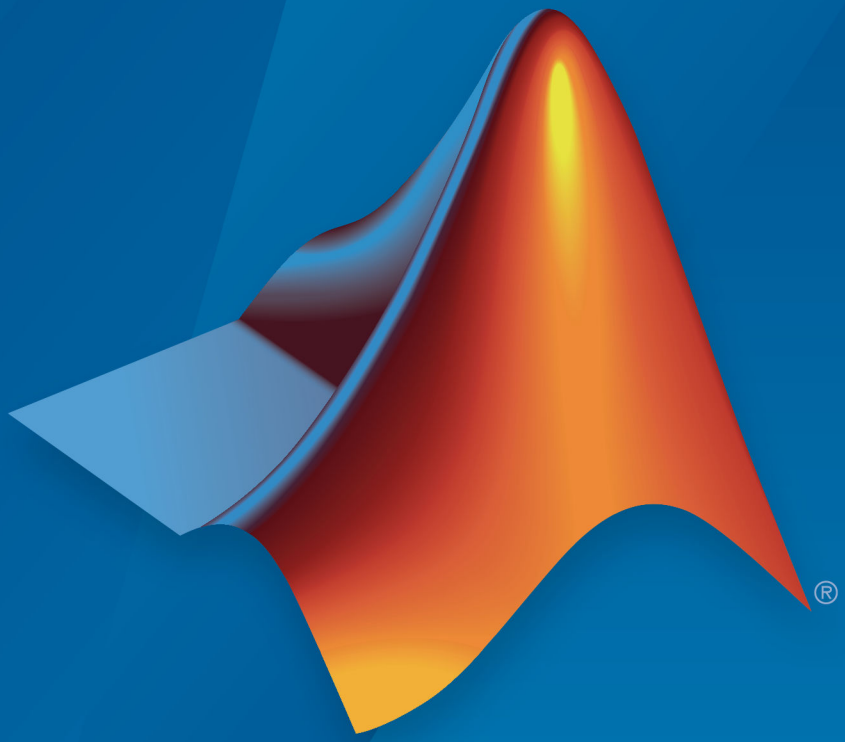
HDL Verifier™

Reference



MATLAB® & SIMULINK®

R2019b

MathWorks®

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

Revision History

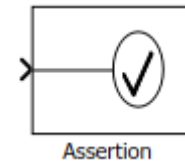
August 2003	Online only	New for Version 1 (Release 13SP1)
February 2004	Online only	Revised for Version 1.1 (Release 13SP1)
June 2004	Online only	Revised for Version 1.1.1 (Release 14)
October 2004	Online only	Revised for Version 1.2 (Release 14SP1)
December 2004	Online only	Revised for Version 1.3 (Release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

Contents

Blocks — Alphabetical List

Assertion

Generate SystemVerilog assertions from Simulink assertion
Library: 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`.

See Also

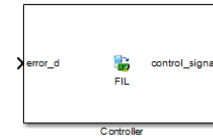
Topics

“Generate SystemVerilog Assertions from Simulink Test Bench”

Introduced in R2018a

FIL Simulation

Simulate HDL code on FPGA hardware from Simulink
Library: 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.

See Also

Topics

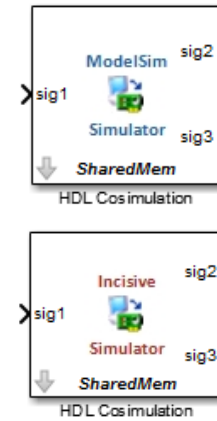
“FPGA-in-the-Loop Simulation”
“FPGA-in-the-Loop Simulation Workflows”
“FIL Simulation with HDL Workflow Advisor for Simulink”
“Block Generation with the FIL Wizard”

Introduced in R2012b

HDL Cosimulation

Cosimulate HDL design by connecting Simulink with HDL simulator

Library: HDL Verifier / For Use with Cadence Incisive
HDL Verifier / For Use with Mentor Graphics ModelSim



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

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.
- 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.

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. You can add and remove ports, and configure their data types and sample times, by changing the block parameters. The **Ports** tab displays the HDL signals that correspond to the ports. You can add, remove, and change the order of the ports. 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. See “Get Signal Information from HDL Simulator” for a detailed description of this feature.

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.

When you import VHDL® signals from the HDL simulator, HDL Verifier returns the signal names in all capitals.

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

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 Incisive®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.

`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.

`Sample Time` — Time between reading samples on an output port
1 (default) | integer

Time interval between consecutive samples applied to an output port.

Simulink deposits an input port signal on an HDL simulator signal at the specified sample rate. Conversely, Simulink reads an output port signal from a specified HDL simulator signal at the specified sample rate.

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.

- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. You must specify an explicit sample time for each output port.

The exact interpretation of the output port sample time depends on the settings of the **Timescales** parameters of the HDL Cosimulation block. See also “Simulation Timescales”.

Dependencies

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

`Data Type` — Data type for output signal

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

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.

Dependencies

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

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

`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

Clocks

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”.

`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/clk` or `manchester.clk`.

For information about and requirements for path specifications 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.

`Active Clock Edge` — HDL clock edge used to sample signals
`Rising` (default) | `Falling`

Select `Rising` or `Falling` to specify either a rising-edge clock or a falling-edge clock.

`Period` — 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).

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-defined tick interval)
- *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”.

`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. 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

`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

`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 Incisive).

`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 Incisive).

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.

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 or Cadence Incisive®.

Each of the HDL Cosimulation blocks cosimulates a hardware component by applying input signals to, and reading output signals from, an HDL model that executes under an HDL simulator. See “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, to check the requirements for using the HDL Cosimulation block for code generation, select **Simulation > Update Diagram**.

HDL Architecture

This block has a single, 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

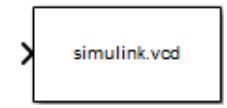
“Import HDL Code for HDL Cosimulation Block”
“Create Simulink Model for Component Cosimulation”
“Create a Simulink Cosimulation Test Bench”
“Run a Simulink Cosimulation Session”
“Simulation Timescales”
“Clock, Reset, and Enable Signals”

Introduced in R2008a

To VCD File

Generate value change dump (VCD) file

Library: HDL Verifier / For Use with Cadence Incisive
HDL Verifier / For Use with Mentor Graphics ModelSim



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 postsimulation 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.

`$version HDL` `Verifier version 1.0 $ end`	Version of the To VCD File block that generated the file.
`$timescale 1 ns $ end`	Timescale used during the simulation.
`$scope module manchestermodel $end`	Scope of module being dumped.

`$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`
	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.

`$upscope $end`	Marks a change to the next highest level in the HDL design hierarchy.
`$enddefinitions $end`	Marks the end of the header and definitions section.
`#0`	Simulation start time.

VCD File Content	Description

`$dumpvars`
` 0!`
` 0"`
` 0#`
` 0$`
`$end`
	Lists the values of all defined variables at time 0.

`#630` ` 1!`	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 `Original Data` changed from 0 to 1.

`.`
`.`
`.`
`#1160`
` 1#`
` 1$`
	At 116 nanoseconds, the values of signals `Recovered Data` and `Data Validity` changed from 0 to 1.

`$dumpoff`
` x!`
` x"`
` x#`
` x$`
`$end`
	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.

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”

Introduced in R2008a

System Objects — Alphabetical List

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? (MATLAB).

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 (MATLAB).

`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 much 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 use a MATLAB System object and a 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 ISE or Quartus II to the system path for the current MATLAB session.

For Xilinx FPGA boards, run

```
>>hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\13.1\ISE_DS\ISE\bi
```

This example assumes that the Xilinx ISE executable is C:\Xilinx\13.1\ISE_DS\ISE\bin \nt64\ise.exe. Substitute with your actual executable if it is different.

For Altera boards, run

```
>>hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath','C:\altera\11.0\quartus\b
```

This example assumes that the Altera Quartus II executable is C:\altera\11.0\quartus\bin \quartus.exe. Substitute with your actual executable if it is different.

Copy FFT HDL Files

Copy the HDL files for the FFT example into your local directory

```
copyFILDemoFiles('fft');
```

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');
```

```
### Generating iMPACT command file
### Checking iMPACT tool
### Start loading bitstream "S:\MATLAB\demo\fft8_fil\fft8_fil.bit"
### Loading bitstream "S:\MATLAB\demo\fft8_fil\fft8_fil.bit" completed successfully
```

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
```

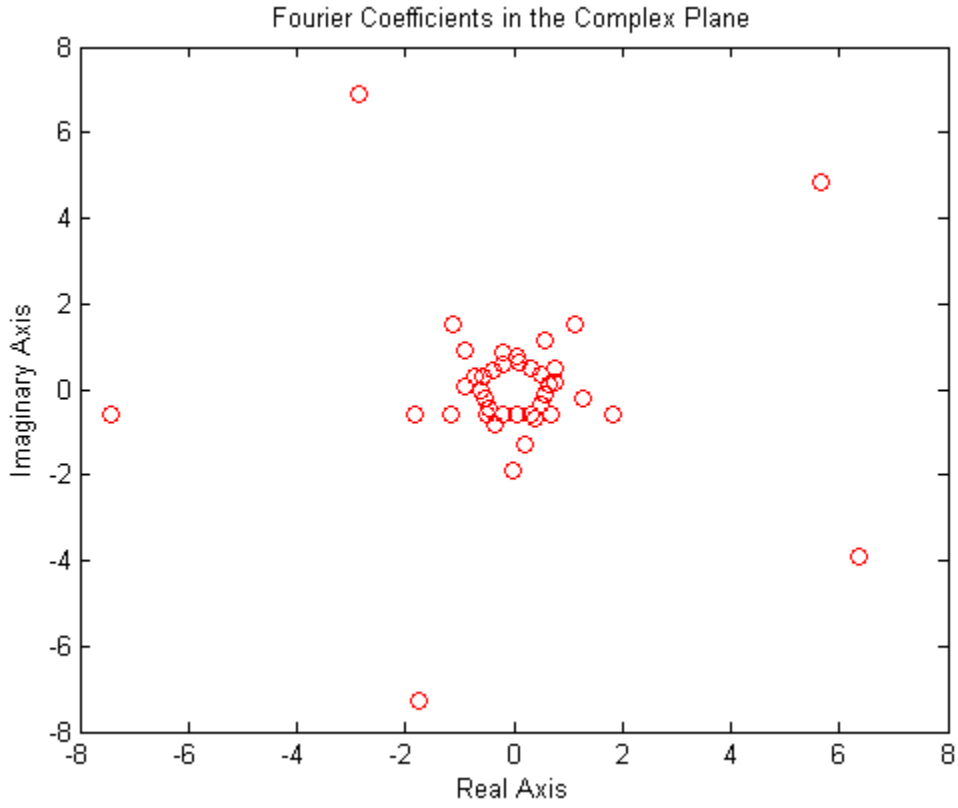
Display the Fourier Coefficients

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.
```



See Also

FIL Simulation | FPGA-in-the-Loop Wizard

Topics

“FPGA-in-the-Loop Simulation Workflows”

Introduced in R2012b

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? (MATLAB).

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', ... ,
'OutputFractionLangths',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 (MATLAB).

`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 postsimulation 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 Mentor Graphics® ModelSim®/QuestaSim® or Cadence® Incisive®/Xcelium® to cosimulate a Viterbi decoder implemented in VHDL.

Set Simulation Parameters and Instantiate Communication System Objects

If you are using Incisive/Xcelium, set simulator variable to 'Incisive'

```
Simulator = 'Incisive';

% or if you are using ModelSim/QuestaSim, set simulator variable to
% 'ModelSim'
Simulator = 'ModelSim';

% The following code sets up the simulation parameters and instantiates the
% system objects that represent the channel encoder, BPSK modulator, AWGN
% channel, BPSK demodulator, and error rate calculator. Those objects
% comprise the system around the Viterbi decoder and can be thought of as
% the test bed for the Viterbi HDL implementation.

EsNo = 0;    % Energy per symbol to noise power spectrum density ratio in dB
FrameSize = 1024;  % Number of bits in each frame

% Convolution Encoder
hConEnc = comm.ConvolutionalEncoder;
% BPSK Modulator
hMod    = comm.BPSKModulator;
% AWGN channel
hChan   = comm.AWGNChannel('NoiseMethod', ...
                           'Signal to noise ratio (Es/No)',...
                           'SamplesPerSymbol',1,...
```

```
                              'EsNo',EsNo);
% BPSK demodulator
hDemod  = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio',...
                               'Variance',0.5*10^(-EsNo/10));
% Error Rate Calculator
hError  = comm.ErrorRate('ComputationDelay',100,'ReceiveDelay', 58);
```

Instantiate the Cosimulation System Object

The hdlcosim function returns an HDL cosimulation System object, which represents the HDL implementation of the Viterbi decoder in this simulation system.

```
switch Simulator
    case 'ModelSim'
        hDec    = hdlcosim('InputSignals', {'/viterbi_block/In1','/viterbi_block/In2'},
                           'OutputSignals', {'/viterbi_block/Out1'}, ...
                           'OutputSigned', false, ...
                           'OutputFractionLengths', 0, ...
                           'TCLPreSimulationCommand', 'force /viterbi_block/clk_enable
                           'TCLPostSimulationCommand', 'echo "done";', ...
                           'PreRunTime', {10,'ns'}, ...
                           'Connection', {'Shared'}, ...
                           'SampleTime', {10,'ns'});
    case 'Incisive'
        hDec    = hdlcosim('InputSignals', {'/viterbi_block/In1','/viterbi_block/In2'},
                           'OutputSignals', {'/viterbi_block/Out1'}, ...
                           'OutputSigned', false, ...
                           'OutputFractionLengths', 0, ...
                           'TCLPreSimulationCommand', 'force :clk B"0" -after 0ns B"1"
                           'TCLPostSimulationCommand', 'echo "done";', ...
                           'PreRunTime', {10,'ns'}, ...
                           'Connection', {'Shared'}, ...
                           'SampleTime', {10,'ns'});
end
```

Launch HDL Simulator

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('Waiting for HDL simulator to launch ...');
switch Simulator
    case 'ModelSim'
        vsim('tclstart',viterbi_tclcmds_modelsim('vsimmatlabsysobj'));
```

```
    case 'Incisive'
        nclaunch('tclstart',viterbi_tclcmds_incisive('hdlsimmatlabsysobj'));
end
Timeout=450;
processid = pingHdlSim(Timeout);
% Check if HDL simulator is ready for Cosimulation.
assert(ischar(processid),['Timeout: HDL simulator took more than ', num2str(Timeout),'
disp('Ready for cosimulation ...');
```

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     = step(hConEnc, data);
    modSignal       = step(hMod, encodedData);
    receivedSignal  = step(hChan, modSignal);
    demodSignalSD   = step(hDemod, receivedSignal);
    quantizedValue  = fi(4-demodSignalSD,0,3,0);
    input1          = quantizedValue(1:2:2*FrameSize);
    input2          = quantizedValue(2:2:2*FrameSize);
    receivedBits    = step(hDec,input1, input2);
    errors          = step(hError, data, double(receivedBits));
end
```

Display the Bit-Error Rate

The Bit-Error Rate is displayed for the Viterbi decoder.

```
sprintf('Bit Error Rate is %d\n',errors(1))
```

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;
```

```
% This concludes the "Verify Viterbi Decoder Using MATLAB System Object and
% HDL Simulator".
```

See Also

Cosimulation Wizard | HDL Cosimulation

Topics

“Create a MATLAB System Object”

Introduced in R2012b

Functions — Alphabetical List

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 or string. 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 or string.

Data Types: `char` | `string`

See Also

`pingHdlSim`

Introduced in R2008a

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 callback function that compiles the HDL code and launches the HDL simulator.

Open the Cosimulation Wizard App

- Simulink Toolstrip: In the **Apps** tab, under **Verification, Validation and Test**, click the **Cosim Wizard** icon.
- MATLAB command prompt: Enter `cosimWizard`.

Examples

- “Verify Raised Cosine Filter Design Using MATLAB”
- “Verify Raised Cosine Filter Design Using Simulink”

See Also

Topics

“Verify Raised Cosine Filter Design Using MATLAB”
“Verify Raised Cosine Filter Design Using Simulink”
“Import HDL Code for MATLAB Function”
“Import HDL Code for MATLAB System Object”
“Import HDL Code for HDL Cosimulation Block”

Introduced in R2012b

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`

See Also

`mvl2dec`

Introduced in R2008a

dpigen

Generate SystemVerilog DPI component from MATLAB function

Syntax

```
dpigen fcn -args args
dpigen fcn -args args -testbench tb_name -options options files -c -
launchreport -FixedpointDataType type
```

Description

`dpigen fcn -args args` generates a SystemVerilog DPI component shared library from MATLAB function `fcn` and all the functions that `fcn` calls. It 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 -args args -testbench tb_name -options options files -c -launchreport -FixedpointDataType 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 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.
- `-FixedpointDataType` specifies the SystemVerilog data type to use for fixed-point type ports.

When generating a DPI component, it creates a shared library specific to that host platform. For instance 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 Incisive run_tb_incisive.sh
### Generating test bench simulation script for Cadence Xcelium run_tb_xcelium.sh
### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh
### Generating test bench simulation script for Vivado Simulator run_tb_vivado.bat
```

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 `dipgen` 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 Incisive run_tb_incisive.sh
### Generating test bench simulation script for Cadence Xcelium run_tb_xcelium.sh
### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh
### Generating test bench simulation script for Vivado Simulator run_tb_vivado.bat
```

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.

`-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.

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
`-I include_path`	Specifies the path to folders containing headers and library files needed for `codegen`, specified as a character vector or string scalar. Add *`include_path`* to the beginning of the code generation path. For example: `-I E:\HDLTools\ModelSim\10.2c-mw-0\questa_sim\include` *`include_path`* 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, `dpigen` might not find files on this path. When converting MATLAB code to C/C++ code, `dpigen` searches the code generation path first. Alternatively, you can specify the include path with the `files` input argument.

Option flag	Option value
`-config config`	Specify a custom configuration object using `coder.config('dll')`. The DPI component must be a shared library. 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** Not all the options in the `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.
`-o output`	Specify the name of the generated component as a character vector or string scalar. The `dpigen` function adds a platform-specific extension to this name for the shared library.
`-d dir`	Specify the output folder. All generated files are placed in *`dir`*. By default, files are placed in `./codegen/dll/<function>`. For example, when `dpigen` compiles the function `fun.m`, the generated code is placed in `./codegen/dll/fun`.

Option flag	Option value
`-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 `GLOBALS`. 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 `-globals` option, `dpigen` checks for the variables in the MATLAB global workspace. If you do not supply an initial value, `dpigen` 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.

`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
`.c`	Custom C file
`.cpp`	Custom C++ file
`.h`	Custom header file (included by all generated files)
`.o`	Object file
`.obj`	Object file
`.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`.

`-FixedpointDataType` — generated SystemVerilog data type for fixed-point ports
`Compatible C Type` | `Bit Vector` | `Logic Vector`

Select the SystemVerilog data type that will be used for ports that have fixed-point data. 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.

See Also

`codegen`

Introduced in R2014b

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®, Microsemi®, 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 **FIL Wizard** icon.
- 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.

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”

Introduced in R2012b

hdldaemon

Control MATLAB server that supports interactions with HDL simulator

Syntax

```
hdldaemon
hdldaemon(Name,Value)
hdldaemon(Option)

s=hdldaemon( ___ )
```

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=hdldaemon( ___ )` 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=hdldaemon('status')
```

```
x =
           comm: 'sockets'
    connections: 0
         ipc_id: '4449'
```

Returned message for a shared memory connection:

```
x=hdldaemon('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 "Implementing the Filter Component of an 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 `matlabcp`/`matlabtb` `tnow` 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 `double` 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 Incisive).

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.

See Also

`nclaunch` | `vsim`

Topics

“Implementing the Filter Component of an Oscillator in MATLAB®”
“Start the HDL Simulator from MATLAB”

Introduced in R2008a

hdlsimmatlab

Load instantiated HDL design for verification with Cadence Incisive and MATLAB

Syntax

```
hdlsimmatlab <instance> [<ncsim_args>]
```

Description

The `hdlsimmatlab` command loads the specified instance of an HDL design for verification and sets up the Cadence Incisive simulator so it can establish a communication link with MATLAB. The Cadence Incisive simulator opens a simulation workspace as it loads the HDL design.

This command may be run from the HDL simulator prompt or from a Tcl script shell (`tclsh`).

This command is issued in the HDL simulator.

Arguments

`<instance>`

Specifies the instance of an HDL design to load for verification.

`<ncsim_args>`

Specifies one or more `ncsim` command arguments. For details, see the description of `ncsim` in the Cadence Incisive simulator documentation.

Examples

The following command loads the module instance `parse` from library `work` for verification and sets up the Cadence Incisive simulator so it can establish a communication link with MATLAB:

```
tclshell> hdlsimmatlab work.parse
```

Introduced in R2008a

hdlsimulink

Load instantiated HDL module for cosimulation with Cadence Incisive and Simulink

Syntax

```
hdlsimulink instance -socket tcp_spec <ncsim_args>
```

Description

Note Issue this command in Cadence Incisive, not in MATLAB.

`hdlsimulink instance -socket tcp_spec <ncsim_args>` loads the specified instance of HDL design for cosimulation and sets up the Cadence Incisive simulator so it can establish a shared communication link with Simulink. The Cadence Incisive 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 Cadence Incisive, 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 Cadence Incisive

Instance of HDL design to load for cosimulation.

`ncsim_args` — ncsim command arguments
Cadence Incisive command arguments

Specify one or more `ncsim` command line arguments. Do not use `-GUI`, `-BATCH`, or `-TCL`. For details, see the description of `ncsim` in the Cadence Incisive 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 Cadence Incisive 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.

See Also

`nclaunch` | `vsimulink`

Introduced in R2008a

matlabcp

Associate MATLAB component function with instantiated HDL design

Syntax

```
matlabcp <instance>
[<time-specs>]
[-socket <tcp-spec>]
[-rising <port>[,<port>...]]
[-falling <port> [,<port>,...]]
[-sensitivity <port>[,<port>,...]]
[-mfunc <name>]
[-use_instance_obj]
[-argument]
```

Description

The `matlabcp` command has the following characteristics:

- 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`, all previously scheduled events initiated by `matlabcp` on `foo` are canceled.

This command is issued in the HDL simulator.

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

Notes 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 you selected for `hdldaemon` when you issue a link request with the `matlabcp` command in the HDL simulator.

Arguments

`<instance>`

Specifies an instance of an HDL design that is associated with a MATLAB function. By default, `matlabcp` 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` (note that hierarchy names are ignored; for example, if your instance name is `top.myfirfilter`, `matlabcp` would associate only `myfirfilter` with the MATLAB function). Alternatively, you can specify a different MATLAB function with `-mfunc`.

Note Do not specify an instance of an HDL module that has already been associated with a MATLAB function (via `matlabcp` or `matlabtb`). If you do, the new association overwrites the existing one.

`<time-specs>`

Specifies a combination of time specifications consisting of any or all of the following:

`<timen>,...`	Specifies one or more discrete time values at which the HDL simulator calls the specified MATLAB function. Each time value is relative to the current simulation time. Even if you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Separate multiple time values by a space. For example: `matlabtb vlogtestbench_top 10 ns, 10 ms, 10 sec` The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero. **Note** For time-based parameters, you can specify any standard time units (`ns`, `us`, and so on). If you do not specify units, the command treats the time value as a value of HDL simulation ticks.
`-repeat <time>`	Specifies that the HDL simulator calls the MATLAB function repeatedly based on the specified `<timen>,...` pattern. The time values are relative to the value of `tnow` at the time the HDL simulator first calls the MATLAB function.
`-cancel <time>`	Specifies a time at which the specified MATLAB function stops executing. The time value is relative to the value of `tnow` 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 `nomatlabtb` call. **Note** The `-cancel` option works only with the <time-specs> arguments. It does not affect any of the other scheduling arguments for `matlabcp`.

Note Place time specifications after the `matlabcp` instance and before any additional command arguments; otherwise the time specifications are ignored.

All time specifications for the `matlabcp` functions appear as a number and, optionally, a time unit:

- fs (femtoseconds)
- ps (picoseconds)
- ns (nanoseconds)
- us (microseconds)
- ms (milliseconds)
- sec (seconds)
- no units (tick)

`-socket <tcp_spec>`

Specifies that HDL Verifier 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 `-socket <tcp_spec>` on the command line. The communication mode that you specify with the `matlabcp` command must match the communication mode that you issued with the `hdldaemon` command.

`-rising <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function on the rising edge (transition from `'0'` to `'1'`) of any of the specified signals. Specify `-rising` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- VHDL: Rising edge is {0 or L} to {1 or H}.
- Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them 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.

`-falling <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals experiences a falling edge—changes from `'1'` to `'0'`. Specify `-falling` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- VHDL: Falling edge is {1 or H} to {0 or L}.
- Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them 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.

`-sensitivity <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals of any type can appear in the sensitivity list and can be positioned at any level in the HDL model hierarchy.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them 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.

`-mfunc <name>`

The name of the MATLAB function that is associated with the HDL module instance you specify for `instance`. By default, the HDL Verifier software invokes a MATLAB function that has the same name as the specified HDL instance. Thus, if the names are the same, you can omit the `-mfunc` option. 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.

`-use_instance_obj`

Instructs the function specified with the argument `-mfunc` to use an HDL instance object passed by HDL Verifier to the function. This argument has the fields shown in the following table. See “Writing Functions Using the HDL Instance Object” for examples.

Field	Read/Write Access	Description
`tnext`	Write only	Used to schedule a callback during the set time value. This field is equivalent to old `tnext`. For example: `hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9` will schedule a callback at time equals 5 nanoseconds from `tnow`.
`userdata`	Read/Write	Stores state variables of the current `matlabcp` instance. You can retrieve the variables the next time the callback of this instance is scheduled.

`simstatus`	Read only	Stores the status of the HDL simulator. The HDL Verifier software sets this field to `'Init'` during the first callback for this particular instance and to `'Running'` thereafter. `simstatus` is a read-only property.
		`>> hdl_instance_obj.simstatus`
		`ans=`
		`Init`

`instance`	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:
		In the HDL simulator:
		`hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj`
		In MATLAB:
		`>> hdl_instance_obj.instance`
		`ans=`
		`osc_top`

Field	Read/Write Access	Description

`argument`	Read only	Stores the argument set by the `-argument` option of `matlabcp`. For example:
		`matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo`
		The link software supports the `-argument` option only when it is used with `-use_instance_obj`, otherwise the argument is ignored. argument is a read-only property.
		`>> hdl_instance_obj.argument`
		`ans=`
		`foo`

`portinfo`	Read only	Stores information about the VHDL and Verilog ports 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 port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information". `hdl_instance_obj.portinfo.`*`field1.field2.field3`* **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`.

Field	Read/Write Access	Description

`tscale`	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. `tscale` is a read-only property.
		
		`>> hdl_instance_obj.tscale`
		
		`ans=`
		`    1.0000e-009`
		
		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`.

`tnow`	Read only	Stores the current time. `tnow` is a read-only property. `hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;`

`portvalues`	Read/Write	Stores the current values of and sets new values for the output and input ports for a `matlabcp` instance. For example:
		
		`>> hdl_instance_obj.portvalues`
		
		`ans =`
		`Read Only Input ports:`
		`    clk_enable: []`
		`    clk: []`
		`    reset: []`
		`Read/Write Output ports:`
		`    sine_out: [22x1 char]`

`linkmode`	Read only	Stores the status of the callback. The HDL Verifier software sets this field to `'testbench'` if the callback is associated with `matlabtb` and `'component'` if the callback is associated with `matlabcp`. `linkmode` is a read-only property.
		
		`>> hdl_instance_obj.linkmode`
		
		`ans=`
		`    component`

`-argument`

Used to pass user-defined arguments from the `matlabcp` invocation on the HDL side to the MATLAB function callbacks. Supported with `-use_instance_obj` only. See the field listing under the `-use_instance_obj` property.

Examples

The following examples demonstrate some ways you might use the `matlabcp` function.

Using `matlabcp` with the `-mfunc` option to Associate an HDL Component with a MATLAB Function of a Different Name

This example explicitly associates the Verilog module `vlogtestbench_top.u_matlab_component` with the MATLAB function `vlogmatlabc` using the `-mfunc` option. The `'-socket'` option specifies using socket communication on port 4449.

```
hdlsim>matlabcp vlogtestbench_top.u_matlab_component -mfunc vlogmatlabc -socket 4449
```

Using `matlabcp` with Explicit Times and the `-cancel` Option

This example includes explicit times with the `-cancel` option.

```
hdlsim>matlabcp vlogtestbench_top 1e6 fs  3 2e3 ps -repeat 3 ns -cancel 7ns
```

Using `matlabcp` with Rising and Falling Edges

This example implicitly associates the Verilog module, `vlogtestbench_top`, with the MATLAB function `vlogtestbench_top`, and also uses rising and falling edges.

```
hldsim> matlabcp vlogtestbench_top  1 2 3 4 5 6 7 -rising outclk3
          -falling u_matlab_component/inoutclk
```

Introduced in R2008a

matlabtb

Schedule MATLAB test bench session for instantiated HDL module

Syntax

```
matlabtb <instance>
[<time-specs>]
[-socket <tcp-spec>]
[-rising <port>[,<port>...]]
[-falling <port> [,<port>,...]]
[-sensitivity <port>[,<port>,...]]
[-mfunc <name>]
[-use_instance_obj]
[-argument]
```

Description

The `matlabtb` command has the following characteristics:

- 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`, all previously scheduled events initiated by `matlabtb` on `foo` are canceled.

This command is issued in the HDL simulator.

MATLAB test bench functions mimic stimuli passed to entities in the HDL model. You force stimulus from MATLAB or HDL scheduled with `matlabtb`.

Notes 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.

Arguments

`<instance>`

Specifies the instance of an HDL module that the HDL Verifier software associates with a MATLAB test bench function. By default, `matlabtb` associates the instance with 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` (note that hierarchy names are ignored; for example, if your instance name is `top.myfirfilter`, `matlabtb` would associate only `myfirfilter` with the MATLAB function). Alternatively, you can specify a different MATLAB function with `-mfunc`.

Note Do not specify an instance of an HDL module that has already been associated with a MATLAB function (via `matlabcp` or `matlabtb`). If you do, the new association overwrites the existing one.

`<time-specs>`

Specifies a combination of time specifications consisting of any or all of the following:

`<timen>,...`	Specifies one or more discrete time values at which the HDL simulator calls the specified MATLAB function. Each time value is relative to the current simulation time. Even if you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Separate multiple time values by a space. For example: `matlabtb vlogtestbench_top 10 ns, 10 ms, 10 sec` The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero. **Note** For time-based parameters, you can specify any standard time units (`ns`, `us`, and so on). If you do not specify units, the command treats the time value as a value of HDL simulation ticks.
`-repeat <time>`	Specifies that the HDL simulator calls the MATLAB function repeatedly based on the specified `<timen>,...` pattern. The time values are relative to the value of `tnow` at the time the HDL simulator first calls the MATLAB function. For example: `matlabtb vlogtestbench_top 5 ns -repeat 10 ns` The MATLAB function executes at time equals 0 ns, 5 ns, 15 ns, 25 ns, and so on.

`-cancel <time>`	Specifies a time at which the specified MATLAB function stops executing. The time value is relative to the value of `tnow` 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 `nomatlabtb` call. **Note** The `-cancel` option works only with the <time-specs> arguments. It does not affect any of the other scheduling arguments for `matlabtb`.

Note Place time specifications after the `matlabtb` instance and before any additional command arguments; otherwise the time specifications are ignored.

All time specifications for the `matlabtb` functions appear as a number and, optionally, a time unit:

- fs (femtoseconds)
- ps (picoseconds)
- ns (nanoseconds)
- us (microseconds)
- ms (milliseconds)
- sec (seconds)
- no units (tick)

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. When you provide TCP/IP information for `matlabtb`, you can choose a TCP/IP port number or TCP/IP port alias or service name for the `<tcp_spec>` parameter. If you are setting up communication between computers, you must also specify the name or Internet address of the remote host that is running the MATLAB server (`hdldaemon`).

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of

communication and takes effect if you do not specify `-socket <tcp_spec>` on the command line.

Note The communication mode that you specify with the `matlabtb` command must match what you specify for the communication mode when you issue the `hldaemon` 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".

`-rising <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function on the rising edge (transition from `'0'` to `'1'`) of any of the specified signals. Specify `-rising` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- VHDL: Rising edge is {0 or L} to {1 or H}.
- Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them 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.

`-falling <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals experiences a falling edge—changes from `'1'` to `'0'`. Specify `-falling` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- VHDL: Falling edge is {1 or H} to {0 or L}.
- Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them 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.

`-sensitivity <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals of any type can appear in the sensitivity list and can be positioned at any level of the HDL design.

If you specify the option with no signals, the interface is sensitive to value changes for all signals.

Note Use of this option for `INOUT` ports can result in double calls.

For example:

```
-sensitivity /randnumgen/dout
```

The MATLAB function executes if the value of `dout` changes.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them 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.

`-mfunc <name>`

The name of the associated MATLAB function. If you omit this argument, `matlabtb` associates the HDL module instance to a MATLAB function that has the same name as the HDL instance. If you omit this argument and `matlabtb` does not find a MATLAB function with the same name, the command generates an error message.

`-use_instance_obj`

Instructs the function specified with the argument `-mfunc` to use an HDL instance object passed by HDL Verifier to the function. This argument has the fields shown in the following table. See“Writing Functions Using the HDL Instance Object” for examples.

Field	Read/Write Access	Description
`tnext`	Write only	Used to schedule a callback during the set time value. This field is equivalent to old `tnext`. For example: `hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9` will schedule a callback at time equals 5 nanoseconds from `tnow`.
`userdata`	Read/Write	Stores state variables of the current `matlabcp` instance. You can retrieve the variables the next time the callback of this instance is scheduled.

`simstatus`	Read only	Stores the status of the HDL simulator. The HDL Verifier software sets this field to `'Init'` during the first callback for this particular instance and to `'Running'` thereafter. `simstatus` is a read-only property.
		`>> hdl_instance_obj.simstatus`
		`ans=`
		`Init`

`instance`	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:
		In the HDL simulator:
		`hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj`
		In MATLAB:
		`>> hdl_instance_obj.instance`
		`ans=`
		`osc_top`

Field	Read/Write Access	Description

`argument`	Read only	Stores the argument set by the `-argument` option of `matlabcp`. For example:
		`matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo`
		The link software supports the `-argument` option only when it is used with `-use_instance_obj`, otherwise the argument is ignored. argument is a read-only property.
		`>> hdl_instance_obj.argument`
		`ans=`
		`foo`

`portinfo`	Read only	Stores information about the VHDL and Verilog ports 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 port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information". `hdl_instance_obj.portinfo.`*`field1.field2.field3`* **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`.

Field	Read/Write Access	Description

`tscale`	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. `tscale` is a read-only property.
		
		`>> hdl_instance_obj.tscale`
		
		`ans=`
		`    1.0000e-009`
		
		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`.

`tnow`	Read only	Stores the current time. `tnow` is a read-only property. `hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;`

`portvalues`	Read/Write	Stores the current values of and sets new values for the output and input ports for a `matlabcp` instance. For example:
		
		`>> hdl_instance_obj.portvalues`
		
		`ans =`
		`Read Only Input ports:`
		`    clk_enable: []`
		`    clk: []`
		`    reset: []`
		`Read/Write Output ports:`
		`    sine_out: [22x1 char]`

`linkmode`	Read only	Stores the status of the callback. The HDL Verifier software sets this field to `'testbench'` if the callback is associated with `matlabtb` and `'component'` if the callback is associated with `matlabcp`. `linkmode` is a read-only property.
		
		`>> hdl_instance_obj.linkmode`
		
		`ans=`
		`    component`

`-argument`

Used to pass user-defined arguments from the `matlabtb` instantiation on the HDL side to the MATLAB function callbacks. Supported with `-use_instance_obj` only. See the field listing for argument under the `-use_instance_obj` property.

Examples

The following examples demonstrate some ways you might use the `matlabtb` function.

Using `matlabtb` with the `-socket` Argument and Time Parameters

The following command starts the HDL simulator client component of HDL Verifier, associates an instance of the entity, `myfirfilter`, with the MATLAB function `myfirfilter`, and begins 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 repeatedly every 10 nanoseconds:

```
hdlsim> matlabtb myfirfilter 5 ns -repeat 10 ns -socket 4449
```

Applying Rising Edge Clocks and State Changes with `matlabtb`

The following command starts the HDL simulator client component of HDL Verifier, and begins a remote TCP/IP socket-based session using 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
```

Specifying a MATLAB Function Name and Sensitizing Signals with `matlabtb`

The following command starts the HDL simulator client component of the HDL Verifier software. The `'-mfunc'` option specifies the MATLAB function to connect to and the `'-`

`socket'` option specifies the port number for socket connection mode. `'-sensitivity'` 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
```

Introduced in R2008a

matlabtbeval

Call specified MATLAB function once and immediately on behalf of instantiated HDL module

Syntax

```
matlabtbeval <instance> [-socket <tcp_spec>]
[-mfunc <name>]
```

Description

The `matlabtbeval` command has the following characteristics:

- 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.
- Executes the specified MATLAB function once and immediately on behalf of the specified module instance.

This command is issued in the HDL simulator.

Note The `matlabtbeval` command executes the MATLAB function immediately, while `matlabtb` provides several options for scheduling MATLAB function execution.

Notes The communication mode that you specify for `matlabtbeval` 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 `matlabtbeval` command in the HDL simulator.

Arguments

`<instance>`

Specifies the instance of an HDL module that is associated with a MATLAB function. By default, `matlabtbeval` associates the HDL module instance with a MATLAB function that has the same name as the HDL module instance. For example, if the HDL module instance is `myfirfilter`, `matlabtbeval` associates the HDL module instance with the MATLAB function `myfirfilter`. Alternatively, you can specify a different MATLAB function with the `-mfunc` property.

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. For TCP/IP socket communication on a single computer, the `<tcp_spec>` can consist of just a TCP/IP port number or service name (alias). If you are setting up communication between computers, you must also specify the name or Internet address of the remote host.

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

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”.

`-mfunc <name>`

The name of the associated MATLAB function. If you omit this argument, `matlabtbeval` associates the HDL module instance with a MATLAB function that has the same name as the HDL module instance. If you omit this argument and `matlabtbeval` does not find a MATLAB function with the same name, the command displays an error message.

Examples

This example starts the HDL simulator client component of the link software, associates an instance of the module `myfirfilter` with the function `myfirfilter.m`, and uses a local TCP/IP socket-based communication link to TCP/IP port 4449 to execute the function `myfirfilter.m`:

```
hdlsim> matlabtbeval myfirfilter -socket 4449:
```

Introduced in R2008a

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 multivalued logic vector.

```
mvl2dec('010111')
```

```
ans = 23
```

Find the decimal integer equivalent for multivalued logic vector with one or more values other than 0 and 1. The function returns NaN.

```
mvl2dec('x01201')
```

```
ans = NaN
```

Find the decimal integer equivalent for 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
`true`	The input is a signed multivalued logic. The function assumes that the first character `mv_logic_char(1)` is a signed bit of a 2's complement number.
`false`	The input is an unsigned multivalued logic.

Data Types: `logical`

See Also

`dec2mvl`

Introduced in R2008a

nclaunch

Start and configure Cadence Incisive simulators for use with HDL Verifier software

Syntax

```
nclaunch('PropertyName','PropertyValue'...)
```

Description

`nclaunch('PropertyName','PropertyValue'...)` starts the Cadence Incisive simulator for use with the MATLAB and Simulink features of the HDL Verifier software. The first folder in the Cadence Incisive simulator matches your MATLAB current folder if you do not specify an explicit `rundir` parameter.

After you call this function, you can use HDL Verifier functions for the HDL simulator (for example, `hdlsimmatlab`, `hdlsimulink`) to do interactive debug setup.

The property name/property value pair settings allow you to customize the Tcl commands used to start the Cadence Incisive simulator, the `ncsim` 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. You must use a property name/property value pair with `nclaunch`.

Name-Value Pair Arguments

`hdlsimdir`

Specifies the path name to the Cadence Incisive simulator executable to be started.

- `pathname`

  Start a different version of the Cadence Incisive simulator or if the version of the simulator you want to run does not reside on the system path.

Default: The first version of the simulator that the function finds on the system path.

`hdlsimexe`

Specifies the name of a Cadence Incisive simulator executable.

- `simexename`

  Custom-built simulator executable.

Default: `ncsim`

`libdir`

This property creates an entry in the startup Tcl file that points to the folder with the shared libraries for the Cadence Incisive simulator to communicate with MATLAB when the Cadence Incisive simulator runs on a machine that does not have MATLAB.

- `folder`

  Folder containing MATLAB shared libraries.

`libfile`

Specifies the library file to use for HDL simulation. 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. See "Cosimulation Libraries" for versions of the library built using other compilers.

- `library_file_name`

  The particular library file to use for HDL simulation.

Default: The version of the library file that was built using the same compiler that MATLAB itself uses.

`rundir`

Specifies the folder containing the HDL simulator executable.

- `dirname`

  Where to run the HDL simulator.

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 folder does *not* exist, you will get an error.

Default: The current working folder

`runmode`

Specifies how to start the HDL simulator.

- `mode`

  This property accepts the following valid values:

  - `'Batch'`: Start the HDL simulator in the background with no window.
  - `'Batch with Xterm'`: Run HDL simulator in an non-interactive Xterm window.
  - `'CLI'`: Start the HDL simulator in an interactive terminal window.
  - `'GUI'`: Start the HDL simulator with the graphical user interface.

Default: `'GUI'`

`socketsimulink`

Specifies TCP/IP socket communication between the Cadence Incisive simulator and Simulink. For shared memory, omit `-socket <tcp-spec>` on the command line.

- `tcp_spec`

  TCP/IP port number or service name (alias)

Default: Shared memory

`starthdlsim`

Determines whether the Cadence Incisive simulator is launched.

This function creates a startup Tcl file which contains pointers to MATLAB and Simulink shared libraries. To run the Cadence Incisive simulator manually, see “Start the HDL Simulator from MATLAB”.

- `yes`

Launches the Cadence Incisive simulator and creates a startup Tcl file.

- `no`

  Does not launch the Cadence Incisive simulator , but still creates a startup Tcl file.

Default: `yes`

`startupfile`

Specify the name and location of the Tcl script generated by `nclaunch`. The generated Tcl script, when executed, compiles and launches the HDL simulator. You can edit and use the generated file in a regular shell outside of MATLAB. For example:

```
sh> tclsh compile_and_launch.tcl
```

- `pathname`

  Filename and path for generated Tcl script. If the file name already exists on the specified path, that file's contents are overwritten.

Default: Generates a filename of `compile_and_launch.tcl` in the folder specified by `rundir`.

`tclstart`

Specifies one or more Tcl commands to execute before the Cadence Incisive simulator launches. You must specify at least one command; otherwise, no action occurs.

- `tcl_commands`

  A command character vector or a cell array of commands.

  Note You must type `exec` in front of non-Tcl system shell commands. For example:

  ```
  exec -ncverilog -64bit -c +access+rw +linedebug top.v
  hdlsimulink -gui work.top
  ```

Examples

Start Cosimulation Session with Simulink

Compile design and start Simulink.

```
nclaunch('tclstart',{'exec ncverilog -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 ncverilog -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 ncvlog -64bit vlogtestbench_top.v'
tclcmd{2} = 'exec ncelab -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 ncvlog -64bit vlogtestbench_top.v'    'exec ncelab -64bit -access +wc vlogtestbench_top'


tclcmd =

    'exec ncvlog -64bit vlogtestbench_top.v'    'exec ncelab -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 `ncsim` Tcl shell.
- Issues a call to `matlabcp`, which associates the function `vlogmatlabc` to the module instance `u_matlab_component`.
- Assumes that the `hdldaemon` in MATLAB is listening on port 32864
- Instructs the `run` function to run 50 resolution units (ticks).

Introduced in R2008a

nomatlabtb

End active MATLAB test bench and MATLAB component sessions

Syntax

```
nomatlabtb
```

Description

The `nomatlabtb` command ends all active MATLAB test bench and MATLAB component sessions that were previously initiated by `matlabtb` or `matlabcp` commands.

This command is issued in the HDL simulator.

Note This command should be called before shutting down `hdldaemon` or `hdldaemon` will block shutdown until the call occurs.

Examples

The following command ends all MATLAB test bench and MATLAB component sessions:

```
hdlsim> nomatlabtb
```

See Also

`matlabcp` | `matlabtb`

Introduced in R2008a

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.

See Also

`hdldaemon` | `waitForHdlClient`

Introduced in R2012b

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`

See Also

`hdldaemon`

Introduced in R2008a

programFPGA

Package: `hdlverifier`

Load programming file 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.

See Also

`hdlverifier.FILSimulation`

Topics

“FIL Simulation with HDL Workflow Advisor for MATLAB”

Introduced in R2010b

tclHdlSim

Execute Tcl command in Incisive or ModelSim simulator

Syntax

```
tclHdlSim(tclCmd)
tclHdlSim(tclCmd,portNumber)
tclHdlSim(tclCmd, portname, hostname)
```

Description

`tclHdlSim(tclCmd)` executes a Tcl command on the Incisive or ModelSim simulator using a shared connection during a Simulink cosimulation session.

`tclHdlSim(tclCmd,portNumber)` executes a Tcl command on the Incisive or ModelSim simulator by connecting to the local host on port `portNumber`.

`tclHdlSim(tclCmd, portname, hostname)` executes a Tcl command on the Incisive or ModelSim simulator by connecting to the host `hostname` on port `portname`.

The Incisive or ModelSim simulator must be connected to MATLAB and Simulink using the HDL Verifier software for this function to work (see either `vsimulink` or `hdlsimulink`).

You may 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 Incisive).

To execute a Tcl command on the Incisive or ModelSim simulator during a MATLAB cosimulation session, use `hdldaemon('tclcmd','command')`.

Examples

The following function call displays a message in the HDL simulator command window using port 5678 on host name `msuser`:

```
>>tclHdlSim('puts "Done"', '5678', 'msuser')
```

See Also

`hldaemon` | `nclaunch` | `vsim`

Introduced in R2008a

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
- `vmatlabsysobj`: 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 ModelSimby 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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).

`socketmatlabsysobj` — 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 `'socketmatlabsysobj'` 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` — LaunchModelSim 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`
- `vmatlabsysobj`

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.

See Also

`matlabtb` | `vsimmatlab` | `vsimulink`

Introduced in R2008a

uvmbuild

Generate UVM test bench from Simulink model

Syntax

```
uvmbuild(dut,sequence,scoreboard)
```

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

Examples

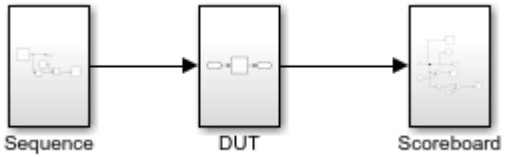
Use uvmbuild to generate UVM Test Bench

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');
```



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');
```

```
### Starting DPI subsystem generation for UVM test bench
### Starting build procedure for model: DUT
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\ex8763
### Generating DPI C Wrapper C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\ex8763
### Generating UVM module package C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\e
### Generating SystemVerilog module C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda
### Generating makefiles for: DUT_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: DUT
### Starting build procedure for model: Sequence
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\ex8763
### Generating DPI C Wrapper C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\ex8763
### Generating UVM module package C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\e
### Generating SystemVerilog module C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda
### Generating makefiles for: Sequence_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: Sequence
### Starting build procedure for model: Scoreboard
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\ex8763
### Generating DPI C Wrapper C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\ex8763
### Generating UVM module package C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda\e
### Generating SystemVerilog module C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\27\tpf8a53cda
### Generating makefiles for: Scoreboard_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: Scoreboard
### Starting UVM test bench generation for model: hdlv_uvmbuild
### Generating UVM transaction object C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53c
```

```
### Generating UVM interface C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex8763
### Generating UVM sequence C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex87636
### Generating UVM sequence transaction C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a5
### Generating UVM driver C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex876366
### Generating UVM monitor C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex87636
### Generating UVM agent C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex87636604
### Generating UVM scoreboard C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex876
### Generating UVM environment C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex87
### Generating UVM test C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex87636604/
### Generating UVM top C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex87636604/h
### Generating UVM test package C:/TEMP/Bdoc19b_1192687_6748/ibF7BE2B/27/tpf8a53cda/ex8
### Generating UVM test bench simulation script for Mentor Graphics QuestaSim/Modelsim
```

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`

See Also

`dpigen`

Topics

“UVM Component Generation Overview”

Introduced in R2019b

vsimmatlab

Load instantiated HDL module for verification with ModelSim and MATLAB

Syntax

```
vsimmatlab <instance> [<vsim_args>]
```

Description

The `vsimmatlab` command 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 HDL module's packages and architectures.

This command is generally issued in the HDL simulator. It also may be run from the HDL simulator prompt or from a Tcl script shell (`tclsh`).

Arguments

`<instance>`

Specifies the instance of an HDL module to load for verification.

`<vsim_args>`

Specifies one or more ModelSim `vsim` command arguments. For details, see the description of `vsim` in the ModelSim documentation.

Examples

The following command loads the HDL module instance `parse` from library `work` for verification and sets up ModelSim so it can establish a communication link with MATLAB:

```
ModelSim> vsimmatlab work.parse
```

Introduced in R2008a

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.

See Also

`vsim`

Introduced in R2008a

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.

See Also

`hdldaemon` | `notifyMatlabServer`

Introduced in R2012b