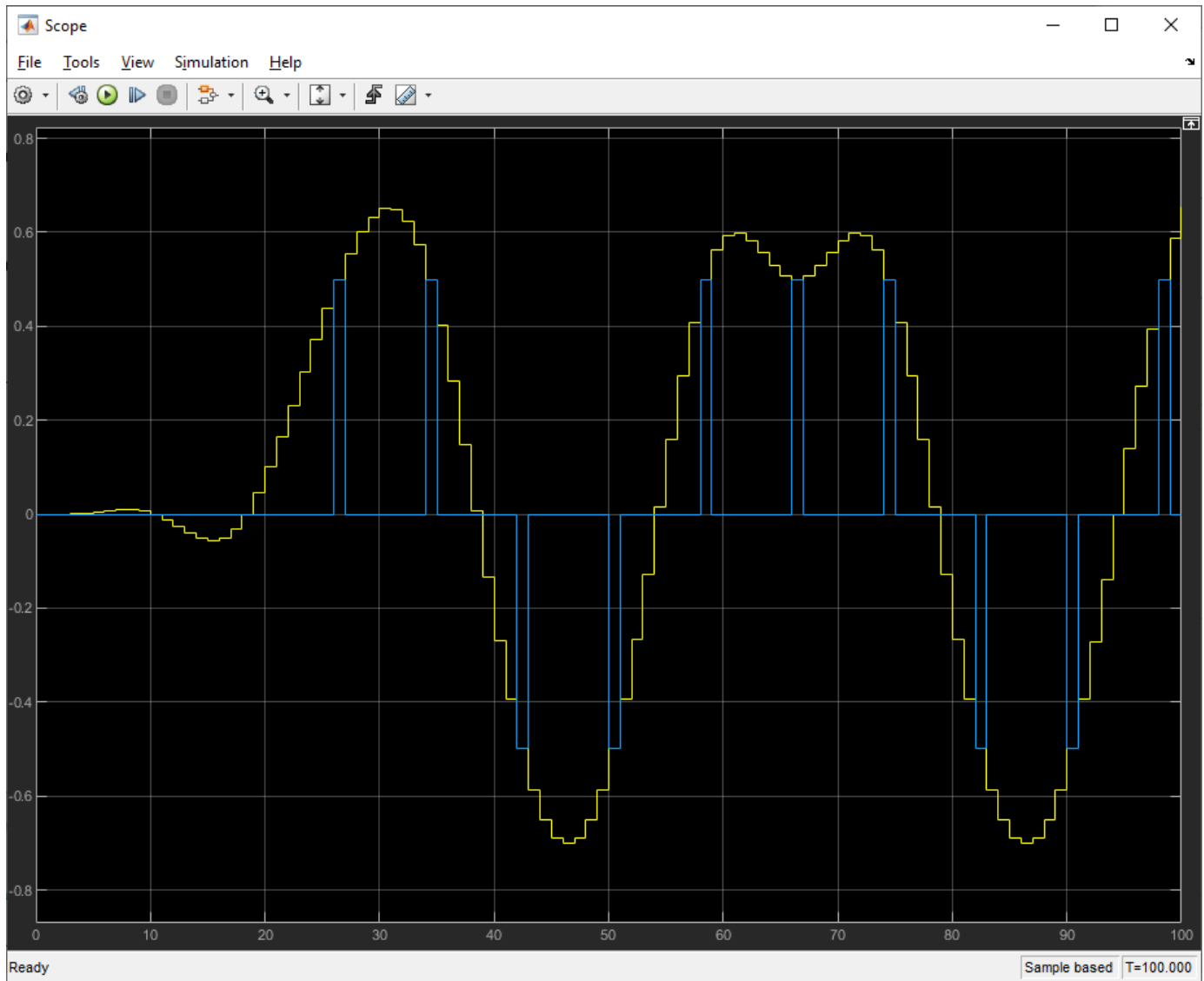
Run Cosimulation and Verify HDL Design

- 1 Start the HDL simulator by double-clicking the block labeled **Launch HDL Simulator**.
- 2 When the HDL simulator is ready, return to Simulink and start the simulation.
- 3 Determine timescale. Since the `AutoTimeScale` property is set to automatically determine the timescale at start of simulation, 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.



4. Click **OK** to close the Timescale Details dialog box. Restart the Simulink simulation and verify the results from the scope in the test bench model.

The scope displays both the delayed version of input to the raised cosine filter and that filter's output. If you sample the output of this filter directly, no inter-symbol-interference occurs.



Version History

Introduced in R2022b

See Also

Cosimulation Wizard

portInterface

Package: hdlverifier

Display port specifications

Syntax

```
portInterface(cosimConfigObj)
```

Description

`portInterface(cosimConfigObj)` displays a table with the port specifications configured by the `cosimConfigObj` object.

Input Arguments

cosimConfigObj — Cosimulation configuration

`cosimulationConfiguration` object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration` | `specifyClock` | `specifyInput` | `specifyOutput` | `specifyReset` | `specifyUnused`

runWorkflow

Package: hdlverifier

Execute cosimulation workflow and generate required artifacts

Syntax

```
runWorkflow(cosimConfigObj)
runWorkflow(cosimConfigObj,RestartFromStep=1)
```

Description

`runWorkflow(cosimConfigObj)` executes all the steps in the workflow to create a cosimulation block or System object and the required scripts as configured in the `cosimulationConfiguration` object.

`runWorkflow(cosimConfigObj,RestartFromStep=1)` executes the workflow steps starting from the specified step.

Examples

Run Workflow Using Default Configuration Values.

Set simulator, workflow, top level model, and Verilog files, and run the workflow to generate an HDL Cosimulation block with automatically determined configuration values.

```
c = cosimulationConfiguration('Xcelium','Simulink','fir_filt')
c.HDLFiles = {'./rcosflt_rtl.v','Verilog'};
runWorkflow(c);
```

Generate Artifacts Based on Previous Cosimulation Wizard Execution

Restore a previous workflow saved in './mygoodrun/cosimWizard_mytop.mat'.

```
c = cosimulationConfiguration('./mygoodrun/cosimWizard_mytop.mat');
runWorkflow(c);
```

Restart Workflow From Step 1

Restore a previous workflow saved in 'previousRun.mat'.

Restore workflow from previous run and execute `runWorkflow`.

```
c = cosimulationConfiguration('previousRun.mat');
runWorkflow(c);
```

```
----- Step 1-----
Select the type of cosimulation you want to do. If the HDL simulator executable
```

you want to use is not on the system path in your environment, you must specify its location.

----- Step 2-----

Add all VHDL, Verilog, and/or script files to be used in cosimulation to the following table. If the file type cannot be automatically detected or the detection result is incorrect, specify the correct file type in the table. If possible, we will determine the compilation order automatically using HDL simulator provided functionality. Then the HDL files can be added in any order. Index in position 2 exceeds array bounds. Index must not exceed 1.

Error in CosimWizardPkg.FileSelection/EnterStep

Error in cosimulationConfiguration/l_Step2 (line 809)
obj.stepH.EnterStep(obj.ddgH)

Error in cosimulationConfiguration/runWorkflow (line 473)
feval(['l_Step' currStepStr], obj); %wizH, stepH, ddgH, inArgs);

After encountering an error, specify HDL files, and restart execution from step 1.

```
c.ResetPortRegularExpression = 'rst_p';
runWorkflow(c, 'RestartFromStep', 1);
```

----- Step 1-----

Select the type of cosimulation you want to do. If the HDL simulator executable you want to use is not on the system path in your environment, you must specify its location.

----- Step 2-----

Add all VHDL, Verilog, and/or script files to be used in cosimulation to the following table. If the file type cannot be automatically detected or the detection result is incorrect, specify the correct file type in the table. If possible, we will determine the compilation order automatically using HDL simulator provided functionality. Then the HDL files can be added in any order.

----- Step 3-----

HDL Verifier has automatically generated the following HDL compilation commands. You can customize these commands with optional parameters as specified in the HDL simulator documentation but they are sufficient as shown to compile your HDL code for cosimulation. The HDL files will be compiled when you click Next.

Compiling HDL files. Please wait ...

Compiling HDL design

Reading pref.tcl

2021.4

Create design library

vlib work

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

Errors: 0, Warnings: 1

Create and open project

project new . compile_project

Loading project compile_project

project open compile_project

Add source files to project

set SRC1 ". "

.

project addfile "\$SRC1/rcosflt_rtl.v"

Calculate compilation order

project calculateorder

QuestaSim-64 vlog 2021.4 Compiler 2021.10 Oct 13 2021

Start time: 12:49:55 on Jul 18,2022

vlog -work work -vopt C:/examples/rcosflt_rtl.v

```
# -- Compiling module rcosflt_rtl
#
# Top level modules:
#   rcosflt_rtl
# End time: 12:49:55 on Jul 18,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Compile of rcosflt_rtl.v was successful.
# All compile dependencies have been resolved.
set compcmd [project compileall -n]
# vlog -work work -vopt -stats=none {C:/examples/rcosflt_rtl.v}
# Close project
project close
# reading modelsim.ini
# Compile all files and report error
if [catch {eval $compcmd}] {
    exit -code 1
}
# QuestaSim-64 vlog 2021.4 Compiler 2021.10 Oct 13 2021
# -- Compiling module rcosflt_rtl
#
# Top level modules:
#   rcosflt_rtl
#
# <EOF>
...done
----- Step 4-----
Specify the name of the HDL module for cosimulation. The Cosimulation Wizard will
launch the HDL simulator, load the specified module, and populate the port list
of that HDL module before the next step. Use "Shared Memory" communication
method if your firewall policy does not allow TCP/IP socket communication.
Elaborating and Loading HDL simulation image. Please wait ...
Waiting for HDL Simulator to startup ...
120 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
119 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
118 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
117 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
Waiting for HDL Simulator to startup ...
116 seconds to time-out ...
To stop this process, press Ctrl+C in MATLAB console.
...done
----- Step 5-----
Specify all input and output port types. Input signals that are identified a
s 'Clock' and 'Reset' signals will be forced in the HDL simulator through
Tcl commands. You can specify the timing parameters for forced 'Clock' and
'Reset' signals in the next step. If you want to drive your HDL clock and
reset signals with Simulink signals, mark them as 'Input'.
----- Step 6-----
Set the sample time and data type for each output port. You can specify
the sample time as -1, which means that it will be inherited via back
propagation in the Simulink model. Back propagation may fail in certain
circumstances; click Help for details.
```

----- Step 7-----
 Set clock and reset parameters here. The time in these tables refers to time in the HDL simulator.
 Please wait while generating waveforms.
 ...done

----- Step 8-----
 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).

----- Step 9-----
 When you click Finish, the Cosimulation Wizard performs the following actions:

- Creates and opens a new Simulink model containing an HDL Cosimulation block configured to your specifications.
- Generates the scripts to compile your HDL code and launch the HDL simulator according to the choices you made with this assistant.
- (If you check the box below) Configures the HDL Cosimulation block to assist you in setting the simulation timescale when you cosimulate with the generated block for the first time. If you do not check the box below, the timescale is set to the default of 1 Simulink second = 1 second in the HDL simulator, or you may change it below.

Generating blocks ... Please wait.
 ...done

Input Arguments

cosimConfigObj — Cosimulation configuration

cosimulationConfiguration object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

RestartFromStep — Resume workflow from step

0 (default) | 1

Resume execution of the cosimulation workflow from the specified step.

- 0 — restart workflow from the last step in previous run. If this is the first execution of a workflow, it means that it starts from step one.
- 1 — restart workflow from the beginning. Use this option when encountering an error, so that the workflow goes through all the steps again.

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration`

specifyClock

Package: hdlverifier

Assign clock ports to cosimulation block or System object

Syntax

```
specifyClock(cosimConfigObj,name)
specifyClock(__,Name=Value)
specifyClock(__,name,[])
```

Description

`specifyClock(cosimConfigObj,name)` explicitly maps the HDL port named `name` as a clock port in the generated block or System object. The attributes for the clock inherit default values from the `default_clock_definition` row of the ClockPorts table.

`specifyClock(__,Name=Value)` sets properties using one or more name-value arguments in addition to the input arguments in the previous syntax. Unspecified arguments inherit the value from 'default_clock_definition'.

To change the default clock attributes, specify values for the 'default_clock_definition'.

`specifyClock(__,name,[])` clears the definition for the clock port (or ports) specified in `name`. For example: `specifyClock(c,'clk',[])` clears the definition for the clock named `clk` in the ClockPorts table.

Input Arguments

cosimConfigObj — Cosimulation configuration

cosimulationConfiguration object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

name — HDL clock port name

string | character vector | cell array

Name of the HDL port to map as a clock port in the generated HDL Cosimulation block or `hdlverifier.HDLCosimulation` System object, specified as a string or a character vector. For multiple clocks, specify a cell array of port names.

Example: `clk`

Example: `{'clk1','clk2','clk3'}`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `specifyClock(c, 'clk1', Period=30)` assigns the port named `clk1` a period of 30 time units as defined by `HDLTimeUnit`.

Edge — Active clock edge

Rising (default) | Falling

Active clock edge, specified as Rising or Falling.

Period — Clock period

10 (default) | positive integer

Clock period, specified as a positive integer of `HDLTimeUnit` units.

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration` | `portInterface`

specifyInput

Assign HDL input ports to cosimulation block or System object

Syntax

```
specifyInput(cosimConfigObj,name)  
specifyInput(__,name,[])
```

Description

`specifyInput(cosimConfigObj,name)` explicitly maps the HDL port named `name` as an input in the generated block or System object.

`specifyInput(__,name,[])` clears the definition for the input port (or ports) specified in `name`. For example: `specifyInput(c,'in1',[])` clears the definition for the input named `in1` in the `InputDataPorts` table.

Input Arguments

cosimConfigObj — Cosimulation configuration

`cosimulationConfiguration` object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

name — HDL input port name

`string` | character vector | cell array

Name of the HDL port to map as an input in the generated HDL Cosimulation block or `hdlverifier.HDLCosimulation` System object, specified as a string or a character vector. For multiple inputs, specify a cell array of port names.

Example: `'data'`

Example: `{'in1','in2','in3'}`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration` | `portInterface`

specifyOutput

Package: hdlverifier

Assign HDL output ports to cosimulation block or System object

Syntax

```
specifyOutput(cosimConfigObj,name)
specifyOutput(__,Name=Value)
specifyOutput(__,name,[])
```

Description

`specifyOutput(cosimConfigObj,name)` explicitly maps the HDL port named `name` as an output in the generated block or System object. The attributes for the output inherit default values from the 'default_output_definition' row of the OutputDataPorts table.

`specifyOutput(__,Name=Value)` sets properties using one or more name-value arguments in addition to the input arguments in the previous syntax. Unspecified arguments inherit the value from 'default_output_definition'.

To change the default output attributes, specify values for the 'default_output_definition'.

`specifyOutput(__,name,[])` clears the definition for the output port (or ports) specified in `name`. For example: `specifyOutput(c,out1,[])` clears the definition for the output named `out1` in the OutputDataPorts table.

Input Arguments

cosimConfigObj — Cosimulation configuration

cosimulationConfiguration object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

name — HDL output port name

string | character vector | cell array

Name of the HDL port to map as an output in the generated HDL Cosimulation block or `hdlverifier.HDLCosimulation` System object, specified as a string or a character vector. For multiple outputs, specify a cell array of port names.

Example: `name='data_out'`

Example: `name={'out1','out2','out3'}`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `specifyOutput(c, {'dout1', 'dout2'}, DataType='Fixedpoint', Signed=true, FractionLength=10);` creates two output ports with the default `SampleTime`, with signed `Fixedpoint` data type and fraction length of 10.

SampleTime — Time between reading samples on an output port

1 (default) | nonnegative numeric | -1

Time interval between consecutive samples applied to an output port, specified in seconds.

- Specify a nonnegative numeric value as the output port sample time
- Specify -1 for Simulink workflows to inherit the sample time via back propagation in the Simulink model.

Simulink reads a value from the associated HDL simulator signal at the sample rate specified here.

In general, Simulink handles port sample periods as follows:

- If you connect an input port to a signal that has an explicit sample period, based on forward propagation, Simulink applies that rate to the port.
- If you connect an input port to a signal that does not have an explicit sample period, Simulink assigns a sample period that is equal to the least common multiple (LCM) of all identified input port sample periods in the model.

You must specify an explicit sample time for each output port.

The HDL time corresponding to the Simulink sample time hits depends on the `TimeScale` setting. See “Simulation Timescales” for more information.

Datatype — Data type for output signal

'Fixedpoint' | 'Double' | 'Single' | 'Inherit'

Data type of the output signal, specified as 'Fixedpoint', 'Double', 'Single', or 'Inherit' (for Simulink workflow only).

When building a Simulink workflow, you can select 'inherit' to automatically determine the data type. The HDL Cosimulation block checks that the inherited word length matches the word length queried from the HDL simulator. If they do not match, Simulink generates an error message. For example, if you connect a Signal Specification block to an output, `Inherit` forces the data type specified by the Signal Specification block onto the output port.

If Simulink cannot determine the data type of the signal connected to the output port, it queries the HDL simulator for the data type of the port. For example, if the HDL simulator returns the VHDL data type `STD_LOGIC_VECTOR` for a signal of size `N` bits, the data type `ufixN` is forced on the output port. The implicit fraction length is 0.

You can also assign an explicit data type, with optional fraction length. By explicitly assigning a data type, you can force fixed-point data types on output ports of the HDL Cosimulation block or system object. For example, for an 8-bit output port, setting `Signed` to `true` and setting `FractionLength` to 5 forces the data type to `sfix8_En5`. The width is always inherited from the HDL simulator.

Example: `DataType='Fixedpoint'`

Signed — Sign of outputs

true (default) | false

Sign of the outputs, specified as `true` (signed) or `false` (unsigned).

Example: `true` - All outputs have a signed value.

FractionLength — Output fraction length

0 (default) | integer

Output fraction length, in bits, specified as an integer.

If you do not specify this property, the fraction length inherits the value from `'default_output_definition'`.

Example: `10` — The specified output has a fraction length of 10 bits.

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration` | `portInterface`

specifyReset

Package: hdlverifier

Assign reset ports to cosimulation block or System object

Syntax

```
specifyReset(cosimConfigObj,name)
specifyReset(__,Name=Value)
specifyCReset(__,name,[])
```

Description

`specifyReset(cosimConfigObj,name)` explicitly maps the HDL port named `name` as a reset port in the generated block or system object. The attributes for the reset signal inherit default values from the 'default_reset_definition' row of the ResetPorts table.

`specifyReset(__,Name=Value)` sets properties using one or more name-value arguments in addition to the input arguments in the previous syntax. Unspecified arguments inherit the value from 'default_reset_definition'.

To change the default reset attributes, specify values for the 'default_reset_definition'.

`specifyCReset(__,name,[])` clears the definition for the reset port (or ports) specified in `name`. For example: `specifyReset(c,'rst1',[])` clears the definition for the reset named `rst1` in the ResetPorts table.

Input Arguments

cosimConfigObj — Cosimulation configuration

cosimulationConfiguration object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

name — HDL reset port name

string | character vector | cell array

Name of the HDL port to map as a reset port in the generated HDL Cosimulation block or `hdlverifier.HDLCosimulation` System object, specified as a string or a character vector. For multiple outputs, specify a cell array of port names.

Example: `name='reset_in'`

Example: `name={'rst1','rst2','rst3'}`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `specifyReset(c, 'rst', InitialValue=1, Duration=54)`; creates a reset signal with an initial value of 1 and duration of 54.

InitialValue – Initial value of reset signal

1 (default) | 0

Initial value of reset signal, specified as 0 or 1.

Duration – Duration of reset signal

8 (default) | positive integer

Duration of the reset signal, specified as a positive integer of HDLTimeUnit units.

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration` | `portInterface`

specifyUnused

Package: hdlverifier

Label HDL ports as unused ports

Syntax

```
specifyUnused(cosimConfigObj,name)  
specifyUnused(__,name,[])
```

Description

`specifyUnused(cosimConfigObj,name)` labels the HDL port named `name` as unused in the generated block or System object.

`specifyUnused(__,name,[])` clears the definition for the unused port (or ports) specified in `name`. For example: `specifyUnused(c,'unused1',[])` clears the definition for the input named `unused1` in the `UnusedPorts` table.

Input Arguments

cosimConfigObj — Cosimulation configuration

cosimulationConfiguration object

Cosimulation configuration, specified as a `cosimulationConfiguration` object.

name — HDL unused port name

string | character vector | cell array

Name of the HDL port to label as an unused port in the generated HDL Cosimulation block or `hdlverifier.HDLCosimulation` System object, specified as a string or a character vector. For multiple outputs, specify a cell array of port names.

Example: `name='clk_out'`

Example: `name={'unused1','unused2','unused3'}`

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`cosimulationConfiguration` | `portInterface`

svdpiConfiguration

Configure workflows for UVM and SystemVerilog component generation from MATLAB

Description

The `svdpiConfiguration` object controls the creation of a universal verification methodology (UVM) component or a SystemVerilog DPI component from a MATLAB function.

Creation

Syntax

```
svdpiObj = svdpiConfiguration()
svdpiObj = svdpiConfiguration(ComponentKind)
```

Description

`svdpiObj = svdpiConfiguration()` creates an `svdpiConfiguration` object for a sequential module.

`svdpiObj = svdpiConfiguration(ComponentKind)` creates an `svdpiConfiguration` object for a SystemVerilog module or a UVM component specified by `ComponentKind`.

Properties

Code Generation

CoderConfiguration — a `coder.config` object

configuration object

Specify a custom configuration object using `coder.config('dll')`. The configuration object build type must be set as dynamic library. See `coder.config`.

Component Information

ComponentKind — Kind of SystemVerilog or UVM component to generate

'sequential-module' (default) | 'sequential-module-varsize' | 'combinational-module' | 'uvm-predictor' | 'uvm-sequence' | 'uvm-scoreboard' | 'custom'

Select a built-in template for SystemVerilog DPI or UVM component generation, specified as '*template-name*'. For a customized template, specify 'custom'.

You can override values of built-in template variables through this configuration object. A common use of overrides is to ensure compatibility of the generated code with any existing testbench or component library by avoiding type-name conflicts.

Common overrides for all templates include:

- `ComponentTypeName`, `TestBenchTypeName` — override the default values by setting the `ComponentTypeName` and `TestBenchTypeName` properties in the `svdpiConfiguration` object.
- `ComponentPackageName` — override the default value by setting the `TemplateDictionary` property in the `svdpiConfiguration` object.

Optional template-specific overrides:

- UVM sequence:
 - `SequenceTransTypeName`, `SequencerTypeName`, `SequenceCount`, `SequenceFlushCount` — override the default values by setting the `TemplateDictionary` property in the `svdpiConfiguration` object.
- UVM predictor:
 - `InputTransTypeName`, `OutputTransTypeName` — override the default values by setting the `TemplateDictionary` property in the `svdpiConfiguration` object.
- UVM scoreboard:
 - `InputTransTypeName`, `OutputTransTypeName`, `ConfigObjTypeName` — override the default values by setting the `TemplateDictionary` property in the `svdpiConfiguration` object.
 - `PREDICTOR_INPUTS`, `MONITOR_INPUTS`, `CONFIG_OBJECT_INPUTS` — map HDL ports to groups by using the `addPortGroup` object function with the `svdpiConfiguration` object.

For more information about the template engine, see “SystemVerilog and UVM Template Engine”.

ComponentTypeName — Type name for the main module or component

MATLAB function name (default) | string | character vector

Component type name, specified as a string or character vector. The `dpigen` function uses this argument to name the generated component and the SystemVerilog package files. If you do not specify a component type name, the component type name is the MATLAB function name.

TestBenchTypeName — Type name for the test bench component

tb_function (default) | string | character vector

Test bench type name, specified as a string or character vector. The `dpigen` function uses this argument to name the generated SystemVerilog test bench and its associated files. If you do not specify a component type name, the test bench type name is uses the name of the MATLAB test bench function.

In the code below, the `dpigen` function generates a predictor component, and creates a test bench module for it using the provided test bench function name (`my_tb`).

```
c = svdpiConfiguration('uvm-predictor');
dpigen fooBar -testbench my_tb -config c;
```

To override that test bench name, specify the desired SystemVerilog name. In this example it is specified as `pulse_framed_tb`.

```
c = svdpiConfiguration('uvm-predictor');
c.TestBenchTypeName = 'pulse_framed_tb';
dpigen fooBar -testbench my_tb -config c;
```

Template

TemplateDictionary — List of name and value pairs defined in the template file

cell array of name-value pairs

Each template defines a template dictionary, which declares template-specific variables. Assign values to these variables as a cell array of variable names followed by values.

The template files expand tokens of the form %<Name> with Value. Names and values must be strings or character arrays.

Override default template values here such as transaction type names for UVM components or sequence counts for a UVM sequence component. To see an example, go to [Override Template Variable Values](#) on page 3-0 .

Example:

```
c = svdpiConfiguration('uvm-sequence');
c.TemplateDictionary = {
    'SequenceCount',    '15',
    'SequenceFlushCount', '2',
};
```

In the template file, the line:

```
repeat (%<SequenceCount>)
```

will be replaced with:

```
repeat (15)
```

PortGroups — Display custom lists of port names used by template files

string | character vector

This property is read-only.

A port group represents a section of the generated interface that logically belongs together. For example:

- All inputs to a module belong to the ALL_INPUTS port group in the built-in templates.
- All inputs to a UVM scoreboard module that originate in the monitor belong to the MONITOR_INPUTS port group in the UVM scoreboard template.
- Configuration inputs to a scoreboard belong to the CONFIG_OBJECT_INPUTS interface.

Several built-in groups exist such as ALL_INPUTS and ALL_OUTPUTS. Templates utilize port groups to generate wrapper code specific to that group.

You can modify a port group by using the `addPortGroup` or `removePortGroup` functions respectively.

Example:

```
c = svdpiConfiguration('uvm-scoreboard');
addPortGroup(c, 'PREDICTOR_INPUTS', {'PeakSq','Location','FilterOut_re','FilterOut_im'});
addPortGroup(c, 'MONITOR_INPUTS', {'PeakSqImplIn','LocationImplIn','FilterOutImpl_re','FilterOutImpl_im'});
addPortGroup(c, 'CONFIG_OBJECT_INPUTS', 'pErrorPercentThreshold');
```

In the template file, the configuration object definition can include the following code to allow randomization of variables:

```
%<BEGIN_FOREACH_PORT CONFIG_OBJECT_INPUTS>  
%<PORT_RAND_VAR_DECL>  
%<END_FOREACH_PORT>
```

ComponentTemplateFiles — Template files to use when generating module or component cell array

The path to the template files to use when processing a module or component, specified as a cell array of one or more templates.

A template file can generate several files per component. Template file location and name can use an absolute path or relative path. Relative paths are converted to absolute paths.

Dependencies

To write this property, set the `ComponentKind` property to 'custom'. Otherwise, this property is read-only.

TestBenchTemplateFiles — Template files to use when generating test bench cell array

The path to the template files to use when processing a test bench for a component, specified as a string or as a cell array for multiple templates.

A template file can generate several files per component. Filenames can use an absolute path or relative path. Relative paths are converted to absolute paths.

Dependencies

To write this property, set the `ComponentKind` property to 'custom'. Otherwise, this property is read-only.

Object Functions

`addPortGroup` Add port group mapping to `svdpiConfiguration` object
`removePortGroup` Remove port group mapping from `svdpiConfiguration` object

Examples

Generate SystemVerilog DPI Component from MATLAB Function

This example shows how to generate a SystemVerilog DPI (SVDPI) component from the `sineWaveGen` function by using the default template in HDL Verifier™.

Use Default Template to Create SVDPI Module

Create a configuration object with the default template, and use it with the `dpigen` function. Note the generated SystemVerilog files:

- `sineWaveGen.sv`
- `sineWaveGen_pkg.sv`

```

c=svdpiConfiguration();
dpigen -config c -args {0,0} sineWaveGen

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Rename the generated module to myDut. Note the generated SystemVerilog files:

- myDut.sv
- myDut_pkg.sv

```

c.ComponentTypeName = 'myDut';
dpigen -config c -args {0,0} sineWaveGen

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Use Template to Create UVM Predictor

Create a configuration object with the UVM predictor template, and use it with the dpigen function. Note the generated SystemVerilog files:

- predictor_input_trans.sv
- predictor_output_trans.sv
- sinWave_predictor_pkg.sv
- sinWave_predictor.sv

```

c = svdpiConfiguration('uvm-predictor');
c.ComponentTypeName = 'sinWave_predictor';
dpigen sineWaveGen -config c -args {0,0}

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Override Template Variable Values

Now, change the generated SystemVerilog transaction names.

- Override the default predictor_input_trans and rename it sineWaveTrans.
- Override the default predictor_output_trans and rename it sineWaveOut.

To assign new values to the InputTransTypeName and OutputTransTypeName variables in the template dictionary, set the TemplateDictionary property.

```
c.TemplateDictionary = {
    'InputTransTypeName', 'sineWaveTrans',
    'OutputTransTypeName', 'sineWaveOut'
};
dpigen sineWaveGen -config c -args {0,0}

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.
```

Version History

Introduced in R2023a

See Also

dpigen

Topics

“SystemVerilog and UVM Template Engine”

“Use Templates to Create SystemVerilog DPI and UVM Components”

addPortGroup

Package: hdlverifier

Add port group mapping to svdpiConfiguration object

Syntax

```
addPortGroup(svdpiConfig, groupName, portNames)
```

Description

addPortGroup(svdpiConfig, groupName, portNames) adds a port group named name to the svdpiConfiguration object, and assigns the ports listed in portNames to that group.

Examples

Add Ports to Scoreboard

Create a configuration object using the UVM scoreboard template. Then add port groups to the object:

- inFromMon — monitor inputs to the scoreboard
- inFromPred — predictor inputs to the scoreboard
- inErrorThreshold — configuration inputs to the scoreboard

```
c = svdpiConfiguration('uvm-scoreboard');
addPortGroup(c, 'MONITOR_INPUTS', 'inFromMon'); % single input
addPortGroup(c, 'PREDICTOR_INPUTS', {'inFromPred1', 'inFromPred2'}); % array of 2ports
addPortGroup(c, 'CONFIG_OBJECT_INPUTS', {'inErrorThreshold'});
```

Input Arguments

svdpiConfig — SystemVerilog DPI configuration

svdpiConfiguration object

SystemVerilog DPI configuration, specified as an svdpiConfiguration object.

groupName — HDL port group name

string | character vector

Add a port group to the svdpiConfiguration object. Specify the name of the port group and a list of ports as a string or character vector, separated by a comma.

Example: name='CONFIG_OBJECT_INPUTS', {'pErrorThreshold'} adds an input port group named 'CONFIG_OBJECT_INPUTS', with the ports listed in {'pErrorThreshold'}

portNames — HDL ports assigned to port group

string | character vector | cell array

Specify names of HDL ports to assign to the port group.

- A string or character vector for a single port
- A cell array of strings or character vectors for multiple ports

Version History

Introduced in R2023a

See Also

`svdpiConfiguration` | `removePortGroup`

Topics

“SystemVerilog and UVM Template Engine”

removePortGroup

Package: hdlverifier

Remove port group mapping from svdpiConfiguration object

Syntax

```
removePortGroup(svdpiConfig, groupName)
```

Description

removePortGroup(svdpiConfig, groupName) removes a port group named name from the svdpiConfiguration object.

Examples

Add and Remove Ports from Scoreboard

Create a configuration object using the UVM scoreboard template. Then add port groups to the object:

- inFromMon — monitor inputs to the scoreboard
- inFromPred — predictor inputs to the scoreboard
- inErrorThreshold — configuration inputs to the scoreboard

```
c = svdpiConfiguration('uvm-scoreboard');
addPortGroup(c, 'MONITOR_INPUTS', 'inFromMon'); % single input
addPortGroup(c, 'PREDICTOR_INPUTS', {'inFromPred1', 'inFromPred2'}); % array of 2 ports
addPortGroup(c, 'CONFIG_OBJECT_INPUTS', inErrorThreshold);
```

Now, remove the configuration port group from the object:

```
c.removePortGroup('CONFIG_OBJECT_INPUTS');
```

Input Arguments

svdpiConfig — SystemVerilog DPI configuration

svdpiConfiguration object

SystemVerilog DPI configuration, specified as an svdpiConfiguration object.

groupName — HDL port group name

string | character vector

Remove a port group from the svdpiConfiguration object. Specify the name of the port group as a string or character vector.

Example: name= 'CONFIG_OBJECT_INPUTS'

Version History

Introduced in R2023a

See Also

svdpiConfiguration | addPortGroup

Topics

“SystemVerilog and UVM Template Engine”

uvmcodegen.uvmconfig

UVM configuration object

Description

The `uvmcodegen.uvmconfig` object is a universal verification methodology (UVM) configuration object. Use this object to configure UVM generation options such as the HDL simulation timescale.

Creation

Syntax

```
cfgUvm = uvmcodegen.uvmconfig
cfgUvm = uvmcodegen.uvmconfig(Name, Value)
```

Description

`cfgUvm = uvmcodegen.uvmconfig` creates a default UVM configuration object that configures parameters for generated SystemVerilog code.

`cfgUvm = uvmcodegen.uvmconfig(Name, Value)` sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, `uvmcodegen.uvmConfig('timescale', '1ps/1ps')` specifies a UVM configuration object with a timescale signature of one picosecond for the time unit and one picosecond for the HDL simulation precision.

Properties

timescale — Timescale compiler directive

'1ns/1ns' (default) | T_u/T_p

HDL simulator timescale directive, specified as ' T_u/T_p ', where T_u is the time unit, and T_p is the time precision.

Example: `10us/100ns` specifies a time unit of ten microseconds with an HDL simulation precision of one hundred nanoseconds.

Data Types: `char` | `string`

buildDirectory — Output directory

'.\uvm_build' (default) | character vector | string scalar

Output directory for the generated files, specified as a character vector or string scalar that represents a relative path or absolute path to the output directory.

Example: `C:\UVM\my_uvm_build` specifies the name of the directory in which the `uvmbuild` function places the generated UVM and DPI files.

Data Types: `char` | `string`

Examples

Configure Generated UVM Timescale

Configure the generated UVM test bench to a timescale of 1ns/1ps. In this case, the time unit is one nanosecond, with simulation precision of one picosecond.

```
cfgUVM=uvmcoden.gen.uvmconfig('timescale','1ns/1ps')
```

```
cfgUVM =  
  uvmconfig with properties:  
    timescale: '1ns/1ps'  
    buildDirectory: './uvm_build'
```

Version History

Introduced in R2020b

See Also

uvmbuild

Topics

“Customize Generated UVM Code”

Functions

breakHdlSim

Execute stop command in HDL simulator from MATLAB

Syntax

```
breakHdlSim()  
breakHdlSim(portNumber)  
breakHdlSim(portNumber,hostName)
```

Description

`breakHdlSim()` executes the `stop` command on the HDL simulator on the local host. Use this function to:

- Unblock the HDL simulator after it loads the simulation and before Simulink starts the simulation.
- Unblock the HDL simulator to add more signals to the waveform window when the simulation is in progress.

When you use `breakHdlSim`, you must specify the current connection information to the HDL simulator.

`breakHdlSim(portNumber)` executes the `stop` command in the HDL simulator on the port `portNumber`.

`breakHdlSim(portNumber,hostName)` executes the `stop` command in the HDL simulator on the host `hostName`.

Examples

Execute Stop Command in HDL Simulator from MATLAB

Stop the HDL simulator that is running on the local host.

```
>> breakHdlSim()
```

Stop the HDL simulator that is running on port 1234.

```
>> breakHdlSim('1234')
```

Stop the HDL simulator that is running on port 1234 and host mylinux.

```
>> breakHdlSim('1234','mylinux')
```

Input Arguments

portNumber — Port number to connect

character vector | string scalar

Port number to connect, specified as a character vector or string scalar. The HDL simulator attempts to connect to a host on the specified port number.

Data Types: char | string

hostName — Name of host to connect

character vector | string scalar

Name of the host to connect, specified as a character vector or string scalar.

Data Types: char | string

Version History

Introduced in R2008a

See Also

pingHdlSim | hlddaemon | vsim

Topics

“Run a Simulink Cosimulation Session”

Cosimulation Wizard

Generate a cosimulation block or System object from existing HDL files

Description

Run your HDL design as part of a Simulink model, or MATLAB script. The **Cosimulation Wizard** generates a cosimulation block, System object, or MATLAB callback function. In addition, the wizard creates MATLAB scripts that can compile the HDL and launch the HDL simulator.

Open the Cosimulation Wizard App

- Simulink Toolstrip: In the **Apps** tab, under **Verification, Validation and Test**, click the **HDL Verifier** icon to open the **HDL Verifier** tab. Then, select **HDL Cosimulation** in the **Mode** section, and click **Import HDL Files**.
- MATLAB command prompt: Enter `cosimWizard`.

Examples

- “Verify Raised Cosine Filter Design Using MATLAB”
- “Verify Raised Cosine Filter Design Using Simulink”
- “Cosimulation Wizard for MATLAB System Object”

Version History

Introduced in R2012b

See Also

HDL Cosimulation | `hdlverifier.VivadoHDLCosimulation` | `hdlverifier.HDLCosimulation`

Topics

“Verify Raised Cosine Filter Design Using MATLAB”
“Verify Raised Cosine Filter Design Using Simulink”
“Cosimulation Wizard for MATLAB System Object”
“Supported Data Types”
“Import HDL Code for MATLAB Function”
“Import HDL Code for MATLAB System Object”
“Import HDL Code for HDL Cosimulation Block”

dec2mvl

Convert decimal to binary character vector

Syntax

```
bits = dec2mvl(d)
bits = dec2mvl(d,n)
```

Description

`bits = dec2mvl(d)` converts the decimal integer `d` to a binary character vector `bits`. `d` must be an integer smaller than 2^{52} .

`bits = dec2mvl(d,n)` returns a binary character vector with at least `n` bits.

Examples

Convert Decimal Integers to Multivalued Logic

Find the multivalued logic vector for a positive decimal integer.

```
bits = dec2mvl(23)
```

```
bits =
'10111'
```

Find the multivalued logic vector for a negative decimal integer.

```
bits = dec2mvl(-23)
```

```
bits =
'101001'
```

Find the multivalued logic vector for a negative decimal integer. Specify the minimum number of bits to be returned at the output.

```
bits = dec2mvl(-23,8)
```

```
bits =
'11101001'
```

Input Arguments

d — Decimal number to be converted

decimal integer

Decimal number to convert, specified as a decimal integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

n — Minimum number of bits to return

nonnegative integer

Minimum number of bits to return, specified as a nonnegative integer.

If n is greater than the number of bits required to represent b , the remaining $(n-b)$ upper bits in the output are padded with:

- 0s if input d is a nonnegative integer
- 1s if input d is a negative integer

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2008a

See Also

`mvl2dec`

External Websites

<https://standards.ieee.org/standard/1164-1993.html>

dpigen

Generate UVM or SystemVerilog DPI component from MATLAB function

Syntax

```
dpigen fcn -args args
dpigen fcn -config config -args args
dpigen fcn -args args -testbench tb_name options files -c -launchreport -
PortsDataType type -ComponentTemplateType template_type
```

Description

`dpigen fcn -args args` generates a DPI component shared library from MATLAB function `fcn` and all the functions that `fcn` calls.

- `.dll` for shared libraries on Microsoft® Windows® systems
- `.so` for shared libraries on Linux® systems

The `dpigen` function also generates a SystemVerilog package file, which contains the function declarations.

The argument `-args args` specifies the type of inputs the generated code can accept. The generated DPI component is specialized to the class and size of the inputs. Using this information, `dpigen` generates a DPI component that emulates the behavior of the MATLAB function.

`fcn` and `-args args` are required input arguments. The MATLAB function must be on the MATLAB path or in the current folder.

`dpigen fcn -config config -args args` generates a SystemVerilog or a UVM component based on the template and settings specified in an `svdpiConfiguration` object. For more information about SystemVerilog and UVM templates, see “SystemVerilog and UVM Template Engine”.

`dpigen fcn -args args -testbench tb_name options files -c -launchreport - PortsDataType type -ComponentTemplateType template_type` generates a SystemVerilog DPI component shared library according to the options specified. You can specify zero or more optional arguments, in any order.

- `-testbench tb_name` also generates a test bench for the SystemVerilog DPI component. The MATLAB test bench must be on the MATLAB path or in the current folder.
- `options` specifies additional options for the compiler and code generation.
- `files` specifies custom files to include in the generated code.
- `-c` generates C code only.
- `-launchreport` generates and opens a code generation report.
- `-PortsDataType` specifies the SystemVerilog data type to use for ports.
- `-ComponentTemplateType` specifies whether the design is sequential or combinational.

When generating a DPI component, it creates a shared library specific to that host platform. For example, 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 in Windows. For porting the generated component from Windows to Linux, see “Port Generated Component and Test Bench to Linux”.

Examples

Generate DPI Component and Test Bench

Generate a DPI component and test bench for the function `fun.m` and its associated test bench, `fun_tb.m`. The `dpigen` function compiles the component automatically using the default compiler. The `-args` option specifies that the first input type is a `double` and the second input type is an `int8`.

```
dpigen -testbench fun_tb.m -I E:\HDLTools\ModelSim\10.2c-mw-0\questa_sim\include fun.m
      -args {double(0),int8(0)}

### Generating DPI-C Wrapper fun_dpi.c
### Generating DPI-C 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 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
```

Generate DPI Component and Test Bench Without Compiling

Generate a DPI component and a test bench for the function `fun.m` and its associated test bench, `fun_tb.m`. To prevent the `dpigen` function from compiling the library, include the `-c` option. Send the source code output to 'MyDPIProject'.

```
dpigen -c -d MyDPIProject -testbench fun_tb.m fun.m -args {double(0),int8(0)}

### Generating DPI-C Wrapper fun_dpi.c
### Generating DPI-C 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
### Generating SystemVerilog test bench fun_tb.sv
### Generating test bench simulation script for Mentor Graphics ModelSim/QuestaSim run_tb_mq.do
### 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
```

Generate SystemVerilog DPI Component from MATLAB Function

This example shows how to generate a SystemVerilog DPI (SVDPI) component from the `sineWaveGen` function by using the default template in HDL Verifier™.

Use Default Template to Create SVDPI Module

Create a configuration object with the default template, and use it with the `dpigen` function. Note the generated SystemVerilog files:

- `sineWaveGen.sv`
- `sineWaveGen_pkg.sv`

```

c=svdpiConfiguration();
dpigen -config c -args {0,0} sineWaveGen

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Rename the generated module to myDut. Note the generated SystemVerilog files:

- myDut.sv
- myDut_pkg.sv

```

c.ComponentTypeName = 'myDut';
dpigen -config c -args {0,0} sineWaveGen

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Use Template to Create UVM Predictor

Create a configuration object with the UVM predictor template, and use it with the dpigen function. Note the generated SystemVerilog files:

- predictor_input_trans.sv
- predictor_output_trans.sv
- sinWave_predictor_pkg.sv
- sinWave_predictor.sv

```

c = svdpiConfiguration('uvm-predictor');
c.ComponentTypeName = 'sinWave_predictor';
dpigen sineWaveGen -config c -args {0,0}

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdl
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Override Template Variable Values

Now, change the generated SystemVerilog transaction names.

- Override the default predictor_input_trans and rename it sineWaveTrans.
- Override the default predictor_output_trans and rename it sineWaveOut.

To assign new values to the InputTransTypeName and OutputTransTypeName variables in the template dictionary, set the TemplateDictionary property.

```

c.TemplateDictionary = {
    'InputTransTypeName', 'sineWaveTrans',
    'OutputTransTypeName', 'sineWaveOut'
};
dpigen sineWaveGen -config c -args {0,0}

### Generating DPI-C Wrapper C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-ex1
### Generating DPI-C Wrapper header file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlv
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating source code file C:\TEMP\Bdoc23a_2213998_3568\ib570499\36\tpba0c4ad2\hdlverifier-c
### Generating makefiles for: sineWaveGen_dpi
Code generation successful.

```

Input Arguments

fcn — Name of MATLAB function

character vector | string scalar

Name of MATLAB function to generate the DPI component from, specified as a character vector or string scalar. The MATLAB function must be on the MATLAB path or in the current folder.

-config — SystemVerilog DPI configuration

svdpiConfiguration object | coder.config object

Specify custom configuration parameters using an `svdpiConfiguration` object. The object includes custom C-code generation parameters using a `coder.config('dll')` object and specification of the kind of SystemVerilog component to generate, such as `sequential-module`, `uvm-predictor`, `uvm-sequence`, or `custom`.

To avoid using conflicting options, do not combine a configuration object with command-line options. Usually the `config` object offers more options than the command-line flags.

Note The option to specify a `coder.config` object will be deprecated in a future release. Transition to using an `svdpiConfiguration` object.

Not all the options in the `coder.config` object are compatible with the DPI feature. If you try to use an incompatible option, an error message informs you of which options are not compatible.

-args args — Data type and size of MATLAB function inputs

cell array

Data type and size of MATLAB function inputs, specified as a cell array. Specify the input types that the generated DPI component accepts. `args` is a cell array specifying the type of each function argument. Elements are converted to types using `coder.typeof`. This argument is required.

This argument has the same functionality as the `codegen` function argument `args`. `args` applies only to the function, `fcn`.

Example: `-args {double(0),int8(0)}`

-testbench tb_name — MATLAB test bench used to generate test bench for generated DPI component

character vector | string scalar

MATLAB test bench used to generate test bench for generated DPI component, specified as a character vector or string scalar. The `dpigen` function uses this test bench to generate a SystemVerilog test bench along with data files and execution scripts. The MATLAB test bench must be on the MATLAB path or in the current folder.

The `-testbench` argument requires a Fixed-Point Designer™ license.

Example: `-testbench My_Test_bench.m`

options — Compiler and code generation options

character vector | string scalar

Compiler and codegen options, specified as a character vector or string scalar. These options are a subset of the options for `codegen`. The `dpigen` function gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the right-most option prevails.

You can specify zero or more optional arguments, in any order. For example:

```
dpigen -c -d MyDPIProject -testbench fun_tb.m fun.m -args {double(0),int8(0)}
-launchreport
```

Option flag	Option value
<code>-I include_path</code>	<p>Specifies the path to folders containing headers and library files needed for <code>codegen</code>, specified as a character vector or string scalar. Add <i>include_path</i> to the beginning of the code generation path.</p> <p>For example:</p> <pre>-I E:\HDLTools\ModelSim\10.2c-mw-0\questa_sim\include</pre> <p><i>include_path</i> must not contain spaces, which can lead to code generation failures in certain operating system configurations. If the path contains non 7-bit ASCII characters, such as Japanese characters, <code>dpigen</code> might not find files on this path.</p> <p>When converting MATLAB code to C/C++ code, <code>dpigen</code> searches the code generation path first.</p> <p>Alternatively, you can specify the include path with the <code>files</code> input argument.</p>

Option flag	Option value
-o output	Specify the name of the generated component as a character vector or string scalar. The <code>dpigen</code> function adds a platform-specific extension to this name for the shared library: <ul style="list-style-type: none"> • <code>.dll</code> for C/C++ dynamic libraries on Microsoft Windows systems • <code>.so</code> for C/C++ dynamic libraries on Linux systems
-d dir	Specify the output folder. All generated files are placed in <i>dir</i> . By default, files are placed in <code>./codegen/dll/<function></code> . For example, when <code>dpigen</code> compiles the function <code>fun.m</code> , the generated code is placed in <code>./codegen/dll/fun</code> .
-globals globals	Specify initial values for global variables in MATLAB files. The global variables in your function are initialized to the values in the cell array <code>GLOBALS</code> . The cell array provides the name and initial value of each global variable. If you do not provide initial values for global variables using the <code>-globals</code> option, <code>dpigen</code> checks for the variables in the MATLAB global workspace. If you do not supply an initial value, <code>dpigen</code> generates an error. MATLAB Coder and MATLAB each have their own copies of global data. For consistency, synchronize their global data whenever the two products interact. If you do not synchronize the data, their global variables might differ.
-rowmajor	Specify this option to generate code that uses row-major array layout in all functions. If this option is not specified, the generated code uses column-major array layout. To override the used array layout for a specific function and the functions it calls, specify <code>coder.rowMajor</code> or <code>coder.columnMajor</code> in the body of the function.

files — Custom files to include in the generated code

character vector | string scalar

Custom files to include in the generated code, each file specified as a character vector or string scalar. The files build along with the MATLAB function specified by `fcn`. List each file separately, separated by a space. The following extensions are supported.

File Type	Description
<code>.c</code>	Custom C file
<code>.cpp</code>	Custom C++ file
<code>.h</code>	Custom header file (included by all generated files)
<code>.o</code>	Object file
<code>.obj</code>	Object file

File Type	Description
.a	Library file
.so	Library file
.lib	Library file

In Windows, if your MATLAB function contains matrix or vector output or input arguments, use the `files` option to specify the library (.lib) that contains the ModelSim DPI definitions. Otherwise, you must manually modify the generated Makefile (*.mk) and then compile the library separately.

-c – Option to generate C code only

character vector | string scalar

Option to generate C code without compiling the DPI component, specified as the character vector `-c`. If you do not use the `-c` option, `dpigen` tries to compile the DPI component using the default compiler. To select a different compiler, use the `-config` option and refer to the `codegen` documentation for instructions on specifying the different options.

-launchreport – Option to generate and open a code generation report

character vector | string scalar

Option to generate and open a code generation report, specified as the character vector `-launchreport`.

-PortsDataType – generated SystemVerilog data type for ports

Compatible C Type | Bit Vector | Logic Vector

Select the SystemVerilog data type that will be used for ports. Choose from three possible values:

- `CompatibleCType` - Generate a compatible C type interface for the port.
- `BitVector` - Generate a bit vector type interface for the port.
- `LogicVector` - Generate a logic vector type interface for the port.

This table shows the MATLAB data-type in the left column, and the generated SystemVerilog type for each value of `PortsDataType`.

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		

-ComponentTemplateType – Option to generate combinational SystemVerilog DPI component

Sequential (default) | Combinational

Select the DPI component template for the SystemVerilog wrapper.

- Sequential - to specify a sequential design, with registers.
- Combinational - to specify a combinational design, with no registers.

Dependencies

When the config argument is set to a svdpiConfiguration object, this argument is ignored.

If your function contains persistent variable it is considered a sequential design.

Version History

Introduced in R2014b

R2023a: Support for svdpiConfiguration object

Generate a UVM component or a SystemVerilog DPI component from a MATLAB function. Use the new svdpiConfiguration object to specify the kind of component and a template to use for code generation. Select a built-in template for common component kinds, or create your own custom template.

R2023a: Not recommended

Behavior change in future release

Using the config argument with a coder.config object is not recommended. Use the new svdpiConfiguration object instead. This object adds support for using the “SystemVerilog and UVM Template Engine”, and it allows generation of UVM components from a MATLAB function in addition to SystemVerilog DPI components.

R2022a: Support for combinational designs

Generate a DPI component for a combinational MATLAB function so that outputs immediately reflect changes in the inputs.

Use the the dpigen function with the new -ComponentTemplateType argument set to Combinational.

```
dpigen myCombFunction -args {0,0} -ComponentTemplateType Combinational
```

R2020b: Function input renamed

Behavior changed in R2020b

The `FixedpointDataType` name-value pair argument has been renamed `PortsDataType`.

R2018a: Support for row-major arrays

You can now generate DPI-C code in a row major array. To specify this layout, at the MATLAB command prompt, enter:

```
dpigen -rowmajor
```

Alternatively, insert `coder.rowMajor` in a function body.

See Also

`svdpiConfiguration` | `codegen` | `uvmbuild`

Topics

“Generate DPI Component Using MATLAB”

“SystemVerilog and UVM Template Engine”

“Template Engine Language Syntax”

filProgramFPGA

Load programming file onto FPGA

Syntax

```
filProgramFPGA(fpgaTool,programFile,chainPosition)
```

Description

`filProgramFPGA(fpgaTool,programFile,chainPosition)` loads a bit file onto an FPGA using the toolchain specified by `fpgaTool`.

Examples

Load Programming File to Xilinx FPGA

Program a Xilinx FPGA.

First, set your system environment for accessing Xilinx tools from MATLAB by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado',...  
                'ToolPath','C:\Vivado\2021.1\Win\bin\vivado')
```

Next, program the FPGA using the programming file `/dir/mybitstream.bit` at JTAG chain position 1.

```
filProgramFPGA('Xilinx Vivado','/dir/mybitstream.bit',1);
```

Load Programming File to Intel FPGA

Program an Intel FPGA.

First, set your system environment for accessing Intel tools from MATLAB by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Altera Quartus II',...  
                'ToolPath','C:\altera\20.1\quartus\bin\quartus.exe');
```

Next, program the FPGA using the programming file `/dir/mybitstream.bit` at JTAG chain position 1.

```
filProgramFPGA('Intel','/dir/mybitstream.bit',1);
```

Input Arguments

fpgaTool — FPGA and toolchain vendor

'Xilinx ISE' | 'Xilinx Vivado' | 'Intel' | 'Microchip'

FPGA toolchain vendor, specified as 'Intel', 'Xilinx ISE', 'Xilinx Vivado', or 'Microchip'.

Data Types: char | string

programFile — Name of programming file

character vector | string scalar

Name of the programming file to load to the FPGA, specified as a character vector or string scalar.

Example: 'arty.runs\impl_1\design_1.bit'

Data Types: char | string

chainPosition — JTAG chain position

nonnegative integer

JTAG chain position, specified as a nonnegative integer.

- If you specify `fpgaTool` as 'Xilinx ISE' or 'Xilinx Vivado', `chainPosition` is a required input.
- If you specify `fpgaTool` as 'Intel', specifying `chainPosition` is optional. If you do not specify `chainPosition` in this case, the function assumes a value of 1.
- If you specify `fpgaTool` as 'Microchip', the function ignores `chainPosition`.

Version History

Introduced in R2011a

See Also

`hdlsetuptoolpath`

Topics

“System Object Generation with the FIL Wizard”

“Access FPGA Memory Using JTAG-Based AXI Manager”

FPGA-in-the-Loop Wizard

Generate an FPGA-in-the-loop (FIL) block or System object from existing HDL files

Description

FPGA-in-the-loop (FIL) enables you to run a Simulink or MATLAB simulation that is synchronized with an HDL design running on an Xilinx, Microchip, or Altera® FPGA board.

This link between the simulator and the board enables you to:

- Verify HDL implementations directly against algorithms in Simulink or MATLAB.
- Apply data and test scenarios from Simulink or MATLAB to the HDL design on the FPGA.
- Integrate existing HDL code with models under development in Simulink or MATLAB.

Open the FPGA-in-the-Loop Wizard App

- Simulink Toolstrip: In the **Apps** tab, under **Verification, Validation and Test**, click the **HDL Verifier** icon. Select **HDL Cosimulation** on the left pane, and click **Import HDL Files**.
- MATLAB command prompt: Enter `filWizard`. You provide the HDL code and all related information for creating a FIL block for simulation with an FPGA device.

Examples

- “Block Generation with the FIL Wizard”
- “System Object Generation with the FIL Wizard”

Programmatic Use

`filWizard(filename)` relaunches the FIL Wizard using a configuration file from a previous session. At the end of each FIL Wizard session, the tool saves a MAT-file that contains the session information. You can use this MAT-file to restore the session later.

Version History

Introduced in R2012b

See Also

Topics

- “Block Generation with the FIL Wizard”
- “System Object Generation with the FIL Wizard”
- “FPGA-in-the-Loop Simulation”
- “FPGA-in-the-Loop Simulation Workflows”

hdldaemon

Control MATLAB server that supports interactions with HDL simulator

Syntax

```
hdldaemon
hdldaemon(Name,Value)
hdldaemon(Option)

s=hlddaemon( ___ )
```

Description

`hdldaemon` starts the HDL Link MATLAB server using shared memory inter-process communication. Only one `hdldaemon` per MATLAB session can be running at any given time.

`hdldaemon(Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

- If you do not specify memory type, the server starts using shared memory.
- If you specify the socket `Name, Value` argument, the server starts using socket memory.

Note If server is already running, issuing `hdldaemon` with these arguments shuts down the current server and then starts a new server session using shared memory (unless socket is specified).

`hdldaemon(Option)` accepts a single optional input. Only one option may be specified in a single call. You must establish the server connection before calling `hdldaemon` with one of these options.

`s=hlddaemon(___)` returns the server status connection in structure `s`, using any of the input arguments in the previous syntaxes.

Examples

Start MATLAB Server With Shared Memory

Start the MATLAB server using shared memory communication and use an integer representation of time.

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

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

Start MATLAB Server With Socket Communication

Start MATLAB server and specify socket communication on port 4449.


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

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

Check Server Status

With one or more connections:

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

```
HDLDaemon socket server is running on port 4449 with 1 connections
```

With no connections:

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

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

Server has not been started:

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

```
HDLDaemon is NOT running
```

Check Connection Information

Check connection information for communication mode, number of existing connections, and the interprocess communication identifier (`ipc_id`) the MATLAB server is using for a link.

Returned message for a socket connection:

```
x=hlddaemon('status')
```

```
x =
      comm: 'sockets'
 connections: 0
      ipc_id: '4449'
```

Returned message for a shared memory connection:

```
x=hlddaemon('status')
```

```
x =
      comm: 'shared memory'
 connections: 0
      ipc_id: '\\.\pipe\E505F434-F023-42a6-B06D-DEFD08434C67'
```

You can examine `ipc_id` by entering it at the MATLAB command prompt:

```
x.ipc_id
 '\\.\pipe\E505F434-F023-42a6-B06D-DEFD08434C67'
```

Shut Down Server

Shut down server without shutting down MATLAB.

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

```
HDLDaemon server was shutdown
```

Issue Tcl Commands

Issue simple or complex Tcl commands.

Simple example:

```
hdldaemon('tclcmd','puts "This is a test"')
```

Complex example:

```
tclcmd = {'cd ',unixprojdir},...
         'vlib work',... % create library (if applicable)
         ['vcom -performdefaultbinding ' unixsrcfile1],...
         ['vcom -performdefaultbinding ' unixsrcfile2],...
         ['vcom -performdefaultbinding ' unixsrcfile3],...
         'vsimmatlab work.osc_top ',...
         'matlabcp u_osc_filter -mfunc oscfilter',...
         'add wave sim:/osc_top/clk',...
         'add wave sim:/osc_top/clk_enable',...
         'add wave sim:/osc_top/reset',...
         ['add wave -height 100 -radix decimal -format analog-step -scale 0.001 -offset 50000 ',...
         'sim:/osc_top/osc_out'],...
         ['add wave -height 100 -radix decimal -format analog-step -scale 0.00003125 -offset 50000 ',...
         'sim:/osc_top/filter1x_out'],...
         ['add wave -height 100 -radix decimal -format analog-step -scale 0.00003125 -offset 50000 ',...
         'sim:/osc_top/filter4x_out'],...
         ['add wave -height 100 -radix decimal -format analog-step -scale 0.00003125 -offset 50000 ',...
         'sim:/osc_top/filter8x_out'],...
         'force sim:/osc_top/clk_enable 1 0',...
         'force sim:/osc_top/reset 1 0, 0 120 ns',...
         'force sim:/osc_top/clk 1 0 ns, 0 40 ns -r 80ns',...
         };
```

This example is taken from “Implement Filter Component of Oscillator in MATLAB”. See the full example for use of this complex Tcl command in context.

Input Arguments

Option — Server option to shut down MATLAB server or display server status

```
'kill' | 'stop' | 'status'
```

Server option to shut down MATLAB server or display server status, specified as one of these character vectors:

'kill'	Shuts down the MATLAB server without shutting down MATLAB.
'stop'	Shuts down the MATLAB server without shutting down MATLAB. There is no difference between using 'kill' and 'stop'.

'status' Displays status of the MATLAB server. You can also use `s=hdldaemon('status')`, which displays MATLAB server status and returns status in structure `s`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `'time','int64','quiet','true'` specifies time values are returned as 64-bit integers and output messages are suppressed.

time — Instruction to MATLAB server on how it should send and return time values

'sec' (default) | 'int64'

Instruction to MATLAB server on how it should send and return time values, specified as the comma-separated pair consisting of 'time' and one of these values:

'int64'	Specifies that the MATLAB server send and return time values in the MATLAB function callbacks as 64-bit integers representing the number of simulation steps. See the <code>matlabcp/matlabtb tnow</code> parameter reference (“MATLAB Function Syntax and Function Argument Definitions”).
'sec'	Specifies that the MATLAB server sends and returns time values in the MATLAB function callbacks as <code>double</code> values that HDL Verifier scales to seconds based on the current HDL simulation resolution.

If server is already running, issuing `hdldaemon` with the `time` parameter alone will shut down the current server and start the server up again using shared memory.

Example: `'time','int64'`

quiet — Indicator to suppress printing diagnostic messages

'false' (default) | 'true'

Indicator to suppress printing diagnostic messages, specified as the comma-separated pair consisting of 'quiet' and one of the following values:

'true'	Suppress printing diagnostic messages.
'false'	Do not suppress printing diagnostic messages.

Errors still appear. Use this option to suppress the MATLAB server shutdown message when using `hdldaemon` to get an unused socket number. If server is already running, issuing `hdldaemon` with the `quiet` parameter alone will shut down the current server and start the server up again using shared memory.

Example: `'quiet','true'`

socket — TCP/IP port used for communication

0 | port number | character vector alias

TCP/IP port used for communication, specified as the comma-separated pair consisting of 'socket' and a value. The value can be either 0, indicating that the host automatically chooses a valid TCP/IP port, an explicit port number (1024 < port < 49151) or a service (alias) name from /etc/services file.

If you specify the operating system option (0), use `hdldaemon('status')` to acquire the assigned socket port number.

Example: 'socket',4449

tclcmd — Tcl command transmitted to all connected clients

character vector | string scalar

Tcl command transmitted to all connected clients, specified as any valid Tcl command character vector or string scalar.

The Tcl command you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the character vector cannot include commands such as `start`, `stop`, or `restart` (for ModelSim) or `run`, `stop`, or `reset` (for Xcelium).

Note You can issue this command only after the software establishes a server connection.

Caution Do not call `hdldaemon('tclcmd', 'Tcl command')` from inside a `matlabtb` or `matlabcp` function. Doing so results in a race condition, and the simulator hangs.

Example: 'tclcmd','puts' '"done"'

Output Arguments

s — Structure containing information about the connection

'comm' | 'connections' | 'ipc_id'

Structure containing information about the connection. The structure contains the following variables:

'comm'	Either 'shared memory' or 'sockets'
'connections'	Number of open connections
'ipc_id'	If shared memory, file system name for the shared memory communication channel. If socket, the TCP/IP port number.

Version History

Introduced in R2008a

See Also

`nclaunch` | `vsim`

Topics

“Implement Filter Component of Oscillator in MATLAB”

“Start the HDL Simulator from MATLAB”

hdlsimmatlab

Load instantiated HDL module for verification with Cadence Xcelium and MATLAB

Syntax

```
hdlsimmatlab instance
hdlsimmatlab instance <xmsim_args>
```

Description

Note Use this command in Cadence Xcelium, not in MATLAB.

`hdlsimmatlab instance` loads the specified instance of an HDL design for verification and sets up the Xcelium simulator so it can establish a communication link with MATLAB. The Xcelium simulator opens a simulation workspace as it loads the HDL design.

You can run this command from the HDL simulator prompt or from a Tcl script shell (`tclsh`).

`hdlsimmatlab instance <xmsim_args>` uses additional Xcelium command line arguments.

Examples

Load Instantiated HDL Module for Cosimulation with MATLAB

In Xcelium, load the HDL module instance `parse` from the library `work`, establishing communication with MATLAB.

```
tclshell> hdlsimmatlab work.parse
```

Input Arguments

instance — Instance of HDL module to load for cosimulation

HDL instance name (as required by Xcelium)

Instance of the HDL module to load for cosimulation, specified as an HDL instance name (as required by Xcelium).

Example: `work.parse`

Data Types: `char` | `string`

xmsim_args — xmsim command arguments

xmsim command arguments (as required by Xcelium)

xmsim command arguments (as required by Xcelium). For details, see the description of `xmsim` in the Xcelium documentation.

Version History

Introduced in R2008a

See Also

[nclaunch](#) | [hdlsimulink](#) | [hdlsimmatlabsysobj](#)

hdlsimmatlabsysobj

Load instantiated HDL module for cosimulation with Cadence Xcelium and MATLAB System object

Syntax

```
hdlsimmatlabsysobj instance
hdlsimmatlabsysobj instance <xmsim_args>
hdlsimmatlabsysobj instance -socket tcp_spec <xmsim_args>
```

Description

Note Use this command in Cadence Xcelium, not in MATLAB.

`hdlsimmatlabsysobj instance` loads the specified instance of an HDL design for cosimulation and sets up Cadence Xcelium to establish a shared communication link with a MATLAB System object. Xcelium opens a simulation workspace and displays a series of messages in the command window as it loads the HDL module packages and architectures.

To generate the `hdlsimmatlabsysobj` function, you must first execute the `nc launch` function in MATLAB.

`hdlsimmatlabsysobj instance <xmsim_args>` uses additional Xcelium command line arguments.

`hdlsimmatlabsysobj instance -socket tcp_spec <xmsim_args>` establishes a communication link with a MATLAB System object over a Transmission Control Protocol (TCP) socket. This setting overrides the setting specified with the MATLAB `nc launch` function.

Examples

Load Instantiated HDL Model for Cosimulation with MATLAB System object

In Xcelium, load the HDL module instance `parse` from the library `work`, establishing communication with the MATLAB System object.

```
tclshell> hdlsimmatlabsysobj -gui work.parse
```

Input Arguments

instance — Instance of HDL module to load for cosimulation

HDL instance name, as required by Xcelium

Instance of the HDL module to load for cosimulation.

xmsim_args — xmsim command arguments

xmsim command arguments

`xmsim` command arguments, as required by Xcelium. For details, see the description of `xmsim` in the Xcelium documentation.

tcp_spec — TCP/IP socket communication

TCP/IP port number | TCP/IP service name | internet address

TCP/IP socket communication for the link between Xcelium and MATLAB, specified as a TCP/IP port name or service name. If the MATLAB server is running on a remote host, you must also specify the name or internet address of the remote host. When this input argument is not specified, the function uses shared memory communication. This setting overrides the setting specified with the MATLAB `nclaunch` function.

Version History

Introduced in R2012b

See Also

`nclaunch` | `hdlsimulink` | `hdlsimmatlab`

hdlsimulink

Load instantiated HDL module for cosimulation with Cadence Xcelium and Simulink

Syntax

```
hdlsimulink instance -socket tcp_spec <xmsim_args>
```

Description

Note Issue this command in Cadence Xcelium, not in MATLAB.

`hdlsimulink instance -socket tcp_spec <xmsim_args>` loads the specified instance of HDL design for cosimulation and sets up the Cadence Xcelium simulator so it can establish a shared communication link with Simulink. The Xcelium simulator opens a simulation workspace into which it loads the HDL design.

To generate the `hdlsimulink` function, you must first invoke the `nclaunch` function in MATLAB.

Examples

Load Instantiated HDL Model for Cosimulation with Simulink

In Xcelium, load the HDL module instance `parse` from the library `work`. This action also establishes communication with Simulink and opens a Tcl script shell.

```
tclshell> hdlsimulink -gui work.parse
```

Input Arguments

instance — Instance of HDL design

HDL instance name, as required by Xcelium

Instance of HDL design to load for cosimulation.

xmsim_args — xmsim command arguments

Xcelium command arguments

Specify one or more `xmsim` command line arguments. Do not use `-GUI`, `-BATCH`, or `-TCL`. For details, see the description of `xmsim` in the Xcelium simulator documentation.

tcp_spec — TCP/IP socket communication

TCP/IP port number | TCP/IP service name | internet address

TCP/IP socket communication for the link between the Xcelium simulator and Simulink, specified as a TCP/IP port name or service name. If the MATLAB server is running on a remote host, you must also specify the name or internet address of the remote host. When this input argument is not specified, the function uses shared memory communication. This setting overrides the setting specified with the MATLAB `nclaunch` function.

Version History

Introduced in R2008a

See Also

`nclaunch` | `vsimulink`

matlabcp

Associate MATLAB component function with instantiated HDL design

Syntax

```
matlabcp instance
matlabcp instance time-specs
matlabcp instance ___ pair1 ... pairN
```

Description

Note Enter this command in the HDL simulator, not in MATLAB.

`matlabcp instance` performs these actions:

- Starts the HDL simulator client component of the HDL Verifier software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Creates a process that schedules invocations of the specified MATLAB function.
- Cancels any pending events scheduled by a previous `matlabcp` command that specified the same instance. For example, if you issue the command `matlabcp` for instance `foo`, `matlabcp` cancels all previously scheduled events initiated by `matlabcp` on `foo`.

Issue this command in the HDL simulator.

MATLAB component functions simulate the behavior of modules in an 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. For more information about MATLAB component functions, see “Create a MATLAB Component Function”.

Note The communication mode that you specify for `matlabcp` must match the communication mode you specified for `hdldaemon` when you established the server connection.

For socket communications, specify the port number that you selected for `hdldaemon` when you issue a link request with the `matlabcp` command in the HDL simulator.

`matlabcp instance time-specs` adds time specifications for scheduling the specified MATLAB function.

`matlabcp instance ___ pair1 ... pairN` specifies one or more additional specifications as pairs consisting of a valid specification name and its value. For example, `-mfunc vlogmatlabcp` specifies for the MATLAB function `vlogmatlabcp` to be associated with the specified HDL module. You can specify these pairs with or without `time-specs`.

Examples

Use matlabcp with -mfunc Option to Associate HDL Component with MATLAB Function of Different Name

Associate the Verilog module `vlogtestbench_top.u_matlab_component` with the MATLAB function `vlogmatlabcp` by using the `-mfunc` option. The `-socket` option specifies to use socket communication on port 4449.

```
hdlsim>matlabcp vlogtestbench_top.u_matlab_component -mfunc vlogmatlabcp -socket 4449
```

Use matlabcp with Explicit Times and -cancel Option

Associate the Verilog module `vlogtestbench_top` with the MATLAB function `vlogtestbench_top`, specifying explicit times with the `-cancel` option.

```
hdlsim>matlabcp vlogtestbench_top 1e6 fs 3 2e3 ps -repeat 3 ns -cancel 7ns
```

Use matlabcp with Rising and Falling Edges

Associate the Verilog module `vlogtestbench_top` with the MATLAB function `vlogtestbench_top`, specifying rising and falling edges.

```
hdlsim> matlabcp vlogtestbench_top 1 2 3 4 5 6 7 -rising outclk3
        -falling u_matlab_component/inoutclk
```

Input Arguments

instance — Instance of HDL module

character vector | string scalar

Instance of an HDL module that is associated with a MATLAB function, specified as a character vector or string scalar that indicates an HDL module instance. By default, the `matlabcp` command associates the instance to a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabcp` associates the instance with the MATLAB function `myfirfilter`. The command ignores hierarchy names. For example, if the instance is `top.myfirfilter`, the `matlabcp` command associates only `myfirfilter` with the MATLAB function. To associate the specified instance with a MATLAB function that differs from the instance name, use the `-mfunc` specification as in the `pair1 ... pairN` argument.

Note If you specify an instance of an HDL module that is already associated with a MATLAB function (via `matlabcp` or `matlabtb`) the new association overwrites the existing one.

Data Types: `char` | `string`

time-specs — Schedule execution of MATLAB function

space-separated list of one or more time specifications

Space-separated list of one or more time specifications, specified as a space-separated list of one or more time specifications listed in this table.

Time Specification	Description
<code>time_1 time_2 ... time_n</code>	<p>Specify one or more discrete times at which the HDL simulator calls the specified MATLAB function. The specified times are relative to the current simulation time. If you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Each <code>time_i</code> consists of a number indicating the time value and an optional time unit of:</p> <ul style="list-style-type: none"> • fs (femtoseconds) • ps (picoseconds) • ns (nanoseconds) • us (microseconds) • ms (milliseconds) • sec (seconds) <p>If you do not specify a time unit, the command treats the time value as a value of HDL simulation ticks. Separate each <code>time_i</code> by a space.</p> <p>For example, this code specifies for the MATLAB function <code>vlogmodel_top</code> to execute at time 0 (initial execution) and then at 10 nanoseconds, 10 milliseconds, and 10 seconds.</p> <pre>matlabcp vlogmodel_top 10 ns, 10 ms, 10 sec</pre>
<code>-repeat <time></code>	<p>Specify that the HDL simulator calls the MATLAB function repeatedly based on the specified times. The time values are relative to the value of <code>tnow</code> at the time the HDL simulator first calls the MATLAB function.</p>
<code>-cancel <time></code>	<p>Specify a single time at which the specified MATLAB function stops executing. The time value is relative to the value of <code>tnow</code> at the time the HDL simulator first calls the MATLAB function. If you do not specify a cancel time, the application calls the MATLAB function until you finish the simulation, quit the session, or issue a <code>nomatlabtb</code> call.</p> <hr/> <p>Note The <code>-cancel</code> option works only with the <code><time-specs></code> arguments. It does not affect any of the other scheduling arguments for <code>matlabcp</code>.</p>

Note Place time specifications after the `matlabcp` instance and before any additional command arguments; otherwise the time specifications are ignored.

pair1 ... pairN — Additional specifications

space-separated list of one or more specification pairs

Additional specifications, specified as a space-separated list of one or more specification pairs. A specification pair consists of a name and value. This table shows valid name and value options for these pairs.

Specification Pairs

Name	Value	Description
-socket	Communication mode that matches the communication mode issued with the <code>hdldaemon</code> command	<p>Specify for HDL Verifier to use TCP/IP sockets to communicate between the HDL simulator and MATLAB. Shared memory is the default mode of communication and takes effect if you do not specify <code>-socket <tcp_spec></code>. The communication mode that you specify with the <code>matlabcp</code> command must match the communication mode that you specified with the <code>hdldaemon</code> command.</p> <p>For more information on choosing TCP/IP socket ports, see "TCP/IP Socket Ports".</p>
-rising	Comma-separated list of one or more signal names	<p>Specify <code>-rising</code> with the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on). This pair indicates that the application calls the specified MATLAB function on the rising edge (transition from '0' to '1') of any of the specified signals.</p> <ul style="list-style-type: none"> • In VHDL, a transition from 0 or L to 1 or H determines a rising edge. • In Verilog, a transition from 0 to x, z, or 1, or from x or z to 1 determines a rising edge. <p>Note When specifying signals with the <code>-rising</code> pair, specify the signals in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.</p>
-falling	Comma-separated list of one or more signal names	<p>Specify the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on). Indicates that the application calls the specified MATLAB function whenever any of the specified signals experience a falling edge (a transition from '1' to '0').</p> <p>For determining signal transition in:</p> <ul style="list-style-type: none"> • In VHDL, a transition from 1 or H to 0 or L determines a falling edge. • In Verilog, a transition from 1 to x, z, or 0, or from x or z to 0 determines a falling edge. <p>Note When specifying signals with the <code>-falling</code> pair, specify the signals in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.</p>

Name	Value	Description
-sensitivity	Comma-separated list of one or more signal names	<p>Specify the path names of one or more signals. This pair indicates that the application calls the specified MATLAB function whenever any of the specified signals change state. Signals of any type can appear in the sensitivity list and can be positioned at any level in the HDL model hierarchy.</p> <hr/> <p>Note When specifying signals with the -sensitivity pair, specify the signals in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.</p>
-mfunc	MATLAB function name	<p>Specify the name of the MATLAB function that is associated with the HDL module instance you specify for instance. By default, the HDL Verifier software calls a MATLAB function that has the same name as the specified HDL instance. If the names are the same, you can omit the -mfunc pair. If the names are not the same, use this argument when you call matlabcp. If you omit this argument and matlabcp does not find a MATLAB function with the same name, the command generates an error message.</p>
-use_instance_obj	Structure with fields as described in the table in -use_instance_obj Fields	<p>This pair instructs the function specified with the argument -mfunc to use an HDL instance object passed by HDL Verifier to the function. This value has the fields shown in the applicable table. See “Writing Functions Using the HDL Instance Object” for examples.</p>
-argument	HDL instance argument	<p>Pass user-defined arguments from the matlabcp command on the HDL side to the MATLAB function callbacks. This pair is supported with -use_instance_obj only. For more details, see the field in the table -use_instance_obj Fields.</p>

-use_instance_obj Fields

Field	Read or Write Access	Description
<code>tnext</code>	Write-only	Schedule a callback during the set time. This field is equivalent to old <code>tnext</code> . For example, this code schedules a callback 5 nanoseconds from <code>tnow</code> . <code>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</code>
<code>userdata</code>	Read or Write	Store state variables of the current <code>matlabcp</code> instance. You can retrieve the variables the next time the callback of this instance is scheduled.
<code>simstatus</code>	Read-only	Store the status of the HDL simulator. The HDL Verifier software sets this field to <code>Init</code> during the first callback for this particular instance and to <code>Running</code> thereafter. <code>simstatus</code> is a read-only field. <code>>> hdl_instance_obj.simstatus</code> <code>ans=</code> <code> Init</code>
<code>instance</code>	Read-only	Store the full path of the Verilog or VHDL instance associated with the callback. <code>instance</code> is a read-only field. The value of this field equals that of the module instance specified with the function call. For example: In the HDL simulator: <code>hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj</code> In MATLAB: <code>>> hdl_instance_obj.instance</code> <code>ans=</code> <code> osc_top</code>
<code>argument</code>	Read-only	Store the argument that is set by the <code>-argument</code> pair. For example: <code>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</code> The link software supports the <code>-argument</code> option only when it is used with the <code>-use_instance_obj</code> pair. Otherwise, the command ignores the argument. <code>argument</code> is a read-only property. <code>>> hdl_instance_obj.argument</code> <code>ans=</code> <code> foo</code>

Field	Read or Write Access	Description
portinfo	Read-only	<p>Store information about the VHDL and Verilog ports that are associated with this instance. portinfo is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the portinfo structure passes information such as the type, direction, and size of the port. For more information on port data, see “Gaining Access to and Applying Port Information”.</p> <pre>hdl_instance_obj.portinfo.field1.field2.field3</pre> <p>Note When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.</p>
tscale	Read-only	<p>Store the resolution limit (tick) in seconds of the HDL simulator. tscale is a read-only property.</p> <pre>>> hdl_instance_obj.tscale ans= 1.0000e-009</pre> <p>Note When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.</p>
tnow	Read-only	<p>Store the current time. tnow is a read-only property.</p> <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + faststrate;</pre>
portvalues	Read or Write	<p>Store the current values of and set new values for the output and input ports for a matlabcp instance. For example:</p> <pre>>> hdl_instance_obj.portvalues ans = Read Only Input ports: clk_enable: [] clk: [] reset: [] Read/Write Output ports: sine_out: [22x1 char]</pre>
linkmode	Read-only	<p>Store the status of the callback. HDL Verifier sets this field to 'testbench' if the callback is associated with the matlabb command and 'component' if the callback is associated with the matlabcp command. linkmode is a read-only property.</p> <pre>>> hdl_instance_obj.linkmode ans= component</pre>

Version History

Introduced in R2008a

See Also

matlabtb | hlddaemon | nomatlabtb

matlabtb

Schedule MATLAB test bench session for instantiated HDL module

Syntax

```
matlabtb instance
matlabtb instance time-specs
matlabtb instance ___ pair1 ... pairN
```

Description

Note Enter this command in the HDL simulator, not in MATLAB.

`matlabtb instance` performs the following actions:

- Starts the HDL simulator client component of the HDL Verifier software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Creates a process that schedules invocations of the specified MATLAB function.
- Cancels any pending events scheduled by a previous `matlabtb` command that specified the same instance. For example, if you issue the command `matlabtb` for instance `foo`, `matlabtb` cancels all previously scheduled events initiated by `matlabtb` on `foo`.

This command is issued in the HDL simulator.

MATLAB test bench functions mimic stimuli passed to entities in the HDL model. Force stimulus from MATLAB or HDL scheduled with `matlabtb`.

Note The communication mode that you specify for `matlabtb` must match the communication mode you specified for `hdldaemon` when you established the server connection.

For socket communications, specify the port number you selected for `hdldaemon` when you issue a link request with the `matlabtb` command in the HDL simulator.

`matlabtb instance time-specs` adds time specifications for scheduling the specified MATLAB function.

`matlabtb instance ___ pair1 ... pairN` specifies one or more additional specifications as pairs consisting of a valid specification name and its value. For example, `-mfunc vlogmatlabbc` specifies for the MATLAB function `vlogmatlabbc` to be associated with the specified HDL module. You can specify these pairs with or without `time-specs`.

Examples

Use `matlabtb` with `-socket` Argument and Time Parameters

Start the HDL simulator client component of HDL Verifier, associate an instance of the entity `myfirfilter` with the MATLAB function `myfirfilter`, and begin a local TCP/IP socket-based test bench session using TCP/IP port 4449. Based on the specified test bench stimuli, `myfirfilter.m` executes 5 nanoseconds from the current time and then executes repeatedly every 10 nanoseconds.

```
hdlsim> matlabtb myfirfilter 5 ns -repeat 10 ns -socket 4449
```

Apply Rising Edge Clocks and State Changes

Start the HDL simulator client component of HDL Verifier. Begin a remote TCP/IP socket-based session using the remote MATLAB host computer named `computer123` and TCP/IP port 4449. Based on the specified test bench stimuli, `myfirfilter.m` executes 10 nanoseconds from the current time, each time the signal `/top/fclk` experiences a rising edge, and each time the signal `/top/din` changes state.

```
hdlsim> matlabtb /top/myfirfilter 10 ns -rising /top/fclk -sensitivity /top/din  
-socket 4449@computer123
```

Specify a MATLAB Function Name and Sensitizing Signals

Start the HDL simulator client component of the HDL Verifier software. The `-mfunc` option specifies the MATLAB function to connect to and the `-socket` pair specifies the port number for socket connection mode. The `-sensitivity` pair indicates that the test bench session is sensitized to the signal `sine_out`.

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

Input Arguments

instance — Instance of HDL module associated with MATLAB test bench function

character vector | string scalar

Instance of an HDL module that is associated with a MATLAB function, specified as a character vector or string scalar that indicates an HDL module instance. By default, the `matlabtb` command associates the instance to a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabtb` associates the instance with the MATLAB function `myfirfilter`. The command ignores hierarchy names. For example, if the instance is `top.myfirfilter`, the `matlabtb` command associates only `myfirfilter` with the MATLAB function. To associate the specified instance with a MATLAB function that differs from the instance name, use the `-mfunc` specification as in the `pair1 ... pairN` argument.

Note If you specify an instance of an HDL module that is already associated with a MATLAB function (via `matlabcp` or `matlabtb`) the new association overwrites the existing one.

Data Types: `char` | `string`

time-specs — Schedule execution of MATLAB function

space-separated list of one or more time specifications

Space-separated list of one or more time specifications, specified as a space-separated list of one or more time specifications listed in this table.

Time Specification	Description
<i>time_1</i> <i>time_2</i> ... <i>time_n</i>	<p>Specify one or more discrete times at which the HDL simulator calls the specified MATLAB function. The specified times are relative to the current simulation time. If you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Each <i>time_i</i> consists of a number indicating the time value and an optional time unit of:</p> <ul style="list-style-type: none"> • fs (femtoseconds) • ps (picoseconds) • ns (nanoseconds) • us (microseconds) • ms (milliseconds) • sec (seconds) <p>If you do not specify a time unit, the command treats the time value as a value of HDL simulation ticks. Separate each <i>time_i</i> by a space.</p> <p>For example, this code specifies for the MATLAB function <code>vlogmodel_top</code> to execute at time 0 (initial execution) and then at 10 nanoseconds, 10 milliseconds, and 10 seconds.</p> <pre>matlabtb vlogmodel_top 10 ns, 10 ms, 10 sec</pre>
<code>-repeat <time></code>	Specify that the HDL simulator calls the MATLAB function repeatedly based on the specified times. The time values are relative to the value of <code>tnow</code> at the time the HDL simulator first calls the MATLAB function.
<code>-cancel <time></code>	<p>Specify a single time at which the specified MATLAB function stops executing. The time value is relative to the value of <code>tnow</code> at the time the HDL simulator first calls the MATLAB function. If you do not specify a cancel time, the application calls the MATLAB function until you finish the simulation, quit the session, or issue a <code>nomatlabtb</code> call.</p> <p>Note The <code>-cancel</code> option works only with the <code><time-specs></code> arguments. It does not affect any of the other scheduling arguments for <code>matlabtb</code>.</p>

Note Place time specifications after the `matlabtb` instance and before any additional command arguments; otherwise the time specifications are ignored.

pair1 ... pairN — Additional specifications

space-separated list of one or more specification pairs

Additional specifications, specified as a space-separated list of one or more specification pairs. A specification pair consists of a name and value. This table shows valid options for these pairs.

Specification Pairs

Name	Value	Description
-socket	Communication mode that matches the communication mode issued with the <code>hdldaemon</code> command	<p>Specify for HDL Verifier to use TCP/IP sockets to communicate between the HDL simulator and MATLAB. Shared memory is the default mode of communication and takes effect if you do not specify <code>-socket <tcp_spec></code>. The communication mode that you specify with the <code>matlabtb</code> command must match the communication mode that you specified with the <code>hdldaemon</code> command.</p> <p>For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.</p> <p>Note The communication mode that you specify with the <code>matlabtb</code> command must match what you specify for the communication mode when you issue the <code>hdldaemon</code> command in MATLAB. For more information on modes of communication, see “Communications for HDL Cosimulation”. For more information on establishing the MATLAB end of the communication link, see “Start the HDL Simulator from MATLAB”.</p>
-rising	Comma-separated list of one or more signal names	<p>Specify <code>-rising</code> with the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on). This pair indicates that the application calls the specified MATLAB function on the rising edge (transition from '0' to '1') of any of the specified signals.</p> <ul style="list-style-type: none"> • In VHDL, a transition from 0 or L to 1 or H determines a rising edge. • In Verilog, a transition from 0 to x, z, or 1, or from x or z to 1 determines a rising edge. <p>Note When specifying signals with the <code>-rising</code> pair, specify the signals in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.</p>

Name	Value	Description
-falling	Comma-separated list of one or more signal names	<p>Specify the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on). Indicates that the application calls the specified MATLAB function whenever any of the specified signals experience a falling edge(a transition from '1' to '0').</p> <p>For determining signal transition in:</p> <ul style="list-style-type: none"> • In VHDL, a transition from 1 or H to 0 or L determines a falling edge. • In Verilog, a transition from 1 to x, z, or 0, or from x or z to 0 determines a falling edge. <p>Note When specifying signals with the -falling pair, specify the signals in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.</p>
-sensitivity	Comma-separated list of one or more signal names	<p>Specify the path names of one or more signals. This pair indicates that the application calls the specified MATLAB function whenever any of the specified signals change state. Signals of any type can appear in the sensitivity list and can be positioned at any level in the HDL model hierarchy.</p> <p>Note When specifying signals with the -sensitivity pair, specify the signals in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.</p>
-mfunc	MATLAB function name	<p>Specify the name of the MATLAB function that is associated with the HDL module instance you specify for instance. By default, the HDL Verifier software calls a MATLAB function that has the same name as the specified HDL instance. If the names are the same, you can omit the -mfunc pair. If the names are not the same, use this argument when you call <code>matlabtb</code>. If you omit this argument and <code>matlabtb</code> does not find a MATLAB function with the same name, the command generates an error message.</p>
-use_instance_obj	Structure with fields as described in the table in -use_instance_obj Fields	<p>This pair instructs the function specified with the argument -mfunc to use an HDL instance object passed by HDL Verifier to the function. This value has the fields shown in the applicable table. See “Writing Functions Using the HDL Instance Object” for examples.</p>

Name	Value	Description
-argument	HDL instance argument	Pass user-defined arguments from the matlabtb command on the HDL side to the MATLAB function callbacks. This pair is supported with -use_instance_obj only. For more details, see the field in the table -use_instance_obj Fields.

-use_instance_obj Fields

Field	Read or Write Access	Description
<code>tnext</code>	Write-only	Schedule a callback during the set time. This field is equivalent to old <code>tnext</code> . For example, this code schedules a callback 5 nanoseconds from <code>tnow</code> . <code>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</code>
<code>userdata</code>	Read or Write	Store state variables of the current <code>matlabtb</code> instance. You can retrieve the variables the next time the callback of this instance is scheduled.
<code>simstatus</code>	Read-only	Store the status of the HDL simulator. The HDL Verifier software sets this field to <code>Init</code> during the first callback for this particular instance and to <code>Running</code> thereafter. <code>simstatus</code> is a read-only field. <code>>> hdl_instance_obj.simstatus</code> <code>ans=</code> <code> Init</code>
<code>instance</code>	Read-only	Store the full path of the Verilog or VHDL instance associated with the callback. <code>instance</code> is a read-only field. The value of this field equals that of the module instance specified with the function call. For example: In the HDL simulator: <code>hdlsim> matlabtb osc_top -mfunc oscfilter use_instance_obj</code> In MATLAB: <code>>> hdl_instance_obj.instance</code> <code>ans=</code> <code> osc_top</code>
<code>argument</code>	Read-only	Store the argument that is set by the <code>-argument</code> pair. For example: <code>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</code> The link software supports the <code>-argument</code> option only when it is used with the <code>-use_instance_obj</code> pair. Otherwise, the command ignores the argument. <code>argument</code> is a read-only property. <code>>> hdl_instance_obj.argument</code> <code>ans=</code> <code> foo</code>

Field	Read or Write Access	Description
portinfo	Read-only	<p>Store information about the VHDL and Verilog ports that are associated with this instance. <code>portinfo</code> 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 type, direction, and size of the port. For more information on port data, see “Gaining Access to and Applying Port Information”.</p> <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>
tscale	Read-only	<p>Store the resolution limit (tick) in seconds of the HDL simulator. <code>tscale</code> is a read-only property.</p> <pre>>> hdl_instance_obj.tscale 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	<p>Store the current time. <code>tnow</code> is a read-only property.</p> <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + faststrate;</pre>
portvalues	Read or Write	<p>Store the current values of and set new values for the output and input ports for a <code>matlabtb</code> instance. For example:</p> <pre>>> hdl_instance_obj.portvalues ans = Read Only Input ports: clk_enable: [] clk: [] reset: [] Read/Write Output ports: sine_out: [22x1 char]</pre>
linkmode	Read-only	<p>Store the status of the callback. HDL Verifier sets this field to 'testbench' if the callback is associated with the <code>matlabtb</code> command and 'component' if the callback is associated with the <code>matlabcp</code> command. <code>linkmode</code> is a read-only property.</p> <pre>>> hdl_instance_obj.linkmode ans= component</pre>

Version History

Introduced in R2008a

See Also

matlabcp | hlddaemon | nomatlabtb

matlabtbeval

Call MATLAB function once and immediately on behalf of instantiated HDL module

Syntax

```
matlabtbeval instance
matlabtbeval instance -mfunc function_name
matlabtbeval instance -socket tcp_spec
```

Description

Note Enter this command in the HDL simulator, not in MATLAB. This command is available only after the HDL simulator loads the MATLAB library.

`matlabtbeval instance` performs these actions:

- Starts the HDL Simulator client component of HDL Verifier.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Executes the specified MATLAB function once and immediately on behalf of the specified module instance.

Note The `matlabtbeval` command executes the MATLAB function immediately, whereas the `matlabtb` command provides options for scheduling MATLAB function execution.

`matlabtbeval instance -mfunc function_name` associates the HDL instance with the MATLAB function specified by `function_name`.

`matlabtbeval instance -socket tcp_spec` establishes a communication link with a MATLAB function over a transmission control protocol (TCP) socket.

Examples

Connect to myfirfilter and Execute

Start the HDL simulator client component of the link software, associate an instance of the module `myfirfilter` with the function `myfirfilter.m`, and use a local TCP/IP socket-based communication link to TCP/IP port 4449 to execute the function `myfirfilter.m`.

```
hdlsim> matlabtbeval myfirfilter -socket 4449:
```

Input Arguments

instance — Instance of HDL module

character vector | string scalar

Instance of an HDL module that is associated with a MATLAB function, specified as a character vector or string scalar that indicates an HDL module instance. By default, the `matlabtbeval` command associates the instance to a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabtbeval` associates the instance with the MATLAB function `myfirfilter`. The command ignores hierarchy names. For example, if the instance is `top.myfirfilter`, the `matlabtbeval` command associates only `myfirfilter` with the MATLAB function. To associate the specified instance with a MATLAB function that differs from the instance name, use the `function_name` input.

Note If you specify an instance of an HDL module that is already associated with a MATLAB function (via `matlabcp` or `matlabtb`) the new association overwrites the existing one.

Data Types: `char` | `string`

function_name — Name of MATLAB function to associate with HDL instance

character vector | string scalar

Name of the MATLAB function to associate with the HDL instance, specified as a character vector and string scalar. If you omit this argument, `matlabtbeval` associates the instance with a MATLAB function that has the same name as the `instance` input. If you omit this argument and `matlabtbeval` does not find a MATLAB function with the same name as `instance`, the command displays an error message.

Data Types: `char` | `string`

tcp_spec — TCP/IP socket communication

TCP/IP port number | TCP/IP service name | Internet address

TCP/IP socket communication for the link between the HDL simulator and MATLAB, specified as a TCP/IP port number, TCP/IP service name, or Internet address. If the MATLAB server is running on a remote host, you must also specify the name or internet address of the remote host. If you omit this argument, `matlabtbeval` uses shared memory communication.

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.

Note The communication mode that you specify with the `matlabtbeval` command must match what you specify for the communication mode when you call the `hdldaemon` command to start the MATLAB server. For more information on communication modes, see “Communications for HDL Cosimulation”. For more information on establishing the MATLAB end of the communication link, see “Start the HDL Simulator from MATLAB”.

Example: `-socket 4040`

Version History

Introduced in R2008a

See Also

`matlabtb` | `matlabcp` | `hdldaemon`

mvl2dec

Convert multivalued logic to decimal

Syntax

```
D = mvl2dec(mv_logic_char)
D = mvl2dec(mv_logic_char,signed)
```

Description

D = mvl2dec(mv_logic_char) converts a multivalued logic to a positive decimal integer.

Note If mv_logic_char contains any character other than '0' or '1', the output returned is NaN.

D = mvl2dec(mv_logic_char,signed) converts a signed multivalued logic to a positive or negative decimal integer.

Examples

Convert Multivalued Logic Vectors to Decimal Integers

Find the decimal integer equivalent for a multivalued logic vector.

```
mvl2dec('010111')
ans = 23
```

Find the decimal integer equivalent for a multivalued logic vector with one or more values that are not 0 or 1. The function returns NaN.

```
mvl2dec('x01201')
ans = NaN
```

Find the decimal integer equivalent for a signed multivalued logic vector. The second input argument indicates that the input is a signed vector.

```
mvl2dec('10111',true)
ans = -9
```

Input Arguments

mv_logic_char — Multivalued logic to convert

character vector | string scalar

Multivalued logic to convert, specified as a character vector or string scalar.

Data Types: `char` | `string`

signed — Implementation of multivalued logic

`false` (0) (default) | `true` (1)

Implementation of the multivalued logic, specified as one of the values in this table

Value	Description
<code>true</code>	The input is a signed multivalued logic. The function assumes that the first character <code>mv_logic_char(1)</code> is a signed bit of a two's complement number.
<code>false</code>	The input is an unsigned multivalued logic.

Data Types: `logical`

Version History

Introduced in R2008a

See Also

`dec2mvl`

External Websites

<https://standards.ieee.org/standard/1164-1993.html>

nclaunch

Start and configure Cadence Xcelium simulator for use with HDL Verifier software

Syntax

```
nclaunch
nclaunch(Name,Value)
```

Description

nclaunch starts the Cadence Xcelium simulator for use with the MATLAB and Simulink features of the HDL Verifier software. The first folder in the Xcelium simulator matches your MATLAB current folder if you do not specify an explicit `rundir` parameter.

nclaunch(Name,Value) specifies name-value pair arguments that allows you to customize the Tcl commands used to start the Xcelium simulator, the `xmsim` executable to be used, the path and name of the Tcl script that stores the start commands, and for Simulink applications, details about the mode of communication to be used by the applications.

After you call this function, you can use HDL Verifier functions for the HDL simulator (for example, `hdlsimmatlab`, `hdlsimulink`) to do interactive debug setup.

Examples

Start Cosimulation Session with Simulink

Compile design and start Simulink.

```
nclaunch('tclstart',{'exec xmvlog -64bit -c +access+rw +linedebug top.v','hdlsimulink' ...
-gui work.top},'socketsimulink','4449','rundir','/proj');
```

In this example, nclaunch performs the following:

- Compiles the design `top.v`: `exec xmvlog -64bit -c +access+rw +linedebug top.v`.
- Starts Simulink with the GUI from the `proj` folder with the model loaded: `hdlsimulink -gui work.top` and `'rundir', '/proj'`.
- Instructs Simulink to communicate with the HDL Verifier interface on socket port 4449: `'socketsimulink','4449'`.

All of these commands are specified in a single character vector as the property value to `tclstart`.

Create Tcl Script to Start HDL Simulator

Create a Tcl script to start the HDL simulator from a Tcl shell using nclaunch.

Specify the name of the Tcl script and the command(s) it includes as parameters to nclaunch:

```
nclaunch('tclstart','xxx','startupfile','myTclscript','starthdlsim','yes')
```

In this example, a Tcl script is created and the command to start the HDL simulator is included. The startup Tcl file is named "myTclscript".

Execute the script in a Tcl shell:

```
shell> Tclsh myTclscript
```

This starts the HDL simulator.

Execute Multiple Tcl Commands When Launching Cosimulation Connection

Build a sequence of Tcl commands that are then executed in a Tcl shell, after calling `nclaunch` from MATLAB.

Assign Tcl command values to the `Tclcmd` parameter of `nclaunch`:

```
Tclcmd{1} = 'exec xmvlog -64bit vlogtestbench_top.v'
Tclcmd{2} = 'exec xmelab -64bit -access +wc vlogtestbench_top'
Tclcmd{3} = ['hdlsimmatlab -gui vlogtestbench_top ' '-input "@matlabcp...
            vlogtestbench_top.u_matlab_component -mfunc vlogmatlabc...
            -socket 32864}' ' '-input "{@run 50}"]

Tclcmd =
    'exec xmvlog -64bit vlogtestbench_top.v' 'exec xmelab -64bit -access +wc vlogtestbench_top'

Tclcmd =
    'exec xmvlog -64bit vlogtestbench_top.v' 'exec xmelab -64bit -access +wc vlogtestbench_top'

Tclcmd =
    [1x31 char]    [1x41 char]    [1x145 char]
```

- `tclcmd{1}` compiles `vlogtestbench_top`.
- `tclcmd{2}` elaborates the model.
- `tclcmd{3}` calls `hdlsimmatlab` in `gui` mode and loads the elaborated `vlogtestbench_top` in the simulator.

Issue the `nclaunch` command, passing the `tclcmd` variable just set:

```
nclaunch('hdlsimdir', 'local.IUS.glnx.tools.bin', 'tclstart', tclcmd);
```

In this example, the `nclaunch` launches the following tasks through the Tcl commands assigned in `tclcmd`:

- Executes the arguments being passed with `-input` (`matlabtb` and `run`) in the `xsim` Tcl shell.
- Issues a call to `matlabcp`, which associates the function `vlogmatlabc` to the module instance `u_matlab_component`.
- Assumes that the `hdl_daemon` in MATLAB is listening on port 32864
- Instructs the `run` function to run 50 resolution units (ticks).

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `nclaunch('runmode','GUI')` starts the HDL simulator with graphical user interface.

hdlsimdir — Path to Cadence Xcelium simulator executable

path name

Path to Xcelium simulator executable, specified as the comma-separated pair consisting of 'hdlsimdir' and a path name. By default, function executes first version of the simulator that the function finds on system path.

Data Types: char

hdlsimexe — Name of Xcelium simulator executable

'xmsim' (default) | character vector

Name of Xcelium simulator executable, specified as the comma-separated pair consisting of 'hdlsimexe' and a simulator name. By default, function uses 'xmsim' simulator.

Data Types: char

libdir — Entry in startup Tcl file

folder name

Entry in startup Tcl file, specified as the comma-separated pair consisting of 'libdir' and a folder name. It points to the folder with the shared libraries for Xcelium simulator to communicate with MATLAB when Xcelium simulator runs on a machine that does not have MATLAB.

Data Types: char

libfile — Library file for HDL simulation

library file name

Library file for HDL simulation, specified as the comma-separated pair consisting of 'libfile' and the library file name. If the HDL simulator links other libraries, including SystemC libraries, that were built using a compiler supplied with the HDL simulator, you can specify an alternate library file with this property. By default, function uses that version of the library file which was built using the same compiler that MATLAB itself uses.

Data Types: char

rundir — Location to run HDL simulator

folder name

Location to run HDL simulator, specified as the comma-separated pair consisting of 'rundir' and a folder name.

The following conditions apply to this name-value pair:

- If the value of `dirname` is "TEMPDIR", the function creates a temporary folder in which it runs the HDL simulator.
- If you specify `dirname` and the directory does *not* exist, you will get an error.

Data Types: char

runmode — Run mode for HDL simulator

'GUI' (default) | 'Batch' | 'Batch with Xterm' | 'CLI'

Run mode for HDL simulator, specified as the comma-separated pair consisting of 'runmode' and one of the following values:

- 'Batch' - Starts HDL simulator in background with no window
- 'Batch with Xterm' - Starts HDL simulator in background with no window
- 'CLI' - Starts HDL simulator in an interactive terminal window
- 'GUI' - Starts HDL simulator with graphical user interface

socketsimulink — TCP/IP socket communication between the Xcelium simulator and Simulink

tcp_spec

TCP/IP socket communication between Xcelium simulator and Simulink, specified as the comma-separated pair consisting of 'socketsimulink' and a port number or service name. By default, function uses shared memory communication.

Data Types: char

starthdlsim — Option to start the Xcelium simulator

'yes' (default) | 'no'

Option to start Xcelium, specified as the comma-separated pair consisting of 'starthdlsim' and one of the following values:

- 'yes' - To create a startup Tcl file after launching Xcelium simulator.
- 'no' - To create a startup Tcl file without launching Xcelium simulator.

startupfile — Name and location of generated Tcl file

path name

Name and location of the generated Tcl file, specified as the comma-separated pair consisting of 'startupfile' and a path name. The generated Tcl script, when executed, compiles and launches the Xcelium simulator.

Data Types: char

tclstart — Execute Tcl commands

tcl commands

Execute TCL commands before Xcelium simulator launches, specified as the comma-separated pair consisting of 'tclstart' and a Tcl command.

Note You must type exec in front of non -Tcl system shell commands. For example:

```
exec -xmvlog -64bit -c +access+rw +linedebug top.v
hdlsimulink -gui work.top
```

You must specify at least one command; otherwise, no action occurs.

Data Types: char

Version History

Introduced in R2008a

nomatlabtb

End active MATLAB test bench and MATLAB component sessions

Syntax

```
nomatlabtb
```

Description

Note Enter this command in the HDL simulator, not in MATLAB.

`nomatlabtb` ends all active MATLAB test bench and MATLAB component sessions that were previously initiated by the `matlabtb` or `matlabcp` functions.

Issue this command in the HDL simulator.

Note Call this command before shutting down `hdldaemon`. Otherwise, `hdldaemon` blocks the shutdown process until you call this command.

Examples

End MATLAB Component and Test Bench Sessions

End all active MATLAB test bench and MATLAB component sessions.

```
hdlsim> nomatlabtb
```

Version History

Introduced in R2008a

See Also

`matlabtb` | `matlabcp` | `hdldaemon`

notifyMatlabServer

Send HDL simulator event ID and process ID to MATLAB server

Syntax

```
notifyMatlabServer eventID -socket tcp_spec
```

Description

Note Issue this command in the HDL simulator, not in MATLAB. It is only available after the HDL simulator loads the MATLAB library.

`notifyMatlabServer eventID -socket tcp_spec` sends the HDL simulator event ID and process identification (PID) to the MATLAB server (`hdldaemon`) using the specified connection methods (socket or shared memory). For MATLAB to receive these IDs, `hdldaemon` must be running with the same communication mode specified by the `notifyMatlabServer` function. The event ID and the PID queue in `hdldaemon`. `notifyMatlabServer` is often used with `waitForHdlClient` to make sure that the HDL simulator is ready to begin or continue processing.

Examples

Send HDL Simulator Event and Process IDs to MATLAB Server

If `EventID = 5` is received within 100 seconds, the function returns the HDL simulator PID. If a time-out occurs, the function returns -1.

```
>> hdldaemon('socket',5002);
...
>> hdlpid = waitForHdlClient(100,5);
```

In the HDL simulator, use the `notifyMatlabServer` command to send event ID 5 to `hdldaemon` running on the same machine using TCP/IP socket port 5002.

```
>> notifyMatlabServer 5 -socket 5002
```

Input Arguments

eventID — Event ID to send to hdldaemon

1 (default) | 32-bit positive integer

Event ID to send to `hdldaemon` specified as a positive integer. This input argument contains the event ID expected by the command `waitForHdlClient` in MATLAB.

tcp_spec — TCP/IP socket communication

TCP/IP port number | TCP/IP service name | internet address

TCP/IP socket communication for the link between the HDL simulator and MATLAB, specified as a TCP/IP port name or service name. If the MATLAB server is running on a remote host, you must also

specify the name or internet address of the remote host. When this input argument is not specified, the function uses shared memory communication.

Version History

Introduced in R2012b

See Also

`waitForHdlClient` | `hdldaemon`

pingHdlSim

Block cosimulation until HDL simulator is ready

Syntax

```
pID = pingHdlSim(timeout)
pID = pingHdlSim(timeout,portnumber)
pID = pingHdlSim(timeout,portnumber,hostname)
```

Description

`pID = pingHdlSim(timeout)` attempts to connect to the HDL simulator using a shared connection. The function blocks cosimulation until the HDL server loads or the specified `timeout` occurs. `pingHdlSim` returns the process ID `pID` of the HDL simulator or `-1` if a timeout occurs. When you automate a cosimulation, use this function to determine if the HDL server is loaded before your script continues the simulation.

`pID = pingHdlSim(timeout,portnumber)` attempts to connect to the local host on the port `portnumber`.

`pID = pingHdlSim(timeout,portnumber,hostname)` attempts to connect to the host `hostname` on port `portnumber`.

Examples

Block Cosimulation Until HDL Simulator Is Ready

The following function call blocks further cosimulation until the HDL server loads or 30 seconds pass.

```
>>pingHdlSim(30)
```

If the server loads within 30 seconds, `pingHdlSim` returns the process ID. Otherwise, `pingHdlSim` returns `-1`.

The following function call blocks further cosimulation on port 5678 until the HDL server loads or 20 seconds pass.

```
>>pingHdlSim(20, '5678')
```

The following function call blocks further cosimulation on port 5678 on host name `msuser` until the HDL server loads or 20 seconds pass:

```
>>pingHdlSim(20, '5678', 'msuser')
```

Input Arguments

timeout — Number of seconds to wait for response

positive scalar

Number of seconds to wait for a response from the HDL simulator, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

portnumber — Port number to connect

character vector | string scalar

Port number to connect, specified as a character vector or string scalar. The HDL simulator attempts to connect to a host on the specified port number.

Data Types: `char` | `string`

hostname — Name of host to connect

character vector | string scalar

Name of the host to connect, specified as a character vector or string scalar.

Data Types: `char` | `string`

Version History

Introduced in R2008a

See Also

`hdldaemon` | `vsim` | `breakHdlSim`

Topics

“Run a Simulink Cosimulation Session”

programFPGA

Package: hdlverifier

Load programming file associated with FILSimulation system object onto FPGA

Syntax

```
programFPGA(filobj)
```

Description

programFPGA(filobj) loads the FPGA through the JTAG cable, using information from the ProgrammingFile, ScanChainPosition, and BoardName properties of the input FILSimulation System object.

Input Arguments

filobj — Instance of FILSimulation

FILSimulation System object

Instance of FILSimulation, specified as a FILSimulation System object.

Version History

Introduced in R2010b

See Also

hdlverifier.FILSimulation

Topics

“FIL Simulation with HDL Workflow Advisor for MATLAB”

tclHdlSim

Execute Tcl command in Xcelium or ModelSim simulator

Syntax

```
tclHdlSim(tclCmd)
tclHdlSim(tclCmd,portNumber)
tclHdlSim(tclCmd,hostname)
```

Description

`tclHdlSim(tclCmd)` executes a Tcl command on the Xcelium or ModelSim simulator using a shared connection during a Simulink cosimulation session.

To use this function, the Xcelium or ModelSim simulator must be connected to MATLAB and Simulink using the HDL Verifier software (see either `vsimulink` or `hdlSimulink`).

To execute a Tcl command on the Xcelium or ModelSim simulator during a MATLAB cosimulation session, use the command `hdlDaemon('tclCmd','command')`.

`tclHdlSim(tclCmd,portNumber)` executes a Tcl command on the Xcelium or ModelSim simulator by connecting to the local host on port `portNumber`.

`tclHdlSim(tclCmd,hostname)` executes a Tcl command on the Xcelium or ModelSim simulator by connecting to the host `hostname`.

Examples

Display Message in HDL Simulator

Display a message in the HDL simulator command window using port 5678 on host name `msuser`.

```
>>tclHdlSim('puts "Done"', '5678', 'msuser')
```

Input Arguments

tclCmd — Tcl command to execute

character vector | string scalar

Tcl command to execute in the HDL simulator, specified as a character vector or string scalar. You can specify any valid Tcl command. The Tcl command you specify cannot include commands that load an HDL simulator project or modify the simulator state. For example, this value cannot include commands such as `start`, `stop`, or `restart` for ModelSim or `run`, `stop`, or `reset` for Xcelium.

Example: `'puts "Hello World!"'`

Data Types: `char` | `string`

portNumber — Port number

character vector | string scalar

TCP/IP port number for socket communication between the HDL-simulator and MATLAB, specified as a character vector or string scalar.

Example: '32864'

hostname — Host name

character vector | string scalar

TCP/IP host name for socket communication between the HDL-simulator and MATLAB, specified as a character vector or string scalar.

Version History

Introduced in R2008a

See Also

hdldaemon | nclaunch | vsim

External Websites

<https://www.tcl-lang.org/>

vsim

Start and configure ModelSim for use with HDL Verifier

Syntax

```
vsim
vsim(Name,Value)
```

Description

`vsim` starts and configures the ModelSim simulator for use with the MATLAB or Simulink cosimulation.

`vsim` creates a startup (or `.do`) file that adds these Tcl commands to ModelSim:

- `vsimmatlab`: link to MATLAB from ModelSim
- `vsimulink`: link to Simulink from ModelSim
- `vsimmatlabsysobj`: link to MATLAB System object from ModelSim

You can use these ModelSim Tcl commands instead of the ModelSim `vsim` command. These commands load instances of VHDL entities or Verilog modules for simulations that use MATLAB or Simulink for verification.

Tip When attempting to automate the cosimulation, use `pingHdlSim` to add a pause between the call to `vsim` and the call to run the simulation.

`vsim(Name,Value)` configures the ModelSim simulator using options specified by one or more name-value pair arguments.

Examples

Start and Configure ModelSim

Change the folder location to the ModelSim project folder, and then call the `vsim` function using the default executable. The function creates a temporary `.do` file in a temporary folder.

Specify the Tcl command `vsimmatlab` by using the `'tclstart'` name-value pair argument. Specify to load an instance of the VHDL entity `parse` in the library `work` for MATLAB verification.

Begin the test bench session for an instance of the entity `parse` by using the `matlabtb` command. Specify TCP/IP socket communication on port 4449 and a test bench timing value of 10 ns.

```
cd VHDLproj % Change folder to ModelSim project folder
vsim('tclstart','vsimmatlab work.parse; matlabtb parse 10 ns -socket 4449')
```

Change the folder location to the ModelSim project folder, and then call the `vsim` function. Specify the use of TCP/IP socket communication on the same computer for links between Simulink and ModelSim by using the `'socketsimulink'` name-value pair argument. Specify using socket port 4449.


```
cd VHDLproj % Change folder to ModelSim project folder
vsim('tclstart','vsimulink work.parse','socketsimulink','4449')
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `vsim('tclstart','vsimulink work.parse','socketsimulink','4449')` specifies executing the `vsimulink` command during startup and using port number 4449 for socket communication between ModelSim and Simulink.

libdir — Path to HDL Verifier HDL libraries

folder name

Path to the HDL Verifier HDL libraries, specified as the comma-separated pair consisting of `'libdir'` and a folder name. The folder contains the libraries that enable ModelSim to communicate with MATLAB when ModelSim runs on a machine that does not have MATLAB installed.

If this property is not specified, the function uses the default path in the MATLAB installation.

libfile — Library file built using compiler

library file name

Library file built using a compiler supplied with the HDL simulator, specified as the comma-separated pair consisting of `'libfile'` and the library file name. The default library file is the version built using the same compiler that MATLAB uses. If the HDL simulator links to other libraries (including SystemC libraries) that are built using a compiler supplied with the HDL simulator, you can specify the library file using this name-value pair argument. See “Cosimulation Libraries” for versions of the library built using other compilers.

Note Do not include the OS-specific library extension in the library file name.

rundir — Location to run HDL simulator

folder name

Location to run the HDL simulator, specified as the comma-separated pair consisting of `'rundir'` and a folder name.

If the value is `“TEMPDIR”`, the function creates a temporary directory to run ModelSim. By default, the function uses the current folder.

runmode — Run mode for HDL simulator

'GUI' (default) | 'Batch' | 'CLI'

Run mode for the HDL simulator, specified as the comma-separated pair consisting of `'runmode'` and one of the values in this table.

Value	Description
'GUI'	Start the HDL simulator with the ModelSim graphical user interface.
'CLI'	Start the HDL simulator in an interactive terminal window.
'Batch'	Start the HDL simulator in the background with no window (Linux) or in a noninteractive command window (Windows).

socketmatlabsobj – TCP/IP socket communication for links between ModelSim and MATLAB

`tcp_spec`

TCP/IP socket communication for links between ModelSim and MATLAB, specified as the comma-separated pair consisting of 'socketmatlabsobj' and a port number or service name. If you are setting up communication between computing systems, you must also specify the internet address or name of the remote host.

Note

- If ModelSim and MATLAB are running on the same computer, you can use shared memory for communication.
 - When this argument is not specified, the function uses shared memory communication. For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.
-

socketsimulink – TCP/IP socket communication for links between ModelSim and Simulink

`tcp_spec`

TCP/IP socket communication for links between ModelSim and Simulink, specified as the comma-separated pair consisting of 'socketsimulink' and a port number or service name. If you are setting up communication between computing systems, you must also specify the name or internet address of the remote host.

Note

- If ModelSim and MATLAB are running on the same computer, you can use shared memory for communication.
 - When this argument is not specified, the function uses shared memory communication. For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.
-

startms – Launch ModelSim from vsim

`yes (default) | no`

Specify `yes` to create a startup Tcl file and launch ModelSim from `vsim`. Specify `no` to create a startup Tcl file without launching ModelSim.

The startup Tcl file contains pointers to MATLAB libraries. To run ModelSim on a machine without MATLAB, copy the startup Tcl file and MATLAB library files to the remote machine and start ModelSim manually. See “Cosimulation Libraries”.

startupfile — Name and location of generated Tcl file

path name

Name and location of the generated Tcl file, specified as the comma-separated pair consisting of 'startupfile' and a path name. Each invocation of vsim creates a Tcl script that is applied during HDL simulator startup. By default, vsim generates the file name `compile_and_launch.tcl` in the folder specified by `rundir`. If the file name already exists, the file contents are overwritten. You can edit and use the generated file in a regular shell outside of MATLAB. For example:

```
sh> vsim -gui -do compile_and_launch.tcl
```

tclstart — Tcl commands to execute during ModelSim startup

tcl commands

Tcl commands to execute during ModelSim startup, specified as the comma-separated pair consisting of 'tclstart' and one of these values:

- vsimmatlab
- vsimulink
- vsimmatlabsysobj

The function appends these commands to the startup file.

vsimdir — Path to ModelSim executable folder

path name

Path to the ModelSim executable folder, specified as the comma-separated pair consisting of 'vsimdir' and a path name. By default, the function uses the first version of `vsim.exe` that it finds on the system path (defined by the `path` variable).

Specify this name-value pair argument if you want to start a different version of the ModelSim simulator, or if the version of the simulator you want to run is not on the system path.

Version History

Introduced in R2008a

See Also

`vsimmatlab` | `matlabtb` | `vsimulink`

uvmbuild

Generate UVM test bench from Simulink model

Syntax

```
uvmbuild(dut, sequence, scoreboard)
uvmbuild(___, Name, Value)
```

Description

`uvmbuild(dut, sequence, scoreboard)` generates a SystemVerilog top module, which includes a universal verification methodology (UVM) test bench and a behavioral design under test (DUT). The UVM test bench includes a sequence, a scoreboard, monitors, and drivers. The `uvmbuild` function maps:

- The Simulink DUT subsystem to a generated SystemVerilog DPI behavioral DUT
- The Simulink sequence subsystem to a UVM sequence block
- The Simulink scoreboard subsystem to a UVM scoreboard

`uvmbuild(___, Name, Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. For example, 'Driver', 'mySLTopModule/myDriver' generates a UVM driver from the Simulink subsystem specified as 'mySLTopModule/myDriver'.

Examples

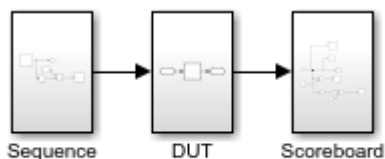
Use uvmbuild to generate UVM Test Bench

Simulink Model Structure

This example uses a Simulink® model, that includes these three subsystems.

- A sequence subsystem, which generates stimulus for the DUT.
- A DUT subsystem, which represents your HDL design.
- A scoreboard subsystem, which collects the outputs and checks them. In this example the DUT is a simple delay block.

```
open_system('hdlv_uvmbuild');
```



Copyright 2019 The Mathworks, Inc.

Generate UVM Test Bench

Generate a UVM test bench from this Simulink model, specifying the paths to the DUT, sequence, and scoreboard subsystems.

```
uvmbuild('hdlv_uvmbuild/DUT', 'hdlv_uvmbuild/Sequence', 'hdlv_uvmbuild/Scoreboard');
```

Observe Generated Output

The `uvmbuild` function creates a directory named `hdlv_uvmbuild_uvmbuild` containing the `uvm_testbench` directory. The `uvm_testbench` directory includes these subdirectories.

- The top directory includes a SystemVerilog top module and generated scripts to execute in your HDL simulation environment.
- The `DPI_dut` directory contains the SystemVerilog-DPI behavioral DUT.
- The `sequence` directory contains the generated sequence transaction type and a UVM sequencer, which drives the transaction to the DUT.
- The `scoreboard` directory contains the generated UVM scoreboard.
- The `uvm_artifacts` directory contains UVM components, such as monitors, drivers, and agents, required for the UVM environment.

Run Generated UVM Test Bench

- 1 Start Modelsim® or Questasim in GUI mode.
- 2 In the HDL simulator, navigate to the top directory: `cd hdlv_uvmbuild_uvmbuild \uvm_testbench\top\`
- 3 In the HDL simulator, enter this command to run your simulation: `do run_tb_mq.do`

Input Arguments

dut — Design under test subsystem

character vector | string scalar

Design under test subsystem, specified as a character vector or string scalar representing a DUT-subsystem name or full block path.

Example: `'hdlv_uvmbuild/DUT'`

Data Types: `char` | `string`

sequence — Sequence subsystem

character vector | string scalar

Sequence subsystem, specified as a character vector or string scalar representing a sequence-subsystem name or full block path.

Example: `'hdlv_uvmbuild/sequence'`

Data Types: `char` | `string`

scoreboard — Scoreboard subsystem

character vector | string scalar

Scoreboard subsystem, specified as a character vector or string scalar representing a scoreboard-subsystem name or full block path.

Example: 'hdlv_uvmbuild/scoreboard'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `uvmbuild('top/dut','top/seq','top/scr','Driver','top/drv','Monitor','top/mon')`

Driver – Driver subsystem

character vector | string scalar

Driver subsystem, specified as a character vector or string scalar representing a driver-subsystem name or full block path. By default, the `uvmbuild` function generates a passthrough UVM driver.

Example: 'hdlv_uvmbuild/driver'

Data Types: char | string

Monitor – Monitor subsystem

character vector | string scalar

Monitor subsystem, specified as a character vector or string scalar representing a monitor-subsystem name or full block path. By default, the `uvmbuild` function generates a passthrough UVM monitor.

Example: 'hdlv_uvmbuild/monitor'

Data Types: char | string

Predictor – Predictor subsystem

character vector | string scalar

Predictor subsystem, specified as a character vector or string scalar representing a predictor subsystem name or full block path.

Example: 'hdlv_uvmbuild/predictor'

Data Types: char | string

Config – UVM configuration parameters

`uvmcodegen.uvmconfig` object

UVM configuration parameters, specified as the comma-separated pair consisting of 'Config' and a `uvmcodegen.uvmconfig` configuration object. Use this value to configure the generated UVM test bench.

Data Types: char | string

SequenceFeedbackBlocks – Sequence Feedback block or blocks

character vector | string | cell array

Sequence Feedback block, specified as a character vector or string scalar representing a Sequence Feedback block name or full block path. For multiple Sequence Feedback blocks, use a cell array.

Example: 'hdlv_uvmbuild/feedback'

Example: {'hdlv_uvmbuild/feedback1', 'hdlv_uvmbuild/feedback2', 'hdlv_uvmbuild/feedback3'}

Data Types: char | string

Version History

Introduced in R2019b

R2023a: Support added for Sequence Feedback blocks

The `uvmbuild` function supports generation of a UVM test bench for models that include a Sequence Feedback block. Use the `SequenceFeedbackBlocks` name-value argument to specify Sequence Feedback blocks in your model.

See Also

`dpigen` | `uvmcodegen.uvmconfig` | Sequence Feedback

Topics

"UVM Component Generation Overview"

vsimmatlab

Load instantiated HDL module for verification with ModelSim and MATLAB

Syntax

```
vsimmatlab instance  
vsimmatlab instance <vsim_args>
```

Description

Note Use this command in ModelSim, not in MATLAB.

`vsimmatlab instance` loads the specified instance of an HDL module for verification and sets up ModelSim so it can establish a communication link with MATLAB. ModelSim opens a simulation workspace and displays a series of messages in the command window as it loads the packages and architectures of the HDL design.

You can run this command from the HDL simulator prompt or from a Tcl script shell (`tclsh`).

`vsimmatlab instance <vsim_args>` uses additional `vsim` command line arguments.

Examples

Load Instantiated HDL Module for Cosimulation with MATLAB

In ModelSim, load the HDL module instance `parse` from the library `work`, establishing communication with MATLAB.

```
ModelSim> vsimmatlab work.parse
```

Input Arguments

instance — Instance of HDL module to load for verification

HDL instance name (as required by ModelSim)

Instance of the HDL module to load for verification, specified as HDL instance name (as required by ModelSim).

Example: `work.parse`

Data Types: `char` | `string`

vsim_args — `vsim` command arguments (as required by ModelSim)

`vsim` command arguments

`vsim` command arguments (as required by ModelSim). For details, see the description of `vsim` in the ModelSim documentation.

Version History

Introduced in R2008a

See Also

`vsim` | `vsimmatlabsysobj`

Topics

“Verify HDL Module with MATLAB Test Bench”

vsimmatlabsysobj

Load instantiated HDL module for cosimulation with ModelSim and MATLAB System object

Syntax

```
vsimmatlabsysobj instance
vsimmatlabsysobj instance <vsim_args>
vsimmatlabsysobj instance -socket tcp_spec
```

Description

Note Use this command in ModelSim, not in MATLAB.

`vsimmatlabsysobj instance` loads the specified instance of an HDL design for cosimulation and sets up ModelSim to establish a shared communication link with a MATLAB System object. ModelSim opens a simulation workspace and displays a series of messages in the command window as it loads the HDL module packages and architectures.

To generate the `vsimmatlabsysobj` function, you must first execute the `vsim` function in MATLAB.

`vsimmatlabsysobj instance <vsim_args>` uses additional `vsim` command line arguments.

`vsimmatlabsysobj instance -socket tcp_spec` establishes a communication link with a MATLAB System object over a Transmission Control Protocol (TCP) socket.

Examples

Load Instantiated HDL Model for Cosimulation with MATLAB System object

In ModelSim, load the HDL module instance `parse` from the library `work`, establishing communication with the MATLAB cosimulation System object.

```
ModelSim> vsimmatlabsysobj work.parse
```

Input Arguments

instance — Instance of HDL module to load for cosimulation

HDL instance name, as required by ModelSim

Instance of the HDL module to load for cosimulation.

vsim_args — vsim command arguments

`vsim` command arguments

`vsim` command arguments, as required by ModelSim. For details, see the description of `vsim` in the ModelSim documentation.

tcp_spec — TCP/IP socket communication

TCP/IP port number | TCP/IP service name | Internet address

TCP/IP socket communication for the link between ModelSim and MATLAB, specified as a TCP/IP port name or service name. If the MATLAB server is running on a remote host, you must also specify the name or internet address of the remote host. When this input argument is not specified, the function uses shared memory communication. This setting overrides the setting specified with the MATLAB `vsim` function.

Version History**Introduced in R2012b****See Also**`vsim` | `vsimmatlab`

vsimulink

Load instantiated HDL module for cosimulation with ModelSim and Simulink

Syntax

```
vsimulink instance -socket tcp_spec <vsim_args>
```

Description

Note Issue this command in ModelSim, not in MATLAB.

`vsimulink instance -socket tcp_spec <vsim_args>` loads the specified instance of the HDL design for cosimulation and sets up ModelSim so it can establish a shared communication link with Simulink. ModelSim opens a simulation workspace and displays a series of messages in the command window as it loads the HDL module packages and architectures.

To generate the `vsimulink` function, you must first invoke the `vsim` function in MATLAB.

Examples

Load Instantiated HDL Model for Cosimulation with Simulink

In ModelSim, load the HDL module instance `parse` from the library `work`, and establish communication with Simulink.

```
ModelSim> vsimulink work.parse
```

Input Arguments

instance — Instance of HDL module to load for cosimulation

HDL instance name, as required by ModelSim

Instance of the HDL module to load for cosimulation.

vsim_args — vsim command arguments

`vsim` command arguments

`vsim` command arguments, as required by ModelSim. For details, see the description of `vsim` in the ModelSim documentation.

tcp_spec — TCP/IP socket communication

TCP/IP port number | TCP/IP service name | internet address

TCP/IP socket communication for the link between ModelSim and Simulink, specified as a TCP/IP port name or service name. If the MATLAB server is running on a remote host, you must also specify the name or internet address of the remote host. When this input argument is not specified, the function uses shared memory communication. This setting overrides the setting specified with the MATLAB `vsim` function.

Version History

Introduced in R2008a

See Also

vsim

waitForHdlClient

Wait until specified event ID is obtained or time-out occurs

Syntax

```
pID = waitForHdlClient(timeout,eventID)
pID = waitForHdlClient(timeout)
pID = waitForHdlClient
```

Description

`pID = waitForHdlClient(timeout,eventID)` waits for the expected HDL simulator `eventID` to arrive at the MATLAB server before processing continues. If the expected `eventID` arrives before the number of seconds specified by `timeOut` the value returned by the HDL simulator is the HDL simulator process ID (PID).

`pID = waitForHdlClient(timeout)` waits for `eventID = 1` for `timeOut` seconds.

`pID = waitForHdlClient` waits for `eventID = 1` for 60 seconds.

Examples

Wait Until Specified Event ID Is Obtained or Time-Out Occurs

Wait for event ID 2 for 120 seconds.

```
>> ID = waitForHdlClient(120,2);
```

Input Arguments

timeout — Number of seconds to wait for response

positive scalar

Number of seconds to wait for a response from the HDL simulator, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

eventID — Event ID expected at MATLAB server

scalar | vector

Event ID expected at the MATLAB server, specified as a scalar or vector. `eventID` must be a positive number less than the maximum value of a 32-bit signed integer. The value must match the event ID sent by the `notifyMatlabServer` command in the HDL simulator.

When specified as a vector the function returns a value when all the elements of the vector have been collected or a time-out occurs. The returned output value is the same size as `eventID`, and each element of the output variable is the detected `pID` of the HDL simulator that corresponds to the event ID.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

pID — Process ID of HDL simulator

scalar | vector

Process ID of the HDL simulator, returned as a scalar or a vector. If a time-out occurs, the pID is returned as -1. The output value depends on the value of eventID.

eventID	pID
scalar	The function returns a scalar representing the detected PID of the HDL simulator.
vector	The function returns a vector the same size as eventID. Each element in the output vector is the detected PID of the HDL simulator. The output is returned only if all elements of the vector are collected or if a time-out occurs.

Version History

Introduced in R2012b

See Also

hdldaemon | notifyMatlabServer

Apps

HDL Verifier

Generate SystemVerilog DPI component from a Simulink subsystem

Description

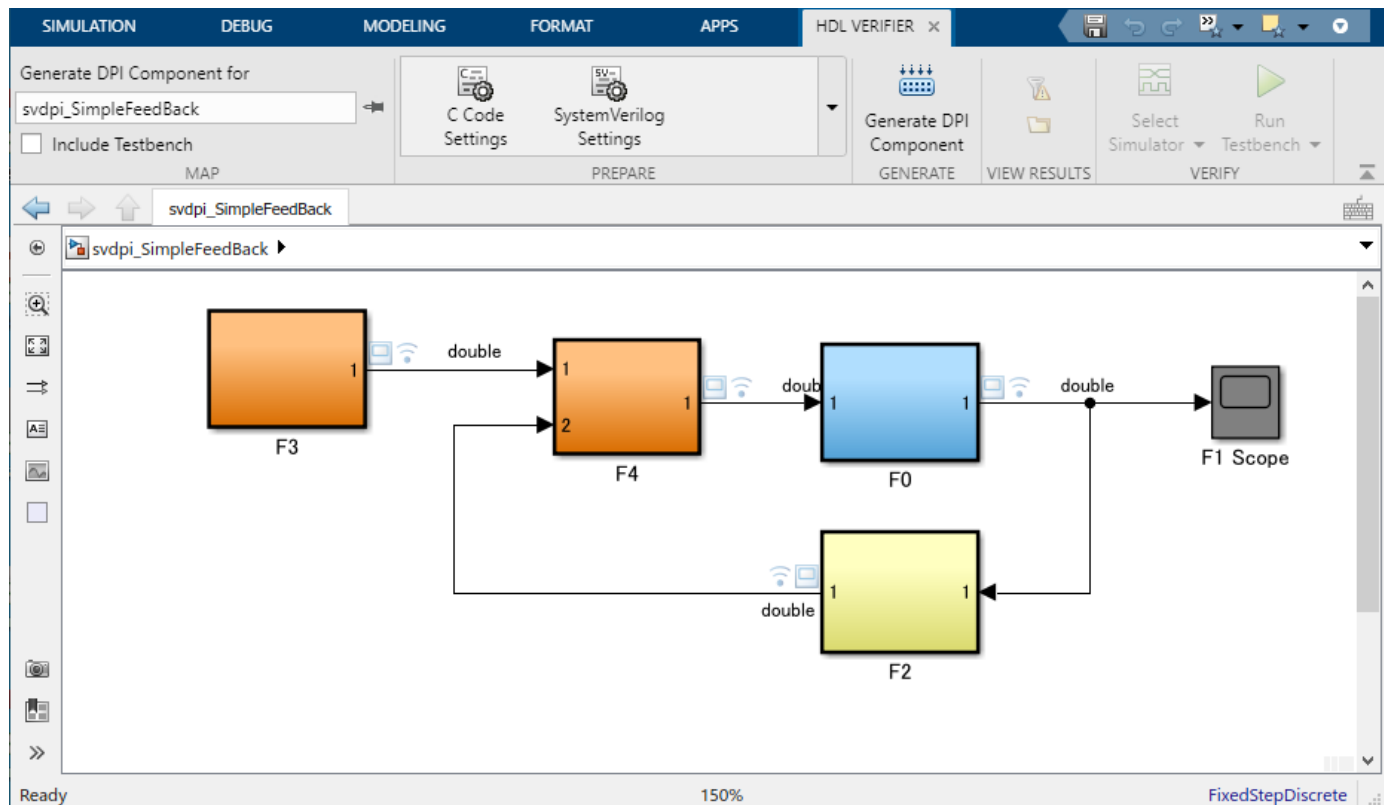
The **HDL Verifier** app enables you to generate a SystemVerilog DPI component from a Simulink subsystem.

The app toolstrip contains five sections that are representative of the HDL Verifier workflow.

- **Map**
- **Prepare**
- **Generate**
- **View results**
- **Verify**

You can use the app to perform these tasks.

- Generate a test bench for your generated DPI component by selecting **Include Testbench** in the **Map** section.
- Set code generation objectives and prepare your model for SystemVerilog DPI code generation by clicking **C Code Settings** or **SystemVerilog Settings** in the **Prepare** section.
- Generate the component by selecting **Generate DPI Component** in the **Generate** section.
- Set up your HDL simulation environment and simulate the generated component by using the options in the **Verify** section.



Open the HDL Verifier App

Simulink Toolstrip: On the **Apps** tab, under **Code verification, validation, and test**, click **HDL Verifier**. The **HDL Verifier** app opens in its own tab on the Simulink Toolstrip.

Examples

- “Get Started with SystemVerilog DPI Component Generation”
- “DPI Component Generation with Simulink”

Version History

Introduced in R2020b

See Also

Apps

Functions

Topics

“Get Started with SystemVerilog DPI Component Generation”
 “DPI Component Generation with Simulink”

“DPI Component Generation with Simulink”

Logic Analyzer

Visualize, measure, and analyze transitions and states over time

Description

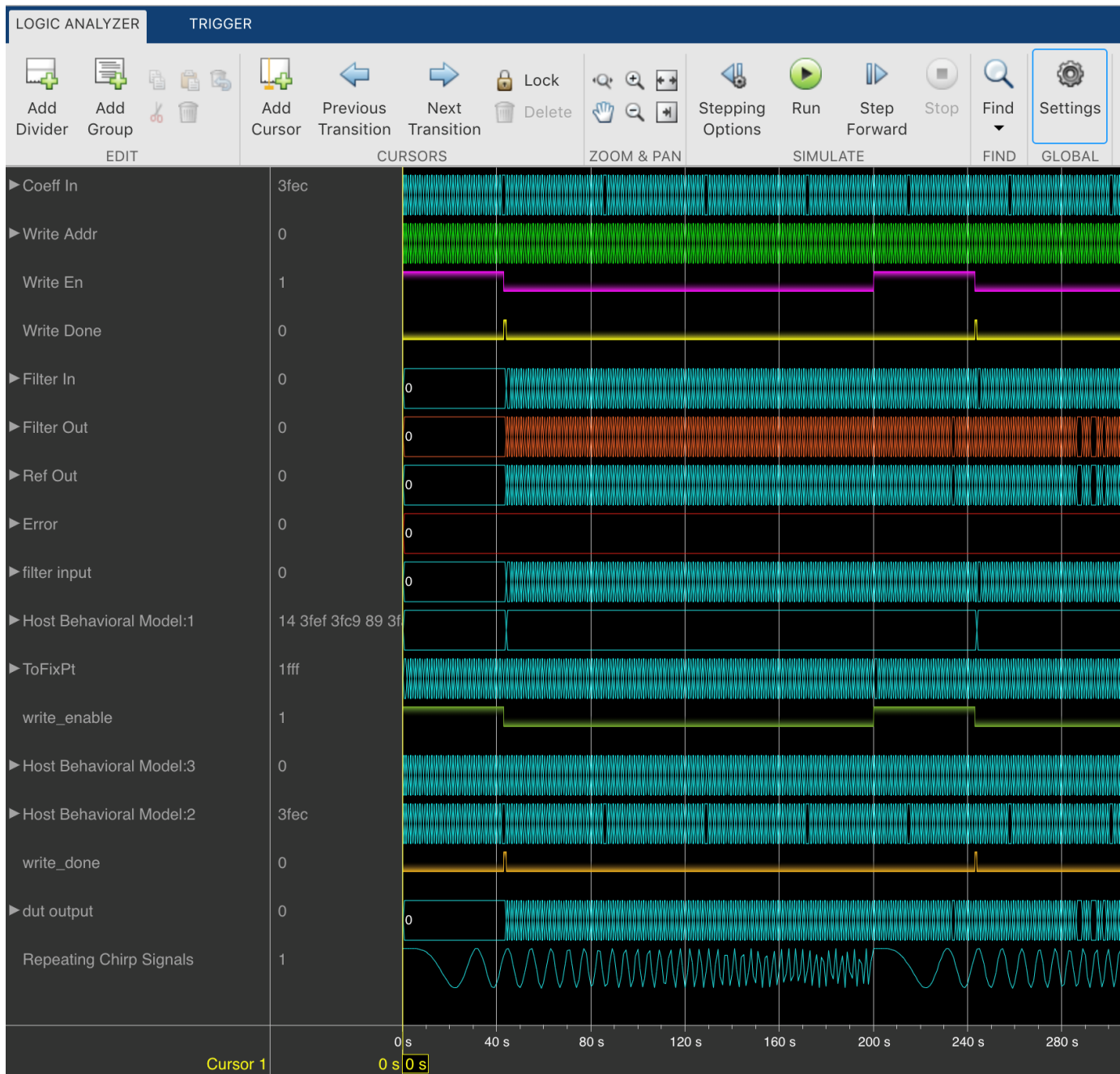
The **Logic Analyzer** is a tool for visualizing and inspecting signals and states in your Simulink model. Using the **Logic Analyzer**, you can:

- Debug and analyze models
- Trace and correlate many signals simultaneously
- Detect and analyze timing violations
- Trace system execution
- Detect signal changes using triggers

For keyboard shortcuts, click [More](#).

Keyboard Shortcuts

Actions	Description	Applicable When
Ctrl+X	Cut	Wave is selected
Ctrl+C	Copy	Wave is selected
Ctrl+V	Paste	Wave is selected
Delete	Delete	Wave is selected
Ctrl+-	Zoom out	Always
Shift+Ctrl+-	Zoom out around active cursor	Always
Ctrl++	Zoom in	Always
Shift+Ctrl++	Zoom in around active cursor	Always
Shift+Ctrl+C	Move display to active cursor	When cursor is not in the display range
Space	Zoom out full	Always
Tab, Right Arrow	Next transition	Digital format wave is selected
Shift+Tab, Left Arrow	Previous transition	Digital format wave is selected
Ctrl+A	Select all waves	Always
Up Arrow	Select wave above selected	Wave is selected
Down Arrow	Select wave below selection	Wave is selected
Ctrl+Up Arrow	Move selected waves up	Wave is selected
Ctrl+Down Arrow	Move selected waves down	Wave is selected
Escape	Unselect all signals	Wave is selected
Page Up	Scroll up	Always
Page Down	Scroll down	Always



Open the Logic Analyzer App

On the Simulink toolstrip Simulation tab, click the **Logic Analyzer** app button. If the button is not displayed, expand the review results app gallery. Your most recent choice for data visualization is saved across Simulink sessions.

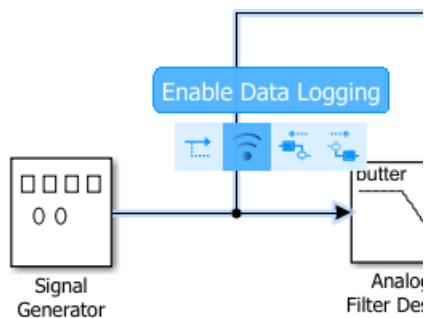
To visualize referenced models, you must open the Logic Analyzer from the referenced model. You should see the name of the referenced model in the Logic Analyzer toolbar.

Examples

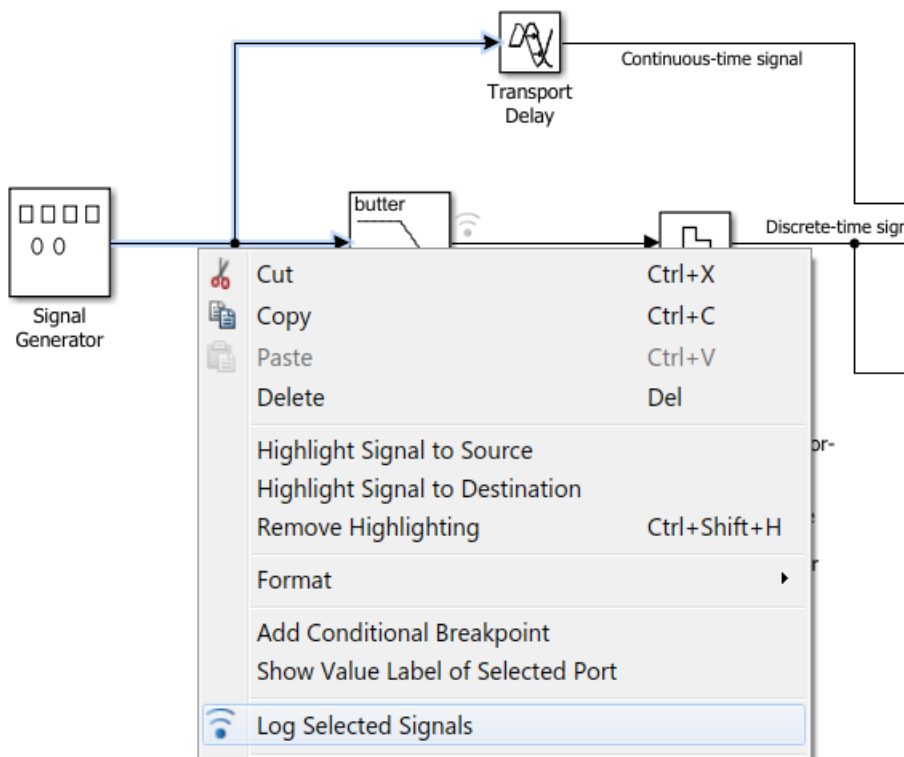
Select Signals to Analyze

The **Logic Analyzer** supports several methods for selecting data to visualize.

- Select a signal in your model. When you select a signal, an ellipsis appears above the signal line. Hover over the ellipsis to view options and then select the **Enable Data Logging** option.



- Right-click a signal in your model to open an options dialog box. Select the **Log Selected Signals** option.



- Use any method to select multiple signal lines in your model. For example, use **Shift**+click to select multiple lines individually or **CTRL+A** to select all lines at once. Then, on the **Signal** tab, select the **Log Signals** button.

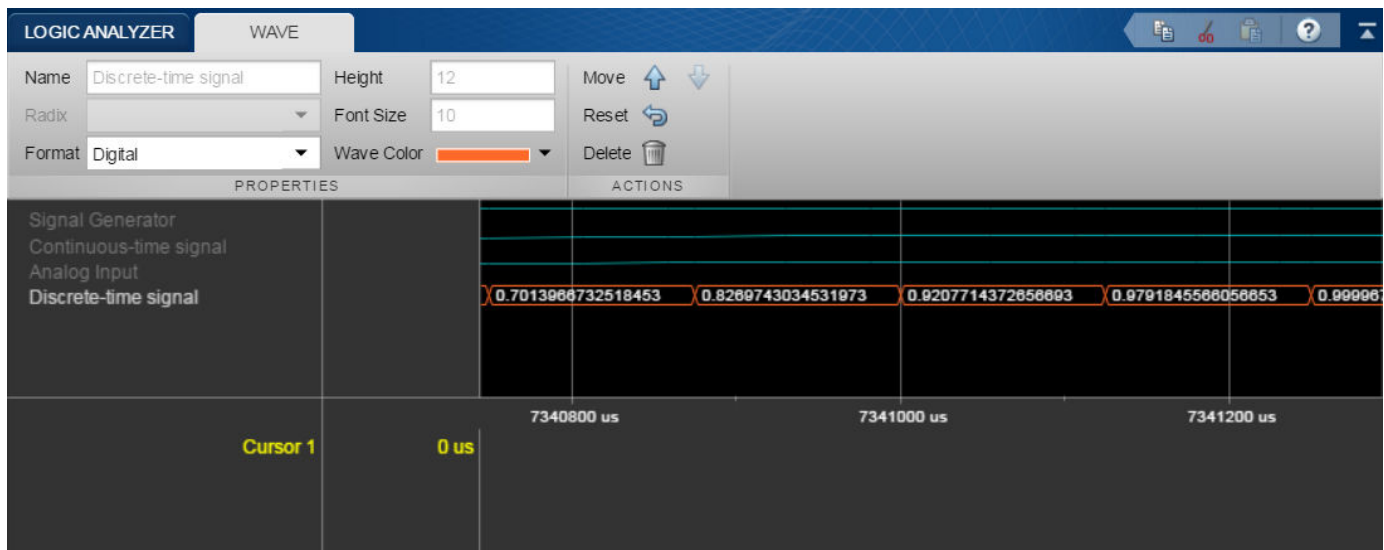


To visualize data in the Logic Analyzer, you must enable signal logging for the model. (Logging is on by default.) To enable signal logging, open **Model Settings** from the toolstrip, navigate to the **Data Import/Export** pane, and select **Signal logging**.

When you open the **Logic Analyzer**, all signals marked for logging are listed. You can add and delete waves from your **Logic Analyzer** while it is open. Adding and deleting signals does not disable logging, only removes the signal from the Logic Analyzer.

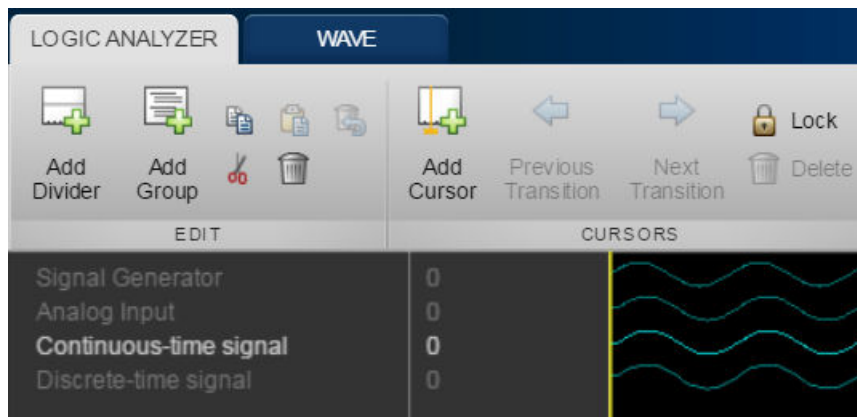
Modify Individual Wave Settings



Open the **Logic Analyzer** and select a wave by double-clicking the wave name. Then from the **Wave** tab, set parameters specific to the individual wave you selected. Any setting made on individual signals supersedes the global setting. To return individual wave parameters to the global settings, click **Reset**.



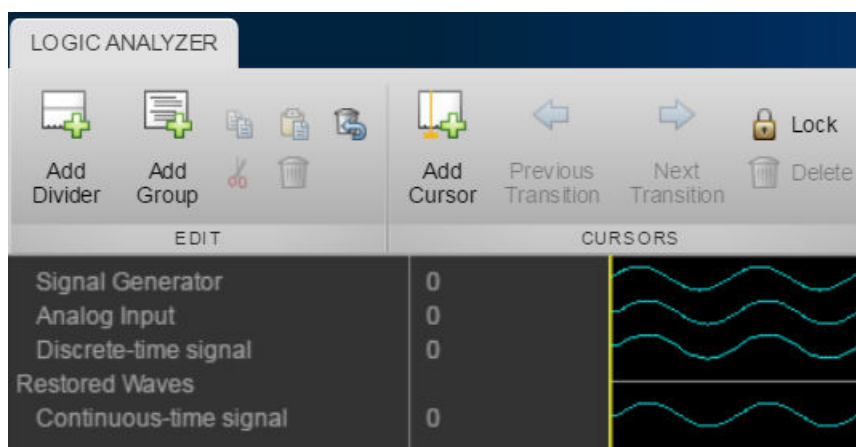
Delete and Restore Waves

- 1 Open the **Logic Analyzer** and select a wave by clicking the wave name.



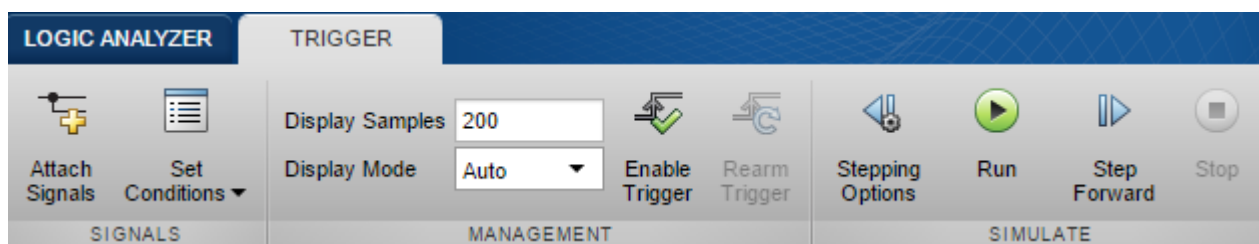
- 2 From the **Logic Analyzer** toolbar, click . The wave is removed from the **Logic Analyzer**.
- 3 To restore the wave, from the **Logic Analyzer** toolbar, click .

A divider named **Restored Waves** is added to the bottom of your channels, with all deleted waves placed below it.



Add Trigger

- 1 Open the **Logic Analyzer** and select the **Trigger** tab.



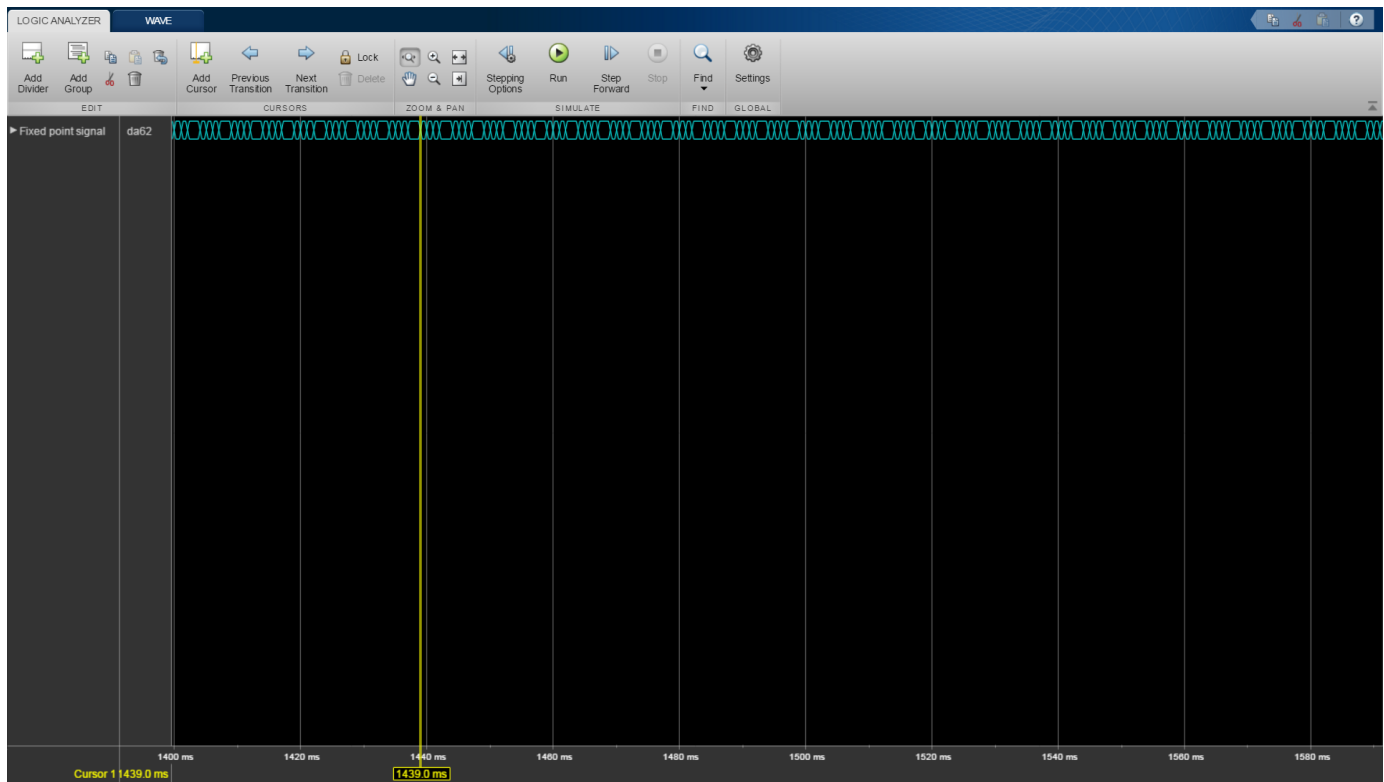
- 2 To attach a signal to the trigger, select **Attach Signals**, then select the signal you want to trigger on. You can attach up to 20 signals to the trigger. Each signal can have only one triggering condition.
- 3 By default, the trigger looks for rising edges in the attached signals. You can set the trigger to look for rising or falling edges, bit sequences, or a comparison value. To change the triggering conditions, select **Set Conditions**.

If you add multiple signals to the trigger, control the trigger logic using the **Operator** option:

- AND - match all conditions.
 - OR - match any condition.
- 4 To control how many samples you see before triggering, set the **Display Samples** option. For example, if you set this option to 500, the **Logic Analyzer** tries to give you 500 samples before the trigger. Depending on the simulation, the **Logic Analyzer** may show more or fewer than 500 samples before the trigger. However, if the trigger is found before the 500th sample, the Logic Analyzer still shows the trigger.
 - 5 Control the trigger mode using **Display Mode**.
 - Once - The **Logic Analyzer** marks only the first location matching the trigger conditions and stops showing updates to the Logic Analyzer. If you want to reset the trigger, select **Rearm Trigger**. Relative to the current simulation time, the **Logic Analyzer** shows the next matching trigger event.
 - Auto - The **Logic Analyzer** marks every location matching the trigger conditions.
 - 6 Before running the simulation, select **Enable Trigger**. A blue cursor appears as time 0. Then, run the simulation. When a trigger is found, the **Logic Analyzer** marks the location with a locked blue cursor.

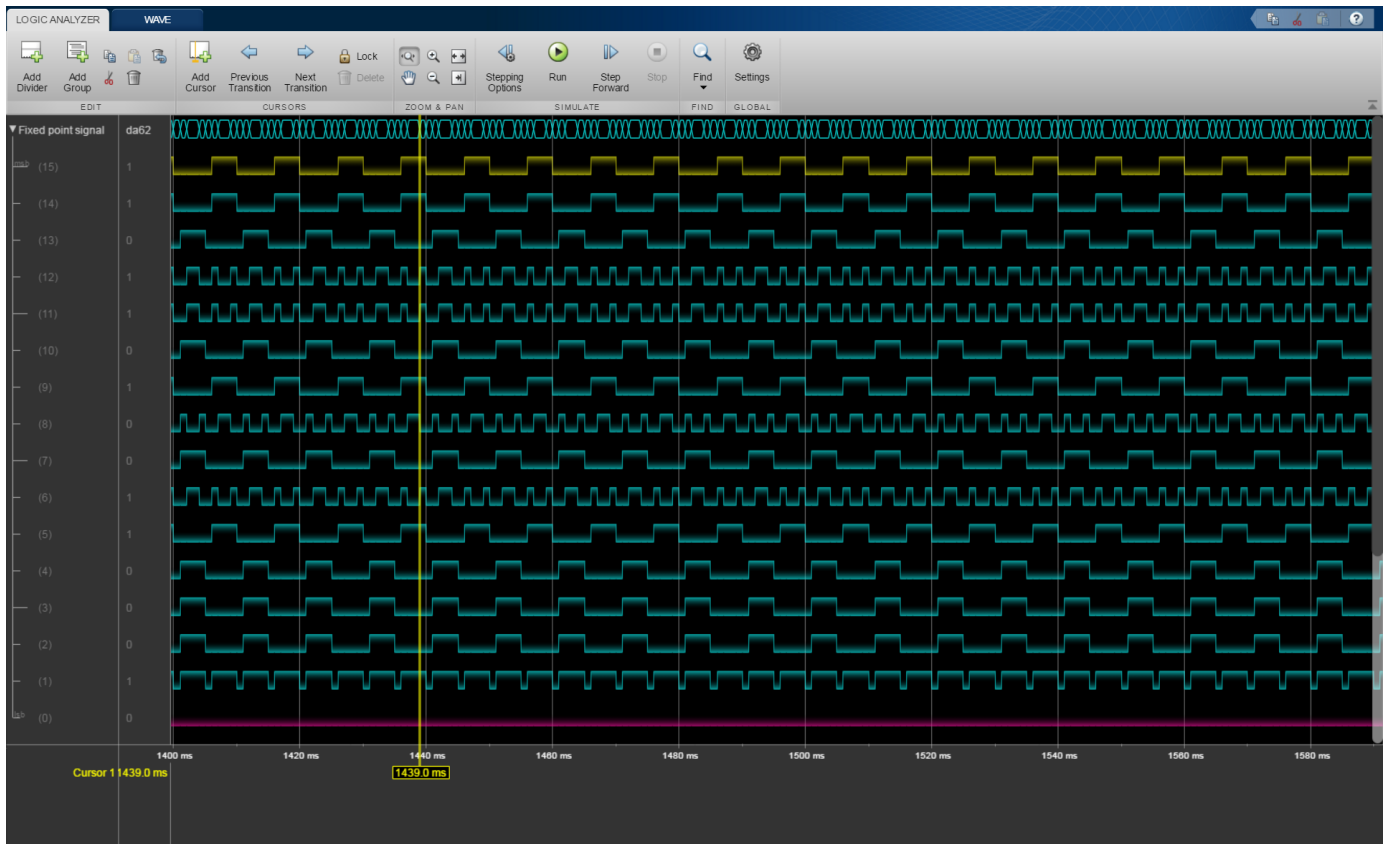
View Bit-Expanded Wave and Reverse Display Order of Bits

The **Logic Analyzer** enables you to bit-expand fixed-point and integer waves.

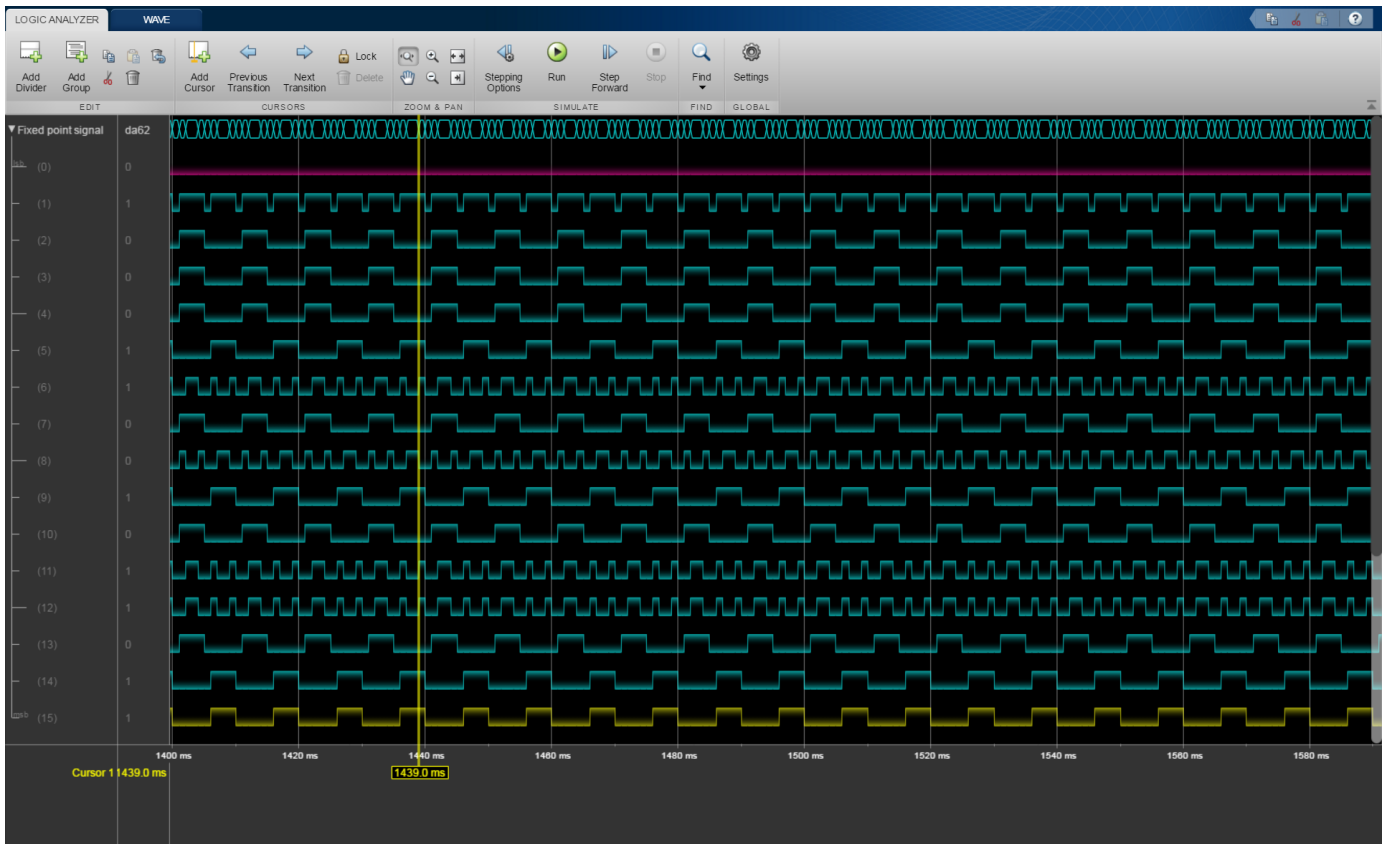


- 1 In the **Logic Analyzer**, click the arrow next to a fixed-point or integer wave to view the bits.

The least significant bit and the most significant bit are marked with **lsb** and **msb** next to the wave names.



- 2 Click Settings, and then select **Display Least Significant bit first** to reverse the order of the displayed bits.



- “Configure Logic Analyzer”
- “Programmable FIR Filter for FPGA” (HDL Coder)
- “Log Simulation Output for States and Data” (Stateflow)
- “View Stateflow States in the Logic Analyzer” (Stateflow)

Limitations

Logging Settings

- If you enable the configuration parameter **Log Dataset data to file**, you cannot stream logged data to the **Logic Analyzer**.
- Signals marked for logging using `Simulink.sdi.markSignalForStreaming` or visualized with a Dashboard Scope do not appear on the **Logic Analyzer**.
- You cannot visualize Data Store Memory block signals in the **Logic Analyzer** if you set the **Log data store data** parameter to on.

Input Signal Limitations

- Signals marked for logging for the **Logic Analyzer** must have fewer than 8000 samples per simulation step.
- The **Logic Analyzer** does not support frame-based processing.

- For 64-bit integers and fixed-point numbers greater than 53 bits, if the numbers are greater than the maximum value of double precision, the transitions between numbers might not display correctly.
- You may see performance degradation in the **Logic Analyzer** for large matrices (greater than 500 elements) and buses with more than 1000 signals.
- The **Logic Analyzer** does not support Stateflow data output.

Graphical Settings

- While the simulation is running, you cannot zoom, pan, or modify the trigger.
- To visualize constant signals, in the settings, you must set the **Format** to **Digital**. Constants marked for logging are visualized as a continuous transition.

Supported Simulation Modes

Mode	Supported	Notes and Limitations
Normal	Yes	
Accelerator	Yes	You cannot use the Logic Analyzer to visualize signals in Model blocks with Simulation mode set to Accelerator.
Rapid Accelerator	Yes	Data is not available in the Logic Analyzer during simulation. If you simulate a model with the simulation mode set to rapid accelerator, after simulation the following signals cannot be visualized in the Logic Analyzer : <ul style="list-style-type: none"> • Multi-instance model reference signals • Nonvirtual bus signals
Processor-in-the-loop (PIL)	No	
Software-in-the-loop (SIL)	No	
External	No	

For more information about these modes, see “How Acceleration Modes Work” (Simulink).

Version History

Introduced in R2016b

Objects

Topics

- “Configure Logic Analyzer”
- “Programmable FIR Filter for FPGA” (HDL Coder)
- “Log Simulation Output for States and Data” (Stateflow)
- “View Stateflow States in the Logic Analyzer” (Stateflow)

