

Vivado Design Suite User Guide

Logic Simulation

UG900 (v2021.2) October 22, 2021

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this [link](#) for more information.



Revision History

The following table shows the revision history for this document.

Section	Revision Summary
10/22/2021 Version 2021.2	
Supported Simulators	Updated Supported Simulators table.
Language and Encryption Support	Updated.
Pointing to the Simulator Install Location	Updated figure.
SECUREIP Simulation Library	Updated Special Considerations for Using SECUREIP Libraries table.
Chapter 3: Simulating with Third-Party Simulators	Updated Supported Third-Party Simulators table. Updated Environment Variable Setting for Third-Party Simulators table.
Dumping SAIF for Power Analysis	Updated.
Scope Window	Updated figures.
Using the launch_simulation Command	Updated examples.
xelab	Updated xelab command syntax options.
xelab, vhdl, and xvlog xsim Command Options	Updated.
xsim Executable Options	xsim Executable Command Options table.
Functional Coverage Report Generator	Updated xcrg command options and description.
Xcelium Simulator Compilation Options	Updated Xcelium Compilation Options table.
Vivado Simulator Elaboration Options	Updated Vivado Simulator Elaboration Options table.
Questa Advanced Simulator Elaboration Options	Questa Advanced Simulator Elaboration Options table.
VCS Simulator Elaboration Options	Updated VCS Elaboration Options table.
Xcelium Simulator Elaboration Options	Updated Xcelium Elaboration Options table.
Supported Features	Updated supported features section.
xsc Compiler	Updated XSC Compiler Switches table.
06/16/2021 Version 2021.1	
Supported Simulators	Updated Supported Simulators table.
Using Versal CIPS VIP	Added new section.
Chapter 3: Simulating with Third-Party Simulators	Updated Supported Third-Party Simulators table. Updated Environment Variable Setting for Third-Party Simulators table.
xelab	Updated xelab command syntax options.
xelab, vhdl, and xvlog xsim Command Options	Updated.
Functional Coverage Report Generator	Updated xcrg command options and description. Updated xcrg example syntax. Updated export_simulation section.
Xcelium Simulator Compilation Options	Updated xcelium compilation options.

Section	Revision Summary
Vivado Simulator Elaboration Options	Updated Vivado simulator elaboration options.
xsc Compiler	Updated XSC compiler switches.

Table of Contents

Revision History.....	2
Chapter 1: Overview.....	9
Navigating Content by Design Process.....	9
Logic Simulation Overview.....	9
Supported Simulators.....	10
Simulation Flow	10
Language and Encryption Support	13
Chapter 2: Preparing for Simulation.....	14
Using Test Benches and Stimulus Files.....	14
Pointing to the Simulator Install Location.....	15
Compiling Simulation Libraries.....	16
Using Xilinx Simulation Libraries.....	21
Using Simulation Settings.....	30
Adding or Creating Simulation Source Files.....	34
Generating a Netlist.....	36
Chapter 3: Simulating with Third-Party Simulators.....	39
Running Simulation Using Third Party Simulators with Vivado IDE.....	40
Dumping SAIF for Power Analysis.....	43
Dumping VCD.....	45
Simulating IP	46
Using a Custom DO File During an Integrated Simulation Run.....	46
Running Third-Party Simulators in Batch Mode.....	48
Chapter 4: Simulating with Vivado Simulator.....	49
Running the Vivado Simulator.....	49
Running Functional and Timing Simulation.....	67
Saving Simulation Results.....	70
Distinguishing Between Multiple Simulation Runs.....	70
Closing a Simulation.....	71
Adding a Simulation Start-up Script File.....	71

Viewing Simulation Messages.....	72
Using the launch_simulation Command.....	74
Re-running the Simulation After Design Changes (relaunch).....	75
Using the Saved Simulator User Interface Settings.....	76

Chapter 5: Analyzing Simulation Waveforms with Vivado

Simulator.....	78
Using Wave Configurations and Windows.....	78
Opening a Previously Saved Simulation Run.....	79
Understanding HDL Objects in Waveform Configurations	80
Customizing the Waveform.....	83
Controlling the Waveform Display	89
Organizing Waveforms.....	93
Analyzing Waveforms.....	95
Analyzing AXI Interface Transactions.....	100

Chapter 6: Debugging a Design with Vivado Simulator..... 115

Debugging at the Source Level.....	115
Forcing Objects to Specific Values.....	119
Power Analysis Using Vivado Simulator.....	127
Using the report_drivers Tcl Command.....	129
Using the Value Change Dump Feature.....	129
Using the log_wave Tcl Command.....	130
Cross Probing Signals in the Object, Wave, and Text Editor Windows.....	132

Chapter 7: Simulating in Batch or Scripted Mode in Vivado

Simulator.....	138
Exporting Simulation Files and Scripts.....	138
Running the Vivado Simulator in Batch Mode.....	144
Elaborating and Generating a Design Snapshot, xelab.....	146
Simulating the Design Snapshot, xsim.....	157
Example of Running Vivado Simulator in Standalone Mode.....	161
Project File (.prj) Syntax.....	162
Predefined Macros.....	163
Library Mapping File (xsim.ini).....	163
Running Simulation Modes.....	164
Using Tcl Commands and Scripts	167
export_simulation	168

export_ip_user_files.....	171
---------------------------	-----

Appendix A: Compilation, Elaboration, Simulation, Netlist, and Advanced Options.....	174
Compilation Options.....	174
Elaboration Options.....	176
Simulation Options.....	178
Netlist Options.....	180
Advanced Simulation Options.....	181
Appendix B: SystemVerilog Support in Vivado Simulator.....	182
Targeting SystemVerilog for a Specific File.....	182
Testbench Feature.....	189
Appendix C: Universal Verification Methodology Support.....	198
Appendix D: VHDL 2008 Support in Vivado Simulator.....	199
Introduction	199
Compiling and Simulating.....	199
Supported Features.....	201
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator.....	203
Introduction.....	203
Compiling C Code.....	203
xsc Compiler.....	204
Binding Compiled C Code to SystemVerilog Using xelab.....	206
Data Types Allowed on the Boundary of C and SystemVerilog.....	206
Mapping for User-Defined Types.....	207
Support for svdpi.h Functions.....	209
DPI Examples Shipped with the Vivado Design Suite.....	217
Appendix F: SystemC Support in Vivado IDE.....	218
Selecting Simulation Model Type.....	218
Protected Models.....	222
Unprotected Models.....	223
SystemC Simulation Using Vivado.....	224
Running SystemC Simulation Using Vivado Simulator.....	226

Appendix G: Automated Testbench Generation for Sub-Design.....	227
generate_vcd_ports.....	227
create_testbench.....	228
Using Automated Testbench Generation on Example Design.....	229
Appendix H: Handling Special Cases.....	233
Using Global Reset and 3-State.....	233
Delta Cycles and Race Conditions.....	235
Using the ASYNC_REG Constraint.....	236
Simulating Configuration Interfaces.....	238
Disabling Block RAM Collision Checks for Simulation.....	241
Dumping the Switching Activity Interchange Format File for Power Analysis.....	242
Skipping Compilation or Simulation.....	242
Appendix I: Value Rules in Vivado Simulator Tcl Commands.....	244
String Value Interpretation.....	244
Vivado Design Suite Simulation Logic.....	244
Appendix J: Vivado Simulator Mixed Language Support and Language Exceptions.....	246
Using Mixed Language Simulation.....	246
VHDL Language Support Exceptions.....	252
Verilog Language Support Exceptions	253
Appendix K: Vivado Simulator Quick Reference Guide.....	256
Appendix L: Using Xilinx Simulator Interface.....	259
Preparing the XSI Functions for Dynamic Linking.....	259
Writing the Test Bench Code.....	261
Compiling Your C/C++ Program.....	262
Preparing the Design Shared Library.....	262
XSI Function Reference.....	263
Vivado Simulator VHDL Data Format.....	268
Vivado Simulator Verilog Data Format.....	271
Appendix M: Additional Resources and Legal Notices.....	274
Xilinx Resources.....	274
Documentation Navigator and Design Hubs.....	274

References.....	274
Links to Additional Information on Third-Party Simulators.....	275
Training Resources.....	276
Please Read: Important Legal Notices.....	276

Overview

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal™ ACAP design process [Design Hubs](#) can be found on the Xilinx.com website. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
 - [Chapter 3: Simulating with Third-Party Simulators](#)
 - [Chapter 4: Simulating with Vivado Simulator](#)
 - [Appendix F: SystemC Support in Vivado IDE](#)

Logic Simulation Overview

Simulation is a process of emulating real design behavior in a software environment. Simulation helps verify the functionality of a design by injecting stimulus and observing the design outputs.

This chapter provides an overview of the simulation process, and the simulation options in the Vivado® Design Suite.

The process of simulation includes:

- Creating test benches, setting up libraries and specifying the simulation settings for Simulation
- Generating a Netlist (if performing post-synthesis or post-implementation simulation)
- Running a Simulation using Vivado simulator or third party simulators. See [Supported Simulators](#) for more information on supported simulators.

Supported Simulators

Following are the supported simulators in the Vivado Design Suite:

Table 1: Supported Simulators

Simulator	Version	Integrated with Vivado Integrated Design Environment
Vivado® Simulator	2021.2	Integrated with the Vivado integrated design environment, where each simulation launch appears as a framework of windows within the Vivado IDE.
Siemens EDA Questa Advanced Simulator	2020.4	Yes
Siemens EDA ModelSim Simulator	2020.4	Yes
Synopsys Verilog Compiler Simulator (VCS)	R-2020.12	Yes
Aldec Rivera-PRO Simulator	2020.10	Yes
Aldec Active-HDL	12.0	Yes
Cadence Xcelium Parallel Simulator	20.09.006	Yes

See the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#)) for the supported versions of third-party simulators.

For more information about the Vivado IDE and the Vivado Design Suite flow, see:

- *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#))
 - *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))
-

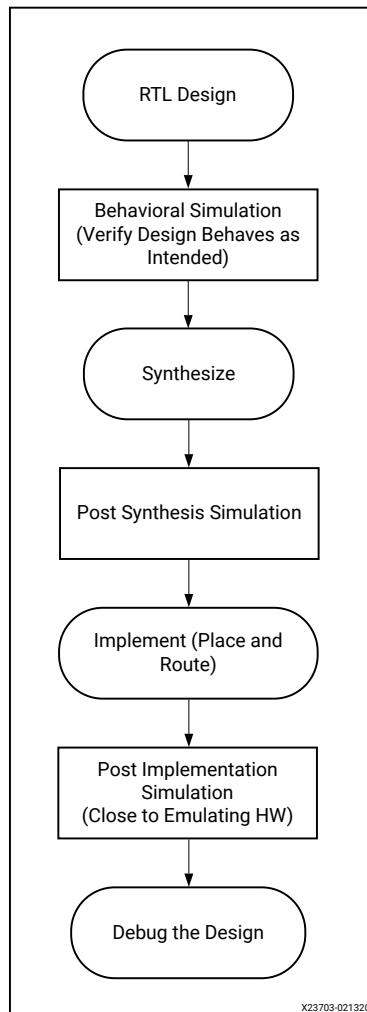
Simulation Flow

Simulation can be applied at several points in the design flow. It is one of the first steps after design entry and one of the last steps after implementation as part of verifying the end functionality and performance of the design.

Simulation is an iterative process and is typically repeated until both the design functionality and timing requirements are satisfied.

The following figure illustrates the simulation flow for a typical design:

Figure 1: Simulation Flow



Behavioral Simulation at the Register Transfer Level

Register Transfer Level (RTL) behavioral simulation can include:

- RTL Code
- Instantiated UNISIM library components
- Instantiated UNIMACRO components
- UNISIM gate-level model (for the Vivado logic analyzer)
- SECUREIP Library

RTL-level simulation lets you simulate and verify your design prior to any translation made by synthesis or implementation tools. You can verify your designs as a module or an entity, a block, a device, or a system.

RTL simulation is typically performed to verify code syntax, and to confirm that the code is functioning as intended. In this step, the design is primarily described in RTL and consequently, no timing information is required.

RTL simulation is not architecture-specific unless the design contains an instantiated device library component. To support instantiation, Xilinx® provides the UNISIM library.

When you verify your design at the behavioral RTL you can fix design issues earlier and save design cycles.

Keeping the initial design creation limited to behavioral code allows for:

- More readable code
- Faster and simpler simulation
- Code portability (the ability to migrate to different device families)
- Code reuse (the ability to use the same code in future designs)

Post-Synthesis Simulation

You can simulate a synthesized netlist to verify that the synthesized design meets the functional requirements and behaves as expected. Although it is not typical, you can perform timing simulation with estimated timing numbers at this simulation point.

The functional simulation netlist is a hierarchical, folded netlist expanded to the primitive module and entity level; the lowest level of hierarchy consists of primitives and macro primitives.

These primitives are contained in the UNISIMS_VER library for Verilog, and the UNISIM library for VHDL.

Related Information

[UNISIM Library](#)

Post-Implementation Simulation

You can perform functional or timing simulation after implementation. Timing simulation is the closest emulation to actually downloading a design to a device. It allows you to ensure that the implemented design meets functional and timing requirements and has the expected behavior in the device.



IMPORTANT! Performing a thorough timing simulation ensures that the completed design is free of defects that could otherwise be missed, such as:

- Post-synthesis and post-implementation functionality changes that are caused by:
 - Synthesis properties or constraints that create mismatches (such as `full_case` and `parallel_case`)
 - UNISIM properties applied in the Xilinx Design Constraints (XDC) file
 - The interpretation of language during simulation by different simulators
 - Dual port RAM collisions
 - Missing, or improperly applied timing constraints
 - Operation of asynchronous paths
 - Functional issues due to optimization techniques
-

Language and Encryption Support

The Vivado simulator supports:

- VHDL, see [IEEE Standard VHDL Language Reference Manual \(IEEE-STD-1076-1993\)](#) and part of [VHDL-2008](#).
- Verilog, see [IEEE Standard Verilog Hardware Description Language \(IEEE-STD-1364-2001\)](#).
- SystemVerilog, see [IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language \(IEEE-STD-1800-2009\)](#).
- IEEE P1735 encryption, see [Recommended Practice for Encryption and Management of Electronic Design Intellectual Property \(IP\) \(IEEE-STD-P1735\)](#).

Preparing for Simulation

This chapter describes the components that you need when you simulate a Xilinx® device in the Vivado® Integrated Design Environment (IDE).

Set up the following before performing the simulation:

- Create a test bench that reflects the simulation actions you want to run.
- Set up an install location in Vivado IDE (if not using the Vivado simulator).
- Compile your libraries (if not using the Vivado simulator).
- Select and declare the libraries you need to use.
- Specify the simulation settings such as target simulator, the simulation top module name, top module (design under test), display the simulation set, and define the compilation, elaboration, simulation, netlist, and advanced options.
- Generate a Netlist (if performing post-synthesis or post-implementation simulation).

Using Test Benches and Stimulus Files

A test bench is Hardware Description Language (HDL) code written for the simulator that:

- Instantiates and initializes the design.
- Generates and applies stimulus to the design.
- Monitors the design output result and checks for functional correctness (optional).

You can also set up the test bench to display the simulation output to a file, a waveform, or to a display screen. A test bench can be simple in structure and can sequentially apply stimulus to specific inputs.

A test bench can also be complex, and can include:

- Subroutine calls
- Stimulus that is read in from external files
- Conditional stimulus
- Other more complex structures

The advantages of a test bench over interactive simulation are that it:

- Allows repeatable simulation throughout the design process
- Provides documentation of the test conditions

The following bullets are recommendations for creating an effective test bench.

- Always specify the `timescale in Verilog test bench files. For example:

```
`timescale 1ns/1ps
```

- Initialize all inputs to the design within the test bench at simulation time zero to properly begin simulation with known values.
- Apply stimulus data after 100 ns to account for the default Global Set/Reset (GSR) pulse used in functional and timing-based simulation.
- Begin the clock source before the Global Set/Reset (GSR) is released.

For more information about test benches, see *Writing Efficient Test Benches* ([XAPP199](#)).



TIP: When you create a test bench, remember that the GSR pulse occurs automatically in the post-synthesis and post-implementation timing simulation. This holds all registers in reset for the first 100 ns of the simulation.

Related Information

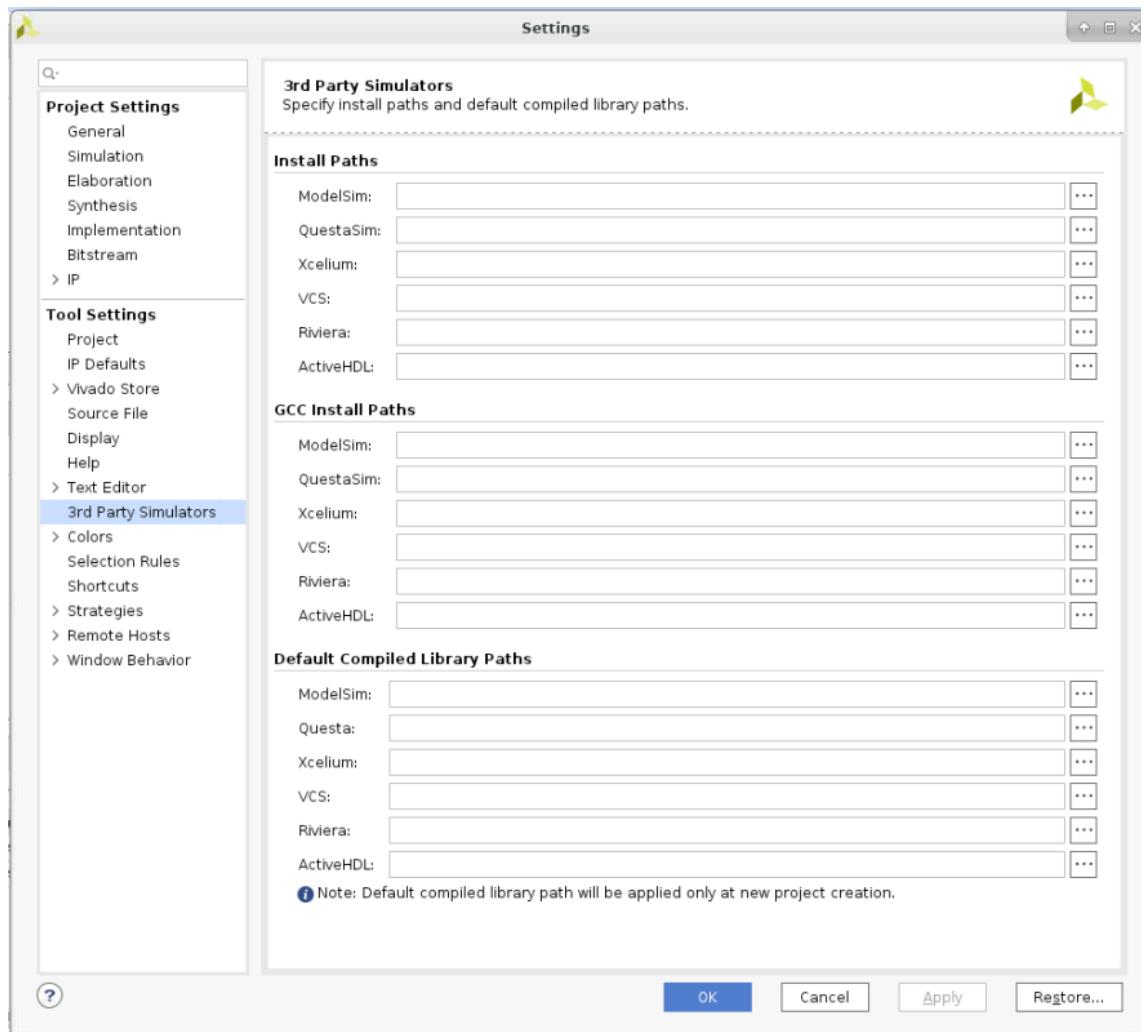
[Using Global Reset and 3-State](#)

Pointing to the Simulator Install Location

To define the installation path:

1. Select **Tools** → **Settings** → **Tool Settings** → **3rd Party Simulators**.
2. In Third-Party simulators tab of the Settings dialog box, select the simulator under the Install Paths as shown in the following figure, and browse to the installation path.
3. Select the appropriate simulator under **Default Compiled Library Paths** and browse to the relevant compiled library paths. You can set the library paths at a later point of time. See [Compiling Simulation Libraries](#) for more information on how to compile libraries for your simulator.

Note: Installing Vivado simulator is part of Vivado IDE Installation. Hence, you do not need to setup an install location for Vivado simulator.



Compiling Simulation Libraries



IMPORTANT! With Vivado simulator, there is no need to compile the simulation libraries. However, you must compile the libraries when using a third-party simulator.

The Vivado Design Suite provides simulation models as a set of files and libraries. Your simulation tool must compile these files prior to design simulation. The simulation libraries contain the device and IP behavioral and timing models. The compiled libraries can be used by multiple design projects.

During the compilation process, Vivado creates a default initialization file that the simulator uses to reference the compiled libraries. The `compile_simlib` command creates the file in the library output directory specified during library compilation. The default initialization file contains control variables that specify reference library paths, optimization, compiler, and simulator settings. If the correct initialization file is not found in the path, you cannot run simulation on designs that include Xilinx primitives.

The name of the initialization file varies depending on the simulator you are using, as follows:

- Questa Advanced Simulator/ModelSim: `modelsim.ini`
- Xcelium: `cds.lib`
- VCS: `synopsys_sim.setup`
- Riviera/Active-HDL: `library.cfg`

For more information on the simulator-specific compiled library file, see the third-party simulation tool documentation.



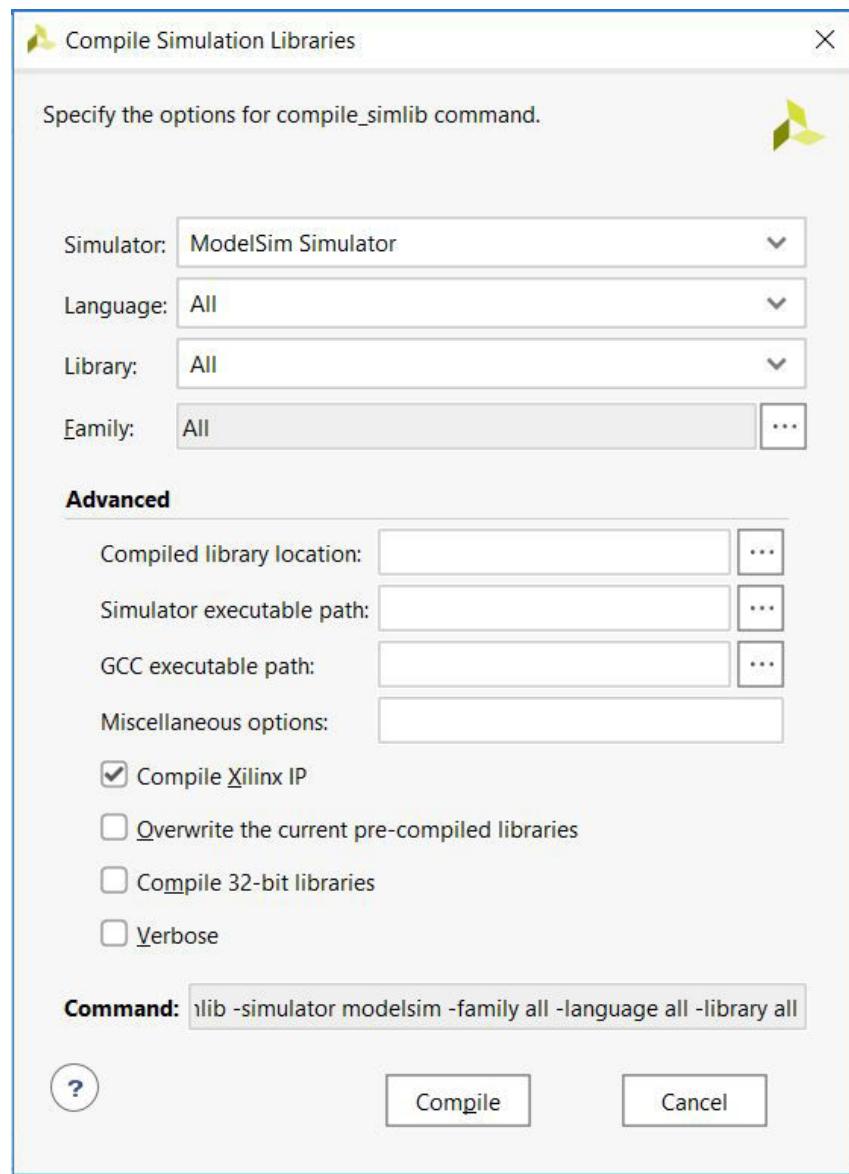
IMPORTANT! *Compilation of the libraries is typically a one-time operation, as long as you are using the same version of tools. However, any change to the Vivado tools or the simulator versions requires that libraries be recompiled.*

You can compile libraries using the Vivado IDE or using Tcl commands, as described in the following sections.

Compiling Simulation Libraries Using Vivado IDE

Select **Tools**→**Compile Simulation Libraries** to open the dialog box shown in the following figure.

Figure 2: Compile Simulation Libraries Dialog Box



Set the following options:

- **Simulator:** From the simulator drop-down menu, select a simulator.
- **Language:** Compiles libraries for the specified language. If this option is not specified, then the language is set to correspond with the selected simulator (above). For multi-language simulators, both Verilog and VHDL libraries are compiled.
- **Library:** Specifies the simulation library to compile. By default, the `compile_simlib` command compiles all simulation libraries.
- **Family:** Compiles selected libraries to the specified device family. All device families are generated by default.

- **Compiled library location:** Specifies the directory path for saving the compiled library results. By default, the libraries are saved in the current working directory in Non-Project mode, and the libraries are saved in the `<project>/<project>.cache/compile_simlib` directory in Project mode. See the *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#)) for more information on Project and Non-Project modes.



TIP: Because the Vivado simulator has pre-compiled libraries, it is not necessary to identify the library location.

- **Simulator executable path:** Specifies the directory to locate the simulator executable. This option is required if the target simulator is not specified in the `$PATH` or `%PATH%` environment variable, or to override the path from the `$PATH` or `%PATH%` environment variable.
- **GCC executable path:** Specifies the directory to locate GCC installation. This option is required if GCC path settings are not done as mentioned in [GCC Path Settings](#). Ignore if you are not using SystemC IP.
- **Miscellaneous Options:** Specify additional options for the `compile_simlib` Tcl command.
- **Compile Xilinx IP:** Enable or disable compiling simulation libraries for Xilinx IP.
- **Overwrite current pre-compiled libraries:** Overwrites the current pre-compiled libraries.
- **Compile 32-bit libraries:** Performs simulator compilation in 32-bit mode instead of the default 64-bit compilation.
- **Verbose:** Temporarily overrides any message limits and return all messages from this command.
- **Command:** Shows the Tcl command equivalent for the options you enter in the dialog box.



TIP: You can use the value of the `Command` field to generate a simulation library in Tcl/non-project mode.

Compiling Simulation Libraries Using Tcl Commands

Alternatively, you can compile simulation libraries using the `compile_simlib` Tcl command. For details, see `compile_simlib` in the *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#)), or type `compile_simlib -help`.

Following are example commands for each third-party simulator:

- **Questa Advanced Simulator:** Generating a simulation library for Questa for all languages and for all libraries and all families in the current directory.

```
compile_simlib -language all -simulator questa -library all -family all
```

- **ModelSim:** Generating simulation library for ModelSim at /a/b/c, where the ModelSim executable path is <simulator_installation_path>.

```
compile_simlib -language all -dir {/a/b/c} -simulator modelsim -  
simulator_exec_path  
{<simulator_installation_path>} -library all -family all
```

- **VCS:** Generating a simulation library for VCS for the Verilog language, for the UNISIM library at /a/b/c.

```
compile_simlib -language verilog -dir {/a/b/c} -simulator vcs_mx -library  
unisim  
-family all
```

- **Xcelium:** Generating a simulation library for Xcelium for the Verilog language, for the UNISIM library at /a/b/c.

```
compile_simlib -language verilog -dir {/a/b/c} -simulator xcelium -  
library unisim  
-family all
```

Changing compile_simlib Defaults

The `config_compile_simlib` Tcl command lets you configure third-party simulator options for use by the `compile_simlib` command.

Tcl Command

```
config_compile_simlib [-cfgopt <arg>] [-simulator <arg>] [-reset] [-quiet]  
[-verbose]
```

Where:

- `-cfgopt <arg>`: Configuration option in form of <simulator>:<language>:<library>:<options>.
- `-simulator`: The name of the simulator whose configuration you want
- `-reset`: Lets you reset all previous configurations for the specified simulator
- `-quiet`: Executes the command without any display to the Tcl Console.
- `-verbose`: Executes the command with all command output to the Tcl Console.

For example, to change the option used to compile the UNISIM VHDL library, type:

```
config_compile_simlib {cxl.modelsim.vhdl.unisim:-source -93 -novopt}
```



IMPORTANT! The `compile_simlib` command compiles Xilinx primitives and Simulation models of Xilinx Vivado IP. Xilinx Vivado IP cores are delivered as an output product when the IP is generated; consequently they are included in the pre-compiled libraries created using `compile_simlib`.

Compiling Patched IP Repository in a New Output Directory Using MYVIVADO

Assume that the patched IP repository is at the following location:

```
'/test/patched_ip_repo/data/ip/xilinx'
```

To compile the default installed IP repository and the repository that is pointed to by MYVIVADO in a new output directory, set the MYVIVADO environment (env) variable to point to this patched IP repository and run `compile_simlib`. `compile_simlib` will process the IP library sources from the default installed repository and the one set by MYVIVADO.

```
% setenv MYVIVADO /test/patched_ip_repo  
% compile_simlib -simulator <simulator> -directory <new_clibs_dir>
```

Compiling Patched IP Repository in an Existing Output Directory Using MYVIVADO

Assume that the patched IP repository is at the following location:

```
'/test/patched_ip_repo/data/ip/xilinx'
```

To compile the repository pointed to by MYVIVADO in an existing output directory where the library was already compiled for the default installed IP repository, set the MYVIVADO env variable to point to this patched IP repository and run `compile_simlib`. `compile_simlib` will process the IP library sources from the repository set by MYVIVADO in the existing output directory.

```
% setenv MYVIVADO /test/patched_ip_repo  
% compile_simlib -simulator <simulator> -directory <existing_clibs_dir>
```

Using Xilinx Simulation Libraries

You can use Xilinx simulation libraries with any simulator that supports the VHDL-93 and Verilog-2001 language standards. Certain delay and modeling information is built into the libraries; this is required to simulate the Xilinx hardware devices correctly.

Use non-blocking assignments for blocks within clocking edges. Otherwise, write code using blocking assignments in Verilog. Similarly, use variable assignments for local computations within a process, and use signal assignments when you want data-flow across processes.

If the data changes at the same time as a clock, it is possible that the simulator will schedule the data input to occur after the clock edge. The data does not go through until the next clock edge, although it is possible that the intent was to have the data clocked in before the first clock edge.



RECOMMENDED: *To avoid such unintended simulation results, do not switch data signals and clock signals simultaneously.*

When you instantiate a component in your design, the simulator must reference a library that describes the functionality of the component to ensure proper simulation. The Xilinx libraries are divided into categories based on the function of the model.

The following table lists the Xilinx-provided simulation libraries:

Table 2: Simulation Libraries

Library Name	Description	VHDL Library Name	Verilog Library Name
UNISIM	Functional simulation of Xilinx primitives.	UNISIM	UNISIMS_VER
UNIMACRO	Functional simulation of Xilinx macros.	UNIMACRO	UNIMACRO_VER
UNIFAST	Fast simulation library.	UNIFAST	UNIFAST_VER
SIMPRIM	Timing simulation of Xilinx primitives.	N/A	SIMPRIMS_VER ¹
SECUREIP	Simulation library for both functional and timing simulation of Xilinx device features, such as the PCIe IP, Gigabit Transceiver etc. You can find the list of IP's under SECUREIP at the following location: <Vivado_Install_Dir>/data/secureip	SECUREIP	SECUREIP
XPM	Functional simulation of Xilinx primitives	XPM	XPM ²

Notes:

1. The SIMPRIMS_VER is the logical library name to which the Verilog SIMPRIM physical library is mapped.
2. XPM is supported as a pre-compiled IP. Hence, you need not add the source file to the project. For third party simulators, the Vivado tools will map to pre-compiled IP generated with `compile_simlib`.



IMPORTANT! You must specify different simulation libraries according to the simulation points. There are different gate-level cells in pre- and post-implementation netlists.

The following table lists the required simulation libraries at each simulation point.

Table 3: Simulation Points and Relevant Libraries

Simulation Point	UNISIM	UNIFAST	UNIMACRO	SECUREIP	SIMPRIM (Verilog Only)	SDF
1. Register Transfer Level (RTL) (Behavioral)	Yes	Yes	Yes	Yes	N/A	No
2. Post-Synthesis Simulation (Functional)	Yes	Yes	N/A	Yes	N/A	N/A
3. Post-Synthesis Simulation (Timing)	N/A	N/A	N/A	Yes	Yes	Yes
4. Post-Implementation Simulation (Functional)	Yes	Yes	N/A	Yes	N/A	N/A
5. Post-Implementation Simulation (Timing)	N/A	N/A	N/A	Yes	Yes	Yes



IMPORTANT! The Vivado simulator uses precompiled simulation device libraries. When updates to libraries are installed the precompiled libraries are automatically updated.

Note: Verilog SIMPRIMS_VER uses the same source as UNISIM with the addition of specify blocks for timing annotation. SIMPRIMS_VER is the logical library name to which the Verilog physical SIMPRIM is mapped.

The following table lists the library locations.

Table 4: Simulation Library Locations

Library	HDL Type	Location
UNISIM	Verilog	<Vivado_Install_Dir>/data/verilog/src/unisims
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unisims
UNIFAST	Verilog	<Vivado_Install_Dir>/data/verilog/src/unifast
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unifast
UNIMACRO	Verilog	<Vivado_Install_Dir>/data/verilog/src/unimacro
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unimacro
SECUREIP	Verilog	<Vivado_Install_Dir>/data/secureip/

The following subsections describe the libraries in more detail.

UNISIM Library

Functional simulation uses the UNISIM library and contains descriptions for device primitives or lowest-level building blocks.



IMPORTANT! By default, the `compile_simlib` command compiles the static simulation files for all the IP's in the IP Catalog.

Encrypted Component Files

The following table lists the UNISIM library component files that let you call precompiled, encrypted library files when you include IP in a design. Include the path you require in your library search path.

Table 5: Component Files

Component File	Description
<Vivado_Install_Dir>/data/verilog/src/unisim_retarget_comp.vp	Encrypted Verilog file
<Vivado_Install_Dir>/data/vhdl/src/unisims/unisim_retarget_VCOMP.vhdp	Encrypted VHDL file



IMPORTANT! Verilog module names and file names are uppercase. For example, module BUFG is `BUFG`. `v`, and module IBUF is `IBUF`. `v`. Ensure that UNISIM primitive instantiations adhere to an uppercase naming convention.

VHDL UNISIM Library

The VHDL UNISIM library is divided into the following files, which specify the primitives for the Xilinx device families:

- The component declarations (`unisim_VCOMP.vhd`)
- Package files (`unisim_PKG.vhd`)

To use these primitives, place the following two lines at the beginning of each file:

```
library UNISIM;
use UNISIM.VCOMPONENTS.all;
```



IMPORTANT! You must also compile the library and map the library to the simulator. The method depends on the simulator.

Note: For Vivado simulator, the library compilation and mapping is an integrated feature with no further user compilation or mapping required.

Note: Starting in Versal™ ACAP, Xilinx is delivering Verilog/SystemVerilog models only for the new primitives. This does mean that a mixed-language environment is needed for VHDL-only designs, like what has been needed in the past for IPs and XPMs. For more information, see [AR76496](#).

Verilog UNISIM Library

In Verilog, the individual library modules are specified in separate HDL files. This allows the `-y` library specification switch to search the specified directory for all components and automatically expand the library.

The Verilog UNISIM library cannot be specified in the HDL file prior to using the module. To use the library module, specify the module name using all uppercase letters.

The following example shows the instantiated module name as well as the file name associated with that module:

- Module BUFG is `BUFG.v`
- Module IBUF is `IBUF.v`

Verilog is case-sensitive, ensure that UNISIM primitive instantiations adhere to an uppercase naming convention.

If you use precompiled libraries, use the correct simulator command-line switch to point to the precompiled libraries. The following is an example for the Vivado simulator:

```
-L unisims_ver
```

Where:

`-L` is the library specification option.

UNIMACRO Library

The UNIMACRO library is used during functional simulation and contains macro descriptions for selected device primitives.



IMPORTANT! You must specify the UNIMACRO library anytime you include a device macro listed in the Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide ([UG953](#)).

VHDL UNIMACRO Library

To use these primitives, place the following two lines at the beginning of each file:

```
library UNIMACRO;
use UNIMACRO.Vcomponents.all;
```

Verilog UNIMACRO Library

In Verilog, the individual library modules are specified in separate HDL files. This allows the `-y` library specification switch to search the specified directory for all components and automatically expand the library.

The Verilog UNIMACRO library does not need to be specified in the HDL file prior to using the modules as is required in VHDL. To use the library module, specify the module name using all uppercase letters. You must also compile and map the library; the method you use depends on the simulator you choose.



IMPORTANT! Verilog module names and file names are uppercase. For example, module BUFG is `BUFG`. v. Ensure that UNIMACRO primitive instantiations adhere to an uppercase naming convention.

SIMPRIM Library

Use the SIMPRIM library for simulating timing simulation netlists produced after synthesis or implementation.



IMPORTANT! Timing simulation is supported in Verilog only; there is no VHDL version of the SIMPRIM library.



TIP: If you are a VHDL user, you can run post synthesis and post implementation functional simulation (in which case no standard default format (SDF) annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the `write_vhdl` Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide ([UG835](#)).

Following is an example for specifying the library for Vivado simulator:

```
-L SIMPRIMS_VER
```

Where:

- `-L` is the library specification option.
- `SIMPRIMS_VER` is the logical library name to which the Verilog SIMPRIM has been mapped.

SECUREIP Simulation Library

Use the SECUREIP library for functional and timing simulation of complex device components, such as GT.

Note: Secure IP Blocks are fully supported in the Vivado simulator without additional setup.

Xilinx leverages the encryption methodology as specified in the IEEE standard *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)* (IEEE-STD-P1735). The library compilation process automatically handles encryption.

Note: See the simulator documentation for the command line switch to use with your simulator to specify libraries.

The following table lists special considerations that must be arranged with your simulator vendor for using these libraries.

Table 6: Special Considerations for Using SECUREIP Libraries

Simulator Name	Vendor	Requirements
Siemens EDA ModelSim SE	Siemens	If design entry is in VHDL, a mixed language license or a SECUREIP OP is required. Contact the vendor for more information.
Siemens EDA Questa Advanced Simulator		
VCS	Synopsys	
Active-HDL	Aldec	If design entry is VHDL, a mixed language license is required.
Riviera-PRO*		



IMPORTANT! See *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)* for the supported version of third-party simulators.

VHDL SECUREIP Library

The UNISIM library contains the wrappers for VHDL SECUREIP. Place the following two lines at the beginning of each file so that the simulator can bind to the entity:

```
Library UNISIM;
UNISIM.VCOMPONENTS.all;
```

Verilog SECUREIP Library

When running a simulation using Verilog code, you must reference the SECUREIP library for most simulators.

If you use the precompiled libraries, use the correct directive to point to the precompiled libraries. The following is an example for the Vivado simulator:

```
-L SECUREIP
```



IMPORTANT! You can use the Verilog SECUREIP library at compile time by using `-f` switch. The file list is available in the following path: `<Vivado_Install_Dir>/data/secureip/secureip_cell.list.f`.

UNIFAST Library

The UNIFAST library is an optional library that you can use during RTL behavioral simulation to speed up simulation runtime.



IMPORTANT! The UNIFAST library is an optional library that you can use during functional simulation to speed up simulation runtime. UNIFAST libraries are supported for 7 series devices only. UltraScale and later device architectures do not support UNIFAST libraries, as all the optimizations are incorporated in the UNISIM libraries by default. UNIFAST libraries cannot be used for sign-off simulations because the library components do not have all the checks/features that are available in a full model.



RECOMMENDED: Use the UNIFAST library for initial verification of the design and then run a complete verification using the UNISIM library.

The simulation run time improvement is achieved by supporting a subset of the primitive features in the simulation mode.

Note: The simulation models check for unsupported attribute values only.

MMCME2

To reduce the simulation runtimes, the fast MMCME2 simulation model has the following changes from the full model:

1. The fast simulation model provides only basic clock generation functions. Other functions, such as DRP, fine phase shifting, clock stopped, and clock cascade are not supported.
2. It assumes that input clock is stable without frequency and phase change. The input clock frequency sampling stops after LOCKED signal is asserted HIGH.
3. The output clock frequency, phase, duty cycle, and other features are directly calculated from input clock frequency and parameter settings.

Note: The output clock frequency is not generated from input-to-VCO clock.

4. The standard and the fast MMCME2 simulation model LOCKED signal assertion times differ.
 - Standard Model LOCKED assertion time depends on the M and D setting. For large M and D values, the lock time is relatively long for a standard MMCME2 simulation model.
 - In the fast simulation model, the LOCKED assertion time is shortened.

DSP48E1

To reduce the simulation runtimes, the fast DSP48E1 simulation model has the following features removed from the full model.

- Pattern Detection
- OverFlow/UnderFlow
- DRP interface support

GTHE2_CHANNEL/GTHE2_COMMON

To reduce the simulation runtimes, the fast GTHE2 simulation model has the following feature differences:

- GTH links must be synchronous with no Parts Per Million (PPM) rate differences between the near and far end link partners.
- Latency through the GTH is not cycle accurate with the hardware operation.
- You cannot simulate the DRP production reset sequence. Bypass it when using the UNIFAST model.

Using Verilog UNIFAST Library

To reduce the simulation runtimes, the fast GTXE2 simulation model has the following feature differences:

- GTX links must be of synchronous with no Parts Per Million (PPM) rate differences between the near and far end link partners.
- Latency through the GTX is not cycle accurate with the hardware operation.

Method 1: Using the complete Verilog UNIFAST library (Recommended)

Method 1 is the recommended method whereby you simulate with all the UNIFAST models.

Use the following Tcl command in Tcl console to enable UNIFAST support (fast simulation models) in a Vivado project environment for the Vivado simulator, ModelSim or VCS:

```
set_property unifast true [current_fileset -simset]
```

See the [UNISIM Library](#) for more information regarding component files.

For more information, see the appropriate third-party simulation user guide.

Method 2: Using specific UNIFAST modules

Recommended for more advanced users who want to specify which modules to simulate with the UNIFAST models.

To specify individual library components, Verilog configuration statements are used. Specify the following in the `config.v` file:

- The name of the top-level module or configuration (for example: `config cfg_xilinx;`)
- The name to which the design configuration applies (for example: `design test_bench;`)
- The library search order for cells or instances that are not explicitly called out (for example: `default liblist unisims_ver unifast_ver;`)
- The map for a particular CELL or INSTANCE to a particular library (For example: `instance testbench.inst.O1 use unifast_ver.MMCME2;`)

Note: For ModelSim (vsim) only `-genblk` is added to hierarchy name (for example: `instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.MMCME2; - VSIM`).

Example `config.v`

```
config cfg_xilinx;
design testbench;
default liblist unisims_ver unifast_ver;
//Use fast MMCM for all MMCM blocks in design
cell MMCME2 use unifast_ver.MMCME2;
//use fast dSO48E1 for only this specific instance in the design
instance testbench.inst.O1 use unifast_ver.DSP48E1;
//If using ModelSim or Questa, add in the genblk to the name
(instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.DSP48E1)
endconfig
```

Using VHDL UNIFAST Library

The VHDL UNIFAST library has the same basic structure as Verilog and can be used with architectures or libraries. You can include the library in the test bench file.

The following example uses a *drill-down* hierarchy with a `for` call:

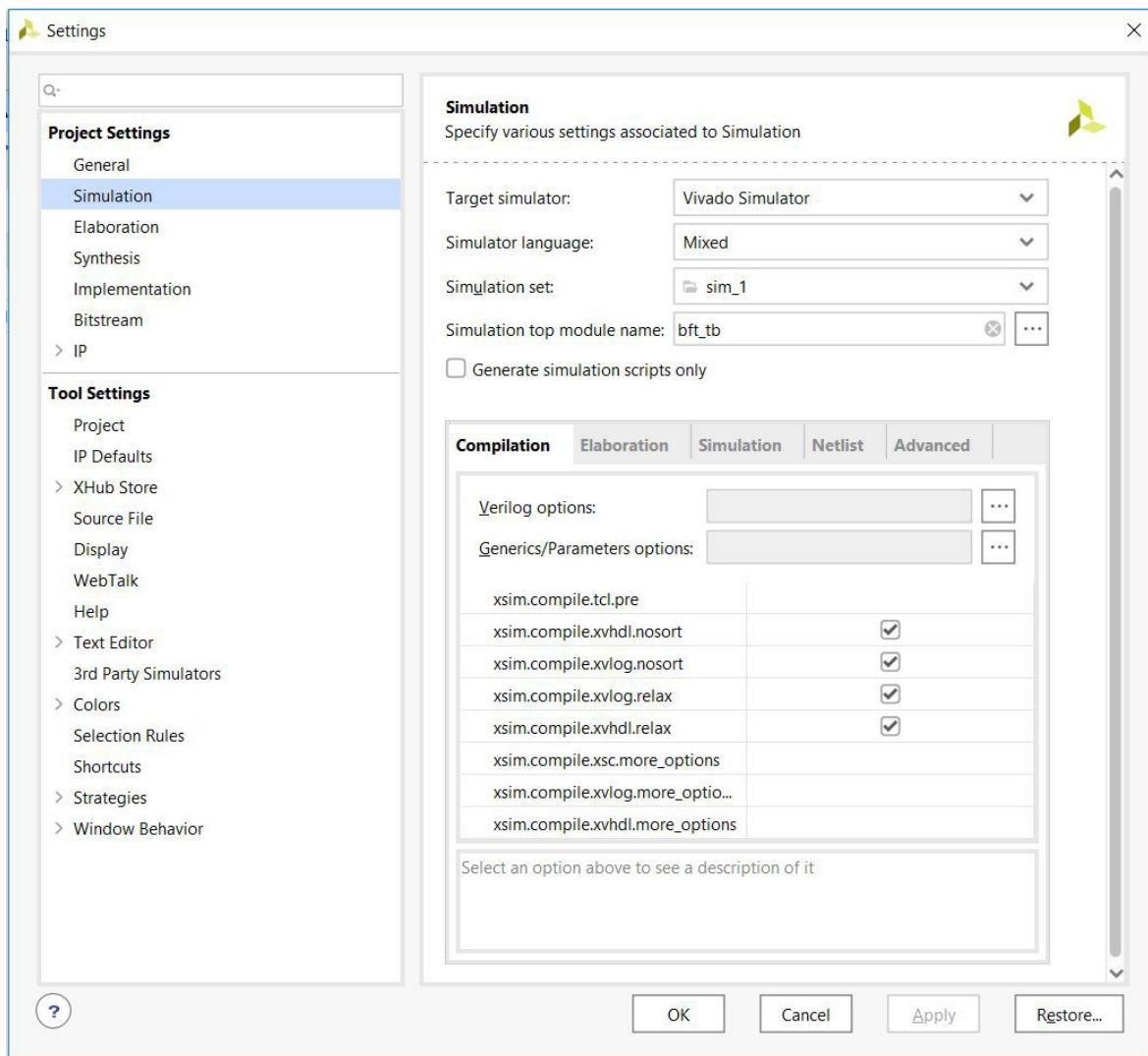
```
library unisim;
library unifast;
configuration cfg_xilinx of testbench
is for xilinx
.. for inst:netlist
... . use entity work.netlist(inst);
.....for inst
.....for all:MMCME2
.....use entity unifast.MMCME2;
.....end for;
.....for O1 inst:DSP48E1;
.....use entity unifast.DSP48E1;
.....end for;
...end for;
..end for;
end for;
end cfg_xilinx;
```

Note: If you want to use a VHDL UNIFAST model, you have to use a configuration to bind the UNIFAST library during elaboration.

Using Simulation Settings

You can use the simulation settings to specify the target simulator, display the simulation set, the simulation top module name, top module (design under test), tabbed listing of compilation, elaboration, simulation, netlist, and advanced options. From the Vivado IDE Flow Navigator, right-click **Simulation** and select **Simulation Settings** to open the Simulation Settings in the Settings dialog box, as shown in the following figure.

Figure 3: Settings Dialog Box



The Settings dialog box includes the following simulation settings:

- **Target simulator:** From the simulator drop-down menu, select a simulator. Vivado® simulator is the default simulator. However, many third-party simulators are also supported.

- **Simulator language:** Select the simulator language mode. The simulation model used for various IPs in your design varies depending on what language the IP supports.
- **Simulation set:** Select the simulation set that the simulation commands use by default.



IMPORTANT! *The compilation and simulation settings for a previously defined simulation set are not applied to a newly-defined simulation set.*

- **Simulation top module name:** Enter an alternate top module to use during simulation.
- **Generate simulation scripts only:** Generates scripts if selected. Simulation is not invoked.
- **Compiled library location:** This option is displayed when you select a third party simulator. This is a directory path for saving the compiled library results. By default, the libraries are saved in the current working directory in Non-Project mode. The libraries are saved in the `<project>/<project>.cache/compile_simlib` directory in project mode.
- **Compilation tab:** This tab defines and manages compiler directives, which are stored as properties on the simulation fileset and used by the `xvlog` and `xvhdl` utilities to compile Verilog and VHDL source files for simulation.

Note: `xvlog` and `xvhdl` are Vivado simulator specific commands. The applicable utilities will change based on the target simulator.

- **Elaboration tab:** This tab defines and manages elaboration directives, which are stored as properties on the simulation fileset and used by the `xelab` utility for elaborating and generating a simulation snapshot. Select a property in the table to display a description of the property and edit the value.

Note: `xelab` is a Vivado simulator specific command. The applicable utilities will change based on the target simulator.

- **Simulation tab:** This tab defines and manages simulation directives, which are stored as properties on the simulation fileset and used by the `xsim` application for simulating the current project. Select a property in the table to display a description of the property and edit the value.
- **Netlist tab:** This tab provides access to netlist configuration options related to SDF annotation of the Verilog netlist and the process corner captured by SDF delays. These options are stored as properties on the simulation fileset and are used while writing the netlist for simulation.
- **Advanced tab:** This tab contains two options:
 - **Enable incremental compilation:** This option enables the incremental compilation and preserves the simulation files during successive run.
 - **Include all design sources for simulation:** By default, this option is enabled. Selecting this option ensures that all the files from design sources along with the files from the current simulation set will be used for simulation. Even if you change the design sources, the same changes will be updated when you launch behavioral simulation.



CAUTION! Changing the settings in the **Advanced** tab should be done only if necessary. The **Include all design sources for simulation** check box is selected by default. Deselecting the box could produce unexpected results. As long as the check box is selected, the simulation set includes Out-of-Context (OOC) IP, IP Integrator files, and DCP.

Note: For detailed information on the properties in the Compilation, Elaboration, Simulation, Netlist, and Advanced tabs, see [Appendix A: Compilation, Elaboration, Simulation, Netlist, and Advanced Options](#).

Understanding the Simulator Language Option

Most Xilinx IP deliver behavioral simulation models for a single language only, effectively disabling simulation for language-locked simulators if you are not licensed for the appropriate language. The `simulator_language` property ensures that an IP delivers a simulation model for any given language. For example, if you are using a single language simulator, you set the `simulator_language` property to match the language of the simulator.

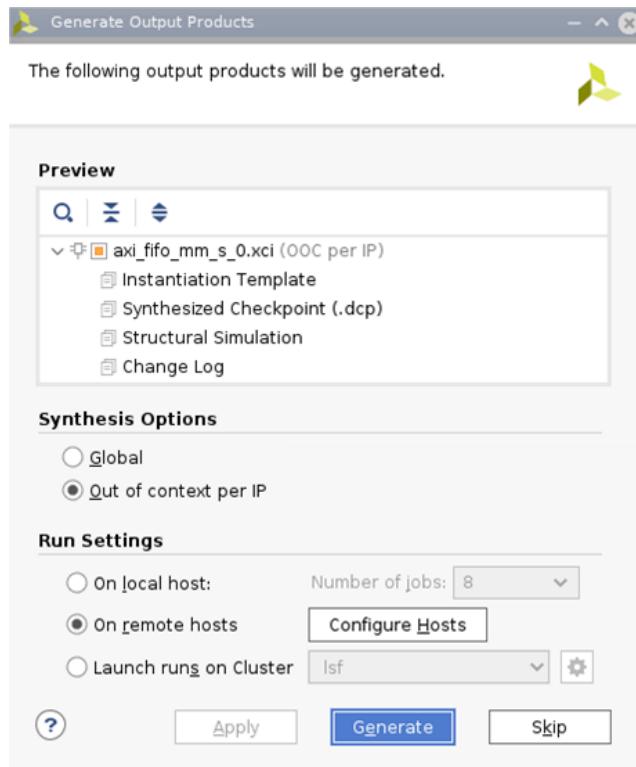
The Vivado Design Suite ensures the availability of a simulation model by using the available synthesis files of an IP to generate a language-specific structural simulation model on demand. For cases in which a behavioral model is missing or does not match the licensed simulation language, the Vivado tools automatically generate a structural simulation model to enable simulation. Otherwise, the existing behavioral simulation model for the IP is used. If no synthesis or simulation files exist, simulation is not supported.

Note: The `simulator_language` property cannot deliver a language-specific simulation netlist file if the generated Synthesized checkpoint (.dcp) is disabled.

1. In the Flow Navigator, click **IP Catalog** to open the IP Catalog.
2. Right-click the appropriate IP and select **Customize IP** from the popup menu.
3. In the Customize IP dialog box, click **OK**.

The Generate Output Products dialog box (shown in the following figure) opens.

Figure 4: Generate Output Products Dialog Box



The following table illustrates the function of the `simulator_language` property.

Table 7: Function of `simulator_language` Property

IP Delivered Simulation Model	<code>simulator_language</code> Value	Simulation Model Used
IP delivers VHDL and Verilog behavioral models	Mixed	Behavioral model (<code>target_language</code>)
	Verilog	Verilog behavioral model
	VHDL	VHDL behavioral model
IP delivers Verilog behavioral model only	Mixed	Verilog behavioral model
	Verilog	Verilog behavioral model
	VHDL	VHDL simulation netlist generated from DCP
IP delivers VHDL behavioral model only	Mixed	VHDL behavioral model
	Verilog	Verilog simulation netlist generated from DCP
	VHDL	VHDL behavioral model
IP delivers no behavioral models	Mixed, Verilog, VHDL	Netlist generated from DCP (<code>target_language</code>)

Notes:

1. Where available, behavioral simulation models always take precedence over structural simulation models. The Vivado tools select behavioral or structural models automatically, based on model availability. It is not possible to override the automated selection.
2. Use the `target_language` property when either language can be used for simulation Tcl: `set_property target_language VHDL [current_project]`

Setting the Simulation Runtime Resolution

Set the simulation run-time resolution using `timescale in test bench. There is no simulator performance gain achieved through use of coarser resolution with the Xilinx simulation models. (In Xilinx simulation models, most simulation time is spent in delta cycles, and delta cycles are not affected by simulator resolution.)



IMPORTANT! Run simulations using a time resolution of 1 fs. Some Xilinx primitive components, such as GT, require a 1 fs resolution to work properly in either functional or timing simulation.

See [Simulation Options](#) for detailed information on Simulation Options in Settings dialog box.



IMPORTANT! Picoseconds are used as the minimum resolution because testing equipment can measure timing only to the nearest picosecond resolution.

Adding or Creating Simulation Source Files

To add simulation sources to a Vivado Design Suite project:

1. Select **File**→**Add Sources**, or click **Add Sources** in the Flow Navigator.

The Add Sources wizard opens.

2. Select **Add or Create Simulation Sources**, and click **Next**.

The Add or Create Simulation Sources dialog box opens. The options are:

- **Add Files:** Invokes a file browser so you can select simulation source files to add to the project.
- **Add Directories:** Invokes directory browser to add all simulation source files from the selected directories. Files in the specified directory with valid source file extensions are added to the project.
- **Create File:** Invokes the Create Source File dialog box where you can create new simulation source files. See this [link](#) in the *Vivado Design Suite User Guide: System-Level Design Entry (UG895)* for more information about project source files.
- Buttons on the side of the dialog box let you do the following:
 - **Remove** : Removes the selected source files from the list of files to be added.
 - **Move Up** : Moves the file up in the list order.
 - **Move Down** : Moves the file down in the list order.

- Check boxes in the wizard provide the following options:
 - - **Scan and add RTL include files into project:** Scans the added RTL file and adds any referenced include files.
 - **Copy sources into project:** Copies the original source files into the project and uses the local copied version of the file in the project.

If you elected to add directories of source files using the Add Directories command, the directory structure is maintained when the files are copied locally into the project.

- **Add sources from subdirectories:** Adds source files from the subdirectories of directories specified in the Add Directories option.
- **Include all design sources for simulation:** Includes all the design sources for simulation.



VIDEO: For a demonstration of this feature, see the [Vivado Design Suite QuickTake Video: Logic Simulation](#).

Working with Simulation Sets

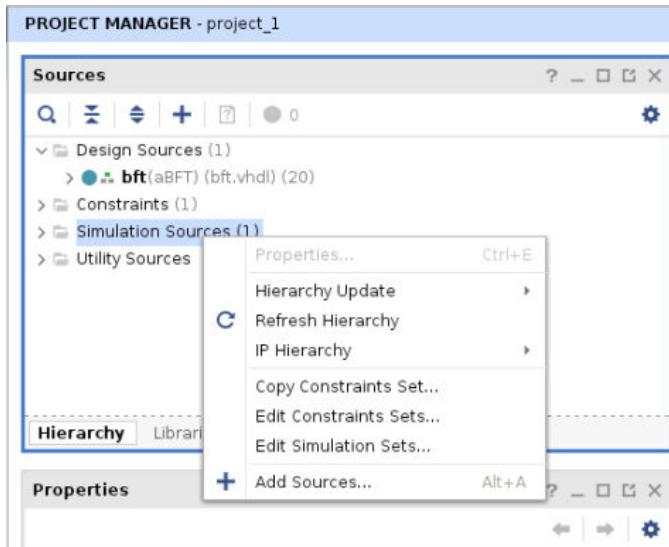
The Vivado IDE stores simulation source files in simulation sets that display in folders in the Sources window, and are either remotely referenced or stored in the local project directory.

The simulation set lets you define different sources for different stages of the design. For example, there can be one test bench source to provide stimulus for behavioral simulation of the elaborated design or a module of the design, and a different test bench to provide stimulus for timing simulation of the implemented design.

When adding simulation sources to the project, you can specify which simulation source set to use.

To edit a simulation set:

1. In the Sources window popup menu, right click **Simulation Sources** and click **Edit Simulation Sets**, as shown in the following figure.



The Add or Create Simulation Sources wizard opens.

2. From the Add or Create Simulation Sources wizard, select **Add Sources**.

This adds the sources associated with the project to the newly-created simulation set.

3. Add additional files as needed.

The selected simulation set is used for the *active* design run.

Generating a Netlist

To run simulation of a synthesized or implemented design run the netlist generation process. The netlist generation Tcl commands can take a synthesized or implemented design database and write out a single netlist for the entire design.

The Vivado Design Suite generates a netlist automatically when you launch the simulator using the IDE or the `launch_simulation` command.

Netlist generation Tcl commands can write SDF and the design netlist. The Vivado Design Suite provides the following Tcl commands:

- `write_verilog`: Verilog netlist
- `write_vhdl`: VHDL netlist
- `write_sdf`: SDF generation



TIP: The SDF values are only estimates early in the design process (for example, during synthesis). As the design process progresses, the accuracy of the timing numbers also progress when there is more information available in the database.

Generating a Functional Netlist

The Vivado Design Suite supports writing out a Verilog or VHDL structural netlist for functional simulation. The purpose of this netlist is to run simulation (without timing) to check that the behavior of the structural netlist matches the expected behavioral model (RTL) simulation.

The functional simulation netlist is a hierarchical, folded netlist that is expanded to the primitive module or entity level; the lowest level of hierarchy consists of primitives and macro primitives.

These primitives are contained in the following libraries:

- UNISIMS_VER simulation library for Verilog simulation
- UNISIMS simulation library for VHDL simulation

In many cases, you can use the same test bench that you used for behavioral simulation to perform a more accurate simulation.

The following Tcl commands generate Verilog and VHDL functional simulation netlist, respectively:

```
write_verilog -mode funcsim <Verilog_Netlist_Name.v>
write_vhdl -mode funcsim <VHDL_Netlist_Name.vhd>
```

Generating a Timing Netlist

You can use a Verilog timing simulation to verify circuit operation after the Vivado tools have calculated the worst-case placed and routed delays.

In many cases, you can use the same test bench that you used for functional simulation to perform a more accurate simulation.

Compare the results from the two simulations to verify that your design is performing as initially specified.

There are two steps to generating a timing simulation netlist:

1. Generate a simulation netlist file for the design.
2. Generate an SDF delay file with all the timing delays annotated.



IMPORTANT! Vivado IDE supports Verilog timing simulation only.



TIP: If you are a VHDL user, you can run post-synthesis and post-implementation functional simulation (in which case no standard default format (SDF) annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the [write_vhdl](#) Tcl command. For usage information, see the Vivado Design Suite Tcl Command Reference Guide ([UG835](#)).

The following is the Tcl syntax for generating a timing simulation netlist:

```
write_verilog -mode timesim -sdf_anno true <Verilog_Netlist_Name>
```

Using Versal CIPS VIP

The Versal™ ACAP Control, Interfaces, and Processing System (CIPS) Verification Intellectual Property (VIP) supports the functional simulation of Versal ACAP applications. It is targeted to enable the functional verification of the programmable logic (PL) by mimicking the processor system (PS)-PL interfaces and OCM memories of the PS logic. This VIP is delivered as a package of System Verilog modules. The VIP operation is controlled by using a sequence of System Verilog tasks. This is supported in the latest version of Vivado. For more information, see *Versal ACAP CIPS Verification IP Data Sheet* ([DS996](#)).

Simulating with Third-Party Simulators

The Vivado® Design Suite supports simulation using third-party tools. Simulation with third-party tools can be performed directly from within the Vivado Integrated Design Environment (IDE) or using a custom external simulation environment.

Table 8: Supported Third-Party Simulators

Third-party Simulators	Red Hat 64-bit Linux	Windows 10 64-bit
Siemens EDA ModelSim SE	Yes	Yes
Siemens EDA Questa Advanced Simulator	Yes	Yes
Cadence Xcelium Parallel Simulator	Yes	NA
Synopsys VCS	Yes	NA
Aldec Active HDL	NA	Yes
Aldec Riviera PRO	Yes	Yes

The *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#)) describes the use of the Vivado IDE.

Please set the following environment variables before running simulation in Vivado IDE.

Table 9: Environment Variable Setting for Third-Party Simulators

Simulator	Linux	Windows
Modelsim	<pre>setenv MODEL_TECH <tool installation path> setenv LM_LICENSE_FILE <license file> setenv PATH \${MODEL_TECH}/bin:\$PATH</pre>	<pre>set MODEL_TECH=<tool installation path> set LM_LICENSE_FILE=<license file> set Path=%MODEL_TECH%\win32;%Path%</pre>
Questa	<pre>setenv MODEL_TECH <tool installation path> setenv LM_LICENSE_FILE <license file> setenv PATH \${MODEL_TECH}/bin:\$PATH</pre>	<pre>set MODEL_TECH=<tool installation path> set LM_LICENSE_FILE=<license file> set Path=%MODEL_TECH%\win32;%Path%</pre>

Table 9: Environment Variable Setting for Third-Party Simulators (cont'd)

Simulator	Linux	Windows
Riviera	<pre>In BASH source <tool_install_path>/etc/setenv source <tool_install_path>/etc/setgcc</pre>	<pre><tool installation path>/etc/ setenv.bat set RIVIERA_BIN=<tool installation path> set Path=%<Riviera install dir>%\bin;%Path% set LM_LICENSE_FILE=<license file></pre>
Active-HDL	NA	<pre>set ACTIVE_BIN=<tool installation path> set Path=%<Active_hdl install dir>%\BIN;%Path% set LM_LICENSE_FILE=<license file></pre>
Xcelium	<pre>setenv CDS_INST_DIR <xcelium_install_dir> setenv LD_LIBRARY_PATH \$CDS_INST_DIR/tools/ xcelium/lib:\$LD_LIBRARY_PATH setenv PATH \$CDS_INST_DIR/tools/xcelium/ bin:\$CDS_INST_DIR/tools/bin:\$PATH setenv CDS_LICENSE_DIR <tool_license></pre>	NA
VCS	<pre>setenv VCS_HOME <tool_install_path> setenv LM_LICENSE_FILE <license_file_path> setenv PATH \${VCS_HOME}/bin:\${PATH}</pre>	NA

Notes:

1. Tool installation path should be added to environment variable `PATH` (irrespective of OS). To simulate SystemC based designs for the supported simulator, provide the required `g++` version installation path as mentioned in [Appendix F: SystemC Support in Vivado IDE](#). The `LD_LIBRARY_PATH` should also include simulator library path.

For links to more information on your third party simulator, see [Links to Additional Information on Third-Party Simulators](#).



IMPORTANT! Use only supported versions of third-party simulators. For more information on supported Simulators and Operating Systems, see the Compatible Third-Party Tools table in the Vivado Design Suite User Guide: Release Notes, Installation, and Licensing ([UG973](#)).

Running Simulation Using Third Party Simulators with Vivado IDE



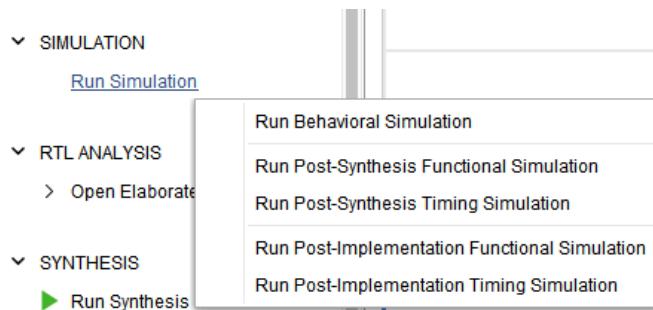
IMPORTANT! Confirm the compiled library location (the path at which `compile_simlib` was invoked or the one you specified with the `-directory` option) before running a third-party simulation.

From the Vivado IDE, you can compile, elaborate, and simulate the design based on the simulation settings and launch the simulator in a separate window.

When you run simulation prior to synthesizing the design, the simulator runs a behavioral simulation. Following each successful design step (synthesis and implementation), the option to run a functional or timing simulation becomes available. You can initiate a simulation run from the Flow Navigator or by typing in a Tcl command.

From the Flow Navigator, click **Run Simulation**, and select the type of simulation you want to run, as shown in the following figure:

Figure 5: Types of Simulation



To use the corresponding Tcl command, type: `launch_simulation`



TIP: This command provides a `-scripts_only` option that can be used to write a DO or SH file, depending on the target simulator. Use the DO or SH file to run simulations outside the IDE.

Note: If you are running VCS simulator outside of Vivado, make sure to use `-full64` switch. Otherwise, the simulator will not run if the design contains Xilinx IP.



IMPORTANT! Use the following command to run the 32-bit Simulator: `set_property 32bit 1 [current_fileset -simset]`

Note: Xilinx Verification IP (VIP) uses SystemVerilog construct. If you are using any IP which instantiates VIP, make sure that your simulator supports SystemVerilog.

Running Timing Simulation Using Third-Party Tools



TIP: Post-Synthesis timing simulation uses the estimated timing delay from the synthesized netlist. Post-Implementation timing simulation uses actual timing delays.

When you run Post-Synthesis and Post-Implementation timing simulation, the simulators include:

- Gate-level netlist containing SIMPRIMS library components
- SECUREIP
- Standard Delay Format (SDF) files

You define the overall design functionality in the beginning. When the design is implemented, accurate timing information is available.

To create the netlist and SDF, the Vivado Design Suite:

- Calls the netlist writer, `write_verilog` with the `-mode timesim` switch and `write_sdf` (SDF annotator)
- Sends the generated netlist to the target simulator

You control these options using Simulation Settings as described in [Using Simulation Settings](#).



IMPORTANT! Post-Synthesis and Post-Implementation timing simulations are supported for Verilog only. There is no support for VHDL timing simulation. If you are a VHDL user, you can run post-synthesis and post-implementation functional simulation (in which case no SDF annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the `write_vhdl` Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide ([UG835](#)).

Post-Synthesis Timing Simulation

When synthesis runs successfully, the **Run Simulation → Post-Synthesis Timing Simulation** option becomes available.

After you select a post-synthesis timing simulation, the timing netlist and the SDF file are generated. The netlist files includes `$sdf_annotate` command so that the generated SDF file is picked up.

Post-Implementation Timing Simulations

When post-implementation is successful, the **Run Simulation → Post-Implementation Timing Simulation** option becomes available.

After you select a post-implementation timing simulation, the timing netlist and the SDF file are generated. The netlist files includes `$sdf_annotate` command so that the generated SDF file is picked up.

Annotating the SDF File for Timing Simulation

When you specified simulation settings, you specified whether or not to create an SDF file and whether the process corner would be set to fast or slow.



TIP: To find the SDF file options settings, in the Vivado IDE Flow Navigator, right-click **Simulation** and select **Simulation Settings**. In the Settings dialog box, select **Simulation** category and click **Netlist** tab.

Based on the specified process corner, the SDF file contains different `min` and `max` numbers.



RECOMMENDED: To run a setup check, create an SDF file with `-process_corner slow`, and use the `max` column from the SDF file.

To run a hold check, create an SDF file with the -process_corner fast, and use the min column from the SDF file. The method for specifying which SDF delay field to use is dependent on the simulation tool you are using. Refer to the specific simulation tool documentation for information on how to set this option.

To get full coverage run all four timing simulations, specify as follows:

1. Slow corner: SDFMIN and SDFMAX
2. Fast corner: SDFMIN and SDFMAX

Running Standalone Timing Simulation

If you are running timing simulation from Vivado IDE, it will add the timing simulation related switches to simulator. If you run standalone timing simulation, make sure to pass the following switch to simulators during elaboration:

For VCS:

```
+pulse_e/<number> and +pulse_r/<number> +transport_int_delays
```

During elaboration (with VCS)

For ModelSim/Questa Advanced Simulator:

```
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0
```

During elaboration (with vsim)



IMPORTANT! The Vivado simulator models use interconnect delays; consequently, additional switches are required for proper timing simulation, as follows: `-transport_int_delays -pulse_r 0 -pulse_int_r 0`. [Table 15: xelab, xvhd, and xvlog Command Options](#) provides descriptions for the these commands.

Dumping SAIF for Power Analysis

The Switching Activity Interchange Format (SAIF) is an ASCII report that assists in extracting and storing switching activity information generated by simulator tools. This switching activity can be back-annotated into the Xilinx power analysis and optimization tools for the power measurements and estimations.

Dumping SAIF in Questa Advanced Simulator/ModelSim

Questa Advanced Simulator/ModelSim uses explicit power commands to dump an SAIF file, as follows:

1. Specify the scope or signals to dump, by typing:

```
power add <hdl_objects>
```

2. Run simulation for specific time (or `run -all`).
3. Dump out the power report, by typing:

```
power report -all filename.saif
```

For more detailed usage or information about each commands, see the ModelSim documentation.

Example DO File

```
power add tb/fpga/*
run 500us
power report -all -bsaif routed.saif
quit
```

Dumping SAIF in VCS

VCS provides power commands to generate SAIF with specific requirements.

1. Specify the scope and signals to be generated, by typing:

```
power <hdl_objects>
```

2. Enable SAIF dumping. You can use the command line in the simulator workspace:

```
power -enable
```

3. Run simulation for a specific time.
4. Disable power dumping and report the SAIF, by typing:

```
power -disable
power -report filename.saif
```

For more detailed usage or information about each command, see the Synopsys VCS documentation.

See [Power Analysis Using Vivado Simulator](#) for more information about Switching Activity Interchange Format (SAIF).

Dumping VCD

You can use a Value Change Dump (VCD) file to capture simulation output. The Tcl commands are based on Verilog system tasks related to dumping values.

Dumping VCD in Questa Advanced Simulator/ ModelSim

Questa Advanced Simulator/ModelSim uses explicit VCD commands to dump a VCS file, as follows:

1. Open the VCD file:

```
vcd file my_vcdfile.vcd
```

2. Specify the scope or signals to dump:

```
vcd add <hdl_objects>
```

3. Run simulation for a specified period of time (or `run -all`).

For more detailed usage or information about each commands, see the [ModelSim documentation](#).

Example DO File:

```
vcd file my_vcdfile.vcd
vcd add -r tb/fpga/*
run 500us
quit
```

Dumping VCD in VCS

In VCS, you can generate a VCD file using the `dumpvar` command. Specify the file name and instance name (by default its complete hierarchy).

```
vcs +vcs+dumpvars+test.vcd
```

Simulating IP

In the following example, the `accum_0.xci` file is the IP you generated from the Vivado® IP catalog. Use the following commands to simulate this IP in VCS:

```
set_property target_simulator VCS [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location>[current_project]
launch_simulation -noclean_dir -of_objects [get_files accum_0.xci]
```

Using a Custom DO File During an Integrated Simulation Run

If you have some specific set of commands (custom DO file) that you want to invoke just before running the simulation, add those commands in a file and pass that using the appropriate command, as shown below:

In Questa Advanced Simulator

```
expans="page">set_property -name {questa.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>}
-objects [get_filesets sim_1]
```

In Modelsim

```
expans="page">set_property -name {modelsim.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>}
-objects [get_filesets sim_1]
```

In VCS

```
expans="page">set_property -name {vcs.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>} -objects
[get_filesets sim_1]
```

In Xcelium

```
expans="page">set_property -name {xcelium.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>}
-objects

expans="page">[get_filesets sim_1]
```

Simulation Step Control Constructs for ModelSim and Questa Advanced Simulator

The following table outlines the constructs used for controlling the step execution based on the do file format:

- **Native do file:** This is a default do file format. In this format, the compile and elaborate shell scripts calls `source <tb>_compile/elaborate.do`. For example:

```
source bft_tb_compile.do 2>&1 | tee -a compile.log
```

The simulate script calls `vsim -64 -c -do do {<tb>_simulate.do}`. For example:

```
$bin_path/vsim -64 -c -do do {bft_tb_simulate.do} -l simulate.log
```

- **Classic do file:** Classic do file format is different from the native do file in compile and elaborate shell scripts. There is no change in the simulate script. In compile and elaborate shell scripts, it calls `vsim -c -do do {<tb>_compile/elaborate.do}`. For example,

```
$bin_path/vsim -64 -c -do do {bft_tb_compile.do} -l compile.log
```

To get this, set `project.writeNativeScriptForUnifiedSimulation` to 0 by invoking `set_param project.writeNativeScriptForUnifiedSimulation 0` on Tcl console command.

This file format is useful for a shared project as the path for Questa Advanced Simulator/ ModelSim utility is hard-coded inside the shell scripts.

Table 10: Simulation Step Control Construct Parameters

Parameter	Description	Default
<code>project.writeNativeScriptForUnifiedSimulation</code>	Write a pure .do file with simulator command only (no Tcl or Shell constructs).	0 (false)
<code>simulator.quitOnSimulationComplete</code>	Quit simulator on simulator completion for ModelSim/Questa Advanced Simulator simulation. To disable quit, set this parameter to false.	1 (true)
<code>simulator.modelsimNoQuitOnError</code>	Do not quit on error or break by default for ModelSim/Questa Advanced Simulator simulation. To quit simulation on error or break, set this parameter to false.	1 (true)

Explanation

- `simulator.quitOnSimulationComplete`: By default, the generated `simulate.do` has `quit -force`. When the simulation is complete in the specified time, the simulator exits. If you do not want the simulator to exit, set `simulator.quitOnSimulationComplete` to 0 by invoking `set_param simulator.quitOnSimulationComplete 0`.

- `simulator.modelsimNoQuitOnErrorHandler`: By default, on error or break, simulator does not exit. If you want to exit the simulator, set the following parameter:

```
set_param simulator.modelsimNoQuitOnErrorHandler 0
```

This adds the following two lines in `<tb>_simulate.do`.

```
onbreak {quit -f}  
onerror {quit -f}
```

Running Third-Party Simulators in Batch Mode

The Vivado Design Suite supports batch or scripted simulation for third party verification. With the design files gathered, and the scripts generated to support your target simulator, you can inspect the scripts and incorporate them into your verification environment. Xilinx recommends that you use the `export_simulation` scripts as a starting point for your simulation flow rather than building a custom API to generate scripts. See [Exporting Simulation Files and Scripts](#) for more information on exporting simulation scripts.

Make sure that you have the correct environment setup for the simulator before running the scripts. See [Using Simulation Settings](#) for more information on configuring your simulator. See the User Guide of your specific simulator for the details of running batch or scripted mode.

Simulating with Vivado Simulator

The Vivado simulator is a Hardware Description Language (HDL) event-driven simulator that supports functional and timing simulations for VHDL, Verilog, SystemVerilog (SV), and mixed VHDL/Verilog or VHDL/SV designs.

The Vivado simulator supports the following features:

- Source code debugging (step, breakpoint, current value display)
- SDF annotation for timing simulation
- VCD dumping
- SAIF dumping for power analysis and optimization
- Native support for HardIP blocks (such as serial transceivers and PCIe®)
- Multi-threaded compilation
- Mixed language (VHDL, Verilog, or SystemVerilog design constructs)
- Single-click simulation re-compile and re-launch
- One-click compilation and simulation
- Built-in support for Xilinx simulation libraries
- Real-time waveform update

See the *Vivado Design Suite Tutorial: Logic Simulation* ([UG937](#)) for a step-by-step demonstration of how to run Vivado simulation.

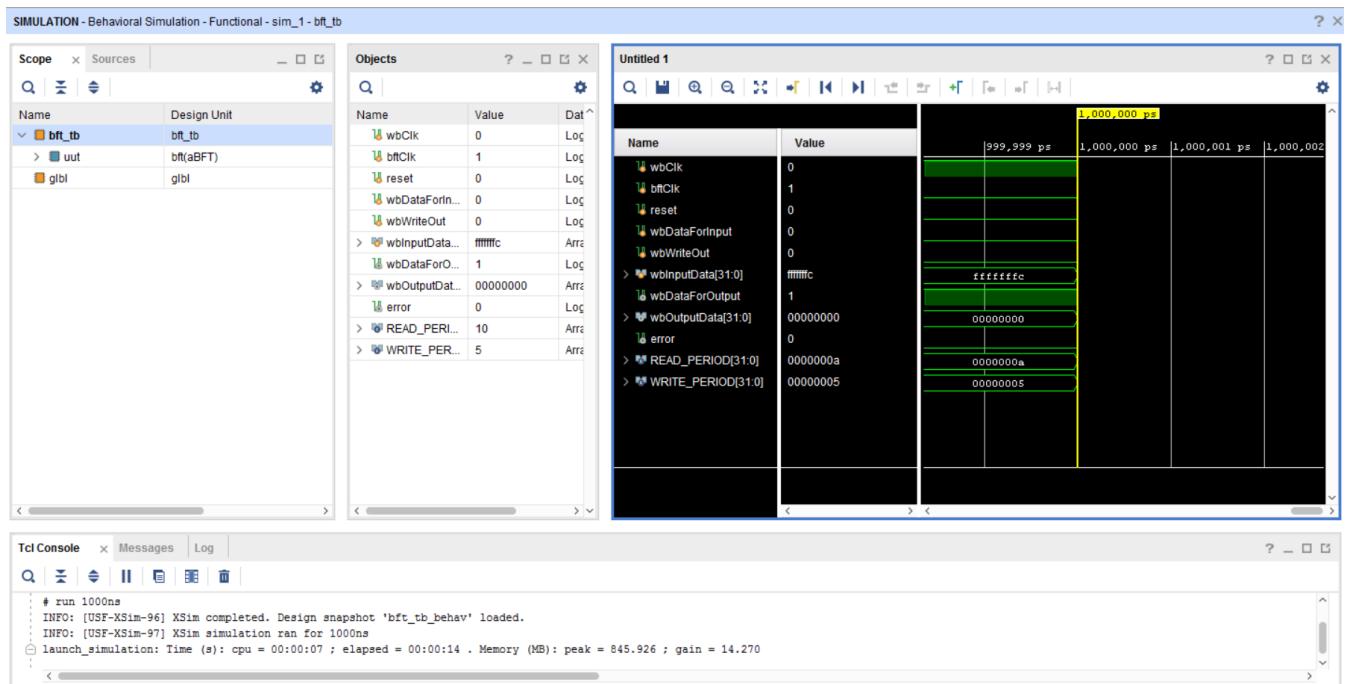
Running the Vivado Simulator



IMPORTANT! If you are using the Vivado simulator, be sure to specify all appropriate project settings for your design before running simulation. For supported third-party simulators, see [Chapter 3: Simulating with Third-Party Simulators](#).

From the Flow Navigator, click **Run Simulation** and select a simulation type to invoke the Vivado simulator workspace, shown in the figure below.

Figure 6: Vivado Simulator Workspace



Main Toolbar

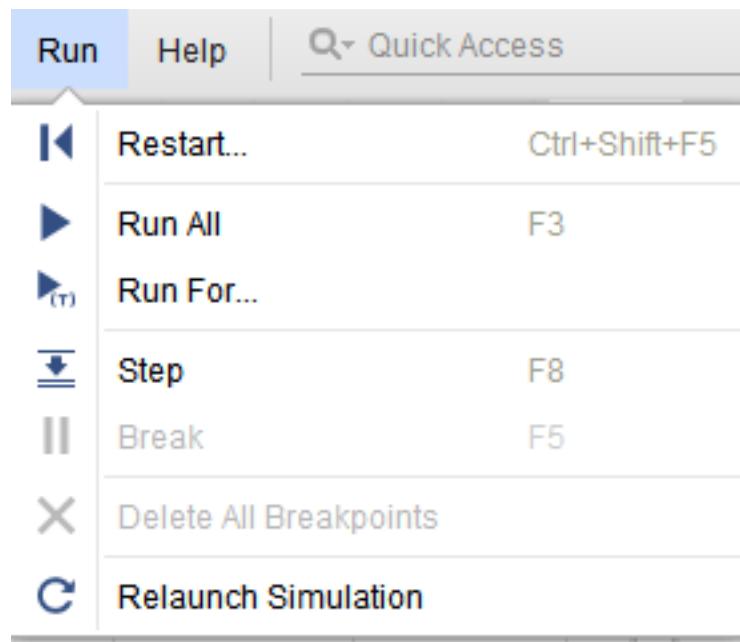
The main toolbar provides one-click access to the most commonly used commands in the Vivado IDE. When you hover over an option, a tool tip appears that provides more information.

Run Menu

The menus provide the same options as the Vivado IDE with the addition of a Run menu after you have run a simulation.

The Run menu for simulation is shown in the following figure.

Figure 7: Simulation Run Menu Options



The Vivado simulator Run menu options:

- **Restart:** Lets you restart an existing simulation from time 0. Tcl Command: `restart`
- **Run All:** Lets you run an open simulation to completion. Tcl Command: `run -all`
- **Run For:** Lets you specify a time for the simulation to run. Tcl Command: `run <time>`



TIP: While you can always specify time units in the run command such as `run 100 ns`, you can also omit the time unit. If you omit the time unit, the Vivado simulator will assume the time unit of the `TIME_UNIT` Tcl property. To view the `TIME_UNIT` property use the Tcl command `get_property time_unit [current_sim]`. To change the `TIME_UNIT` property use the Tcl command `set_property time_unit <unit> [current_sim]`, where <unit> is one of the following: `fs, ps, ns, us, ms, and s`.

- **Step:** Runs the simulation up to the next HDL source line.
- **Break:** Lets you pause a running simulation.
- **Delete All Breakpoints:** Deletes all breakpoints.
- **Relaunch Simulation:** Recompiles the simulation files and restarts the simulation.

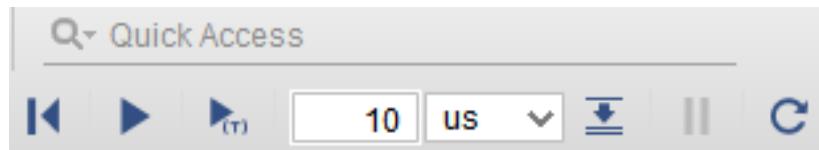
Related Information

[Re-running the Simulation After Design Changes \(relaunch\)](#)

Simulation Toolbar

When you run the Vivado simulator, the simulation-specific toolbar (shown in the figure below) opens to the right of the main toolbar.

Figure 8: Simulation Toolbar



These are the same buttons labeled in [Run Menu](#), above (without the Delete All Breakpoints option), and they are provided for ease of use.

Simulation Toolbar Button Descriptions

Hover over the toolbar buttons for tool-tip descriptions.

- **Restart:** resets the simulation time to zero.
- **Run all:** runs the simulation until it completes all events or until an HDL statement indicates that the simulation should stop.
- **Run For:** runs for a specified period of time.
- **Step:** runs the simulation until the next HDL statement.
- **Break:** Pauses the current simulation.
- **Relaunch:** Recompiles the simulation sources and restarts the simulation (after making code changes, for example).

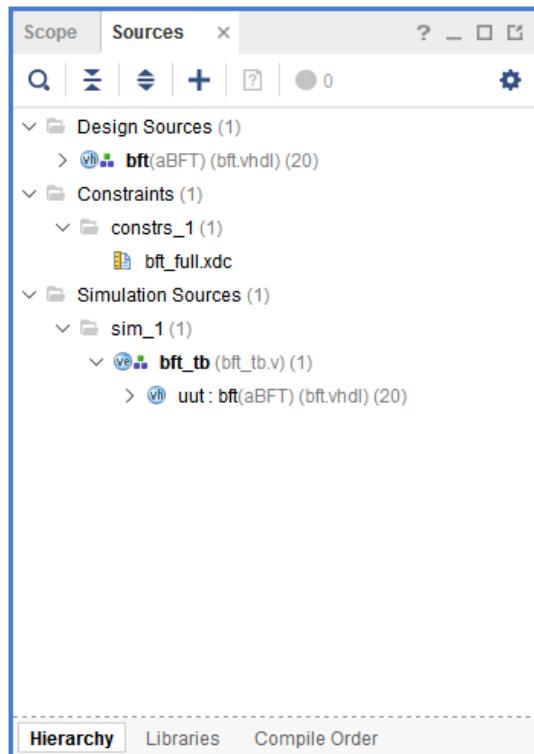
Related Information

[Re-running the Simulation After Design Changes \(relaunch\)](#)

Sources Window

The Sources window displays the simulation sources in a hierarchical tree, with views that show Hierarchy, IP Sources, Libraries, and Compile Order, as shown in the following figure.

Figure 9: Sources Window



The Sources buttons are described by tool tips when you hover the mouse over them. The buttons let you examine, expand, collapse, add to, open, filter and scroll through files.

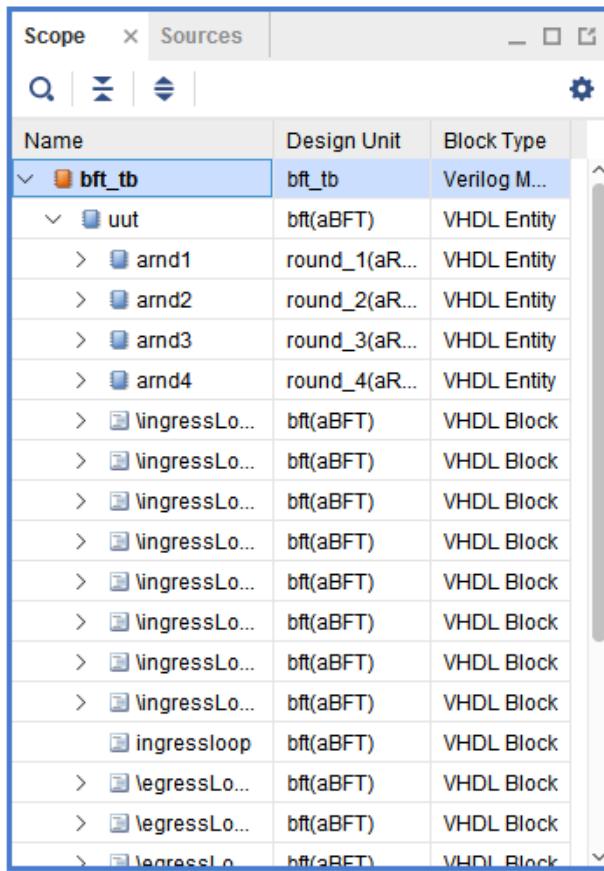
You can also open or add a source file by right-clicking on the source object and selecting the **Open File** or **Add Sources** options.

Scope Window

A scope is a hierarchical partition of an HDL design. Whenever you instantiate a design unit or define a process, block, package, or subprogram, you create a scope.

In the Scope window (shown in the figure below), you can see the design hierarchy. When you select a scope in the Scope window, all HDL objects visible from that scope appear in the Objects window. You can select HDL objects in the Objects window and add them to the waveform viewer.

Figure 10: Scope Window



Filtering Scopes

- Click **Settings** option on the scopes sub-menu to toggle between showing or hiding (check or uncheck) the corresponding scope type.

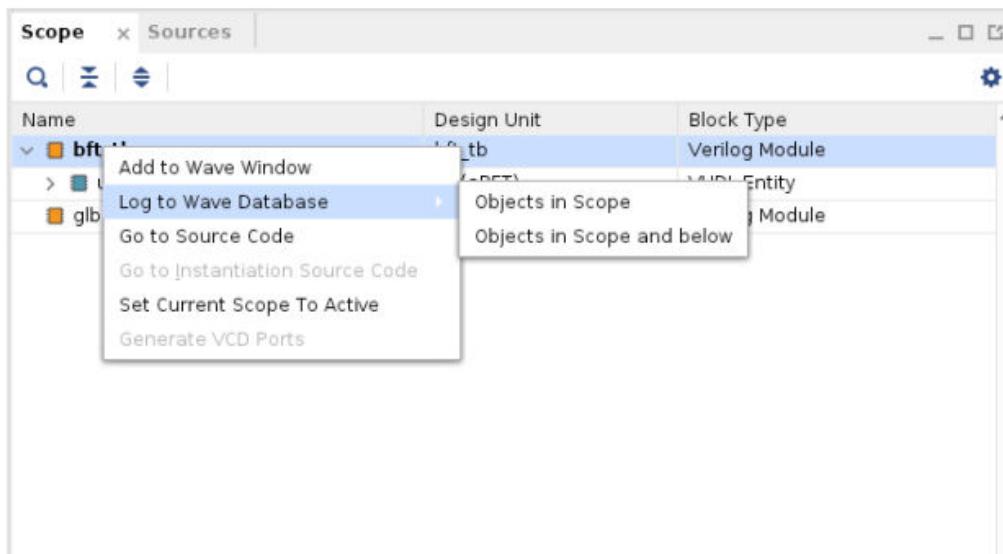
TIP: When you hide a scope using Setting option, all scopes inside that scope are also hidden regardless of type. For example, in the figure above, clicking the Verilog Module button to hide all Verilog module scopes would hide not only the *bft_tb* scope but also *uut* (even though *uut* is a VHDL entity scope).

- To limit the display to scopes containing a specified string, click the **Search** button  and type the string in the text box.

The objects displayed in the Objects window change (or are filtered) based on the current scope. Select the current scope to change the objects in the Objects window.

When you right-click a scope, a menu (shown in the following figure) appears with the following options:

Figure 11: Scope Window Options



- **Add to Wave Window:** Adds all viewable HDL objects of the selected scope to the waveform configuration.



TIP: HDL objects of large bit width can slow down the display of the waveform viewer. You can filter out such objects by setting a display limit on the wave configuration before issuing the Add to Wave Window command. To set a display limit, use the Tcl command `set_property DISPLAY_LIMIT <maximum bit width> [current_wave_config]`.

The Add to Wave Window command might add a different set of HDL objects from the set displayed in the Objects window. When you select a scope in the Scope window, the Objects window might display HDL objects from enclosing scopes in addition to objects defined directly in the selected scope. The Add to Wave Window command, on the other hand, adds objects from the selected scope only.

Alternately, you can drag and drop items in the Objects window into the Name column of the Wave window.



IMPORTANT! The Wave window displays the value changes of an object over time, starting from the simulation time at which the object was added.



TIP: To display object values prior to the time of insertion, the simulation must be restarted. To avoid having to restart the simulation because of missing value changes: issue the `log_wave -r` / Tcl command at the start of a simulation run to capture value changes for all display-able HDL objects in your design. For more information, see [Using the log_wave Tcl Command](#).

Changes to the waveform configuration, including creating the waveform configuration or adding HDL objects, do not become permanent until you save the WCFG file.

- **Go To Source Code:** Opens the source code at the definition of the selected scope.

- **Go To Instantiation Source Code:** For Verilog modules and VHDL entity instances, opens the source code at the point of instantiation for the selected instance.
- **Set Current Scope to Active:** Set the current scope to selected scope. The selected scope becomes the active simulation scope (i.e. `get_property active_scope [current_sim]`). Active simulation scope is the HDL process scope, where the simulation is currently paused. When used by disabling the **follow active scope** in setting, Vivado simulator will remember the last current_scope selection even when simulation proceeds. When a break-point is hit, current_scope will still point to last scope which is set as active scope
- **Log to Wave Database:** You can log either of the following:
 - The objects of current scope
 - The objects of the current scope and all scope below the current scope.



TIP: By default, the Vivado simulator suppresses the logging of large HDL objects. To change the size limit of logged objects, use the `set_property trace_limit <size> [current_sim]` Tcl command, where `<size>` is the number of scalar elements in the HDL object.

In the source code text editor, you can hover over an identifier in the code get the value, as shown in [Scope Window](#).



IMPORTANT! For this feature to work, be sure you have the scope associated with the source code selected in the Scope window.

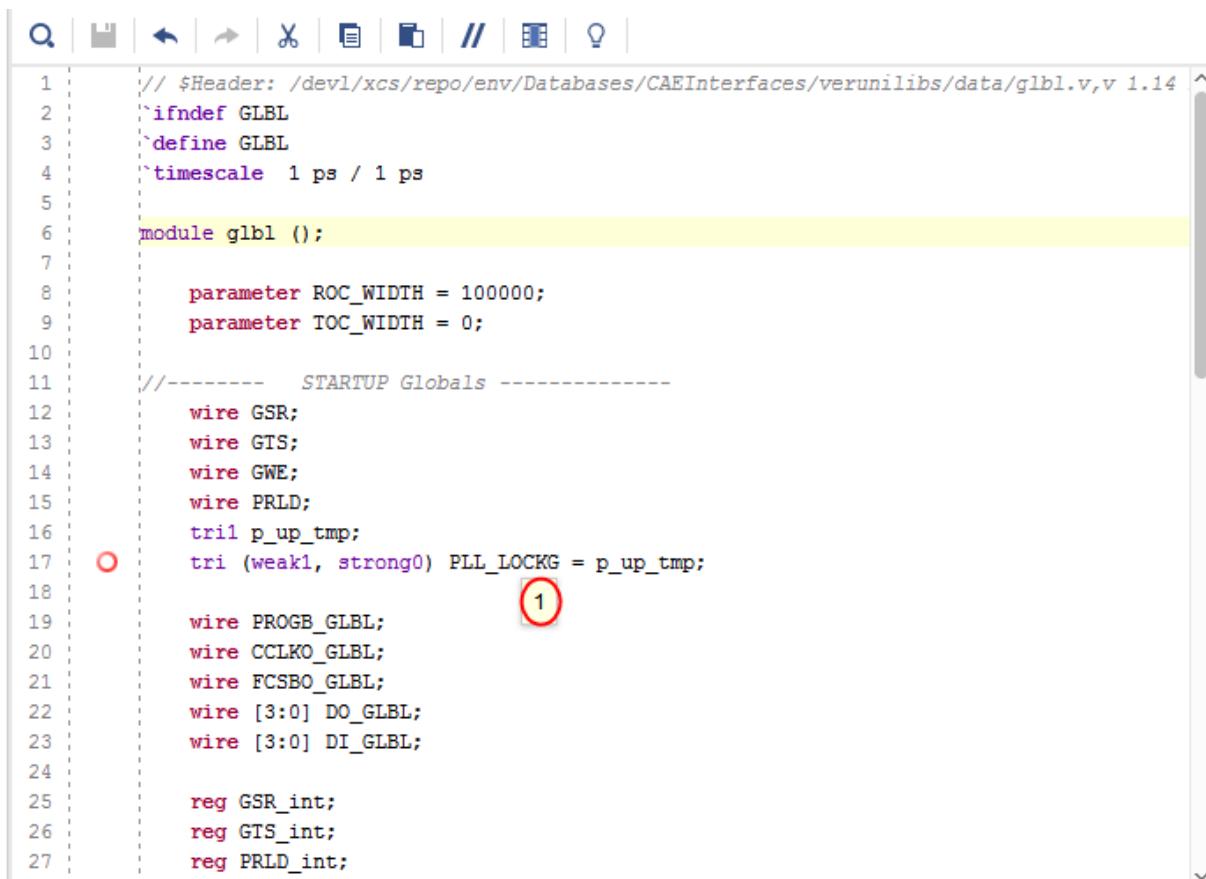


TIP: Because the top module is not instantiated, the Go to Instantiation Source Code right-click option (shown in the figure above) is grayed out when the top module is selected.



TIP: Use `log_wave` to log the objects of current scope or below. Post simulation, you can add any objects on waveform and see the plot starting from time 0 till current simulation.

Figure 12: Source Code with Identifier Value Displayed



```
1 // $Header: /devl/xcs/repo/env/Databases/CAEInterfaces/verunilibs/data/glbl.v,v 1.14 ^
2 `ifndef GLBL
3 `define GLBL
4 `timescale 1 ps / 1 ps
5
6 module glbl ();
7
8     parameter ROC_WIDTH = 100000;
9     parameter TOC_WIDTH = 0;
10
11 //----- STARTUP Globals -----
12     wire GSR;
13     wire GTS;
14     wire GWE;
15     wire PRLD;
16     tri1 p_up_tmp;
17     tri (weak1, strong0) PLL_LOCKG = p_up_tmp;
18     wire PROGB_GLBL; 1
19     wire CCLKO_GLBL;
20     wire FCSBO_GLBL;
21     wire [3:0] DO_GLBL;
22     wire [3:0] DI_GLBL;
23
24     reg GSR_int;
25     reg GTS_int;
26     reg PRLD_int;
```

Additional Scopes and Sources Options

In either the Scope or the Sources window, a search field displays when you select the **Show Search** button .

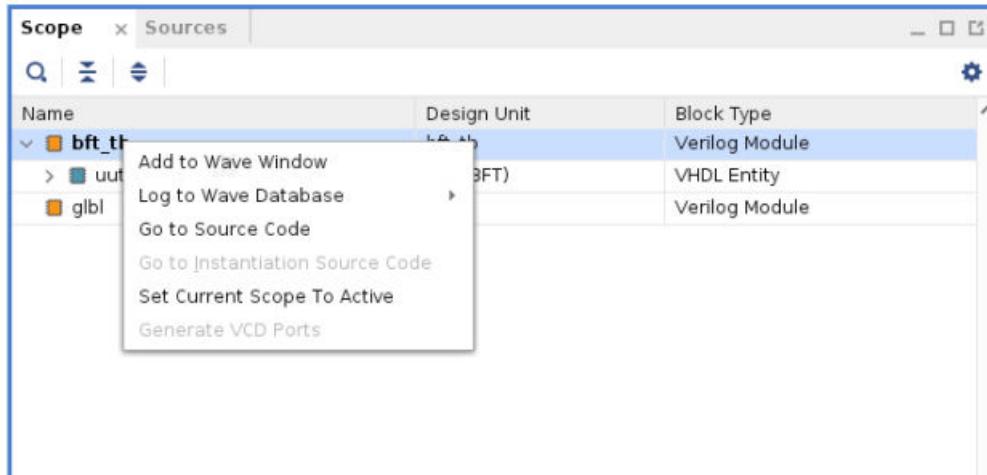
As an equivalent to using the Scope and Objects windows, you can navigate the HDL design by typing the following in the Tcl Console:

```
get_scopes
current_scope
report_scopes

id="ai516872">report_values
```

 **TIP:** To access source files for editing, you can open files from the Scope or Objects window by selecting **Go to Source Code**, as shown in [Scope Window](#).

Figure 13: Context Menu in Scope Window

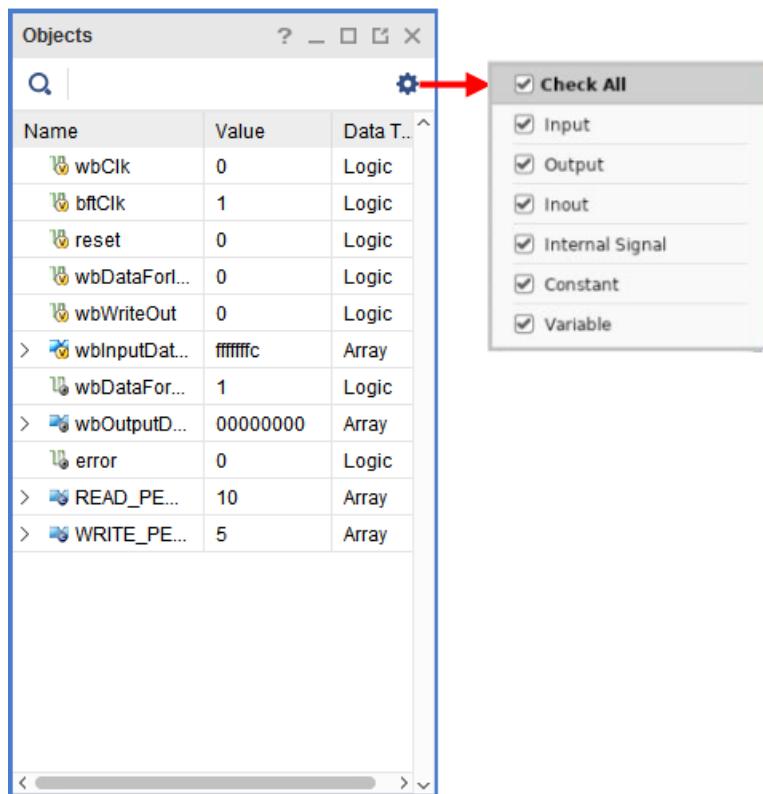


TIP: After you have edited source code and saved the file, you can click the Relaunch button  to recompile and relaunch simulation without having to close and reopen the simulation.

Objects Window

The Objects window displays the HDL simulation objects associated with the scope selected in the Scope window, as shown in the following figure.

Figure 14: Objects Window



Icons beside the HDL objects show the type or port mode of each object. This view lists the Name, Value, and Data Type of the simulation objects.

You can obtain the current value of an object by typing the following in the Tcl Console.

```
get_value <hdl_object>
```



TIP: To limit the number of digits to display for vectors, use the `set_property array_display_limit <bits> [current_sim]` command, where `<bits>` is the number of bits to display.

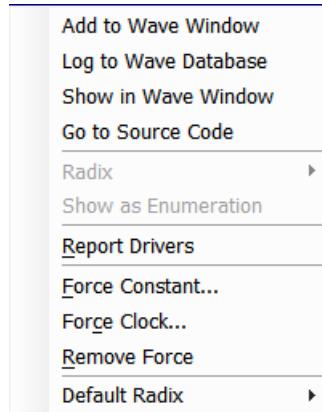
The following are the options available at the top of the Objects window. Click **Settings** to view the selected objects in the Objects window. Use this to filter or limit the contents of the Objects window.

- **Search:** You can use the **Search** option to search for an object name.
- **Settings:** Settings option allows you to display or hide various HDL objects in the Objects window.

Objects Context Menu

When you right-click an object in the Objects window, a context menu (shown in [Objects Window](#)) appears. The options in the context menu are described below.

Figure 15: Context Menu in Objects Window



- **Add to Wave Window:** Add the selected object to the waveform configuration. Alternately, you can drag and drop the objects from the Objects window to the Name column of the wave window.
- **Log to Wave Database:** Logs events of the selected object to the waveform database (WDB) for later viewing in the wave window.



TIP: By default, the Vivado simulator suppresses the logging of large HDL objects. To change the size limit of logged objects, use the `set_property trace_limit <size> [current_sim]` Tcl command, where `<size>` is the number of scalar elements in the HDL object.

- **Show in Wave Window:** Highlights the selected object in the wave window.
- **Default Radix:** Set the default radix for all objects in the Objects window and text editor. The default radix is Hexadecimal. You can change this option from the context menu.

Tcl command:

```
set_property radix <new radix> [current_sim]
```

Where `<new radix>` is any of the following:

- bin
- unsigned (for unsigned decimal)
- hex
- dec (for signed decimal)
- ascii
- oct

- smag (for signed magnitude)

Note: If you need to change the radix of an individual signal, use radix option from the context menu.

- **Radix:** Select the numerical format to use when displaying the value of the selected object in the Objects window and in the source code window.

You can change the radix of an individual object as follows:

1. Right-click an object in the Objects window.
2. From the context menu, select **Radix** and the format you want to use:
 - Default
 - Binary
 - Hexadecimal
 - Octal
 - ASCII
 - Unsigned Decimal
 - Signed Decimal
 - Signed Magnitude



TIP: If you change the radix in the Objects window, it will not be reflected in the wave window.

- **Show as Enumeration:** Select to display the values of a SystemVerilog enumeration signal or variable using enumeration labels.

Note: This menu item is enabled only for SystemVerilog enumerations. If unchecked, all values of the enumeration object display numerically according to the radix set for the object. If checked, those values for which the enumeration declaration defines a label display the label text, and all other values display numerically.

- **Report Drivers:** Display in the Tcl Console a report of the HDL processes that assign values to the selected object.
- **Go To Source Code:** Open the source code at the definition of the selected object.
- **Force Constant:** Forces the selected object to a constant value.
- **Force Clock:** Forces the selected object to an oscillating value.
- **Remove Force:** Removes any force on the selected object.



TIP: If you notice that some HDL objects do not appear in the Waveform Viewer, it is because Vivado simulator does not support waveform tracing of some HDL objects, such as named events in Verilog and local variables.

Related Information

[Using Force Commands](#)

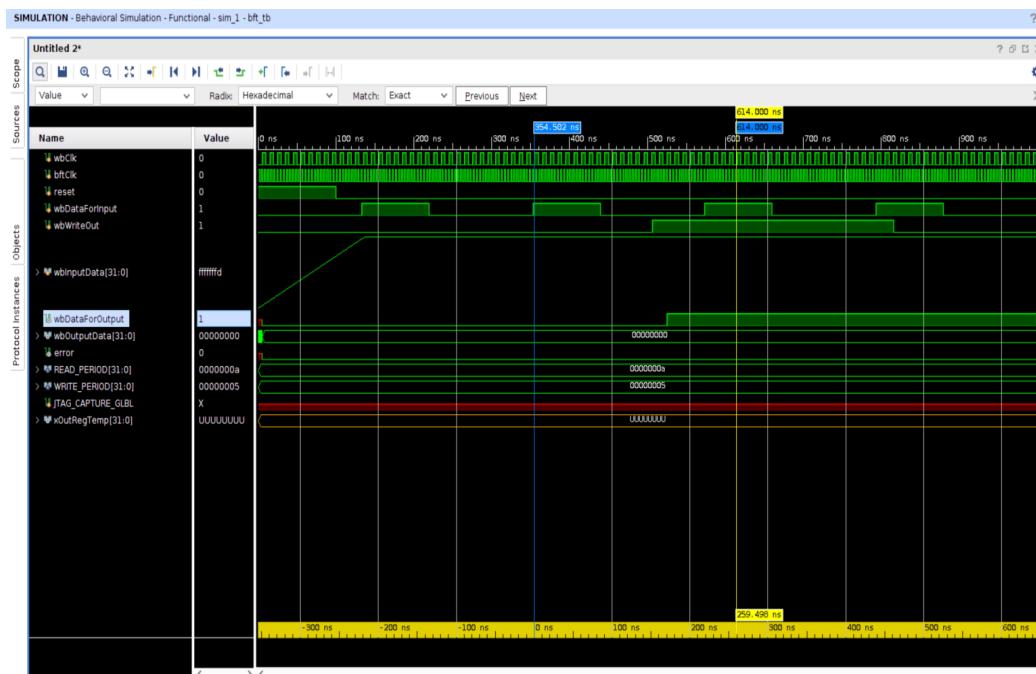
Wave Window

When you invoke the simulator it opens a wave window by default. The wave window displays a new wave configuration consisting of the traceable HDL objects from the top module of the simulation, as shown in [Wave Window](#).



TIP: On closing and reopening a project, you must rerun simulation to view the wave window. If, however, you unintentionally close the default wave window while a simulation is active, you can restore it by selecting **Window**→**Waveform** from the main menu.

Figure 16: Wave Window



To add an individual HDL object or set of objects to the wave window: in the Objects window, right-click an object or objects and select the **Add to Wave Window** option from the context menu (shown in [Objects Window](#)).

To add an object using the Tcl command type: `add_wave <HDL_objects>`.

Using the `add_wave` command, you can specify full or relative paths to HDL objects.

For example, if the current scope is `/bft_tb/uut`, the full path to the reset register under `uut` is `/bft_tb/uut/reset`: the relative path is `reset`.

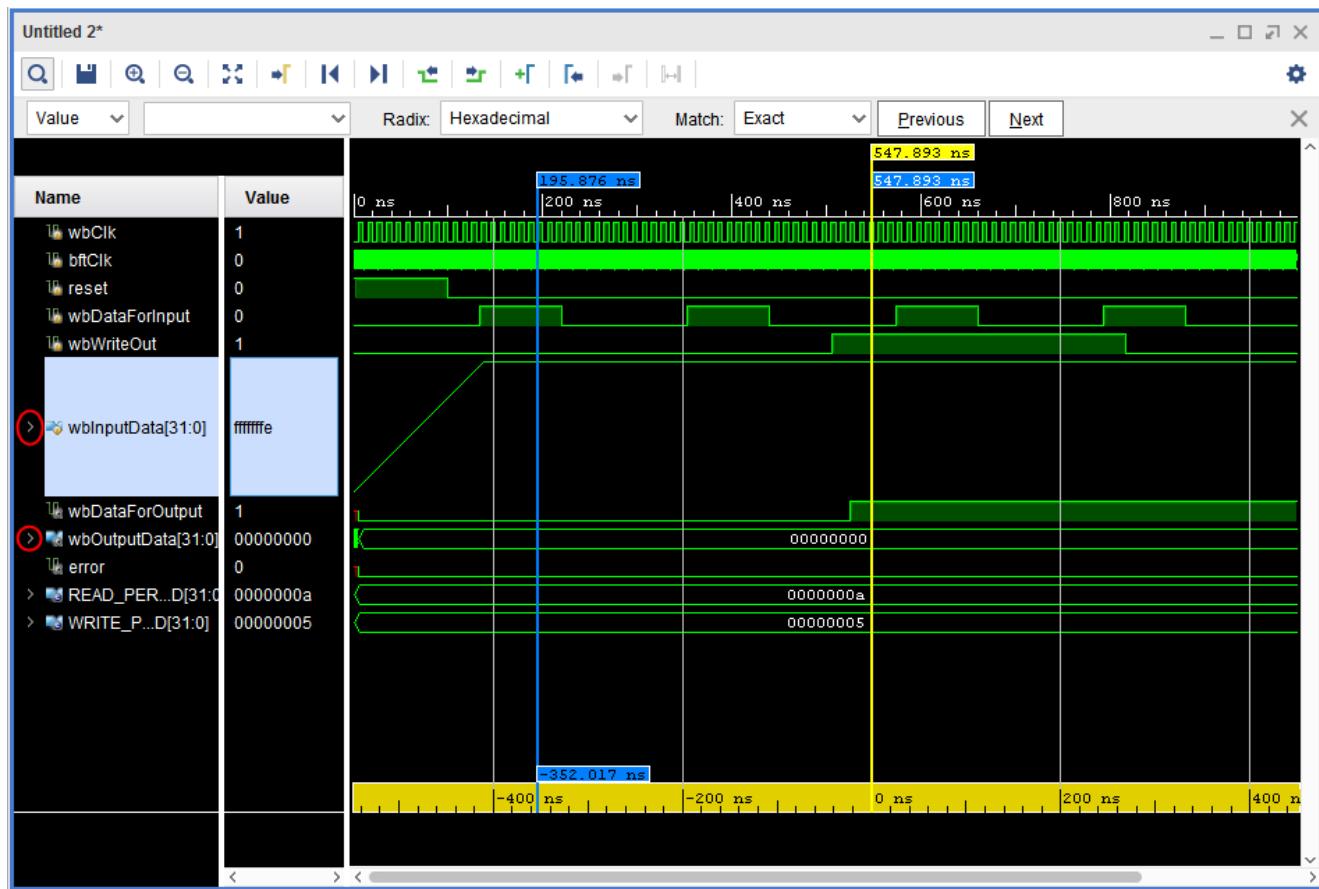


TIP: The `add_wave` command accepts HDL scopes as well as HDL objects. Using `add_wave` with a scope is equivalent to the **Add To Wave Window** command in the Scope window. HDL objects of large bit width can slow down the display of the waveform viewer. You can filter out such objects by setting a display limit on the wave configuration before issuing the Add to Wave Window command. To set a display limit, use the Tcl command `set_property DISPLAY_LIMIT <maximum bit width> [current_wave_config]`.

Wave Objects

The Vivado IDE Wave window is common across a number of Vivado Design Suite tools. An example of the wave objects in a waveform configuration is shown in the following figure.

Figure 17: HDL Objects in Waveform



The Wave window displays HDL objects, their values, and their waveforms, together with items for organizing the HDL objects, such as: groups, dividers, and virtual buses.

Collectively, the HDL objects and organizational items are called a *wave configuration*. The waveform portion of the Wave window displays additional items for time measurement, that include: cursors, markers, and timescale rulers.

The Vivado IDE traces the value changes of the HDL object in the Wave window during simulation, and you use the wave configuration to examine the simulation results.

The design hierarchy and the simulation waveforms are not part of the wave configuration, and are stored in a separate wave database (WDB) file.

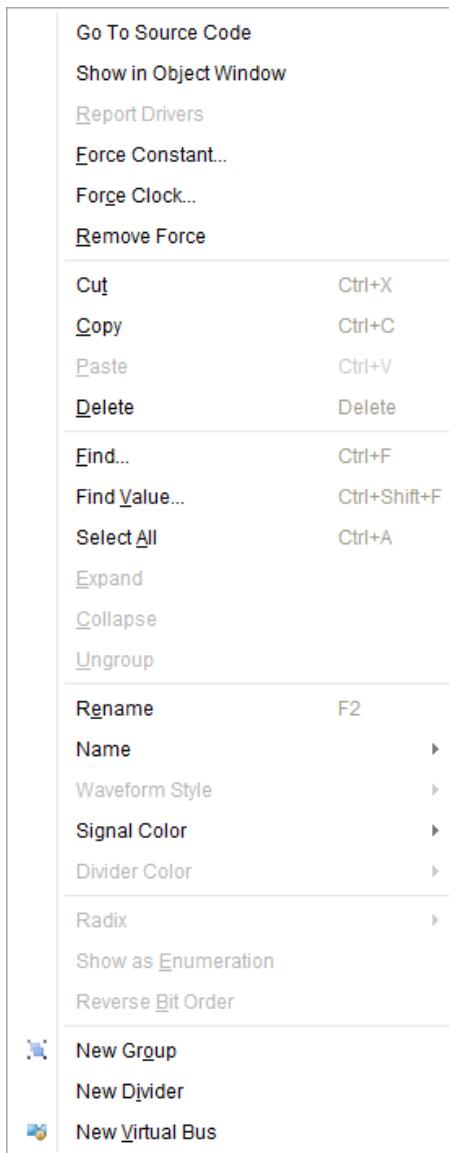
Context Menu in Waveform Window

When you right-click an object in the Waveform window, a context menu (shown in the following figure) appears. See [Understanding HDL Objects in Waveform Configurations](#) for more information on HDL objects in Waveforms. The options in the context menu are described below

- **Go To Source Code:** Opens the source code at the definition of the selected design wave object.
- **Show in Object Window:** Displays the HDL objects for a design wave object in the Objects window.
- **Report Drivers:** Display in the Tcl Console a report of the HDL processes that assign values to the selected wave object.
- **Force Constant:** Forces the selected object to a constant value.
- **Force Clock:** Forces the selected object to an oscillating value.
- **Remove Force:** Removes any force on the selected object.
- **Find:** Opens the Find Toolbar in the Waveform window to search for a wave object by name.
- **Find Value:** Opens the Find Toolbar in the Waveform window to search a waveform for a value.
- **Select All:** Selects all the wave objects in the Waveform window.
- **Expand:** Shows the sub-objects of the selected wave object.
- **Collapse:** Hides the sub-objects of the selected wave object.
- **Ungroup:** Unpacks the selected group or virtual bus.
- **Rename:** Changes the displayed name of the selected wave object.
- **Name:** Changes the display of the name of the selected wave object to show the full hierarchical name (long name), the simple signal or bus name (short name), or a custom name.
- **Waveform Style:** Changes the waveform of the selected design wave object to digital or analog format.
- **Signal Color:** Sets the waveform color of the selected design wave object.
- **Divider Color:** Sets the bar color of the selected divider.
- **Radix:** Sets the radix in which to display values of the selected design wave objects.
- **Show as Enumeration:** Shows values of the selected SystemVerilog enumeration wave object as enumerator labels in place of numbers, whenever possible.

- **Reverse Bit Order:** Reverses the bit order of values displayed for the selected array wave object.
- **New Group:** Packs the selected wave objects into a folder-like group wave object.
- **New Divider:** Creates a horizontal separator in the list of the Waveform window's wave objects.
- **New Virtual Bus:** Creates a new logic vector wave object consisting of the bits of the selected design wave objects.
- **Cut:** Allows you to cut a signal in the Waveform window.
- **Copy:** Allows you to copy a signal in the Waveform window.
- **Paste:** Allows you to paste a signal in the Waveform window.
- **Delete:** Allows you to delete a signal in the Waveform window.

Figure 18: Context Menu of Waveform Objects Window



See [Chapter 5: Analyzing Simulation Waveforms with Vivado Simulator](#) for more information about using the Wave window.

Saving a Waveform Configuration

The new wave configuration is not saved to disk automatically. Select **File** → **Simulation Waveform** → **Save Configuration As** and supply a file name to save a WCFG file.

To save a wave configuration to a WCFG file, type the Tcl command `save_wave_config <filename.wcfg>`.

The specified command argument names and saves the WCFG file.



IMPORTANT! *Zoom settings are not saved with the wave configuration.*

Related Information

- [Using Analog Waveforms](#)
- [Changing the Format of SystemVerilog Enumerations](#)
- [Organizing Waveforms](#)
- [Waveform Object Naming Styles](#)
- [Using Force Commands](#)
- [Searching a Value in Waveform Configuration](#)
- [Grouping Signals and Objects](#)
- [Reversing the Bus Bit Order](#)

Creating and Using Multiple Waveform Configurations

In a simulation session you can create and use multiple wave configurations, each in its own Wave window. When you have more than one Wave window displayed, the most recently-created or recently-used window is the *active window*. The active window, in addition to being the window currently visible, is the Wave window upon which commands external to the window apply. For example: **HDL Objects** → **Add to Wave Window**.

You can set a different Wave window to be the *active* window by clicking the title of the window.

Related Information

- [Distinguishing Between Multiple Simulation Runs](#)
- [Creating a New Wave Configuration](#)

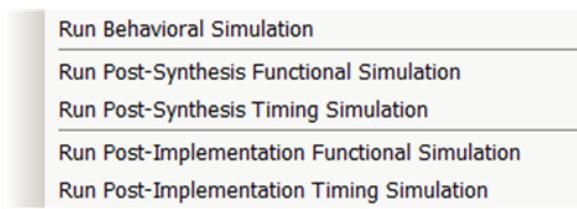
Running Functional and Timing Simulation

As soon as your project is created in the Vivado Design Suite, you can run behavioral simulation. You can run functional and timing simulations on your design after successfully running synthesis and/or implementation. To run simulation: in the Flow Navigator, select **Run Simulation** and choose the appropriate option from the popup menu shown in the figure below.



TIP: Availability of popup menu options is dependent on the design development stage. For example, if you have run synthesis but have not yet run implementation, the implementation options in the popup menu are grayed out.

Figure 19: Simulation Run Options



Running Functional Simulation

Post-Synthesis Functional Simulation

You can view **Run Simulation** → **Post-Synthesis Functional Simulation** option (shown in the previous figure) after completing a successful synthesis run.

After synthesis, the general logic design has been synthesized into device-specific primitives. Performing a post-synthesis functional simulation ensures that any synthesis optimizations have not affected the functionality of the design. After you select a post-synthesis functional simulation, the functional netlist is generated, and the UNISIM libraries are used for simulation.

Post-Implementation Functional Simulations

The **Run Simulation** → **Post-Implementation Functional Simulation** option (shown in the previous figure) becomes available after completing implementation run.

After implementation, the design has been placed and routed in hardware. A functional verification at this stage is useful in determining if any physical optimizations during implementation have affected the functionality of your design.

After you select a post-implementation functional simulation, the functional netlist is generated and the UNISIM libraries are used for simulation.

Running Timing Simulation



TIP: Post-Synthesis timing simulation uses the estimated timing delay from the device models and does not include interconnect delay. Post-Implementation timing simulation uses actual timing delays.

When you run Post-Synthesis and Post-Implementation timing simulation the simulator tools include:

- Gate-level netlist containing SIMPRIMS library components
- SECUREIP
- Standard Delay Format (SDF) files

You defined the overall functionality of the design in the beginning. When the design is implemented, accurate timing information is available.

To create the netlist and SDF, the Vivado Design Suite:

- Calls the netlist writer, `write_verilog` with the `-mode timesim` switch and `write_sdf` (SDF annotator)
- Sends the generated netlist to the target simulator

You control these options using Simulation Settings as described in [Using Simulation Settings](#).



IMPORTANT! Post-Synthesis and Post-Implementation timing simulations are supported for Verilog only. There is no support for VHDL timing simulation. If you are a VHDL user, you can run post synthesis and post implementation functional simulation (in which case no SDF annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the `write_vhdl` Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide ([UG835](#)).



IMPORTANT! The Vivado simulator models use interconnect delays; consequently, additional switches are required for proper timing simulation, as follows: `-transport_int_delays -pulse_r 0 -pulse_int_r 0`

Post-Synthesis Timing Simulation

The **Run Simulation → Post-Synthesis Timing Simulation** option (shown in the previous figure) becomes available after completing a successful synthesis run.

After synthesis, the general logic design has been synthesized into device-specific primitives, and the estimated routing and component delays are available. Performing a post-synthesis timing simulation allows you to see potential timing-critical paths prior to investing in implementation. After you select a post-synthesis timing simulation, the timing netlist and the estimated delays in the SDF file are generated. The netlist files includes `$sdf_annotate` command so that the simulation tool includes the generated SDF file.

Post-Implementation Timing Simulations

The **Run Simulation → Post-Implementation Timing Simulation** option (shown in the previous figure) becomes available after completing implementation run.

After implementation, the design has been implemented and routed in hardware. A timing simulation at this stage helps determine whether or not the design functionally operates at the specified speed using accurate timing delays. This simulation is useful for detecting unconstrained paths, or asynchronous path timing errors, for example, on resets. After you select a post-implementation timing simulation, the timing netlist and the SDF file are generated. The netlist files includes `$sdf_annotate` command so that the generated SDF file is picked up.

When you specified simulation settings, you specified whether or not to create an SDF file and whether the process corner would be set to fast or slow.



TIP: To find the SDF file optional settings, in the Vivado IDE Flow Navigator, right click **Simulation** and select **Simulation Settings**. In the Settings dialog box, select **Simulation** category and click **Netlist** tab.

Based on the specified process corner, the SDF file contains different `min` and `max` numbers.

Run two separate simulations to check for setup and hold violations.

To run a setup check, create an SDF file with `-process_corner slow`, and use the `max` column from the SDF file.

To run a hold check, create an SDF file with the `-process_corner fast`, and use the `min` column from the SDF file. The method for specifying which SDF delay field to use is dependent on the simulation tool you are using. Refer to the specific simulation tool documentation for information on how to set this option.

To get full coverage run all four timing simulations, specify as follows:

- Slow corner: `SDFMIN` and `SDFMAX`
- Fast corner: `SDFMIN` and `SDFMAX`

Saving Simulation Results

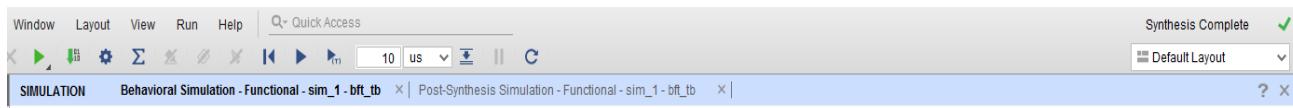
The Vivado simulator saves the simulation results of the objects (VHDL signals, or Verilog `reg` or `wire`) being traced to the Waveform Database (WDB) file (`<filename>.wdb`) in the `<project>.sim/<simset>` directory.

If you add objects to the Wave window and run the simulation, the design hierarchy for the complete design and the transitions for the added objects are automatically saved to the WDB file. You can also add objects to the waveform database that are not displayed in the Wave window using the `log_wave` command. For information about the `log_wave` command, see [Using the log_wave Tcl Command](#).

Distinguishing Between Multiple Simulation Runs

When you have run several simulations against a design, the Vivado simulator displays named tabs at the top of the workspace with the simulation type that is currently in the window highlighted, as shown in the following figure.

Figure 20: Active Simulation Type



Closing a Simulation

To close a simulation, in the Vivado IDE:

Select **File > Exit** or click the **X** at the top-right corner of the project window.



CAUTION! When there are multiple simulations running, clicking the **X** on the blue title bar closes all simulations. To close a single simulation, click the **X** on the small gray or white tab under the blue title bar.

To close a simulation from the Tcl Console, type:

```
close_sim
```

The Tcl command first checks for unsaved wave configurations. If any exist, the command issues an error. Close or save unsaved wave configurations before issuing the `close_sim` command, or add the `-force` option to the Tcl command.

Note: It is always recommended to use `close_sim` command to completely close the simulation before using `close_project` command to close the current project.

Adding a Simulation Start-up Script File

You can add custom Tcl commands in a batch file to the project so that they are run with the simulation. These commands are run after simulation begins. An example of this process is described in the steps below.

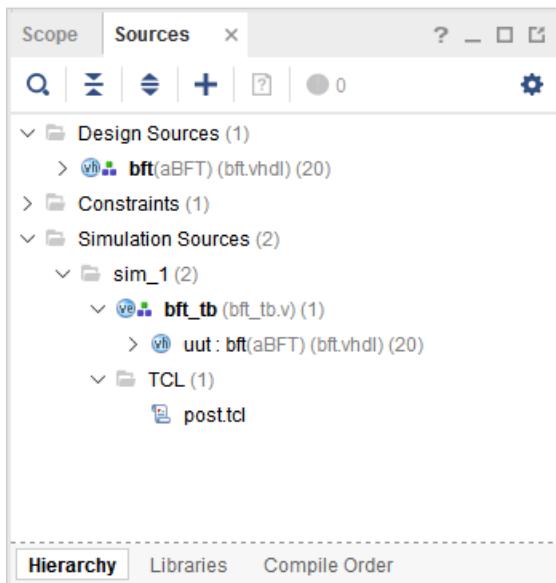
1. Create a Tcl script with the simulation commands you want to add to the simulation source files. For example, if you have a simulation that runs for 1,000 ns, and you want it to run longer, create a file that includes:

```
run 5us
```

Or, if you want to monitor signals that are *not* at the top level (because, by default, only top-level signals are added to the waveform), you can add them to the `post.tcl` script. For example:

```
add_wave /top/I1/<signalName>
```

2. Name the file `post.tcl` and save it.
3. Use the Add Sources option in Flow Navigator to invoke the Add Sources wizard, and select **Add or Create Simulation Sources**.
4. Add the `post.tcl` file to your Vivado Design Suite project as a simulation source. The `post.tcl` file displays in the Simulation Sources folder, as shown in the following figure.



5. From the Simulation toolbar, click the **Relaunch** button .

Simulation runs again, with the additional time you specified in the `post.tcl` file added to the originally specified time. Notice that the Vivado simulator automatically sources the `post.tcl` file after invoking all its commands.

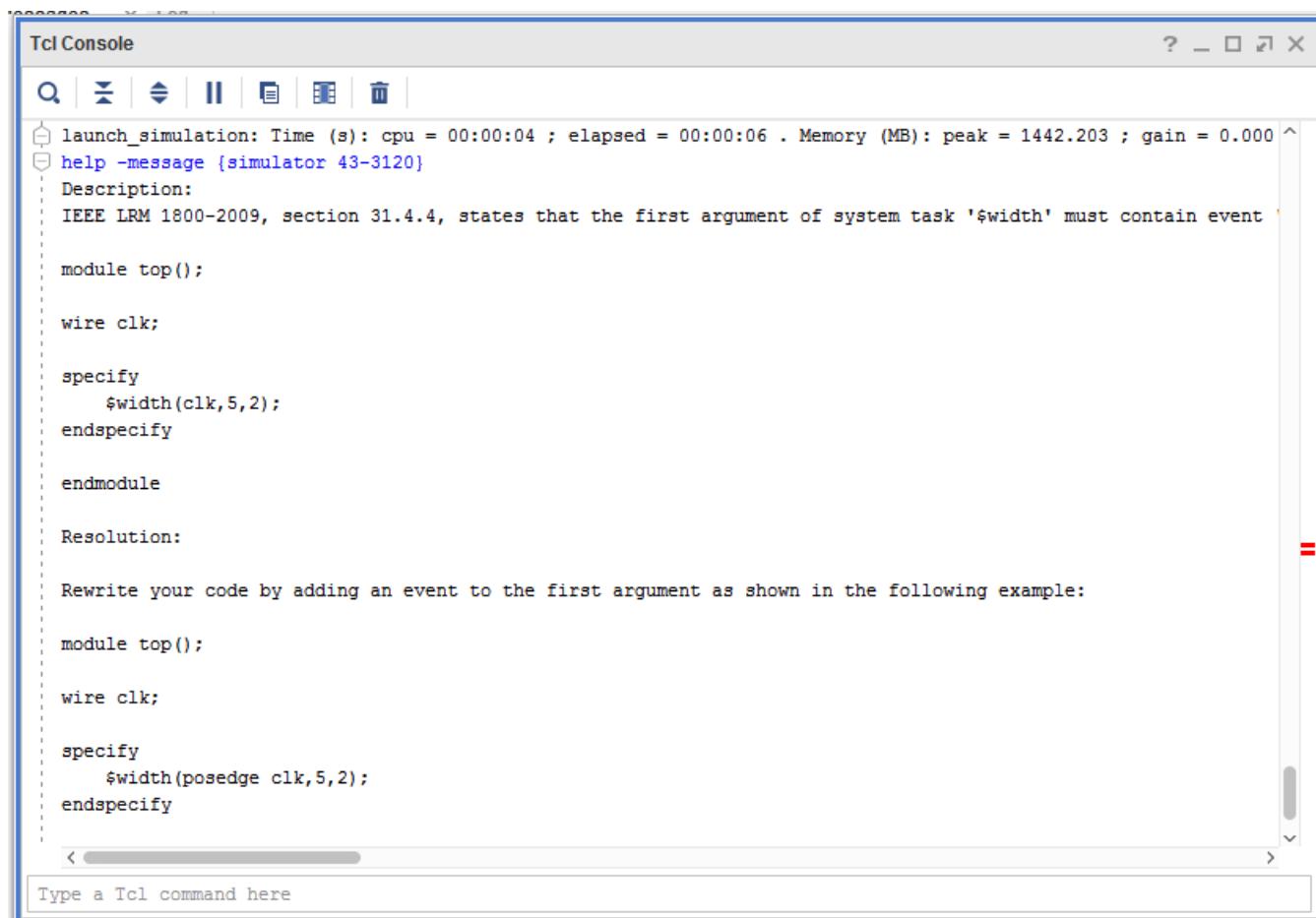
Viewing Simulation Messages

The Vivado IDE contains a message area where you can view informational, warning, and error messages. As shown in the following figure, some messages from the Vivado simulator contain an issue description and a suggested resolution.

To see the same detail in the Tcl Console, type:

```
help -message {message_number}
```

Figure 21: Simulator Message Description and Resolution Information



The screenshot shows the Vivado Tcl Console window. The title bar says "Tcl Console". The main area displays a message from the simulator. The message starts with "launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:06 . Memory (MB): peak = 1442.203 ; gain = 0.000 ^" and then provides a help message for the "help -message {simulator 43-3120}" command. It describes a violation of IEEE LRM 1800-2009, section 31.4.4, which states that the first argument of the system task '\$width' must contain an event. The code example shows a Verilog module "top" with a "specify" block that uses the \$width system task with a single argument. The "Resolution" section suggests rewriting the code to use a two-argument version of \$width, specifically \$width(poposedge clk, 5, 2). The bottom of the window has a "Type a Tcl command here" input field.

An example of such a command is as follows:

```
help -message {simulator 43-3120}
```

Managing Message Output

If your HDL design produces a large number of messages (for example, via the `$display` Verilog system task or `report` VHDL statement), you can limit the amount of text output sent to the Tcl Console and log file. This saves computer memory and disk space. To accomplish this, use the `-maxlogsize` command line option:

1. In the Flow Navigator, right-click on **SIMULATION** and select **Simulation Settings**.
2. In the **Settings** dialog box, add `-maxlogsize <size>` next to `xsim.simulate.xsim.more_options`, where `<size>` is the maximum amount of text output in megabytes.

Using the `launch_simulation` Command

The `launch_simulation` command lets you run any supported simulator in script mode.

The syntax of `launch_simulation` is as follows:

```
launch_simulation [-step <arg>] [-simset <arg>] [-mode <arg>] [-type <arg>]
                  [-scripts_only] [-of_objects <args>] [-absolute_path]
                  [-install_path <arg>] [-noclean_dir] [-quiet] [-verbose]
                  [-gcc_install_path <arg>] [-exec]
```

The following table describes the options of `launch_simulation`.

Table 11: `launch_simulation` Options

Option	Description
<code>[-step]</code>	Launch a simulation step. Values: all, compile, elaborate, simulate. Default: all (launch all steps).
<code>[-simset]</code>	Name of the simulation fileset.
<code>[-mode]</code>	Simulation mode. Values: behavioral, post-synthesis, post-implementation. Default: behavioral.
<code>[-type]</code>	Netlist type. Values: functional, timing. This is only applicable when the mode is set to post-synthesis or post-implementation.
<code>[-scripts_only]</code>	Only generate scripts.
<code>[-of_objects]</code>	Generate compile order file for this object (applicable with <code>-scripts_only</code> option only)
<code>[-absolute_path]</code>	Make all file paths absolute with respect to the reference directory.
<code>[-install_path]</code>	Custom installation directory path.
<code>[-noclean_dir]</code>	Do not remove simulation run directory files.
<code>[-quiet]</code>	Ignore command errors.
<code>[-verbose]</code>	Suspend message limits during command execution.
<code>[-gcc_install_path]</code>	Specify GNU compiler installation directory path for g++/gcc executable
<code>[-exec]</code>	Execute existing script for the step specified with the <code>-step</code> switch.

Examples

- Running behavioral simulation using Vivado simulator

```
create_project project_1 project_1 -part xc7vx485tffg1157-1
add_files -norecurse tmp.v
add_files -fileset sim_1 -norecurse testbench.v
import_files -force -norecurse
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
launch_simulation
```

- Generating script for behavioral simulation with Questa Advanced Simulator.

```
create_project project_1 project_1 -part xc7vx485tffg1157-1
add_files -norecurse tmp.v
add_files -fileset sim_1 -norecurse testbench.v
import_files -force -norecurse
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
set_property target_simulator Questa [current_project]
set_property compxlib.questa_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -scripts_only
```

- Launching post-synthesis functional simulation using Synopsys VCS

```
set_property target_simulator VCS [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -mode post-synthesis -type functional
```

- Running post-implementation timing simulation using Synopsys VCS

```
set_property target_simulator vcs [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location> [current_project]
launch_simulation -mode post-implementation -type timing
```

Re-running the Simulation After Design Changes (relaunch)

While debugging your HDL design with the Vivado simulator, you can determine that your HDL source code needs correction.

Use the following steps to modify your design and re-run the simulation:

1. Use the Vivado code editor or other text editor to update and save any necessary source code changes.
2. Use the Relaunch  button on the Vivado IDE toolbar to re-compile and re-launch the simulation as shown in the following figure. You may alternatively use the `relaunch_sim` Tcl command to re-compile and re-launch the simulation.

Figure 22: relaunch sim option



3. If the modified design fails to compile, an error box appears displaying the reason for failure. The Vivado IDE continues to display the results of the previous run of the simulation in a disabled state. Return to step 1 to correct the errors and re-launch the simulation again.

After the design successfully re-compiles, the simulation starts again.



IMPORTANT! Relaunching may fail for reasons other than compilation errors, such as in the case of a file system error. If the Run buttons on the Simulation toolbar are grayed out after a re-launch, indicating that the simulation is disabled, check the contents of the Tcl Console for possible errors that have prevented the re-launch from succeeding.



CAUTION! You may also re-launch the simulation using Run Simulation in the Flow Navigator or using `launch_simulation` Tcl command. However, using these options may fully close the simulation, discarding waveform changes and simulation settings such as radix customization.

Note: The **Relaunch Simulation** button  will be active only after one successful run of Vivado simulator using `launch_simulation`. The **Relaunch Simulation** button would be grayed out if the simulation is run in a Batch/Scripted mode.

Using the Saved Simulator User Interface Settings

By default, the Vivado simulator saves your configuration changes to a file under the simulation's working directory as you work with the user interface controls and Tcl commands of the Vivado simulator. The settings that are saved include the following:

- The state of the filter buttons and column widths of the Scopes and Objects windows.
- Tcl properties of the simulation, including array display limit, default radix, default time unit for the run command, and trace limit.
- Radixes and the **Show as Enumeration** state that you set on HDL objects in the Objects window.

After you shut down the simulation, the Vivado simulator restores your settings when you reopen and run the Vivado simulator.



IMPORTANT! Turn off the **Clean Up Simulation Files** check box in Vivado's Simulation Settings to ensure that the settings file does not get erased when you relaunch the simulation.



TIP: To revert the settings to their defaults, delete the settings file. You can find the settings file under the Vivado project directory at `<project>.sim/<simset>/<simtype>/xsim.dir/<snapshot>/xsimSettings.ini`. For example, the settings file for the default behavioral simulation run of the BFT example design would reside at `bft.sim/sim_1/behav/xsim.dir/bft_tb_behav/xsimSettings.ini`. Alternatively, turn on the **Clean Up Simulation Files** check box in the Simulation Settings.

Default Settings

A Vivado® project Tcl object supports a few properties that allows you to supply default settings for cleaned up or newly created simulations. These simulations do not already have a settings file. The following list shows the default settings properties of the project:

- XSIM.ARRAY_DISPLAY_LIMIT
- XSIM.RADIX
- XSIM.TIME_UNIT
- XSIM.TRACE_LIMIT

You can view the current values of the properties with the `report_property [current_project]` Tcl command and set the values of the properties with the `set_property <property name> <property value> [current_project]` Tcl command. For example, to set the array display limit to 16, use the following command.

```
set_property xsim.array_display_limit 16 [current_project]
```

When you launch the new or cleaned-up simulation, the simulation Tcl object inherits your project properties. You can verify it with the following Tcl command:

```
report_property [current_sim]
```



IMPORTANT! *The project properties apply only to cleaned-up or newly created simulations. After you have run a simulation of a particular run type and sim set such as `sim_1/behav`, that simulation retains a separate copy of the settings for all subsequent launches. The changes to the project properties can no longer take effect for that simulation. The project properties take effect again only if the simulation is cleaned up or the settings file is deleted.*

Analyzing Simulation Waveforms with Vivado Simulator

In the Vivado® simulator, you can use the waveform to analyze your design and debug your code. The simulator populates design signal data in other areas of the workspace, such as the Objects and the Scope windows.

Typically, simulation is set up in a test bench where you define the HDL objects you want to simulate. For more information about test benches see *Writing Efficient Test Benches* ([XAPP199](#)).

When you launch the Vivado simulator, a wave configuration displays with top-level HDL objects. The Vivado simulator populates design data in other areas of the workspace, such as the Scope and Objects windows. You can then add additional HDL objects, or run the simulation. See [Using Wave Configurations and Windows](#) below.

Using Wave Configurations and Windows

Vivado simulator allows customization of the wave display. The current state of the display is called the *wave configuration*. This configuration can be saved for future use in a WCFG file.

A wave configuration can have a name or be `untitled`. The name shows on the title bar of the wave configuration window. A wave configuration is `untitled` when it has never been saved to a file.

Creating a New Wave Configuration

Create a new waveform configuration for displaying waveforms as follows:

1. Select **File**→**Simulation Waveform**→**New Configuration**.

A new Wave window opens and displays a new, `untitled` waveform configuration. Tcl command: `create_wave_config <waveform_name>`.

2. Add HDL objects to the waveform configuration using the steps listed in [Understanding HDL Objects in Waveform Configurations](#) .

See [Chapter 4: Simulating with Vivado Simulator](#) for more information about creating new waveform configurations. Also see [Creating and Using Multiple Waveform Configurations](#) for information on multiple waveforms.

Opening a WCFG File

Open a WCFG file to use with the simulation as follows:

1. Select **File > Simulation Waveform > Open Configuration**.

The Open Waveform Configuration dialog box opens.

2. Locate and select a WCFG file.

Note: When you open a WCFG file that contains references to HDL objects that are not present in a static simulation HDL design hierarchy, the Vivado simulator ignores those HDL objects and omits them from the loaded waveform configuration.

A Wave window opens, displaying waveform data that the simulator finds for the listed wave objects of the WCFG file.

Tcl command: `open_wave_config <waveform_name>`

Saving a Wave Configuration

After editing, to save a wave configuration to a WCFG file, select **File → Simulation Waveform → Save Configuration As**, and type a name for the waveform configuration.

Tcl command: `save_wave_config <waveform_name>`

Opening a Previously Saved Simulation Run

There are three methods for opening a previously saved simulation using the Vivado Design Suite: an interactive method and a programmatic method.

Standalone mode

You can open WDB file outside Vivado using the following command:

```
xsim <name>.wdb -gui
```



TIP: You can open a WCFG file together with the WDB file by adding `-view <WCFG file>` to the `xsim` command.

Interactive Method

- If a Vivado Design Suite project is loaded, click **Flow** → **Open Static Simulation** and select the WDB file containing the waveform from the previously run simulation.



TIP: A static simulation is a mode of the Vivado simulator in which the simulator displays data from a WDB file in its windows in place of data from a running simulation.

- Alternatively, in the Tcl Console, run: `open_wave_database <name>.wdb`.

Programmatic Method

Create a Tcl file (for example, `design.tcl`) with contents:

```
current_fileset
open_wave_database <name>.wdb
```

Then run it as:

```
vivado -source design.tcl
```



IMPORTANT! Vivado simulator can open WDB files created on any supported operating system. It can also open WDB files created in Vivado Design Suite versions 2014.3 and later. Vivado simulator cannot open WDB files created in versions earlier than 2014.3 of the Vivado Design Suite.

When you run a simulation and display HDL objects in a Wave window, the running simulation produces a waveform database (WDB) file containing the waveform activity of the displayed HDL objects. The WDB file also stores information about all the HDL scopes and objects in the simulated design. In this mode you cannot use commands that control or monitor a simulation, such as run commands, as there is no underlying 'live' simulation model to control.

However, you can view waveforms and the HDL design hierarchy in a static simulation.

Understanding HDL Objects in Waveform Configurations

When you add an HDL object to a waveform configuration, the waveform viewer creates a *wave object* of the HDL object. The wave object is linked to, but distinct from, the associated HDL object.

You can create multiple wave objects from the same HDL object, and set the display properties of each wave object separately.

For example, you can set one wave object for an HDL object named `myBus` to display values in hexadecimal and another wave object for `myBus` to display values in decimal.

There are other kinds of wave objects available for display in a waveform configuration, such as: dividers, groups, and virtual buses.

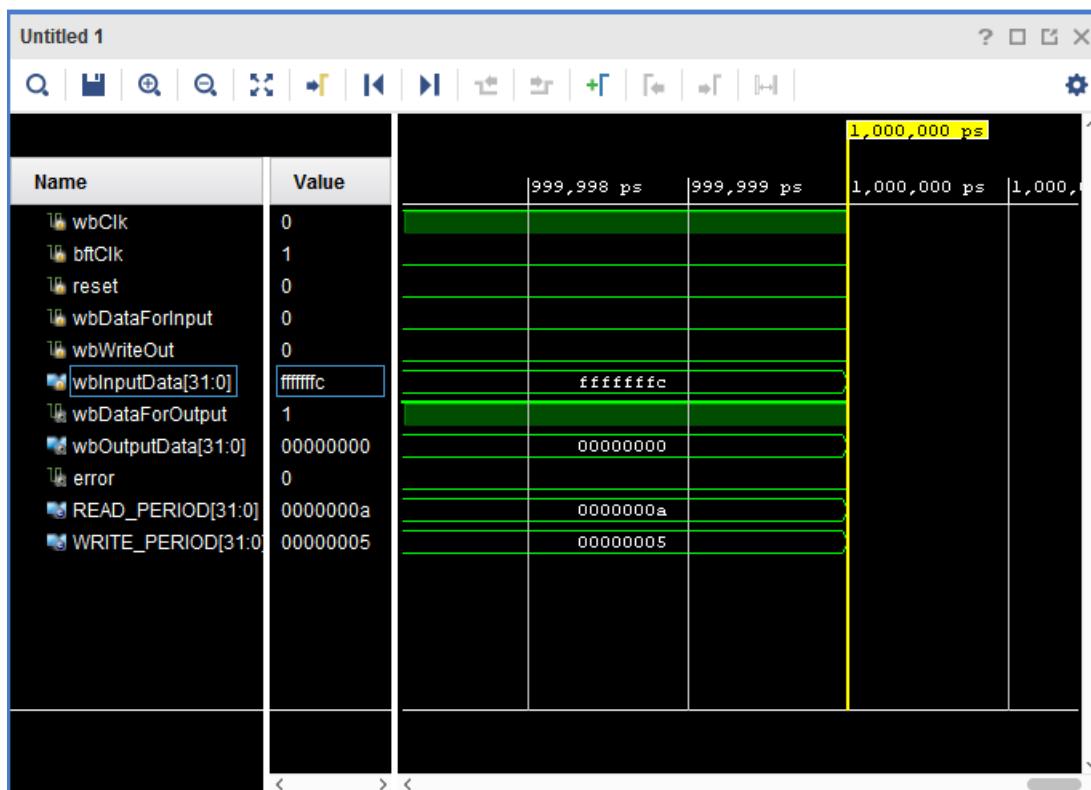
Wave objects created from HDL objects are specifically called *design wave objects*. These objects display with a corresponding icon. For design wave objects, the icon indicates whether the object is a scalar or a compound such as a Verilog vector or VHDL record.

TIP: To view the HDL object for a design wave object in the Objects window, right-click the name of the design wave object and choose **Show in Object Window**.

The following figure shows an example of HDL objects in the waveform configuration window. The design objects display Name and Value.

- **Name:** By default, shows the short name of the HDL object: the name alone, without the hierarchical path of the object. You can change the Name to display a long name with full hierarchical path or assign it a custom name.
 - **Value:** Displays the value of the object at the time indicated in the main cursor of the wave window. You can change the formatting, or radix, of the value independent of the formatting of other design wave objects linked to the same HDL object and independent of the formatting of values displayed in the Objects window and source code window.

Figure 23: Waveform HDL Objects



The Scope window provides the ability to add all viewable HDL objects for a selected scope to the wave window. For information on using the Scope window, see [Scope Window](#).

About Radixes

Understanding the type of data on your bus is important, and to use the digital and analog waveform options effectively, you need to recognize the relationship between the radix setting and the data type.



IMPORTANT! Make a change to the radix setting in the window in which you wish to see the change. A change to the radix of an item in the Objects window does not apply to values in the Wave window or the Tcl Console. For example, the item `wbOutputData[31:0]` can be set to **Signed Decimal** in the objects window, but it remains set to **Binary** in the Wave window.

Changing the Default Radix

The default waveform radix controls the numerical format of values for all wave objects whose radix you did not explicitly set. The waveform radix defaults to **Hexadecimal**.

To change the default waveform radix:

1. In the Waveform window, click the **Settings** button  to open the **Waveform Settings**.
2. On the General page, click the Default Radix drop-down menu.
3. From the drop-down list, select a radix.

Changing the Radix on Individual Objects

To change the radix of a wave object in the Wave window:

1. Right-click the wave object name.
2. Select **Radix** and the format you want from the drop-down menu:
 - Default
 - Binary
 - Hexadecimal
 - Octal
 - ASCII
 - Unsigned Decimal
 - Signed Decimal
 - Signed Magnitude
 - Real



Note: For a description of the usage for Real and Real Settings see [Using Analog Waveforms](#)

3. From the Tcl Console, to change the numerical format of the displayed values, type the following Tcl command:

```
set_property radix <radix> <hdl_object>
```

Where `<radix>` is one the following: `bin`, `unsigned`, `hex`, `dec`, `ascii`, or `oct` and where `<wave_object>` is an object returned by the `add_wave` command.



TIP: If you change the radix in the Wave window, it will not be reflected in the Objects window.

Customizing the Waveform

Using Analog Waveforms

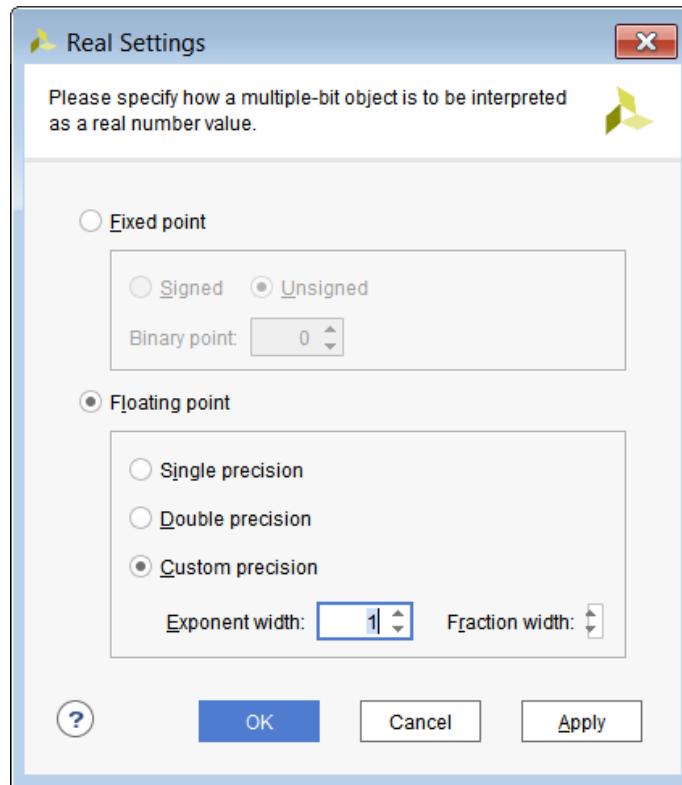
Using Radixes and Analog Waveforms

Bus values are interpreted as numeric values, which are determined by the radix setting on the bus wave object, as follows:

- Binary, octal, hexadecimal, ASCII, and unsigned decimal radices cause the bus values to be interpreted as unsigned integers.
- If any bit in the bus is neither 0 nor 1, the entire bus value is interpreted as 0.
- The signed decimal and signed magnitude radices cause the bus values to be interpreted as signed integers.
- Real radices cause bus values to be interpreted as fixed point or floating point real numbers, based on settings of the Real Settings dialog box.

To set a wave object to the Real radix:

1. In the waveform configuration window, select an HDL object, and right-click to open the popup menu.
2. Select **Radix → Real Settings** to open the **Real Settings** dialog box, shown in the following figure.



You can set the radix of a wave to **Real** to display the values of the object as real numbers. Before selecting this radix, you must choose settings to instruct the waveform viewer how to interpret the bits of the values.

The Real Setting dialog box options are:

- **Fixed Point:** Specifies that the bits of the selected bus wave object(s) is interpreted as a fixed point, signed, or unsigned real number.
- **Binary Point:** Specifies how many bits to interpret as being to the right of the binary point. If Binary Point is larger than the bit width of the wave object, wave object values cannot be interpreted as fixed point, and when the wave object is shown in Digital waveform style, all values show as <Bad Radix>. When shown as analog, all values are interpreted as 0.
- **Floating Point:** Specifies that the bits of the selected bus wave object(s) should be interpreted as an IEEE floating point real number.

Note: Only single precision and double precision (and custom precision with values set to those of single and double precision) are supported.

Other values result in <Bad Radix> values as in Fixed Point. Exponent Width and Fraction Width must add up to the bit width of the wave object, or else <Bad Radix> values result.



TIP: If the row indices separator lines are not visible, you can turn them on in the [Using the Waveform Settings Dialog Box](#), to make them visible.

Displaying Waveforms as Analog



IMPORTANT! When viewing an HDL bus object as an analog waveform—to produce the expected waveform, select a radix that matches the nature of the data in the HDL object. For example:

- If the data encoded on the bus is a 2's-compliment signed integer, you must choose a signed radix.
- If the data is floating point encoded in IEEE format, you must choose a real radix.

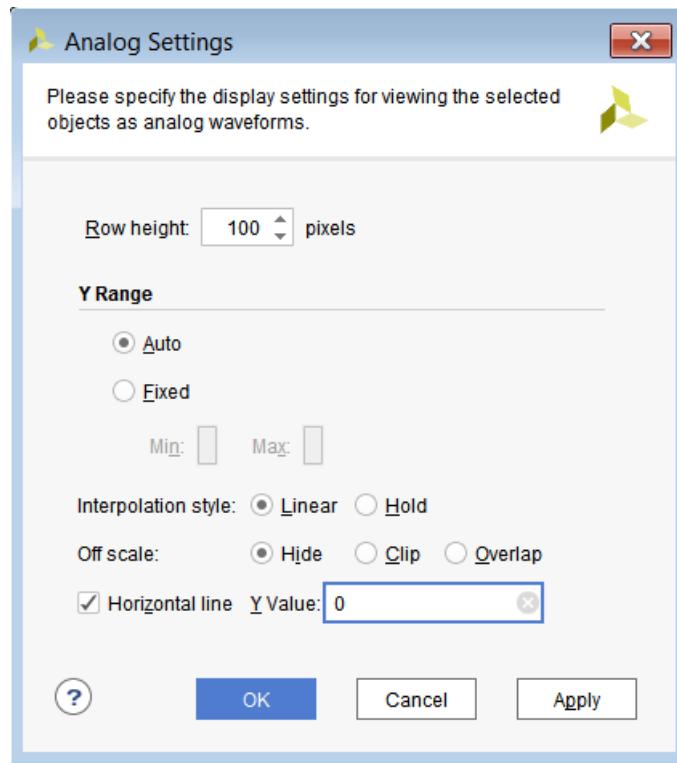
Customizing the Appearance of Analog Waveforms

To customize the appearance of an analog waveform, right-click an HDL object in the Name column of the waveform configuration window and select **Waveform Style** from the drop-down menu. A popup menu appears, showing the following options:

- **Analog:** Sets the waveform to Analog.
- **Digital:** Sets the waveform object to Digital.
- **Analog Settings:** Opens the Analog Settings dialog box (shown in the following figure), which provides options for the analog waveform display.

The Wave window can display analog waveforms only for buses that are 64 bits wide or smaller.

Figure 24: Analog Settings Dialog Box



Analog Settings Dialog Box Option Descriptions

- **Row Height:** Specifies how tall to make the select wave object(s), in pixels. Changing the row height does not change how much of a waveform is exposed or hidden vertically, but rather stretches or contracts the height of the waveform.

When switching between Analog and Digital waveform styles, the row height is set to an appropriate default for the style (20 for digital, 100 for analog).



TIP: If the row indices separator lines are not visible, enable the checkbox in the Waveform Settings to turn them on. [Using the Waveform Settings Dialog Box](#) for information on how to change the options settings. You can also change the row height by dragging the row index separator line to the left and below the waveform name.

- **Y Range:** Specifies the range of numeric values to be shown in the waveform area.
 - **Auto:** Specifies that the range should continually expand whenever values in the visible time range of the window are discovered to lie outside the current range.
 - **Fixed:** Specifies that the time range is to remain at a constant interval.
 - **Min:** Specifies the value displays at the bottom of the waveform area.
 - **Max:** Specifies the value displays at the top.

Both values can be specified as floating point; however, if the wave object radix is integer, the values are truncated to integers.

- **Interpolation Style:** Specifies how the line connecting data points is to be drawn.
 - **Linear:** Specifies a straight line between two data points.
 - **Hold:** Specifies that of two data points, a horizontal line is drawn from the left point to the X-coordinate of the right point, then another line is drawn connecting that line to the right data point, in an L shape.
- **Off Scale:** Specifies how to draw waveform values that lie outside the Y range of the waveform area.
 - **Hide:** Specifies that outlying values are not shown, such that a waveform that reaches the upper or lower bound of the waveform area disappears until values are again within the range.
 - **Clip:** Specifies that outlying values be altered so that they are at the top or bottom of the waveform area, so a waveform that reaches the upper- or lower-bound of the waveform area follows the bound as a horizontal line until values are once again within the range.
 - **Overlap:** Specifies that the waveform be drawn wherever its values are, even if they lie outside the bounds of the waveform area and overlap other waveforms, up to the limits of the Wave window itself.
- **Horizontal Line:** Specifies whether to draw a horizontal rule at the given value. If the checkbox is on, a horizontal grid line is drawn at the vertical position of the specified Y value, if that value is within the Y range of the waveform.

As with Min and Max, the Y value accepts a floating point number but truncates it to an integer if the radix of the selected wave objects is an integer.

Waveform Object Naming Styles

There are options for renaming objects, viewing object names, and changing name displays.

Renaming Objects

You can rename any wave object in the waveform configuration, such as design wave objects, dividers, groups, and virtual buses.

1. Select the object name in the **Name** column.
2. Right-click and select **Rename** from the popup menu.
The Rename dialog box opens.
3. Type the new name in the Rename dialog box, and click **OK**.

Note: Changing the name of a design wave object in the wave configuration does not affect the name of the underlying HDL object.

Changing the Object Display Name

You can display the full hierarchical name (long name), the simple signal or bus name (short name), or a custom name for each design wave object. The object name displays in the Name column of the wave configuration. If the name is hidden:

1. Expand the Name column until you see the entire name.
2. In the Name column, use the scroll bar to view the name.

To change the display name:

1. Select one or more signal or bus names. Use Shift+click or Ctrl+click to select many signal names.
2. Right-click and select **Name** from the drop-down menu. A popup menu appears, showing the following options:
 - **Long** to display the full hierarchical name of the design object.
 - **Short** to display the name of the signal or bus only.
 - **Custom** to display the custom name given to the object when renamed. See [Changing the Object Display Name](#).



TIP: Renaming a wave object changes the name display mode to Custom. To restore the original name display mode, change the display mode to Long or Short, as described above. Long and Short names are meaningful only to design wave objects. Other wave objects (dividers, groups, and virtual buses) display their Custom names by default and display an ID string for their Long and Short names.

Reversing the Bus Bit Order

You can reverse the bus bit order in the wave configuration to switch between MSB-first (big endian) and LSB-first (little endian) bit order for the display of bus values.

To reverse the bit order:

1. Select a bus.
2. Right-click and select **Reverse Bit Order**.

The bus bit order reverses. The Reverse Bit Order command is marked to show that this is the current behavior.



IMPORTANT! The Reverse Bit Order command operates only on the values displayed on the bus. The command does not reverse the list of bus elements that appears below the bus when you expand the bus wave object.



TIP: The index ranges displayed on Long and Short names of buses indicate the bit order in bus elements. For example, after applying Reverse Bit Order on a bus `bus[0:7]`, the bus displays `bus[7:0]`.

Changing the Format of SystemVerilog Enumerations

A SystemVerilog enumeration is an HDL object with numerical values for which text labels are defined to represent specific values. For example, an enumeration might define LABEL1 to represent the value 1 and LABEL2 to represent the value 5. The **Show As Enumeration** option on the context menu lets you specify whether to show enumeration values using their given labels or numerically. In the previous example, if **Show As Enumeration** is on, a value of 5 appears as LABEL2. If the option is off, the value 5 appears as in whatever radix is set for the enumeration, as shown in the Radix menu.

To display enumerations using labels:

1. Select an enumeration
2. Right-click and check **Display As Enumeration**

To display enumerations numerically:

1. Select an enumeration
2. Right-click and uncheck **Display As Enumeration**

Note: Enumeration values for which there is no defined label always display numerically, regardless of the **Display As Enumeration** setting. The **Display As Enumeration** option is enabled only for SystemVerilog enumeration objects.

Controlling the Waveform Display

You can control the waveform display using:

- Resizing handles between the Name, Value, and waveform columns of the wave window
- Scroll combinations with the mouse wheel
- Zoom feature buttons in the Wave window sidebar
- Zoom combinations with the mouse wheel
- Vivado IDE Y-Axis zoom gestures
- Vivado simulation X-Axis zoom gestures. See the *Vivado Design Suite User Guide: Using the Vivado IDE* ([UG893](#)) for more information about using the mouse to pan and zoom.

Note: In contrast to other Vivado Design Suite graphic windows, zooming in a Wave window applies to the X (time) axis independent of the Y axis. As a result, the **Zoom Range X** gesture, which specifies a range of time to which to zoom the window, replaces the **Zoom to Area** gesture of other Vivado Design Suite windows.



TIP: Saving a WCFG file records wave window settings in addition to wave objects and markers. Wave window settings include the Name and Value column widths, zoom level, scroll position, expansion state of groups and buses, and the position of the main cursor.

Using the Column Resizing Handles

To change the width of the Name or Value column, position the mouse over the vertical bar to the right of the column until the mouse cursor changes shape, then drag the mouse left or right to narrow or widen the column as desired.

Note: You may need to widen the Value column first to widen the Name column, if the Value column's width is already at its minimum.

Scrolling with the Mouse Wheel

Click within the wave window to scroll up and down with the mouse wheel. You can also scroll the waveform left and right with the mouse wheel in combination with the Shift key.

Using the Zoom Feature Buttons

There are zoom functions such as Zoom in, Zoom Out, and Zoom Fit as menu buttons in the Wave window that let you zoom in and out of a wave configuration as needed.



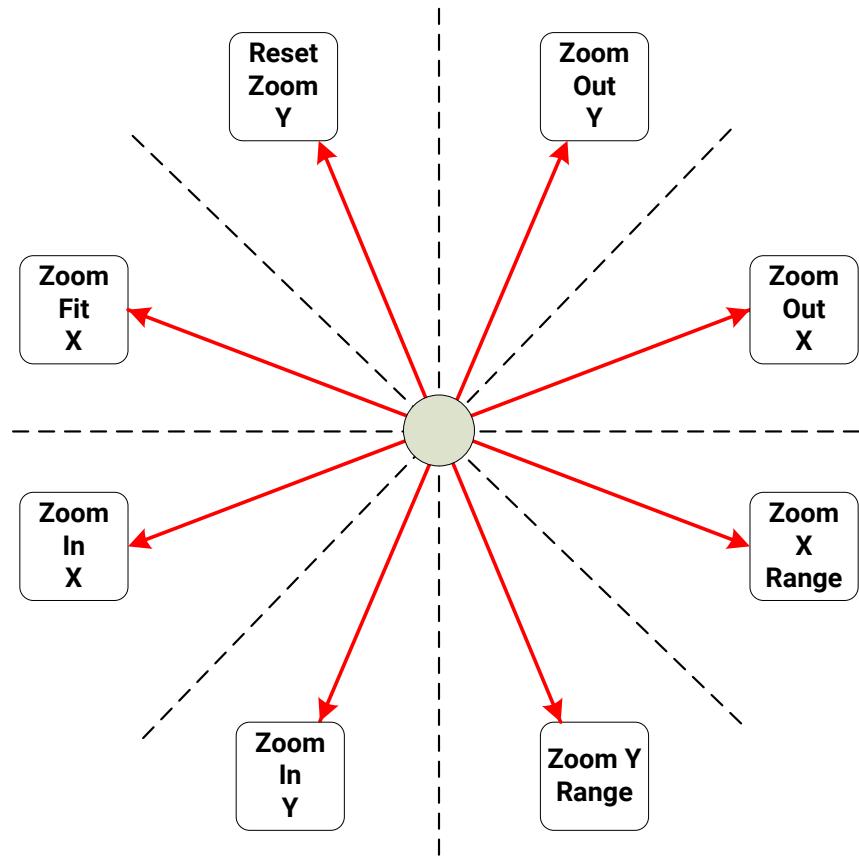
Zooming with the Mouse Wheel

Click within the waveform area and use the mouse wheel in combination with the Ctrl key to zoom in and out, emulating the operation of the dials on an oscilloscope.

Y-Axis Zoom Gestures for Analog Waveforms

In addition to the zoom gestures supported for zooming in the X dimension, when over an analog waveform, additional zoom gestures are available, as shown in the following figure.

Figure 25: Analog Zoom Options



To invoke a zoom gesture, hold down the left mouse button and drag in the direction indicated in the diagram, where the starting mouse position is the center of the diagram.

The additional zoom gestures are:

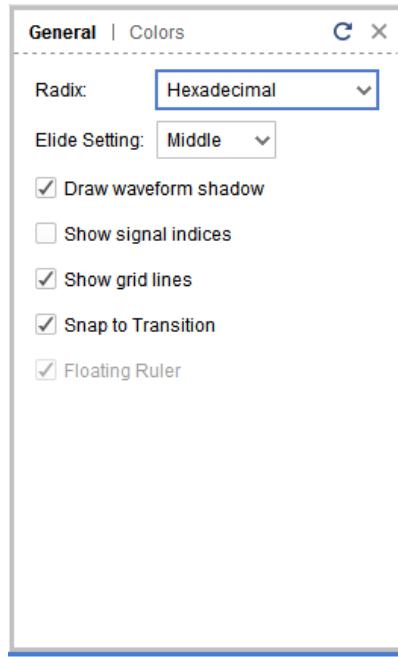
- Zoom Out Y: Zooms out in the Y dimension by a power of 2 determined by how far away the mouse button is released from the starting point. The zoom is performed such that the Y value of the starting mouse position remains stationary.
- Zoom Y Range: Draws a vertical curtain which specifies the Y range to display when the mouse is released.
- Zoom In Y: Zooms in toward the Y dimension by a power of 2 determined by how far away the mouse button is released from the starting point. The zoom is performed such that the Y value of the starting mouse position remains stationary.
- Reset Zoom Y: Resets the Y range to that of the values currently displayed in the Wave window and sets the Y Range mode to Auto.

All zoom gestures in the Y dimension set the Y Range analog settings. Reset Zoom Y sets the Y Range to Auto, whereas the other gestures set Y Range to Fixed.

Using the Waveform Settings Dialog Box

Click the **Settings** button  to open the Waveform Settings as shown in the following figure.

Figure 26: Waveform Settings



From the General tab, you can configure the following Waveform Settings:

- **Radix:** Sets the numerical format to use for newly-created design wave objects.
- **Elide Setting:** Controls truncation of signal names that are too long for the Wave window.
 - **Left** truncates the left end of long names.
 - **Right** truncates the right end of long names.
 - **Middle** preserves both the left and right ends, omitting the middle part of long names.
- **Draw Waveform Shadow:** Creates a shaded representation of the waveform.
- **Show signal indices:** Displays the row numbers to the left of each wave object name. You can drag the lines separating the row numbers to change the height of a wave object.
- **Show grid lines:** Displays the wave window with grid lines.
- **Snap to Transition:** When selected, causes dragged cursors and markers to gravitate to waveform transitions near the mouse cursor. See [Using Cursors](#) for more information.
- **Floating Ruler:** Displays the floating ruler whenever the secondary cursor is visible or a marker is selected. See [Using the Floating Ruler](#) for more information.



TIP: If Floating Ruler option appears disabled (unchecked) in the Settings dialog box, use Shift+Click on the Wave window to make the secondary cursor visible. This action results in enabling the Floating Ruler option in the Settings dialog box.

- From the **Colors** tab, you can set colors of items within the waveform.

Changing the Display of the Time Scale

Right-click above the ruler to display the time scale menu. This menu lets you select how you want to display time values on the time scale.

The following are the options on the timescale menu:

- Auto: The timescale chooses time units suitable for the wave window's zoom level.
- Default: Displays the time units corresponding to the precision of the simulation that was determined when the HDL design was compiled.
- Samples: Displays the time in discrete sample numbers instead of fractions of a second (not available for HDL simulation).
- User: User-defined time units (not available for HDL simulation).
- fs: Displays time units in femtoseconds.
- ps: Displays time units in picoseconds.
- ns: Displays time units in nanoseconds.
- us: Displays time units in microseconds.
- ms: Displays time units in milliseconds.
- s: Displays time units in seconds

Organizing Waveforms

The following subsections describe the options that let you organize information within a waveform.

Grouping Signals and Objects

A Group is an expandable and collapsible container for organizing related sets of wave objects. The Group itself displays no waveform data but can be expanded to show its contents or collapsed to hide them. You can add, change, and remove groups.

To add a Group:

- In a Wave window, select one or more wave objects to add to a group.

Note: A group can include dividers, virtual buses, and other groups.

2. Right-click and select **New Group** from the context menu.

This adds a Group that contains the selected wave object to the wave configuration.

In the Tcl Console, type `add_wave_group` to add a new group.

A Group is represented with the **Group** button . You can move other HDL objects to the group by dragging and dropping the signal or bus name.

The new Group and its nested wave objects saves when you save the waveform configuration file.

You can move or remove Groups as follows:

- Move Groups to another location in the Name column by dragging and dropping the group name.
- Remove a Group by highlighting it, right-click and select **Ungroup** from the popup menu. Wave objects formerly in the Group are placed at the top-level hierarchy in the wave configuration.

Groups can be renamed also; see [Changing the Object Display Name](#).



CAUTION! The Delete key removes a selected group and its nested wave objects from the wave configuration.

Using Dividers

Dividers create a visual separator between HDL objects to make certain signals or objects easier to see. You can add a divider to your wave configuration to create a visual separator of HDL objects, as follows:

1. In a Name column of the Wave window, click a signal to add a divider below that signal.
2. Right-click and select **New Divider**.

The new divider is saved with the wave configuration file when you save the file.

Tcl command: `add_wave_divider`

You can move or delete Dividers as follows:

- To move a Divider to another location in the waveform, drag and drop the divider name.
- To delete a Divider, highlight the divider, and click the Delete key, or right-click and select **Delete** from the context menu.

Dividers can be renamed also; see [Changing the Object Display Name](#).

Defining Virtual Buses

You define a virtual bus to the wave configuration, which is a grouping to which you can add logic scalars and vectors.

The virtual bus displays a bus waveform, whose values are composed by taking the corresponding values from the added scalars and arrays in the vertical order that they appear under the virtual bus and flattening the values to a one-dimensional vector.

To add a virtual bus:

1. In a wave configuration, select one or more wave objects to add to a virtual bus.
2. Right-click and select **New Virtual Bus** from the popup menu.

The virtual bus is represented with the **Virtual Bus** button .

Tcl Command: `add_wave_virtual_bus`

You can move other logical scalars and arrays to the virtual bus by dragging and dropping the signal or bus name.

The new virtual bus and its nested items save when you save the wave configuration file. You can also move it to another location in the waveform by dragging and dropping the virtual bus name.

You can rename a virtual bus; see [Changing the Object Display Name](#).

To remove a virtual bus, and ungroup its contents, highlight the virtual bus, right-click, and select **Ungroup** from the popup menu.



CAUTION! The Delete key removes the virtual bus and nested HDL objects within the bus from the wave configuration.

Analyzing Waveforms

The following subsections describe available features that help you analyze the data within the waveform.

Using Cursors

Cursors are temporary time markers that can be moved frequently for measuring the time between two waveform edges.

Placing Main and Secondary Cursors

You can place the main cursor with a single left-click in the Wave window.

To place a secondary cursor, **Ctrl+Click**, hold the waveform, and drag either left or right. You can see a flag that labels the location at the top of the cursor. Alternatively, you can hold the **Shift** key and click a point in the waveform.

If the secondary cursor is not already on, this action sets the secondary cursor to the present location of the main cursor and places the main cursor at the location of the mouse click.

Note: To preserve the location of the secondary cursor while positioning the main cursor, hold the **Shift** key while clicking. When placing the secondary cursor by dragging, you must drag a minimum distance before the secondary cursor appears.

Moving Cursors

To move a cursor, hover over the cursor until you see the grab symbol, and click and drag the cursor to the new location.

As you drag the cursor in the Wave window, you see a hollow or filled-in circle if the **Snap to Transition** waveform setting is selected, which is the default behavior.

- A hollow circle  under the mouse indicates that you are between transitions in the waveform of the selected signal.
- A filled-in circle  under the mouse indicates that the cursor is locked in on a transition of the waveform under the mouse or on a marker.

A secondary cursor can be hidden by clicking anywhere in the Wave window where there is no cursor, marker, or floating ruler.

Finding the Next or Previous Transition on a Waveform

The Waveform window contains buttons for jumping the main cursor to the next or previous transition of selected waveform or from the current position of the cursor.

To move the main cursor to the next or previous transition of a waveform:

1. Ensure the wave object in the waveform is active by clicking the name.

This selects the wave object, and the waveform display of the object displays with a thicker line than usual.

2. Click the **Next Transition** or **Previous Transition**   buttons in the waveform toolbar (?), or use the right or left keyboard arrow key to move to the next or previous transition, respectively.



TIP: You can jump to the nearest transition of a set of waveforms by selecting multiple wave objects together.

Using Markers

Use a marker when you want to mark a significant event within your waveform in a permanent fashion. Markers let you measure times relevant to that marked event.

You can add, move, and delete markers as follows:

- You add markers to the wave configuration at the location of the main cursor.
 1. Place the main cursor at the time where you want to add the marker by clicking in the Wave window at the time or on the transition.
 2. Right-click **Markers** → **Add Marker** .

A marker is placed at the cursor, or slightly offset if a marker already exists at the location of the cursor. The time of the marker displays at the top of the line.

To create a new wave marker, use the Tcl command:

```
add_wave_marker <time> <timeunit> -name <name of the marker> -into <wcfg file>
```

- You can move the marker to another location in the Wave window using the drag and drop method. Click the marker label (at the top of the marker or marker line) and drag it to the location.
 - As you drag the marker in the Wave window, you see a hollow or filled-in circle if the Snap to Transition option is selected in Waveform Settings window, which is the default behavior.
 - A filled-in circle  indicates that you are hovering over a transition of the waveform for the selected signal or over another marker.
 - For markers, the filled-in circle is white.
 - A hollow circle  indicates that the marker is locked in on a transition of the waveform under the mouse or on another marker.

Release the mouse key to drop the marker to the new location.

- You can delete one or all markers with one command. Right-click over a marker, and do one of the following:
 - Select **Delete Marker** from the popup menu to delete a single marker.
 - Select **Delete All Markers** from the popup menu to delete all markers.

You can also use the Delete key to delete a selected marker.

See the Vivado Design Suite help or the [Vivado Design Suite Tcl Command Reference Guide \(UG835\)](#) for command usage.

Using the Floating Ruler

The floating ruler assists with time measurements using a time base other than the absolute simulation time shown on the standard ruler at the top of the Wave window.

You can display (or hide) the floating ruler and drag it to change the vertical position in the Wave window. The time base (time 0) of the floating ruler is the secondary cursor, or, if there is no secondary cursor, the selected marker.

The floating ruler is visible only when the secondary cursor or a marker is present.

1. Do either of the following to display or hide a floating ruler:
 - Place the secondary cursor.
 - Select a marker.
2. In the Waveform Settings window, enable (check) the **Floating Ruler** option.

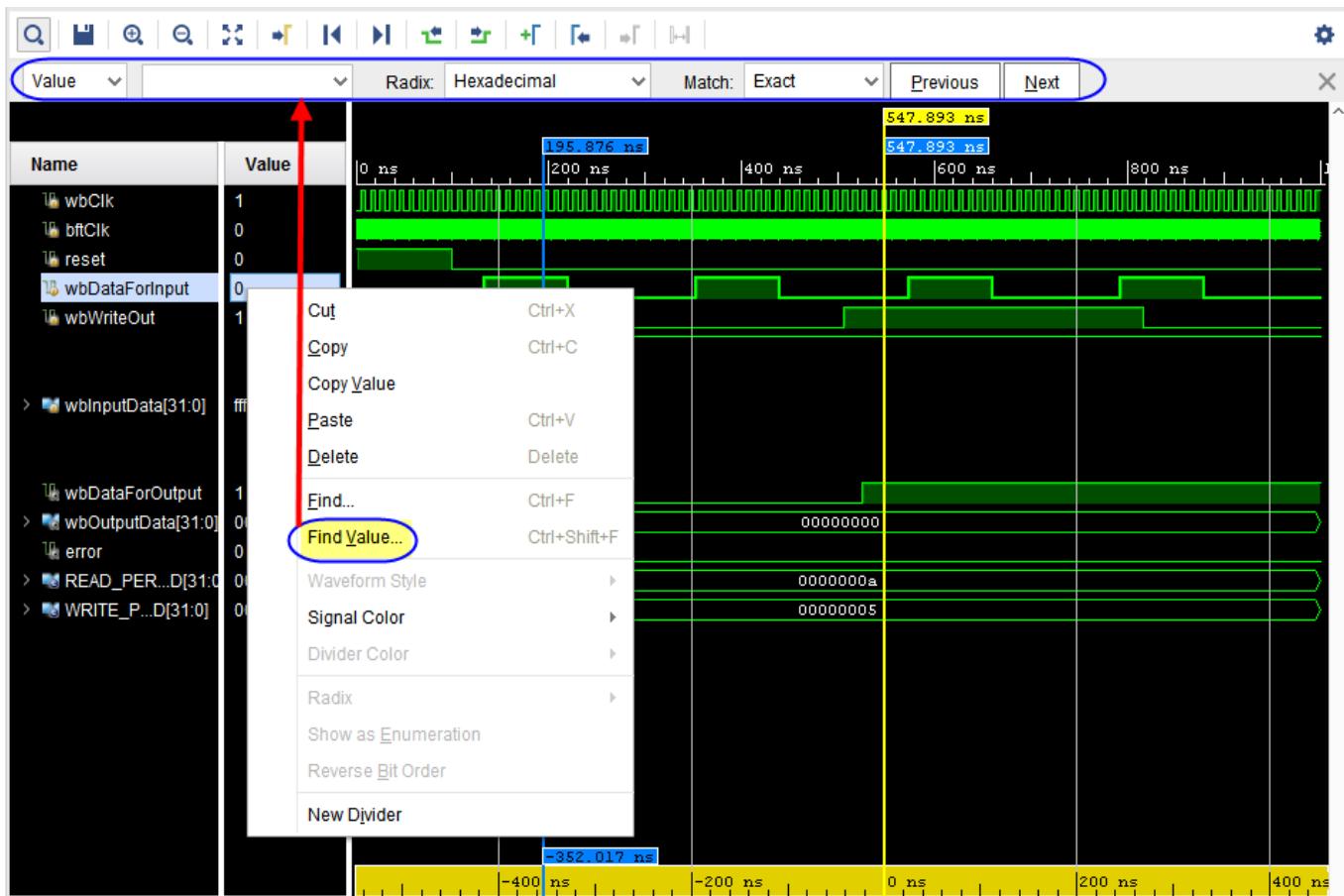
You only need to follow this procedure the first time. The floating ruler displays each time you place the secondary cursor or select a marker.

Uncheck/disable the **Floating Ruler** option to hide the floating ruler.

Searching a Value in Waveform Configuration

The Find Toolbar allows you to search one or more waveforms for a specified value. You can search for either an exact value, such as 23FF, or a pattern that matches a set of values, such as "any value whose first two digits are 23 and whose fourth digit is F."

Figure 27: Find Value option and Find Toolbar



IMPORTANT! This search feature supports only scalar and vector (1-D) wave objects of a “logic” type. Logic types include 2-state and 4-state types of Verilog/SystemVerilog and bit and std_logic of VHDL.

To perform the search:

1. In the Name column, select one or more design wave objects (wave objects that have waveforms).
2. Right-click one of the selected wave objects in either the Name column or Value column and choose the **Find Value** option to activate the Find Toolbar.
3. On the Find Toolbar, choose a radix for your search value from the Radix drop down list. The search feature supports the following radices:
 - Binary
 - Hexadecimal
 - Octal
 - Unsigned Decimal
 - Signed Decimal

4. In the blank text box on the Find Toolbar, enter a value pattern consisting of a string of digits valid for the radix you chose. Valid digits include numeric digits, VHDL MVL 9 literals (U, X, 0, 1, Z, W, L, H, -), and Verilog literals (0, 1, x, z).

Note: If you enter an invalid digit, the text box turns red, and an error message appears at the right side of the toolbar. The set of valid numeric digits depends on the radix. For example, if you chose the Octal radix, numeric digits are those between 0 and 7. Numeric digits for hexadecimal include 0 through 9 and A through F (or a through f). You may enter the special digit “!” to specify a match with any digit value. For example, the Octal value pattern “12.4” matches occurrences of 1234, 1204, and 12X4 encountered in the waveform.

5. Choose a match style from the following options in the Match drop down list:

- **Exact:** Waveform values must contain the same number of digits as in the value pattern to be considered a match. For example, a value pattern of “1234” matches occurrences of 1234 encountered in the waveform but not 123 or 12345.



TIP: With the Exact match style you may omit leading zeros from the value pattern. For example, to find the value 0023 in the waveform, you may specify a value pattern of “0023” or simply “23.”

- **Beginning:** Any waveform value whose beginning digits match the value pattern is considered a match. For example, a value pattern of “1234” matches occurrences of 1234 and 12345 encountered in the waveform but not 1235 or 123. This option is available only for radices binary, octal, and hexadecimal.
- **End:** Any waveform value whose ending digits match the value pattern is considered a match. For example, a value pattern of “1234” matches occurrences of 1234 and 91234 encountered in the waveform but not 1235 or 234. This option is available only for radices binary, octal, and hexadecimal.
- Click the Next button or press the Enter key to move the main cursor forward to the nearest match, or click the Previous button to move the main cursor backward to the nearest match. With multiple wave objects selected, the cursor stops on the nearest match of any of the selected wave objects.



TIP: If there are no matches in the requested direction, the cursor remains stationary and a “Value not found” message appears on the right side of the toolbar.

Analyzing AXI Interface Transactions

If you compose your design as a block design using the Vivado IP integrator, when you launch the Vivado simulator, Vivado automatically imports the AMBA® AXI interfaces from your design into the Vivado simulator as *protocol instances* to be viewed in the wave window. Once added to the wave window, a protocol instance of an AXI interface shows you the data transactions occurring on that interface during simulation.

Understanding Protocol Instances

An AXI interface consists of a standard set of logic signals as defined by Arm® in the AMBA® AXI and ACE Protocol Specification and AMBA® 4 AXI4-Stream Protocol Specification. These signals convey the data transactions encoded as logic events as described by the specification. The Vivado simulator makes those signals available for viewing directly in the wave viewer, however, it can be difficult to visualize what transactions are happening from the signals alone.

To make it easier to view the transactions, the Vivado simulator provides a feature that analyzes the signal activity and produces new signals that summarize the activity at the transaction level. This process is called as *protocol analysis*. For each AXI interface the Vivado simulator creates a new design object called a *protocol instance* that represents the AXI interface and the inputs and outputs of protocol analysis. The protocol instance typically resides in the same scope as its input signals.

Using the IP Integrator to Mark an AXI Interface to View in the Vivado Simulator

The Vivado IP integrator provides a feature for identifying AXI interfaces to display in the Vivado simulator's wave viewer directly from the block design window. Perform the following steps to mark an AXI interface for viewing in the Vivado simulator:

1. Locate the AXI interface you want to view.
2. Right-click the corresponding net connection (orange line as shown in the following figure).
3. Click **Mark Simulation**.

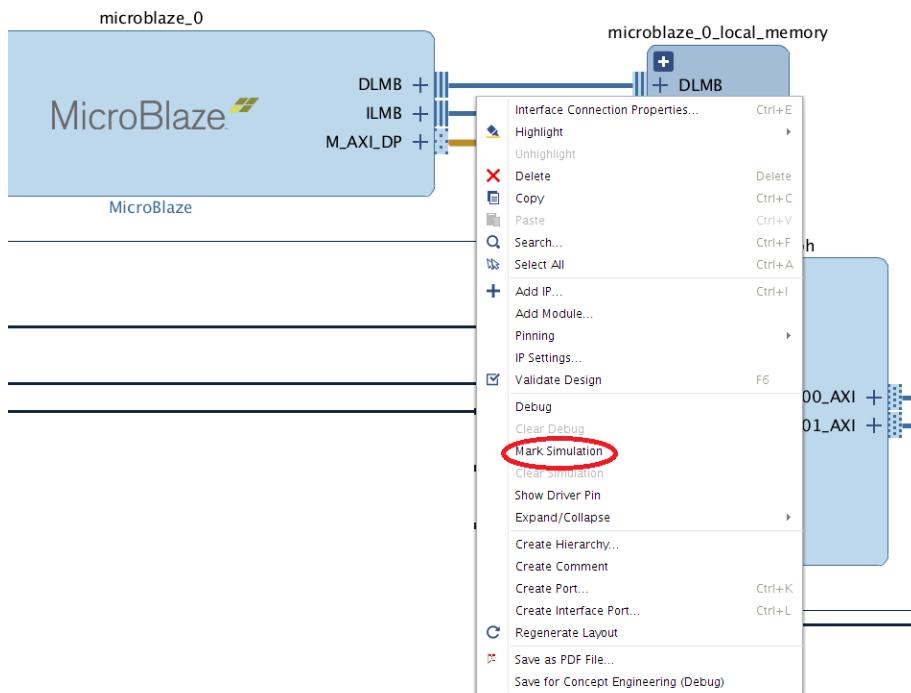
Note: Mark Simulation option can only be applied on an AXI interface.

4. Repeat steps 1-3 to mark additional interfaces.
5. Click **Clear Simulation** option to clear a marked AXI interface.

Note: Clear Simulation option is available only when an AXI interface is marked.

6. Launch the Vivado simulator. Save the block design if prompted.

When the Vivado simulator starts, the interfaces you marked appear in the Wave window. If the Vivado project is customized to open a wave configuration automatically, the marked interfaces are added to the wave configuration if not already present. If the Vivado project is not customized to open a wave configuration, the Vivado simulator creates a default wave configuration containing the marked interfaces in lieu of the usual top-level HDL signal list.



CAUTION! Some AXI interfaces which are internal to an AXI interconnect may not display correctly in the wave viewer depending on the configuration of the interconnect. It is recommended that you mark interfaces only on the boundary of an interconnect.

Finding Protocol Instances in the Vivado Simulator

When the Vivado simulator launches, it scans the design and its input files to locate protocol instances. The results of the scan appear in the Tcl console near the top of the simulator output as shown in the following figure. You can copy a protocol instance path from the Tcl console and paste it into a Tcl command.

Figure 28: Protocol Instances Identified in the Tcl Console

```
TIME RESOLUTION IS 1 ps
INFO: [Wavedata 42-564] Reading protoinst file protoinst_files/base_mb.protoinst
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//axi_gpio_0/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//axi_uartlite_0/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0/M_AXI_DP
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/M00_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/M01_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/S00_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m00_couplers/M_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m01_couplers/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/m01_couplers/M_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/s00_couplers/M_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/s00_couplers/S_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/xbar/M00_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/xbar/M01_AXI
INFO: [Wavedata 42-564] Found protocol instance at /system_tb/base_mb_wrapper/base_mb_i//microblaze_0_axi_periph/xbar/S00_AXI
concurrent system th tcl
```

Finding Protocol Instances in the Objects Window

Protocol instance objects appear in the Objects window in the scope containing the corresponding AXI interface signals. Perform the following steps to find a protocol instance using the Scope window:

1. In the Scope window, select the scope containing AXI interface signals.

Note: The scope hierarchy roughly matches with your block design.

2. Locate the protocol instance for an AXI interface of a block in your block design. Select the scope name that matches with the instance name of your block.

Perform the following steps to find a protocol instance using the Objects window:

1. In the Objects window scroll to the bottom of the list.
2. Locate the protocol instance name that matches with the port name of the AXI interface in your block design. AXI port names usually end with _AXI and are frequently M_AXI or S_AXI.



TIP: Protocol instances have a port mode of Internal Signal. To facilitate searching for protocol instances in the Objects window, you can hide all objects except the Internal Signal object. Click the gear icon, deselect the Select All check box and select the Internal Signal check box. To restore the Objects window select the Select All check box.

Finding Protocol Instances Using a Tcl Command

Protocol instance objects have a Tcl type field of proto_inst. You can use the get_objects Tcl command to locate protocol instances in or under a specific scope.

Use the Following Command to Locate All Protocol Instances in the Design:

```
get_objects /* -r -filter {type==proto_inst}
```

Use the Following Command to Locate All Protocol Instances in a Scope:

```
get_objects <Design scope hierarchy>/* -filter {type==proto_inst}
```

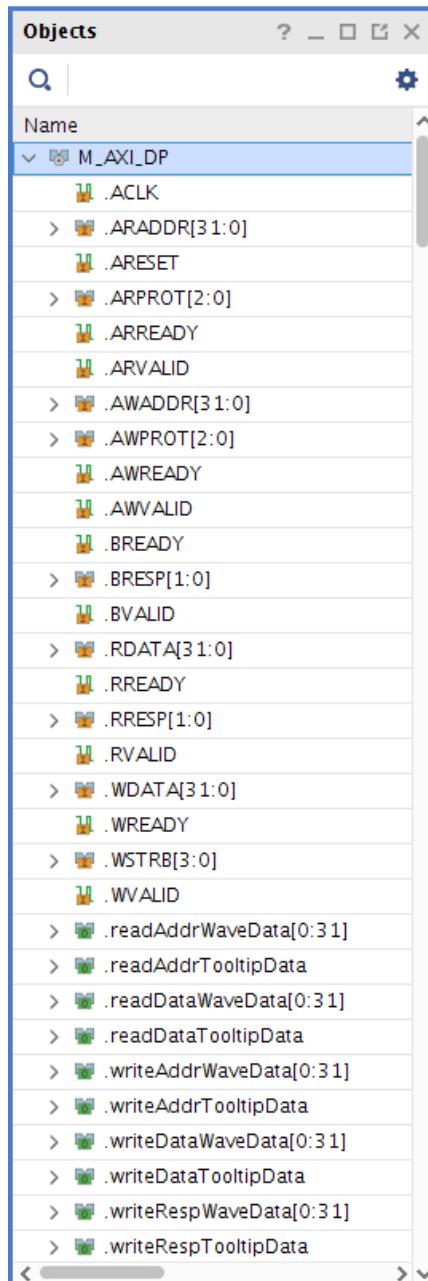
Use the Following Command to Locate All Protocol Instances in or under a Scope:

```
get_objects /system_tb/base_mb_wrapper/base_mb_i/* -r -filter {type==proto_inst}
```

Protocol Instance in the Objects Window

In the Objects window, a protocol instance appears as an aggregate design object that has the same name as the port name of the AXI interface seen in the block design. Click the arrow expand button to view the inputs and outputs of protocol analysis for the protocol instance as shown in the following figure for a protocol instance named M_AXI_DP.

Figure 29: Protocol Instance in the Objects Window

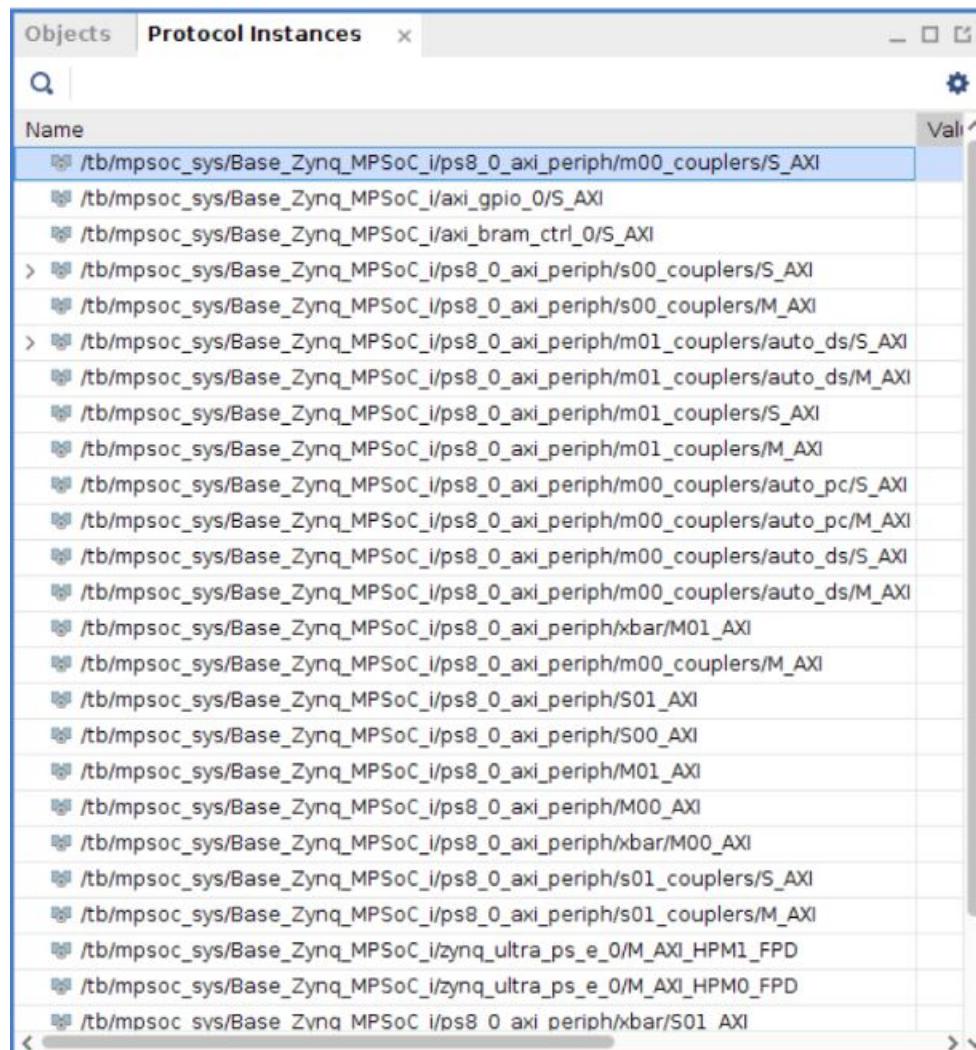


Note: To optimize computer resources, the protocol instance does not display its child objects until the protocol instance is added to the wave window for the first time.

- **Protocol Instances Window:**

The protocol instances window contains the complete protocol instance list present in a given design. It has an absolute path to the protocol instance to ensure that an instance with same name can be differentiated based on the path.

Figure 30: Protocol Instances Window



- **Protocol Instance Inputs:**

The protocol instance input signals shown with an orange icon are the aliases of the AXI signals from the HDL design. Hover over an input to see a tool tip giving full path of the alias as well as the full path of the actual (aliased) signal. From a protocol instance input you can also jump the Scope and Objects window to the actual signal. Perform the following steps to go to the actual signal of a protocol instance input:

1. To view all signal types in the Objects window, click the gear icon and select the **Check All** check box.
2. Right click the protocol instance input.
3. Select **Go To Actual**.



TIP: If you would like to save the protocol instance input signals in a wave configuration (WCFG) file, add the actual signals of the input signals instead. Because, the protocol instance inputs and outputs are created only after a WCFG file is loaded, your saved inputs will be missing from the wave configuration. The protocol instance outputs cannot be saved in a WCFG file.

- **Protocol Instance Outputs:**

The protocol instance outputs are shown with a green O icon. These are special signals of the protocol instance that have no counterparts in the HDL design. The output signals produce events meaningful only to the wave viewer for displaying transactions.



CAUTION! The set of protocol instance output signals and their contents are subject to change from one Vivado release to the next. It is recommended not to depend on the specific behavior of protocol instance output signals when scripting using Tcl.

Adding Protocol Instances to the Wave Window

You can add any protocol instance present in the design to the wave window. Adding a protocol instance to the wave window causes the Vivado simulator to run protocol analysis on the protocol instance inputs starting from the simulation time 0 regardless of how much simulation time has already elapsed. As protocol analysis utilizes the waveform database (WDB), inputs of all protocol instances are always traced in the waveform database even if you have not requested tracing of the inputs or added the protocol instance to the wave window.

In addition to marking an AXI interface in your IP integrator block design as described in [Using the IP Integrator to Mark an AXI Interface to View in the Vivado Simulator](#) section, you can add a protocol instance to the wave window using the Objects window or a Tcl command.



IMPORTANT! Protocol instances may use more computer resources, it is recommended that you add just the protocol instances you currently need. You can always add additional protocol instances at a later time during the simulation without missing the data.

Perform the following steps to add a protocol instance to the Wave window using the Objects window:

1. To locate the protocol instance in the Objects window, see the steps described in the section [Finding Protocol Instances in the Objects Window](#).
2. Add the protocol instance to the Wave window by the following two ways:
 - a. Right click the protocol instance and choose **Add to Wave**.
 - b. Drag and drop the protocol instance to the Name column of the Wave window.

Perform the following steps to add a protocol instance to the Wave window using a Tcl command:

1. To locate the protocol instance in the Objects window, see the steps described in the section [Finding Protocol Instances in the Vivado Simulator](#).
2. Copy the protocol instance path to the clipboard:
 - a. If you have located the protocol instance in the Objects window, left click the protocol instance to select it and copy the protocol instance path.
 - b. If you have located the protocol instance in the Tcl console, use your mouse to select the protocol instance path and copy it.
 - c. If you have located the protocol instance using the `get_objects` Tcl command, use your mouse to select the text of the protocol instance path in the Tcl console and copy it. Alternatively, you can get objects together as described in the following section.
3. Type `add_wave` and paste the protocol instance name.



TIP: If your protocol instance path contains special characters, surround the path with double braces. For example, `add_wave {{path}}`.

Using `get_objects` Programmatically

When you use the `get_objects` Tcl command as described in [Finding Protocol Instances Using a Tcl Command](#), the command returns the protocol instances as a Tcl list. You can store the list in a Tcl variable:

```
set p [get_objects -r /* -filter {type==proto_inst}]
```

and use the list with the `add_wave` Tcl command to add all the protocol instances in the list:

```
add_wave $p
```

or a specific protocol instance from the list using the built-in `lindex` command as shown in the following example that adds the first protocol instance of the list:

```
add_wave [lindex $p 0]
```

Analyzing Protocol Instances in the Wave Window

This section describes the waveform features common to all interface types. See the following protocol specific sections for more information about a specific interface type (AXI Memory Mapped or AXI4 Stream).

Understanding Protocol Instances in the Wave Window

When you add a protocol instance to the wave window, the Vivado simulator creates a hierarchy of wave objects to represent the protocol instance. You cannot change the structure of the hierarchy. The type of AXI interface determines the hierarchy.

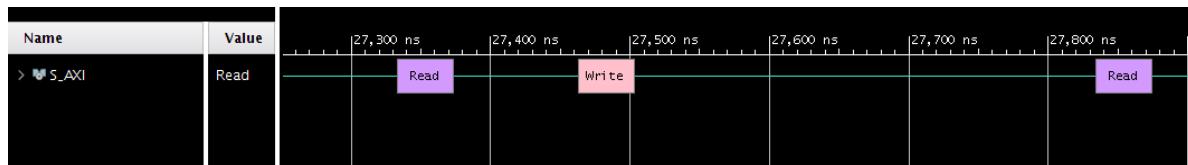


TIP: You may need to view the protocol instance input signals not included in the wave object hierarchy. While you cannot add the signals into the hierarchy, you can add them before or after the hierarchy.

Understanding Transaction Waveforms

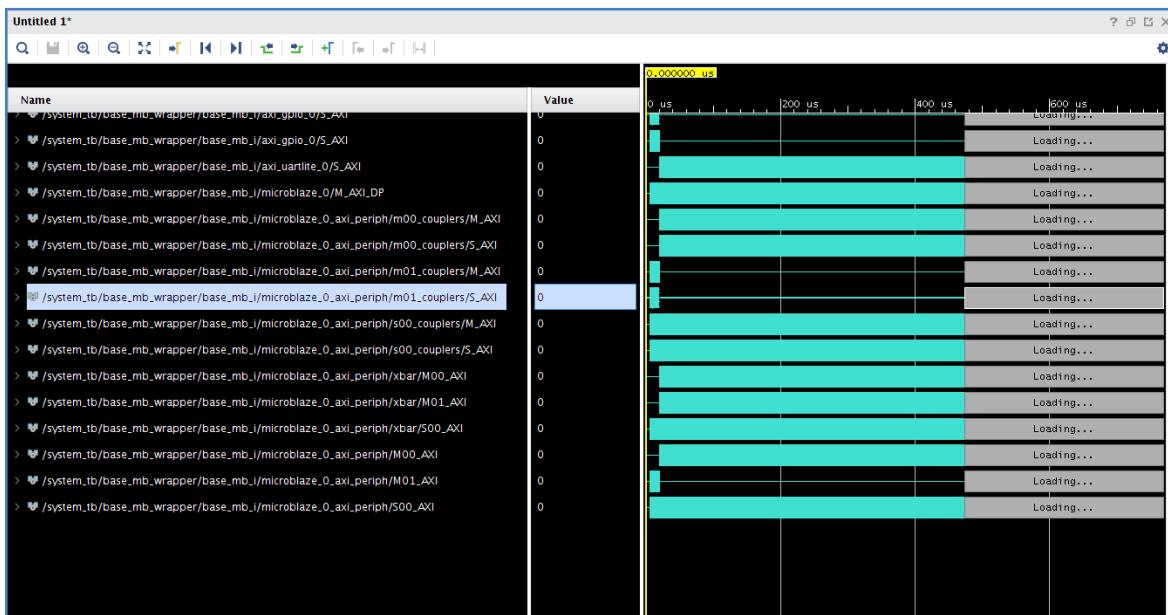
Transaction waveforms differ from other types of waveforms. A transaction waveform displays periods of activity and inactivity of some aspect of the simulated design in contrast with displaying value changes of a signal over time. The following figure shows an example of a transaction waveform. A thin line indicates periods of inactivity, while the rectangles represent periods of activity which are generally called transaction bars. The example in the figure shows three transaction bars.

Figure 31: Transaction Waveform Display



As shown in the following figure, the transaction waveform displays a gray bar with the text Loading while protocol analysis is performed on the inputs of the protocol instance. As the protocol analysis progresses, the gray bars shrink to reveal newly processed transaction data.

Figure 32: Transaction Waveforms Showing Incomplete Protocol Analysis

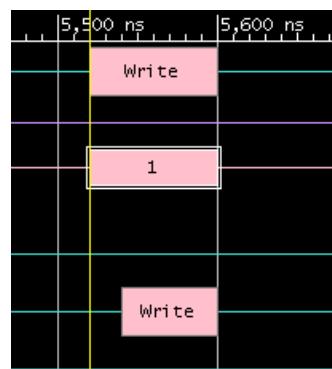


Using Transaction Bars

Selecting Transaction Bars

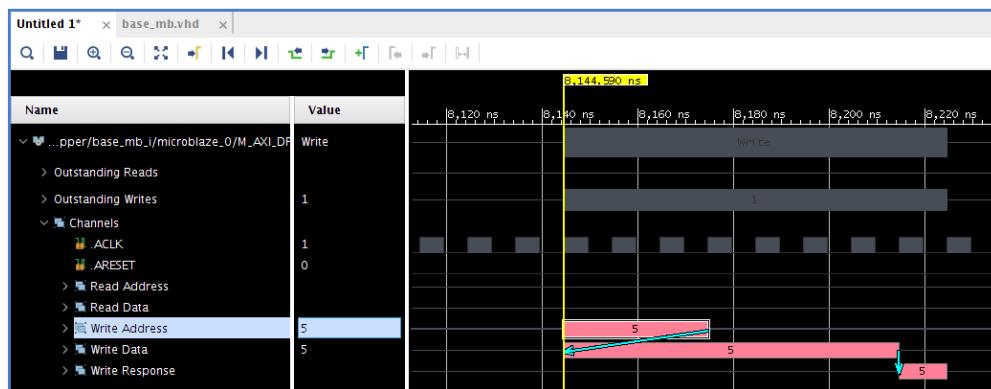
Place your cursor on a transaction bar and left click to select the transaction bar which is highlighted with a double border as shown in the following figure:

Figure 33: A Selected Transaction Bar



If the selected transaction bar is a member of a group of related transaction bars, arrows appear called associations connecting the related transaction bars. The rest of the objects in the wave window appear dim to highlight the group of transaction bars. The following figure shows a selected transaction bar and its related transaction bars connected with associations.

Figure 34: Transaction Bars with Associations



Press the Esc key to clear the transaction bar selection.



TIP: To reposition the main cursor within a transaction waveform, hold the Ctrl key and left click at the desired cursor time.

Navigation Transactions Using Associations

When you click on one end of an association, the selection moves to the transaction bar at the other end of the association, and the wave window scrolls so that the other end is in view.

Note: The wave window may not scroll if the other end is already in view.

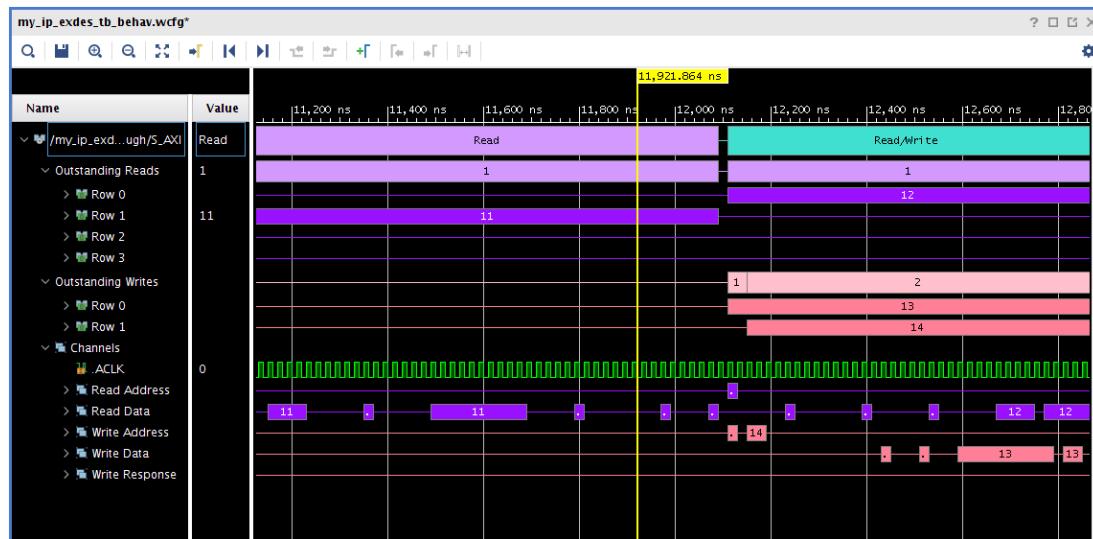
Tool Tips

When you hover over a transaction bar or an association using your mouse, a tool tip displaying extra information about the transaction bar or association may appear depending on the type of protocol instance interface.

Analyzing AXI Memory-Mapped (AXI-MM) Interfaces

This section describes the transaction viewing features specific to AXI-MM protocol instances. A protocol instance of an AXI-MM interface appears in the wave window with a wave object hierarchy as shown in the following figure.

Figure 35: AXI-MM Interface



Understanding the Top Summary Row

The top of the wave object hierarchy of an AXI-MM protocol instance is the top summary row. This transaction waveform shows the overall read and write activity of an AXI interface based on the following rules:

- If one or more AXI read transactions are in progress, the top summary shows a Read transaction bar in purple color.
- If one or more AXI write transactions are in progress, the top summary shows a Write transaction bar in pink color.
- If one or more of AXI read and write transactions are in progress, the top summary shows Read/Write transaction bar in teal color.

An AXI transaction is an abstract concept not to be confused with the graphical transaction bar. It is a complete data exchange carried out using AXI signaling including the Address, Data, and optionally Response phases.



TIP: For performance reasons, the wave viewer does not display the transaction bars in different colors when zoomed out. Instead, it displays all transaction bars in teal color. You need to zoom in to distinguish between read and write transactions.

Understanding the Outstanding Reads and Outstanding Writes Rows

There are a group of outstanding AXI read transactions and a group of write transactions located under the top of the wave object hierarchy of an AXI-MM protocol instance. An AXI transaction is known as outstanding if the interface master has raised `A*VALID` or `WVALID` but the last data phase or optionally response phase has not yet completed. The outstanding reads row shows the current count of outstanding AXI read transactions or is inactive (shown as a thin line) to indicate zero outstanding AXI read transactions. Similarly, the outstanding writes row shows the current count of outstanding AXI write transactions or is inactive to indicate zero outstanding AXI write transactions.

Understanding the Transaction Summary Rows

There is a set of transaction summary rows under each outstanding read and write row labeled as **Row <n>**, where **<n>** is an integer. A transaction summary is a transaction bar that depicts a single AXI transaction starting at the first phase and ending at the last phase of an AXI transaction. The assignment of a transaction summary to a specific numbered row conveys no special meaning. Instead, it prevents overlapping of multiple outstanding AXI transactions in the same row.



TIP: The number of transaction summary rows can increase as simulation progresses. For performance reasons, the wave window updates the rows only when protocol analysis is complete. To see the latest state of the rows during simulation without waiting for the entire simulation to complete, you can pause the simulation and allow the **Loading** bars to disappear.

Each transaction summary is labeled with a sequence number. The first AXI transaction has a sequence number of 1, the second AXI transaction has a sequence number of 2, and so forth. The progression of sequence numbers for reads and writes are separate from each other and from the AXI transactions of all other protocol instances. For example, a particular protocol instance can have an AXI read transaction with the sequence number 16 and a separate AXI write transaction with the sequence number 16.

Understanding Channel Rows

The channels wave object group is collapsed by default. When you expand the group, you see logic signals for the AXI interface clock and reset (if present) and one transaction row for each AXI channel present in the interface.

Note: Not all five channels are necessarily present in an AXI interface. For read only interfaces, the write channels are absent. For write only interfaces, the read channels are absent. Some AXI interfaces that employ the write channels may omit the response channel if the AXI master has no use of the response information.

Each channel row shows a transaction bar summarizing individual handshakes of that AXI channel from VALID to READY, except that the multiple contiguous data beats of the same AXI transaction appear as a single transaction bar. To visually tie all channel transaction bars of an AXI transaction together, each channel transaction bar is tagged with the same sequence number as the corresponding transaction summary. You can expand the channel row to show key AXI signals for that channel.



TIP: You may need to view protocol instance input signals not included in the wave object hierarchy. While you cannot add the signals into the hierarchy, you can add them before or after the hierarchy.



TIP: You may notice that channel transaction bars appear up to one clock cycle after the corresponding AXI signal event. The AXI protocol analyzers consider AXI signal events that occur on or after a positive clock edge to take effect at the following positive clock edge.

When you hover over any channel transaction bar, association, or transaction summary using your mouse, a tool tip appears showing the values of the informational AXI address channel signals from the address phase of the AXI transaction.

Note: Optional AXI address channel signals which are absent from the interface are omitted from the tool tip.

When you select a channel transaction bar, associations appear for all channel transaction bars participating in the same AXI transaction as the selected transaction bar. You can click the tails of the association arrows to follow the progress of the AXI transaction from address phase through response phase. The chain of associations always begins with the address phase transaction even if the data phase precedes the address phase.

Error Conditions

If there is a handshaking error on the interface, you may see a sequence number on a channel transaction consisting of a string of all 9s. This sequence number indicates that the data and/or response phases could not be matched with an address and/or data phase. Common causes are mismatched read/write ID tags and the protocol analyzer being held in reset (ARESET or ARESETn signal active) while the AXI phases are in progress.

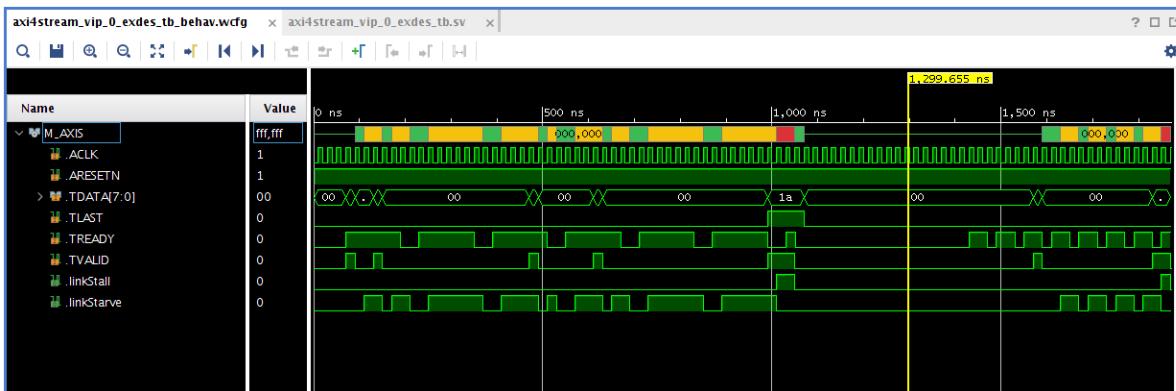


CAUTION! Because certain configurations of an AXI interconnect are optimized for performance rather than transaction debugging, AXI interfaces internal to an AXI interconnect may respect a different reset signal than the one connected to the interface causing transaction errors in the wave viewer. If you observe transaction errors on the interface, it is recommended that you monitor interfaces on the outside of the interconnect instead.

Analyzing AXI4-Stream (AXI-S) Interfaces

This section describes the transaction viewing features specific to AXI-Stream protocol instances. A protocol instance of an AXI-S interface appears in the wave window with a wave object hierarchy as shown in the following figure.

Figure 36: AXI-Stream (AXI-S) Interface



The wave object hierarchy of an AXI-S interface contains only one transaction row which is at the top of the hierarchy. Each transaction bar in the row corresponds to one complete AXI transaction. The text on the transaction consists of the stream identifier from AXI signal TID and the coarse routing information from AXI signal TDEST.

The transaction bar contains color coded stripes to indicate the status of an AXI transaction as described in the following table.

Table 12: AXI Transaction Status

Color	Status	Description
Green	Normal	Data is streaming normally.
Yellow	Starve	Slave is waiting for data from the master.
Red	Stall	Master is producing data faster than slave can consume it.

Under the top transaction row, the wave object hierarchy contains some of the key AXI signals plus stall and starve status signals to indicate stall and starve conditions. The status signals provide the same information as the color stripes do in the transaction row.



TIP: You may notice that the channel transaction bars appear up to one clock cycle after the corresponding AXI signal event. The AXI protocol analyzers consider AXI signal events that occur on or after a positive clock edge to take effect at the following positive clock edge.

Error Conditions

A handshaking error produces an error transaction with the text containing all Fs.

Debugging a Design with Vivado Simulator

The Vivado® Design Suite simulator provides you with the ability to:

- Examine source code
- Set *breakpoints* and run simulation until a breakpoint is reached
- Step over sections of code
- Force waveform objects to specific values

This chapter describes debugging methods and includes Tcl commands that are valuable in the debug process. There is also a flow description on debugging with third-party simulators.

Debugging at the Source Level

You can debug your HDL source code to track down unexpected behavior in the design.

Debugging is accomplished through controlled execution of the source code to determine where issues might be occurring. Available strategies for debugging are:

- Step through the code line by line: For any design at any point in development, you can use the `step` command to debug your HDL source code one line at a time to verify that the design is working as expected. After each line of code, run the `step` command again to continue the analysis. For more information, see [Stepping Through a Simulation](#).
- Set breakpoints on the specific lines of HDL code, and run the simulation until a breakpoint is reached: In larger designs, it can be cumbersome to stop after each line of HDL source code is run. Breakpoints can be set at any predetermined points in your HDL source code, and the simulation is run (either from the beginning of the test bench or from where you currently are in the design) and stops are made at each breakpoint. You can use the Step, Run All, or Run For command to advance the simulation after a stop. For more information, see the section, [Using Breakpoints](#), below.
- Set conditions. The tools evaluate each condition and execute Tcl commands when the condition is true. Use the Tcl command:

```
add_condition <condition> <instruction>
```

See [Adding Conditions](#) for more information.

Stepping Through a Simulation

You can use the `step` command, which executes your HDL source code one line of source code at a time, to verify that the design is working as expected.

The line of code is highlighted and an arrow points to the currently executing line of code.

You can also create breakpoints for additional stops while stepping through your simulation. For more information on debugging strategies in the simulator, see the section, [Using Breakpoints](#), below.

1. To step through a simulation:

- From the current running time, select **Run > Step**, or click the **Step** button .

The HDL associated with the top design unit opens as a new view in the Wave window.

- From the start (0 ns), restart the simulation. Use the **Restart** command to reset time to the beginning of the test bench. See [Chapter 4: Simulating with Vivado Simulator](#).

2. In the waveform configuration window, right-click the waveform or HDL tab and select **Tile Horizontally** see the waveform and the HDL code simultaneously.
3. Repeat the **Step** action until debugging is complete.

As each line is executed, you can see the arrow moving down the code. If the simulator is executing lines in another file, the new file opens, and the arrow steps through the code. It is common in most simulations for multiple files to be opened when running the Step command. The Tcl Console also indicates how far along the HDL code the step command has progressed.

Using Breakpoints

A breakpoint is a user-determined stopping point in the source code that you can use for debugging the design.



TIP: Breakpoints are particularly helpful when debugging larger designs for which debugging with the Step command (stopping the simulation for every line of code) might be too cumbersome and time consuming.

You can set breakpoints in executable lines in your HDL file so you can run your code continuously until the simulator encounters the breakpoint.

Note: You can set breakpoints on lines with executable code only. If you place a breakpoint on a line of code that is not executable, the breakpoint is not added.

1. Run a simulation.

2. Go to your source file and click the hollow circle  to the left of the source line of interest.

A red dot  confirms the breakpoint is set correctly.

After the procedure completes, a simulation breakpoint button opens next to the line of code.

Type the Tcl Command: `add_bp <file_name> <line_number>`

This command adds a breakpoint at `<line_number>` of `<file_name>`. See the Vivado Design Suite help or the *Vivado Design Suite Tcl Command Reference Guide (UG835)* for command usage.

Open the HDL source file.

3. Set breakpoints on executable lines in the HDL source file.

4. Repeat steps 1 and 2 until all breakpoints are set.

5. Run the simulation, using a Run option:

- To run from the beginning, use the **Run > Restart** command.
- Use the **Run > Run All** or **Run > Run For** command.

The simulation runs until a breakpoint is reached, then stops.

The HDL source file displays an arrow, indicating the breakpoint stopping point.

6. Repeat Step 4 to advance the simulation, breakpoint by breakpoint, until you are satisfied with the results.

A controlled simulation runs, stopping at each breakpoint set in your HDL source files.

During design debugging, you can also run the **Run > Step** command to advance the simulation line by line to debug the design at a more detailed level.

You can delete a single breakpoint or all breakpoints from your HDL source code.

To delete a single breakpoint, click the **Breakpoint** button .

To remove all breakpoints, either select **Run > Delete All Breakpoints** or click the **Delete All Breakpoints** button .

To delete all breakpoints:

- Type the Tcl command `remove_bps -all`.

To get breakpoint information on the specified list of breakpoint objects:

- Type the Tcl command `report_bps`

Adding Conditions

To add breakpoints based on a condition and output a diagnostic message, use the following commands:

```
add_condition <condition> <message>
```

Using the Vivado IDE BFT example design, to stop when the `wbClk` signal and the `reset` are both active-High, issue the following command at start of simulation to print a diagnostic message and pause simulation when `reset` goes to 1 and `wbClk` goes to 1:

```
add_condition {reset == 1 && wbClk == 1} {puts "Reset went to high"; stop}
```

In the BFT example, the added condition causes the simulation to pause at 5 ns when the condition is met and "Reset went to high" is printed to the console. The simulator waits for the next step or run command to resume simulation.

-notrace Option

Normally, when you execute the `add_condition` command, the specified Tcl commands also echo to the console, log file, and journal file. The `-notrace` switch causes those commands to execute silently, suppressing the commands (but not their output) from appearing in those three places.

For Example, If you execute the following example command:

```
puts 'Hello'
```

The normal behavior of the above command would be to emit the following output in the console, log file, and journal file:

```
# puts 'Hello'  
Hello
```

When you execute `-notrace` switch, it would produce only the following output:

```
Hello
```

Pausing a Simulation

While running a simulation for any length of time, you can pause a simulation using the **Break** command, which leaves the simulation session open.

To pause a running simulation, select **Simulation > Break** or click the **Break** button .

The simulator stops at the next executable HDL line. The line at which the simulation stopped is displayed in the text editor.

Note: This behavior applies to designs that are compiled with the `-debug <kind>` switch.

Resume the simulation any time using the Run All, Run, or Step commands. See [Stepping Through a Simulation](#) for more information.

Tracing the Execution of a Simulation

You can display a note on the Tcl console for every source line that the simulation encounters while running. This continuous display of encountered lines is called line tracing.

To turn on line tracing, use one of the following Tcl commands:

```
ltrace on  
set_property line_tracing true [current_sim]
```

To turn off line tracing use one of the following Tcl commands:

```
ltrace off  
set_property line_tracing false [current_sim]
```

You can display a note on the Tcl console for every process that the simulation encounters while running. This continuous display of encountered processes is called process tracing.

To turn on process tracing, use one of the following Tcl commands:

```
ptrace on  
set_property process_tracing true [current_sim]
```

To turn off process tracing, use one of the following Tcl commands:

```
ptrace off  
set_property process_tracing false [current_sim]
```

Forcing Objects to Specific Values

Using Force Commands

The Vivado simulator provides an interactive mechanism to force a signal, wire, or register to a specified value at a specified time or period of time. You can also force values on objects to change over a period of time.



TIP: A 'force' is both an *action* (that is, the overriding of HDL-defined behavior on a signal) and also a *Tcl first-class object*, something you can hold in a *Tcl variable*.

You can use force commands on an HDL signal to override the behavior for that signal as defined in your HDL design. You might, for example, choose to override the behavior of a signal to:

- Supply a stimulus to a test bench signal that the HDL test bench itself is not driving
- Correct a bad value temporarily during debugging (allowing you to continue analyzing a problem)

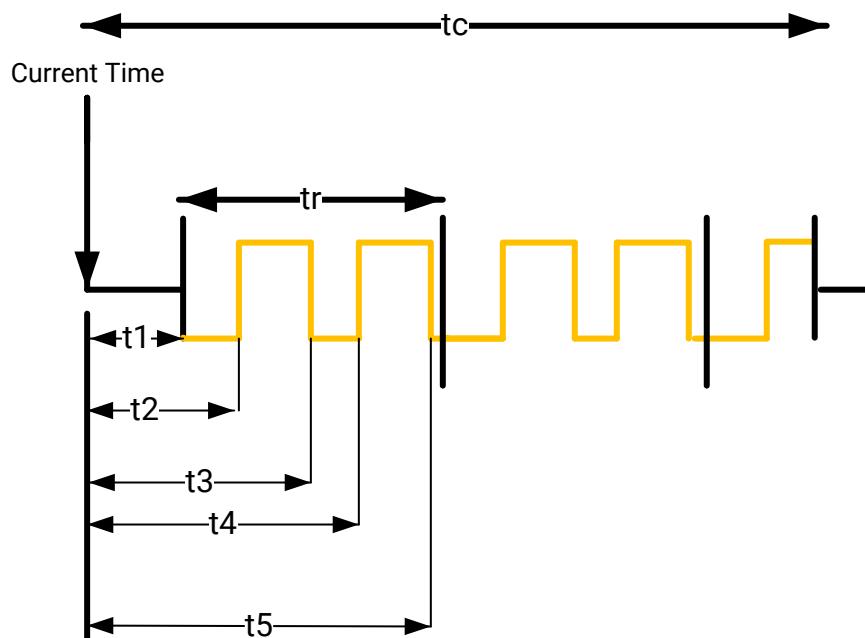
The available force commands are:

- Force Constant
- Force Clock
- Remove Force

The following figure illustrates how the `add_force` functionality is applied given the following command:

```
add_force mySig {0 t1} {1 t2} {0 t3} {1 t4} {0 t5} -repeat_every tr -
cancel_after tc
```

Figure 37: Illustration of -add_force Functionality



You can get more detail on the command by typing the following in the Tcl Console:

```
add_force -help
```

Force Constant

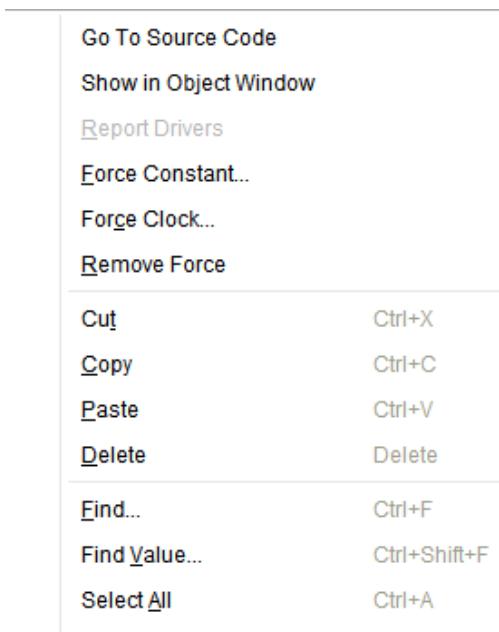
The Force Constant option lets you fix a signal to a constant value, overriding the assignments made within the HDL code or another previously applied constant or clock force.

Force Constant and Force Clock are options in the Objects or wave window right-click menu (as shown in the following figure), or in the text editor (source code).



TIP: Double-click an item in the Objects, Sources, or Scope window to open it in the text editor. For additional information about the text editor, see the Vivado Design Suite User Guide: Using the Vivado IDE (UG893).

Figure 38: Force Options



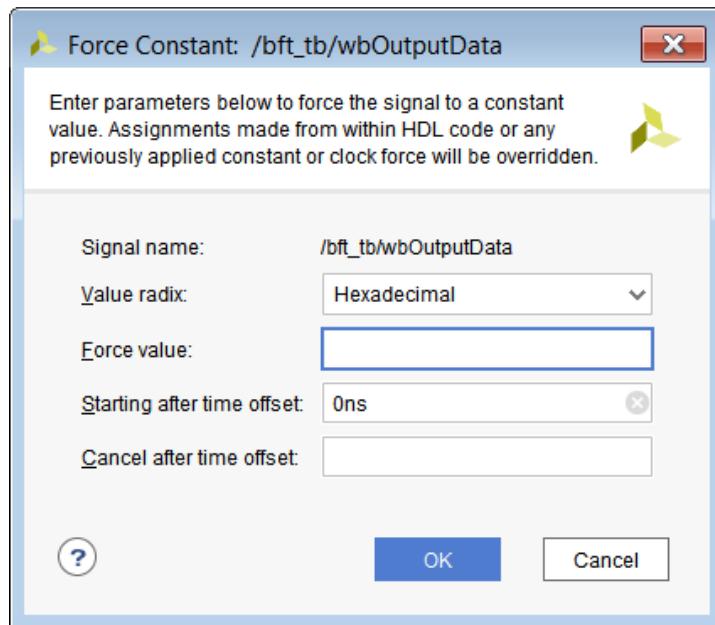
The Force options are disabled for the objects for which the Vivado simulator does not support forcing. The type of object or limitations in the Vivado simulator's modeling for those objects may be the cause for not supporting such objects.



TIP: To force a module or entity port whose Force options are disabled, try forcing its connected actual signal one scope level up. Use the `add_force` Tcl command (for example, `add_force myObj 0`) to view the reason why the options are disabled.

When you select the Force Constant option, the Force Constant dialog box opens so you can enter the relevant values, as shown in the following figure.

Figure 39: Force Constant Dialog Box



The following are Force Constant option descriptions:

- **Signal name:** Displays the default signal name, that is, the full path name of the selected object.
- **Value radix:** Displays the current radix setting of the selected signal. You can choose one of the supported radix types: Binary, Hexadecimal, Unsigned Decimal, Signed Decimal, Signed Magnitude, Octal, and ASCII. The GUI then disallows entry of the values based on the Radix setting. For example: if you choose Binary, no numerical values other than 0 and 1 are allowed.
- **Force value:** Specifies a force constant value using the defined radix value. (For more information about radixes, see [Changing the Default Radix](#) and [Using Analog Waveforms](#).)
- **Starting after time offset:** Starts after the specified time. The default starting time is 0. Time can be a string, such as 10 or 10 ns. When you enter a number without a unit, the Vivado simulator uses the default (ns).
- **Cancel after time offset:** Cancels after the specified time. Time can be a string such as 10 or 10 ns. If you enter a number without a unit, the default simulation time unit is used.

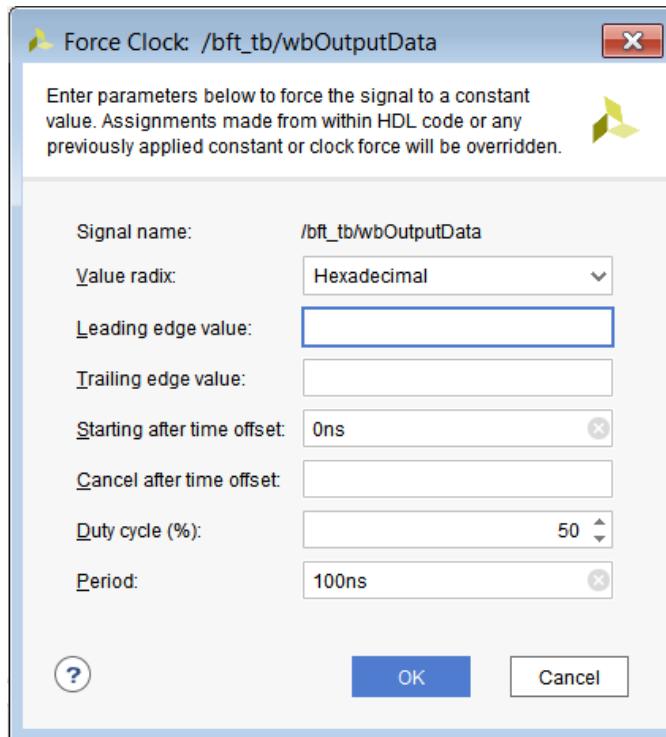
Tcl command:

```
add_force /testbench/TENSOUT 1 200 -cancel_after 500
```

Force Clock

The Force Clock command lets you assign a signal a value that toggles at a specified rate between two states, in the manner of a clock signal, for a specified length of time. When you select the **Force Clock** option in the Objects window menu, the Force Clock dialog box opens, as shown in the following figure.

Figure 40: Force Clock Dialog Box



The options in the Force Clock dialog box are shown below.

- **Signal name:** Displays the default signal name; the full path name of the item selected in the Objects window or waveform.



TIP: The **Force Clock** command can be applied to any signal (not just clock signals) to define an oscillating value.

- **Value radix:** Displays the current radix setting of the selected signal. Select one of the displayed radix types from the drop-down menu: **Binary**, **Hexadecimal**, **Unsigned Decimal**, **Signed Decimal**, **Signed Magnitude**, **Octal**, or **ASCII**.
- **Leading edge value:** Specifies the first edge of the clock pattern. The leading edge value uses the radix defined in Value radix field.
- **Trailing edge value:** Specifies the second edge of the clock pattern. The trailing edge value uses the radix defined in the Value radix field.

- **Starting after time offset:** Starts the force command after the specified time from the current simulation. The default starting time is 0. Time can be a string, such as 10 or 10 ns. If you enter a number without a unit, the Vivado simulator uses the default user unit.
- **Cancel after time offset:** Cancels the force command after the specified time from the current simulation time. Time can be a string, such as 10 or 10 ns. When you enter a number without a unit, the Vivado simulator uses the default simulation time unit.
- **Duty cycle (%):** Specifies the percentage of time that the clock pulse is in an active state. The acceptable value is a range from 0 to 100 (default is 50%).
- **Period:** Specifies the length of the clock pulse, defined as a time value. Time can be a string, such as 10 or 10 ns.

Note: For more information about radices, see [Changing the Default Radix](#) and [Using Analog Waveforms](#).

Example Tcl command:

```
add_force /testbench/TENSOUT -radix bin {0} {1} -repeat_every 10ns -  
cancel_after 3us
```

Remove Force

Remove Force

To remove any specified force from an object use the following Tcl command:

```
remove_forces <force object>  
remove_forces <HDL object>
```

Using Force in Batch Mode

The code examples below show how to force a signal to a specified value using the `add_force` command. A simple verilog circuit is provided. The first example shows the interactive use of the `add_force` command and the second example shows the scripted use.

Example 1: Adding Force

The following code snippet is a Verilog circuit:

```
module bot(input in1, in2,output out1);  
reg sel;  
assign out1 = sel? in1: in2;  
endmodule  
module top;  
reg in1, in2;  
wire out1;  
bot I1(in1, in2, out1);  
initial  
begin  
    #10 in1 = 1'b1; in2 = 1'b0;
```

```

#10 in1 = 1'b0; in2 = 1'b1;
end
initial
  $monitor("out1 = %b\n", out1);
endmodule

```

You can invoke the following commands to observe the effect of `add_force`:

```

xelab -vlog tmp.v -debug all
xsim work.top

```

At the command prompt, type:

```

add_force /top/I1/sel 1
run 10
add_force /top/I1/sel 0
run all

```

You can use the `add_force` Tcl command to force a signal, wire, or register to a specified value:

```

add_force [-radix <arg>] [-repeat_every <arg>] [-cancel_after <arg>] [-quiet]
[-verbose] <hdl_object> <values>...

```

For more info on this and other Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#)).

Example 2: Scripted Use of `add_force` with `remove_forces`

The following is an example Verilog file, `top.v`, which instantiates a counter. You can use this file in the following command example.

```

module counter(input clk,reset,updown,output [4:0] out1);
reg [4:0] r1;
always@(posedge clk)
begin
  if(reset)
    r1 <= 0;
  else
    if(updown)
      r1 <= r1 + 1;
    else
      r1 <= r1 - 1;
end
assign out1 = r1;
endmodule
module top;
reg clk;
reg reset;
reg updown;
wire [4:0] out1;
counter I1(clk, reset, updown, out1);
initial
begin
  reset = 1;
  #20 reset = 0;
end

```

```

initial
begin
    updown = 1; clk = 0;
end
initial
    #500 $finish;
initial
    $monitor("out1 = %b\n", out1);
endmodule

```

Command Example

1. Create a file called `add_force.tcl` with following command:

```

create_project add_force -force
add_files top.v
set_property top top [get_filesets sim_1]
set_property -name xelab.more_options -value {-debug all} -objects
[get_filesets
sim_1]
set_property runtime {0} [get_filesets sim_1]
launch_simulation -simset sim_1 -mode behavioral
add_wave /top/*

```

2. Invoke the Vivado Design Suite in Tcl mode, and source the `add_force.tcl` file.
3. In the Tcl Console, type:

```

set force1 [add_force clk {0 1} {1 2} -repeat_every 3 -cancel_after 500]
set force2 [add_force updown {0 10} {1 20} -repeat_every 30]
run 100

```

Observe that the value of `out1` increments as well as decrements in the Wave window. You can observe the waveforms in the Vivado IDE using the `start_gui` command.

Observe the value of `updown` signal in the Wave window.

4. In the Tcl Console, type:

```

remove_forces $force2
run 100

```

Observe that only the value of `out1` increments.

5. In the Tcl Console, type:

```

remove_forces $force1
run 100

```

Observe that the value of `out1` is not changing because the `clk` signal is not toggling.

Power Analysis Using Vivado Simulator

The Switching Activity Interchange Format (SAIF) is an ASCII report that assists in extracting and storing switching activity information generated by simulator tools. This switching activity can be back-annotated into the Xilinx® power analysis and optimization tools for the power measurements and estimations.

Switching Activity Interchange Format (SAIF) dumping is optimized for Xilinx power tools and for use by the `report_power` Tcl command. The Vivado simulator writes the following HDL types to the SAIF file. Refer to this [link](#) in the *Vivado Design Suite User Guide: Power Analysis and Optimization (UG907)* for additional information.

- Verilog:
 - Input, Output, and Inout ports
 - Internal wire declarations
- VHDL:
 - Input, Output, and Inout ports of type `std_logic`, `std_ulogic`, and `bit` (scalar, vector, and arrays).

Note: A VHDL netlist is not generated in the Vivado Design Suite for timing simulations; consequently, the VHDL sources are for RTL-level code only, and not for netlist simulation.

For RTL-level simulations, only block-level ports are generated and not the internal signals.

For information about power analysis using third-party simulation tools, see [Dumping SAIF for Power Analysis](#), and [Dumping SAIF in VCS](#) in [Chapter 3: Simulating with Third-Party Simulators](#).

Generating SAIF Dumping

Before you use the `log_saif` command, you must call `open_saif`. The `log_saif` command does not return any object or value.

1. Compile your RTL code with the `-debug typical` option to enable SAIF dumping:

```
xvlog -sv <fileName>.sv
xelab xsim mysim -debug typical top -s mysim
```

2. Use the following Tcl command to start SAIF dumping:

```
open_saif <saif_file_name>
```

3. Add the scopes and signals to be generated by typing one of the following Tcl commands:

```
log_saif [get_objects]
```

To recursively log all instances, use the Tcl command:

```
log_saif [get_objects -r *]
```

4. Run the simulation (use any of the run commands).
5. Import simulation data into an SAIF format using the following Tcl command:

```
close_saif
```

Example SAIF Tcl Commands

To log SAIF for:

- All signals in the scope: `/tb: log_saif /tb/*`
- All the ports of the scope: `/tb/UUT`
- Those objects having names that start with `a` and end in `b` and have digits in between:

```
log_saif [get_objects -regexp {^a[0-9]+b$}]
```

- The objects in the `current_scope` and `children_scope`:

```
log_saif [get_objects -r *]
```

- The objects in the `current_scope`:

```
log_saif * or log_saif [get_objects]
```

- Only the ports of the scope `/tb/UUT`, use the command:

```
id="ah453025">log_saif [get_objects -filter {type == in_port || type == out_port || type == inout_port || type == port} /tb/UUT/* ]
```

- Only the internal signals of the scope `/tb/UUT`, use the command:

```
log_saif [get_objects -filter { type == signal } /tb/UUT/* ]
```



TIP: This filtering is applicable to all Tcl commands that require HDL objects.

Dumping SAIF using a Tcl Simulation Batch File

```
sim.tcl:
open_saif xsim_dump.saif
log_saif /tb/dut/*
run all
close_saif
quit
```

Using the `report_drivers` Tcl Command

You can use the `report_drivers` Tcl command to determine what signal is *driving* a value on an HDL object. The syntax is as follows:

```
report_drivers <hdl_object>
```

The command prints drivers (HDL statements doing the assignment) to the Tcl Console along with current driving values on the right side of the assignment to a wire or signal-type HDL object.

You can also call the `report_drivers` command from the Object or Wave window context menu or text editor. To open the context menu (shown in the figure below), right-click any signal and click **Report Drivers**. The result appears in the Tcl console.

Figure 41: Context Menu with Report Drivers Command Option



Using the Value Change Dump Feature

You can use a Value Change Dump (VCD) file to capture simulation output. The Tcl commands are based on Verilog system tasks related to dumping values.

For the VCD feature, the Tcl commands listed in the table below model the Verilog system tasks.

Table 13: Tcl Commands for VCD

Tcl Command	Description
open_vcd	Opens a VCD file for capturing simulation output. This Tcl command models the behavior of <code>\$dumpfile</code> Verilog system task.
checkpoint_vcd	Models the behavior of the <code>\$dumpall</code> Verilog system task.
start_vcd	Models the behavior of the <code>\$dumpon</code> Verilog system task.
log_vcd	Logs VCD for the specified HDL objects. This command models behavior of the <code>\$dumpvars</code> Verilog system task.
flush_vcd	Models behavior of the <code>\$dumpflush</code> Verilog system task.
limit_vcd	Models behavior of the <code>\$dumplimit</code> Verilog system task.
stop_vcd	Models behavior of the <code>\$dumpoff</code> Verilog system task.
close_vcd	Closes the VCD generation.

See the *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#)), or type the following in the Tcl Console:

```
<command> -help
```

Example:

```
open_vcd xsim_dump.vcd
log_vcd /tb/dut/*
run all
close_vcd
quit
```

See [Verilog Language Support Exceptions](#) for more information.

You can use the VCD data to validate the output of the simulator to debug simulation failures.

Related Information

[Vivado Simulator Mixed Language Support and Language Exceptions](#)

Using the log_wave Tcl Command

The `log_wave` command logs simulation output for viewing specified HDL objects with the Vivado simulator waveform viewer. Unlike `add_wave`, the `log_wave` command does not add the HDL object to the waveform viewer (that is, the Waveform Configuration). It simply enables the logging of output to the Vivado simulator Waveform Database (WDB).



TIP: To display object values prior to the time of insertion, the simulation must be restarted. To avoid having to restart the simulation because of missing value changes: issue the `log_wave -r` / Tcl command at the start of a simulation run to capture value changes for all display-able HDL objects in your design.

Syntax:

```
log_wave [-recursive] [-r] [-quiet] [-verbose] <hdl_objects>...
```

Example log_wave TCL Command Usage

To log the waveform output for:

- All signals in the design (excluding those of alternate top modules):

```
log_wave -r /
```

- All signals in a scope: /tb:

```
log_wave /tb/*
```

- Those objects having names that start with a and end in b and have digits in between:

```
log_wave [get_objects -regexp {^a[0-9]+b$}]
```

- All objects in the current scope and all child scopes, recursively:

```
log_wave -r *
```

- Temporarily overriding any message limits and return all messages from the following command:

```
log_wave -v
```

- The objects in the current scope:

```
log_wave *
```

- Only the ports of the scope /tb/UUT, use the command:

```
log_wave [get_objects -filter {type == in_port || type == out_port || type == inout_port || type == port} /tb/UUT/*]
```

- Only the internal signals of the scope /tb/UUT, use the command:

```
log_wave [get_objects -filter {type == signal} /tb/UUT/*]
```

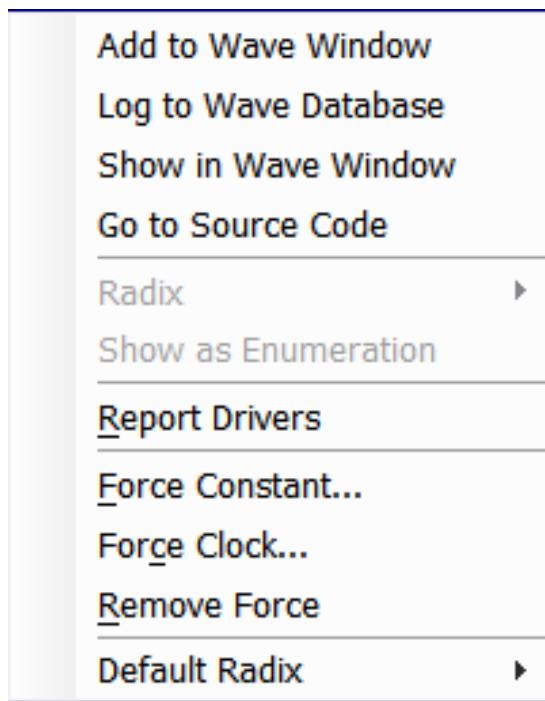
The wave configuration settings; which include the signal order, name style, radix, and color; are saved to the wave configuration (WCFG) file upon demand. See [Chapter 5: Analyzing Simulation Waveforms with Vivado Simulator](#).

Cross Probing Signals in the Object, Wave, and Text Editor Windows

In Vivado simulator, you can do cross probing on signals present in the Objects, Wave, and text editor windows.

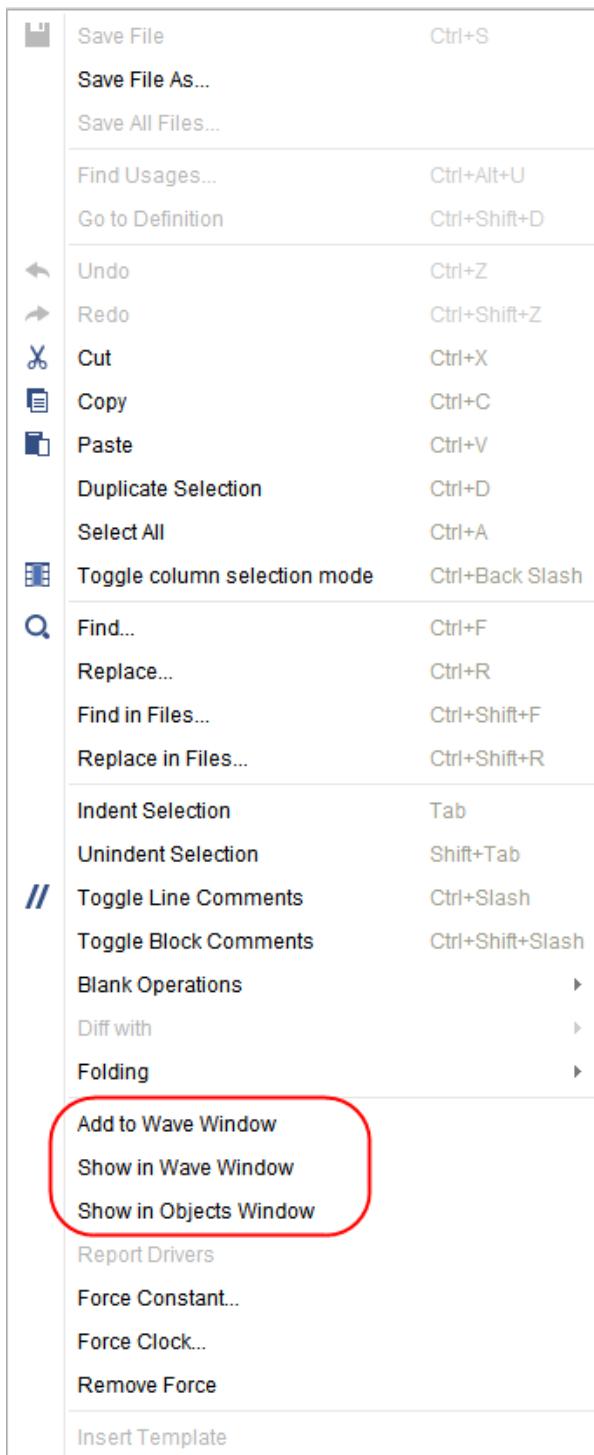
From the Objects window, you can check to see if a signal is present in the Wave window and vice versa. Right-click the signal to open the context menu shown in the following figure. Click **Show in Wave Window** or **Add to Wave Window** (if signal is not yet present in the Wave window).

Figure 42: Objects Window Context Menu Options



You can also cross probe a signal from the text editor. Right-click a signal to open the context menu shown in the figure below. Select **Add to Wave Window**, **Show in Waveform** or **Show in Objects**. The signal then appears highlighted in the Wave or Objects window.

Figure 43: Text Editor Right-Click (Context) Menu



Tool Specific init.tcl

During execution of simulation, Vivado simulator sources the init file present at the following location:

```
$HOME/.xilinx/xsim/xsim_init.tcl
```

It is useful, if you want to set a property across multiple runs. In such a scenario, you can write these inside a tcl file and Vivado simulator will source this tcl file before time 0' during execution.

Subprogram Call-Stack Support

You can now step-through subprogram calls and access automatic (as well as static) variables inside subprogram using `get_value`/`set_value` options.

Currently, you can only access these variables if the subprogram is at the top of the call-stack.

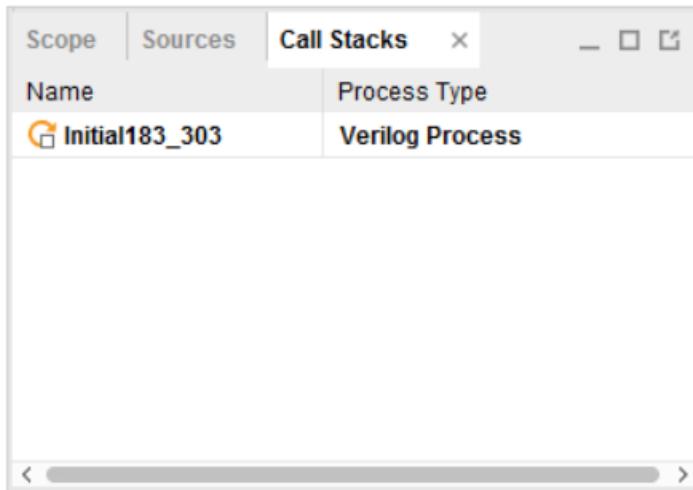
Use the following options to support access to variables at any level of the call-stack.

Call Stacks Window

Call Stacks window shows HDL scopes for all the VHDL/Verilog processes in a design which are waiting inside a subprogram at the current simulation time. This is similar to `get_stacks` Tcl command.

By default, the current process in which simulation is stopped (inside a subprogram) will be selected in the Call Stacks window. However, you can select any other processes waiting in a subprogram. The effect of selecting a process on the Call Stack window is same as selecting a process scope from the Scope window or using `current_scope` Tcl command. When you select a process on the Call Stacks window, the updated process appears in the Scope window, Objects window, Stack Frames window and Locals tab. The process name with absolute path and its type of the selected process is shown in the Call Stacks window.

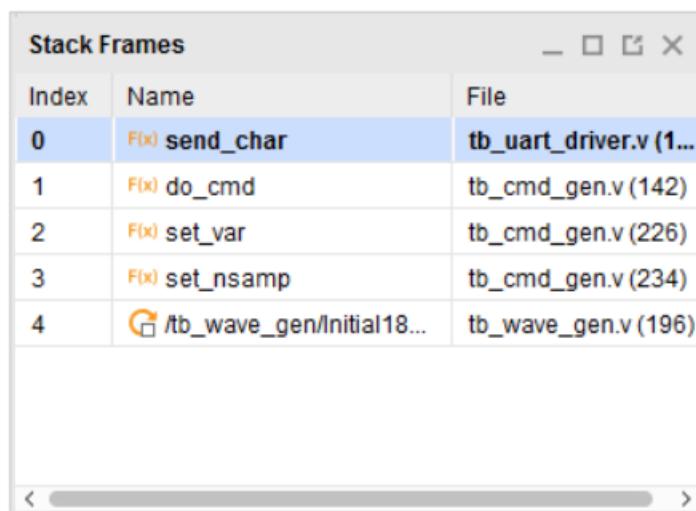
Figure 44: Call Stacks Window



Stack Frames Window

Stack Frames window shows the current HDL process that is waiting inside a subprogram and the subprograms in its call-stack. This is similar to `report_frames` and `current_frame` Tcl commands. In the Stack Frames windows, the most recent subprogram in the current hierarchy is shown at the top, followed by caller subprograms. The caller HDL process is shown the bottom. You can select other frames to be current and the effect is same as the `current_frame -set <selected_frame_index>` Tcl command. The Locals tab in the Objects window follows the subprogram frame selection and shows the static and automatic variables local to the selected subprogram frame. The frame number, subprogram/process name, source file and current line for the selected HDL process is shown in the Stack Frames window.

Figure 45: Stack Frames Window

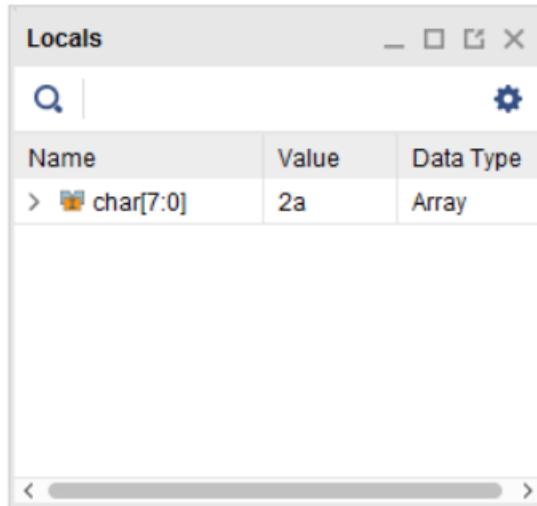


Index	Name	File
0	F(x) send_char	tb_uart_driver.v (1...
1	F(x) do_cmd	tb_cmd_gen.v (142)
2	F(x) set_var	tb_cmd_gen.v (226)
3	F(x) set_nsamp	tb_cmd_gen.v (234)
4	Initial183_303	tb_wave_gen.v (196)

Locals Tab in Objects Window

The Locals tab in Objects window shows the name, value and type of static and automatic variables local to the currently executing (or selected) subprogram. This is similar to `get_objects -local` Tcl command. This window follows the frame selected in the Stack Frames window. For every variable/argument, its name, value and type would be shown in the Locals tab.

Figure 46: Locals Tab in Objects Window



Debugging with Dynamic Type

In the SystemVerilog, there are dynamic types such as Class, Dynamic Array, Queue, and Associative Array etc. These dynamic types are supported in the Vivado simulator. Vivado allows you to probe the dynamic type variables. For example:

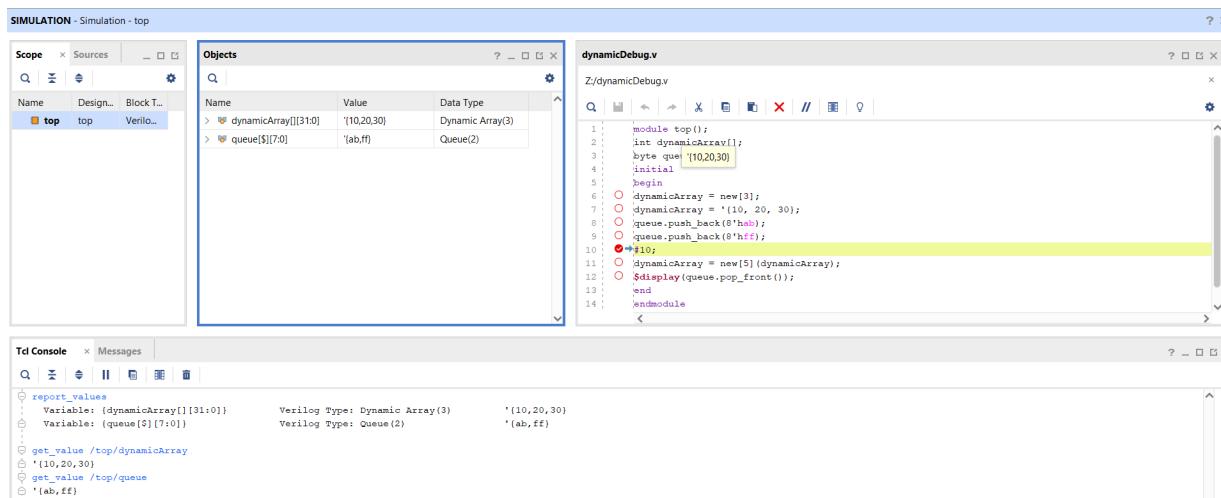
```
module top();
int dynamicArray[];
byte queue[$];
initial
begin
    dynamicArray = new[3];
    dynamicArray = '{10, 20, 30};
    queue.push_back(8'hab);
    queue.push_back(8'hff);
    #10;
    dynamicArray = new[5](dynamicArray);
    $display(queue.pop_front());
end
endmodule
```

You can probe the dynamic type variables using the following windows as shown in the following figure:

- Objects window

- Tcl Console window by using `get_value` and `report_value` commands.
- Tooltip in the Sources window

Figure 47: Probing Dynamic Type



Note: Dynamic types are not supported for tracing waveform (`add_wave`) or for creating waveform data base (`log_wave`).

Simulating in Batch or Scripted Mode in Vivado Simulator

This chapter describes the command line compilation and simulation process.

Vivado supports an integrated simulation flow where the tool can launch Vivado simulator, or a third party simulator from the IDE. However, many users also want to run simulation in batch or scripted mode in their verification environment, which may include system-level simulation, or advanced verification such as UVM. The Vivado Design Suite supports batch or scripted simulation in the Vivado simulator.

This chapter describes a process to gather the needed design files, to generate simulation scripts for your target simulator, and to run simulation in batch mode. The simulation scripts can be generated for a top-level HDL design, or for hierarchical modules, managed IP projects, or block designs from Vivado IP integrator. Batch simulation is supported in both project and non-project script-based flow.

Exporting Simulation Files and Scripts

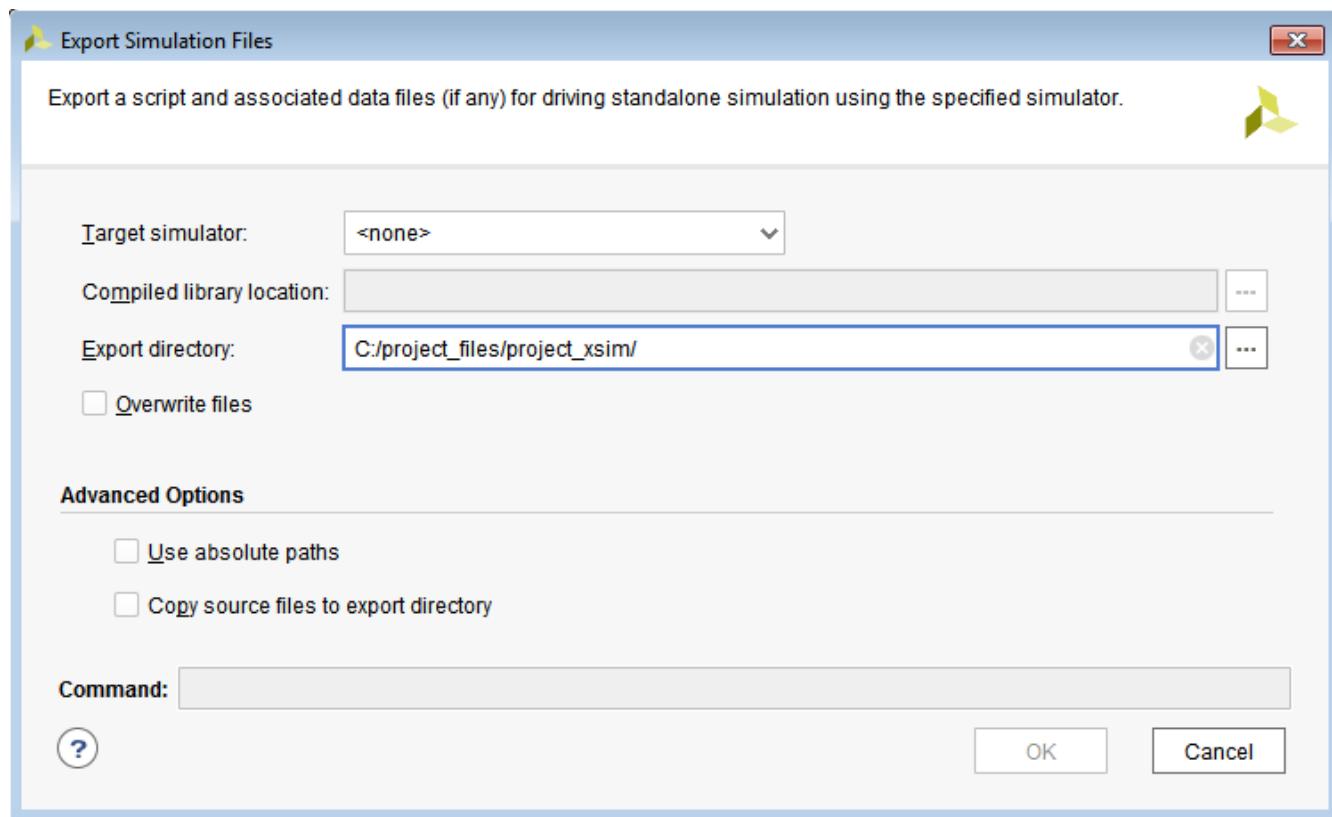
Running a simulation from the command line for either a behavioral or timing simulation requires you to perform the following steps:

1. Identifying and parsing design files.
2. Elaborating and generating an executable simulation snapshot of the design.
3. Running simulation using the executable snapshot.

The Vivado Design Suite provides an Export Simulation command to let you quickly gather the design files required for simulation, and generate the simulation scripts for the top-level RTL design, or a sub-design. The `export_simulation` command will generate scripts for all of the supported third-party simulators, or for the target simulator of your choice.

From within the Vivado IDE, use the **File → Export → Export Simulation** command to open the Export Simulation dialog box as shown in the following figure.

Figure 48: Export Simulation dialog box



The Export Simulation command writes a simulation script file for all supported simulators, or for the specified Target simulator. The generated scripts will contain simulator commands for compiling, elaborating, and simulating the design.

The features of the Export Simulation dialog box include the following:

- **Target simulator:** Specifies all simulators, or a specific simulator to generate command line scripts for. Target simulators include Vivado simulator as well as various supported third-party simulators. Refer to [Chapter 3: Simulating with Third-Party Simulators](#) for more information.
- **Compiled library location:** In order to perform simulation with the script generated by Export Simulation, your simulation libraries must first be compiled with the `compile_simlib` Tcl command. The generated scripts will automatically include the setup files needed for the target simulator from the compiled library directory. Refer to [Compiling Simulation Libraries](#) for more information.



TIP: It is recommended to provide the path to the Compile library location whenever running Export Simulation. This insures that the scripts will always point to the correct simulation libraries.

- **Export directory:** Specifies the output directory for the scripts written by Export Simulation. By default, the simulation scripts are written to the local directory of the current project.

- **Overwrite files:** Overwrites the files of the same name that already exist in the `export` directory.
- **Use absolute paths:** By default, source files and directory paths in the generated scripts will be relative to a reference directory that is defined in the scripts. Use this switch to make file paths in the script absolute rather than relative.
- **Copy source files to export directory:** Copy design files to the output directory. This copies the simulation source files as well as the generated scripts to make the entire simulation folder more portable.
- **Command:** This field provides the Tcl command syntax for the `export_simulation` command that will be run as a result of the various options and settings that you have specified in the Export Simulation dialog box.
- **Help:** For detailed information on various options in Export Simulation files dialog box, click the help button.

The Export Simulation command supports both project and non-project designs. It does not read properties from the current project to query for Verilog defines and include directories. Instead, the Export Simulation command gets directives from the dialog box or from `export_simulation` command options. You must specify the appropriate options to get the results you want. In addition, you must have output products generated for all IP and BD that are used in the top-level design.



IMPORTANT! The `export_simulation` command will not generate output products for IP and BD if they do not exist. Instead it will return an error and exit.

When you click **OK** on the Export Simulation dialog box, the command gets the simulation compile order of all design files required for simulating the specified design object: the top-level design, a hierarchical module, IP core, a block design from Vivado IP integrator, or a Managed IP project with multiple IP. The simulation compile order of the required design files is exported to a shell script with compiler commands and options for the target simulator.

The simulation scripts are written to separate folders in the Export directory as specified in the Export Simulation dialog box. A separate folder is created for each specified simulator, and `compile`, `elaborate`, and `simulate` scripts are written for the simulator.

The scripts generated by the Export Simulation command uses a 3-step process, `analyze/compile`, `elaborate` and `simulate`, that is common to many simulators including the Vivado simulator. However, for ModelSim and Riviera the generated scripts use the two-step process of `compile` and `simulate` that the tool requires.



TIP: To use the two-step process in the Questa simulator, you can start with the scripts generated for ModelSim and modify them as needed.

The Export Simulation command will also copy data files (if any) from the fileset, or from an IP, to the specified export directory. If the design contains Verilog sources, then the generated script will also copy the `glbl.v` file from the Vivado software installation path to the output directory.

```
export_ip_user_files -no_script -force
export_simulation -directory "C:/Data/project_wave1" -simulator all
```

When you run the Export Simulation command from the dialog box, the Vivado IDE actually runs a sequence of commands that defines the base directory (or location) for the exported scripts, exports the IP user files, and then runs the `export_simulation` command.

The `export_ip_user_files` command is run automatically by the Vivado IDE to ensure that all required files needed to support simulation for both core container and non-core container IP, as well as block designs, are available. See this [link](#) in the *Vivado Design Suite User Guide: Designing with IP (UG896)* for more information. While `export_ip_user_files` is run automatically when working with the Export Simulation dialog box, you must be sure to run it manually before running the `export_simulation` command.



TIP: Notice the `-no_script` option is specified when `export_ip_user_files` is run automatically by the Vivado IDE. This is to prevent the generation of simulation scripts for the individual IP and BDs that are used in the top-level design because it can add significant run time to the command. However, you can generate these simulation scripts for individual IP and BD by running `export_ip_user_files` on specified objects (`-of_objects`), or without the `-no_script` option.

The `export_ip_user_files` command sets up the user file environment for IP and block design needed for simulation and synthesis. The command creates a folder called `ip_user_files` which contains instantiation templates, stub files for use with third-party synthesis tools, wrapper files, memory initialization files, and simulation scripts.

The `export_ip_user_files` command also consolidates static simulation files that are shared across all IP and block designs in the project and copies them to an `ipstatic` folder. Many of the IP files that are shared across multiple IP and BDs in a project do not change for specific IP customizations. These static files are copied into the `ipstatic` directory. The scripts created for simulation reference the shared files in this directory as needed. The dynamic simulation files that are specific to an IP customization are copied to the IP folder. See this [link](#), or "Understanding IP User Files" in *Vivado Design Suite User Guide: Designing with IP (UG896)* for more information.



IMPORTANT! The scripts and files generated by the `export_simulation` command point to the files in the `ip_user_files` directory. You must run the `export_ip_user_files` command before you run `export_simulation` or simulation errors may occur.

Exporting the Top Level Design

To create simulation scripts for the top-level RTL design use `export_simulation` and provide the simulation fileset object. In the following example `sim_1` is the simulation fileset, and `export simulation` will create simulation scripts for all the RTL entities, IP, and BD objects in the design.

```
export_ip_user_files -no_script
export_simulation -of_objects [get_filesets sim_1] -directory C:/test_sim \
-simulator questa
```

Exporting IP from the Xilinx Catalog and Block Designs

To generate scripts for an IP, or a Vivado IP integrator block design, you can simply run the command on the IP or block design object:

```
export_ip_user_files -of_objects [get_ips blk_mem_gen_0] -no_script -force
export_simulation -simulator xcelium -directory ./export_script \
-of_objects [get_ips blk_mem_gen_0]
```

Or, export the `ip_user_files` for all IP and BDs in the design:

```
export_ip_user_files -no_script -force
export_simulation -simulator xcelium -directory ./export_script
```

You can also generate simulation scripts for block design objects:

```
export_ip_user_files -of_objects [get_files base_microblaze_design.bd] \
-no_script -force
export_simulation -of_objects [get_files base_microblaze_design.bd] \
-directory ./sim_scripts
```

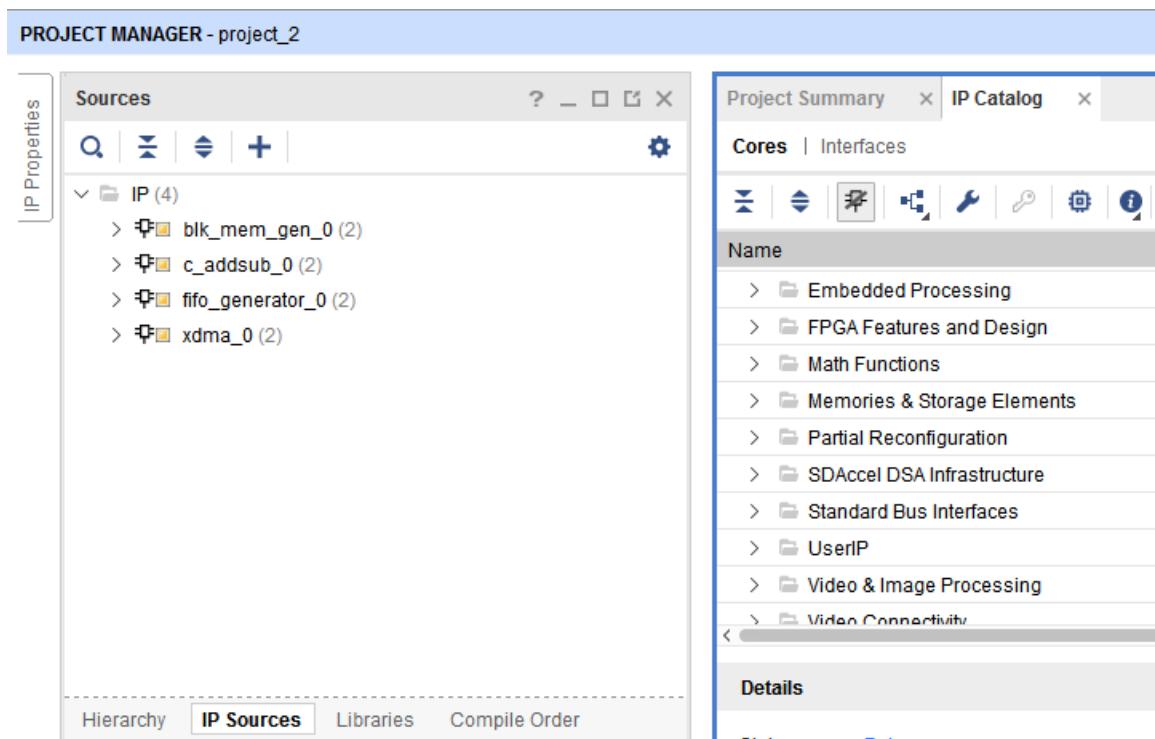


IMPORTANT! You must have output products generated for all IP and BD that are used in the top-level design. The `export_simulation` command will not generate output products for IP and BD if they do not exist. Instead it will return an error and exit.

Exporting a Manage IP Project

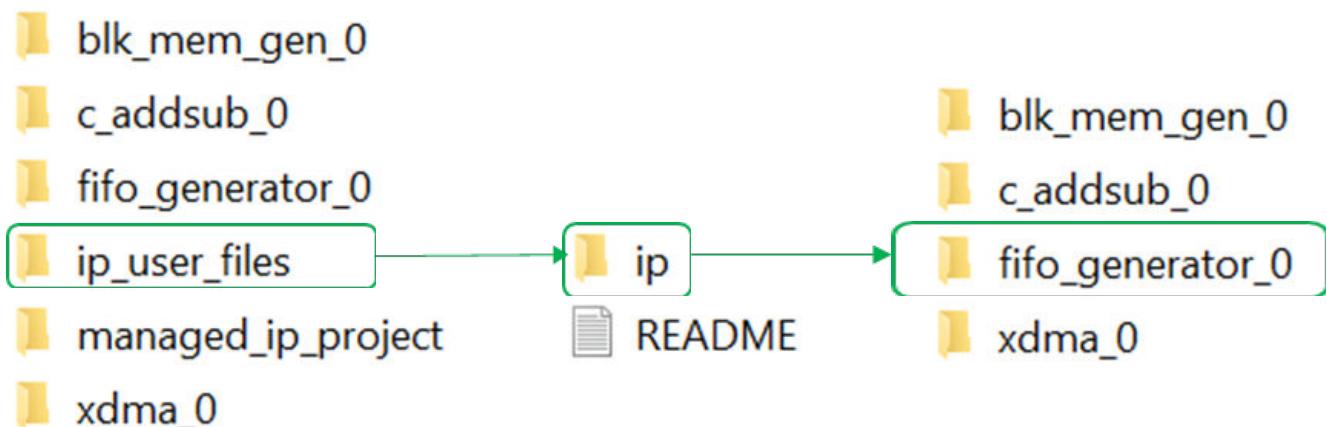
Manage IP project provides users an ability to create and manage a centralized repository of customized IPs. See this [link](#) in the *Vivado Design Suite User Guide: Designing with IP (UG896)* for more information on Manage IP projects. When generating the IP output products for Manage IP projects, the Vivado tool also generates simulation scripts for each IP using the `export_ip_user_files` command as previously discussed.

Figure 49: Managed IP Project



The Managed IP Project shown above features four different customized IP: `blk_mem_gen_0`, `c_addsub_0`, `fifo_generator_0`, `xdma_0`. For this project the Vivado Design Suite creates an `ip_user_files` folder as shown in the following figure.

Figure 50: Managed IP Directory Structure



The `ip_user_files` folder is generated by the `export_ip_user_files` command as previously described. When this command is run on a Manage IP project, it will recursively process all the IP in the project and generate the scripts and other files needed for synthesis and simulation of the IP. The `ip_user_files` folder contains the scripts used for batch simulation, as well as the dynamic and static IP files needed to support simulation.

The simulation scripts for your target simulator, or for all supported simulators, are located in the `./sim_scripts` folder as seen in [Exporting a Manage IP Project](#). You can go to the folder of your target simulator and incorporate the `compile`, `elaborate`, and `simulate` scripts into your simulation flow.

The Vivado tool consolidates all the shared simulation files, used by multiple IP and BD in the design, into a folder called `./ipstatic`. The dynamic files that vary depending on the specifics of an IP customization are located in the `./ip` folder.



TIP: *In addition to exporting all the IP in a Manage IP project, you can use the steps outlined in [Exporting IP from the Xilinx Catalog and Block Designs](#) to generate scripts for individual IP in the project.*

Running the Vivado Simulator in Batch Mode

To run in batch or scripted mode, the Vivado simulator relies on three processes which are supported by the files generated by the `export_simulation` command.

- [Parsing Design Files, xvhdl and xvlog](#)
- [Elaborating and Generating a Design Snapshot, xelab](#)
- [Simulating the Design Snapshot, xsim](#)

For timing simulation, there are additional steps and data required to complete the simulation, as described in the following:

- [Generating a Timing Netlist](#)
- [Running Post-Synthesis and Post-Implementation Simulations](#)

Parsing Design Files, xvhdl and xvlog

The `xvhdl` and `xvlog` commands parse VHDL and Verilog files, respectively. Descriptions for each option are available in [Table 15: xelab, xvhd, and xvlog Command Options](#).

xvhdl

The `xvhdl` command is the VHDL analyzer (parser).

xvhdl Syntax

```
xvhdl
[-cryptodumps]
[-f [-file] <filename>]
[-h [-help]]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=library_dir]]]
[-log <filename>]
[-nolog]
[-prj <filename>]
[-relax]
[-v [verbose] [0|1|2]]
[-version]
[-work <library_name> [=library_dir]]
[-incr]
[-2008]
[-93_mode]
[-nosignalhandlers]
```

This command parses the VHDL source file(s) and stores the parsed dump into a HDL library on disk.

xvhdl Examples

```
xvhdl file1.vhd file2.vhd
xvhdl -work worklib file1.vhd file2.vhd
xvhdl -prj files.prj
```

xvlog

The `xvlog` command is the Verilog parser. The `xvlog` command parses the Verilog source file(s) and stores the parsed dump into a HDL library on disk.

xvlog Syntax

```
xvlog
[-d [define] <name>[=<val>]]
[-cryptodumps]
[-f [-file] <filename>]
[-h [-help]]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=library_dir]]]
[-log <filename>]
[-nolog]
[-noname_unamed_generate]
[-relax]
[-prj <filename>]
[-sourcelibdir <sourcelib_dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-sv]
[-v [verbose] [0|1|2]]
```

```
[-version]
[-work <library_name> [=<library_dir>]
[-incr]
[-nosignalhandlers]
[-uvm_version arg]
```

xvlog Examples

```
xvlog file1.v file2.v
xvlog -work worklib file1.v file2.v
xvlog -prj files.prj
```

Note: `xelab`, `xvlog` and `xvhdl` are not Tcl commands. The `xvlog`, `xvhdl`, `xelab` are Vivado-independent compiler executables. Hence, there is no Tcl command for them.

The simulation launching is Vivado dependent and hence is done through `xsim` Tcl command.

For usage of simulation outside Vivado, an executable by the same name as `xsim` is provided. The `xsim` executable launches Vivado in project less mode and executes `xsim` Tcl command to launch simulation. Hence, to get help on `xvlog`, `xvhdl`, `xelab` form within Vivado IDE, please precede the command with `exec`.

Example: `exec xvlog -help`.

To get help on `xsim`, use `xsim -help`.

Elaborating and Generating a Design Snapshot, `xelab`

Simulation with the Vivado simulator happens in two phases:

- In the first phase, the simulator compiler `xelab`, compiles your HDL model into a snapshot, which is a representation of the model in a form that the simulator can execute.
- In the second phase, the simulator loads and executes (using the `xsim` command) the snapshot to simulate the model. In Non-Project Mode, you can reuse the snapshot by skipping the first phase and repeating the second.

When the simulator creates a snapshot, it assigns the snapshot a name based on the names of the top modules in the model. You can, however, override the default by specifying a snapshot name as an option to the compiler. Snapshot names must be unique in a directory or SIMSET; reusing a snapshot name, whether default or custom, results in overwriting a previously-built snapshot with that name.



IMPORTANT! You cannot run two simulations with the same snapshot name in the same directory or SIMSET.

xelab

The `xelab` command, for given top-level units, does the following:

- Loads children design units using language binding rules or the `-L <library>` command line specified HDL libraries
- Performs a static elaboration of the design (sets parameters, generics, puts generate statements into effect, and so forth)
- Generates executable code
- Links the generated executable code with the simulation kernel library to create an executable simulation snapshot

You then use the produced executable simulation snapshot name as an option to the `xsim` command along with other options to effect HDL simulation.



TIP: `xelab` can implicitly call the parsing commands, `xvlog` and `xvhdl`. You can incorporate the parsing step by using the `xelab -prj` option. See [Project File \(.prj\) Syntax](#) for more information about project files.

Note: `xelab`, `xvlog` and `xvhdl` are not Tcl commands. The `xvlog`, `xvhdl`, `xelab` are Vivado-independent compiler executables. Hence, there is no Tcl command for them.

xelab Command Syntax Options

Descriptions for each option are available in the following codeblock.

```
xelab
[-d [define] <name>[=<val>]
[-debug <kind>]
[-f [-file] <filename>]
[-generic_top <value>]
[-h [-help]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-log <filename>]
[-L [-lib] <library_name> [=<library_dir>]
[-maxdesigndepth arg]
[-mindelay]
[-typdelay]
[-maxarraysize arg]
[-maxdelay]
[-mt arg]
[-nolog]
[-noname_unnamed_generate]
[-notimingchecks]
[-nosdfinterconnectdelays]
[-nospecify]
[-O arg]
[-override_timeunit]
[-override_timeprecision]
[-prj <filename>]
[-pulse_e arg]
[-pulse_r arg]
[-pulse_int_e arg]
```

```
[-pulse_int_r arg]
[-pulse_e_style arg]
[-r [-run]]
[-R [-runall]]
[-rangecheck]
[-relax]
[-s [-snapshot] arg]
[-sdfnowarn]
[-sdfnoerror]
[-sdfroot <root_path>]
[-sdfmin arg]
[-sdftyp arg]
[-sdfmax arg]
[-sourcelibdir <sourcelib dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-stats]
[-timescale]
[-timeprecision_vhdl arg]
[-transport_int_delays]
[-v [verbose] [0|1|2]]
[-version]
[-sv_root arg]
[-sv_lib arg]
[-sv_liblist arg]
[-dpiheader arg]
[-driver_display_limit arg]
[-dpi_absolute]
[-incr]
[-93_mode]
[-nosignalhandlers]
[-dpi_stacksize arg]
[-transform_timing_checkers]
[-a[ --standalone]
[-ignore_assertions]
[-ignore_coverage]
[-cov_db_dir arg]
[-cov_db_name arg]
[-uvm_version arg]
[-report_assertion_pass]
[-dup_entity_as_module]
[-cc_celldefines]
[-cc_libs]
[-cc_type arg]
[-cc_db arg]
[-cc_dir arg]
[-cov_db_dir arg]
[-cov_db_name arg]
[-ignore_localparam_override]
[-sc_lib arg]
[-sc_root arg]
```

xelab Examples

```
xelab work.top1 work.top2 -s cpusim
xelab lib1.top1 lib2.top2 -s fftsim
xelab work.top1 work.top2 -prj files.prj -s pciesim
xelab lib1.top1 lib2.top2 -prj files.prj -s ethernetsim
```

Verilog Search Order

The `xelab` command uses the following search order to search and bind instantiated Verilog design units:

1. A library specified by the `'uselib` directive in the Verilog code. For example:

```
module
  full_adder(c_in, c_out, a, b, sum)
  input c_in,a,b;
  output c_out,sum;
  wire carry1,carry2,sum1;
  `uselib lib = adder_lib
  half_adder adder1(.a(a)..b(b)..c(carry1)..s(sum1));
  half_adder adder1(.a(sum1)..b(c_in)..c(carry2)..s(sum));
  c_out = carry1 | carry2;
endmodule
```

2. Libraries specified on the command line with `-lib` | `-L` switch.
3. A library of the parent design unit.
4. The `work` library.

Verilog Instantiation Unit

When a Verilog design instantiates a component, the `xelab` command treats the component name as a Verilog unit and searches for a Verilog module in the user-specified list of unified logical libraries in the user-specified order.

- If found, `xelab` binds the unit and the search stops.
- If the case-sensitive search is not successful, `xelab` performs a case-insensitive search for a VHDL design unit name constructed as an extended identifier in the user-specified list and order of unified logical libraries, selects the first one matching name, then stops the search.
- If `xelab` finds a unique binding for any one library, it selects that name and stops the search.

Note: For a mixed language design, the port names used in a named association to a VHDL entity instantiated by a Verilog module are always treated as case insensitive. Also note that you cannot use a `defparam` statement to modify a VHDL generic. See [Using Mixed Language Simulation](#), for more information.



IMPORTANT! Connecting a whole VHDL record object to a Verilog object is unsupported.

VHDL Instantiation Unit

When a VHDL design instantiates a component, the `xelab` command treats the component name as a VHDL unit and searches for it in the logical `work` library.

- If a VHDL unit is found, the `xelab` command binds it and the search stops.

- If `xelab` does not find a VHDL unit, it treats the case-preserved component name as a Verilog module name and continues a case-sensitive search in the user-specified list and order of unified logical libraries. The command selects the first matching the name, then stops the search.
- If case sensitive search is not successful, `xelab` performs a case-insensitive search for a Verilog module in the user-specified list and order of unified logical libraries. If a unique binding is found for any one library, the search stops.

`uselib Verilog Directive

The Verilog ``uselib` directive is supported, and sets the library search order.

`uselib Syntax

```
<uselib compiler directive> ::= `uselib [<Verilog-XL uselib directives>]
<lib
directive>]
<Verilog-XL uselib directives> ::= dir = <library_directory> | file =
<library_file>
| libext = <file_extension>
<lib directive> ::= <library reference> {<library reference>}
<library reference> ::= lib = <logical library name>
```

`uselib Lib Semantics

The ``uselib lib` directive cannot be used with any of the Verilog-XL ``uselib` directives. For example, the following code is illegal:

```
`uselib dir=./ file=f.v lib=newlib
```

Multiple libraries can be specified in one ``uselib` directive.

The order in which libraries are specified determines the search order. For example:

```
`uselib lib=mylib lib=yourlib
```

Specifies that the search for an instantiated module is made in `mylib` first, followed by `yourlib`.

Like the directives, such as ``uselib dir`, ``uselib file`, and ``uselib libext`, the ``uselib lib` directive is persistent across HDL files in a given invocation of parsing and analyzing, just like an invocation of parsing is persistent. Unless another ``uselib` directive is encountered, a ``uselib` (including any Verilog XL ``uselib`) directive in the HDL source remains in effect. A ``uselib` without any argument removes the effect of any currently active ``uselib <lib|file|dir|libext>`.

The following module search mechanism is used for resolving an instantiated module or UDP by the Verific Verilog elaboration algorithm:

- First, search for the instantiated module in the ordered list of logical libraries of the currently active ``uselib lib` (if any).
- If not found, search for the instantiated module in the ordered list of libraries provided as `search libraries` in `xelab` command line.
- If not found, search for the instantiated module in the library of the parent module. For example, if module A in library `work` instantiated module B of library `mylib` and B instantiated module C, then search for module C in the `/mylib`, library, which is the library of B (parent of C).
- If not found, search for the instantiated module in the `work` library, which is one of the following:
 - The library into which HDL source is being compiled
 - The library explicitly set as `work` library
 - The default work library is named as `work`

``uselib Examples`

Table 14: ``uselib Examples`

File <code>half_adder.v</code> compiled into logical library named <code>adder_lib</code>	File <code>full_adder.v</code> compiled into logical library named <code>work</code>
<pre>module half_adder(a,b,c,s); input a,b; output c,s; s = a ^ b; c = a & b; endmodule</pre>	<pre>module full_adder(c_in, c_out, a, b, sum) input c_in,a,b; output c_out,sum; wire carry1,carry2,sum1; `uselib lib = adder_lib half_adder adder1(.a(a),.b(b),.c(carry1),.s(sum1)); half_adder adder1(.a(sum1),.b(c_in),.c (carry2),.s(sum)); c_out = carry1 carry2; endmodule</pre>

xelab, xvhd1, and xvlog xsim Command Options

The following table lists the command options for the `xelab`, `xvhdl`, and `xvlog xsim` commands.

Table 15: `xelab`, `xvhdl`, and `xvlog` Command Options

Command Option	Description	Used by Command
<code>-d [define] <name>[=<val>]</code>	Define Verilog macros. Use <code>-d --define</code> for each Verilog macro. The format of the macro is <code><name>[=<val>]</code> where <code><name></code> is name of the macro and <code><value></code> is an optional value of the macro.	<code>xelab</code> Parsing Design Files, xvhd1 and xvlog

Table 15: **xelab**, **xvhd**, and **xvlog** Command Options (cont'd)

Command Option	Description	Used by Command
<code>-debug <kind></code>	<p>Compile with specified debugging ability turned on. The <code><kind></code> options are:</p> <ul style="list-style-type: none"> • <code>typical</code>: Most commonly used abilities, including: <code>line</code> and <code>wave</code>. • <code>line</code>: HDL breakpoint. • <code>wave</code>: Waveform generation, conditional execution, force value. • <code>xlibs</code>: Visibility into Xilinx® precompiled libraries. This option is only available on the command line. • <code>off</code>: Turn off all debugging abilities (Default). • <code>all</code>: Uses all the debug options. 	<code>xelab</code>
<code>-cryptodumps</code>	Encrypt parsed dump of design units being compiled.	Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code> Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code>
<code>-f [-file] <filename></code>	Read additional options from the specified file.	xelab <code>xsim</code> Executable Options Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code> Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code>
<code>-generic_top <value></code>	Override generic or parameter of a top-level design unit with specified value. Example: <code>-generic_top "P1=10"</code>	xelab
<code>-h [-help]</code>	Print this help message.	xelab <code>xsim</code> Executable Options Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code> Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code>
<code>-i [include] <directory_name></code>	Specify directories to be searched for files included using Verilog <code>\include</code> . Use <code>-i --include</code> for each specified search directory.	xelab Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code>
<code>-initfile <init_filename></code>	User-defined simulator initialization file to add to or override settings provided by the default <code>xsim.ini</code> file.	xelab Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code> Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code>
<code>-L [-lib] <library_name> [=<library_dir>]</code>	Specify search libraries for the instantiated non-VHDL design units; for example, a Verilog design unit. Use <code>-L --lib</code> for each search library. The format of the argument is <code><name>[=<dir>]</code> where <code><name></code> is the logical name of the library and <code><library_dir></code> is an optional physical directory of the library.	xelab Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code> Parsing Design Files, <code>xvhdl</code> and <code>xvlog</code>

Table 15: **xelab, xvhd, and xvlog Command Options** (cont'd)

Command Option	Description	Used by Command
-log <filename>	Specify the log file name. Default: <xvlog xvhdl xelab xsim>.log.	xelab xsim Executable Options Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog
-maxarraysize <arg>	Set maximum vhdl array size to be 2^{**n} (Default: n = 28, which is 2^{**28}).	xelab
-maxdelay	Compile Verilog design units with maximum delays.	xelab
-maxdesigndepth <arg>	Override maximum design hierarchy depth allowed by the elaborator (Default: 5000).	xelab
-maxlogsize <arg>	Set the maximum size a log file can reach in MB. The default setting is unlimited.	xsim Executable Options
-mindelay	Compile Verilog design units with minimum delays.	xelab
-mt <arg>	<p>Specifies the number of sub-compilation jobs which can be run in parallel. Possible values are <code>auto</code>, <code>off</code>, or an integer greater than 1.</p> <p>If <code>auto</code> is specified, <code>xelab</code> selects the number of parallel jobs based on the number of CPUs on the host machine. (Default = <code>auto</code>.)</p> <p>Advanced usage: to further control the <code>-mt</code> option, you can set the Tcl property as follows:</p> <pre data-bbox="675 1115 1122 1163">set_property XELAB.MT_LEVEL off N [get_filesets sim_1]</pre>	xelab
-nolog	Suppress log file generation.	xelab xsim Executable Syntax Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog
-noieeewarnings	Disable warnings from VHDL IEEE functions.	xelab
-noname_unnamed_generate	Do not generate name for an unnamed generate block.	xelab Parsing Design Files, xvhd and xvlog
-notimingchecks	Ignore timing check constructs in Verilog specify block(s).	xelab
-nosdfinterconnectdelays	Ignore SDF port and interconnect delay constructs in SDF.	xelab
-nospecify	Ignore Verilog path delays and timing checks.	xelab

Table 15: **xelab, xvhd, and xvlog Command Options** (cont'd)

Command Option	Description	Used by Command
-O <arg>	<p>Enable or disable optimizations.</p> <ul style="list-style-type: none"> -O 0 = Disable optimizations -O 1 = Enable basic optimizations -O 2 = Enable most commonly desired optimizations (Default) -O 3 = Enable advanced optimizations <p>Note: A lower value speeds compilation at expense of slower simulation; a higher value slows compilation but simulation runs faster.</p>	xelab
-override_timeunit	Override timeunit for all Verilog modules, with the specified time unit in -timescale option.	xelab
-override_timeprecision	Override time precision for all Verilog modules, with the specified time precision in -timescale option.	xelab
-pulse_e <arg>	Path pulse error limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_r <arg>	Path pulse reject limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_int_e <arg>	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_int_r <arg>	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_e_style <arg>	<p>Specify when error about pulse being shorter than module path delay should be handled. Choices are:</p> <ul style="list-style-type: none"> ondetect: Report error right when violation is detected. onevent: Report error after the module path delay. <p>Default: onevent</p>	xelab
-prj <filename>	Specify the Vivado simulator project file containing one or more entries of vhdl verilog <work lib> <HDL file name>.	xelab Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog
-r [-run]	Run the generated executable snapshot in command-line interactive mode.	xelab
-rangecheck	Enable run time value range check for VHDL.	xelab
-R [-runall]	Run the generated executable snapshot until the end of simulation.	xelab xsim Executable Syntax
-relax	Relax strict language rules.	xelab Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog

Table 15: **xelab**, **xvhd**, and **xvlog** Command Options (cont'd)

Command Option	Description	Used by Command
<code>-s [-snapshot] <arg></code>	Specify the name of output simulation snapshot. Default is <code><worklib>.<unit></code> ; for example: <code>work.top</code> . Additional unit names are concatenated using <code>#</code> ; for example: <code>work.t1#work.t2</code> .	<code>xelab</code>
<code>-sdfnowarn</code>	Do not emit SDF warnings.	<code>xelab</code>
<code>-sdfnoerror</code>	Treat errors found in SDF file as warning.	<code>xelab</code>
<code>-sdfmin <arg></code>	<code><root>=<file></code> SDF annotate <code><file></code> at <code><root></code> with minimum delay.	<code>xelab</code>
<code>-sdftyp <arg></code>	<code><root>=<file></code> SDF annotate <code><file></code> at <code><root></code> with typical delay.	<code>xelab</code>
<code>-sdfmax <arg></code>	<code><root>=<file></code> SDF annotate <code><file></code> at <code><root></code> with maximum delay.	<code>xelab</code>
<code>-sdfroot <root_path></code>	Default design hierarchy at which SDF annotation is applied.	<code>xelab</code>
<code>-sourcelibdir <sourcelib dirname></code>	Directory for Verilog source files of uncompiled modules. Use <code>-sourcelibdir <sourcelib dirname></code> for each source directory.	<code>xelab</code> Parsing Design Files, xvhdl and xvlog
<code>-sourcelibext <file_extension></code>	File extension for Verilog source files of uncompiled modules. Use <code>-sourcelibext <file extension></code> for source file extension.	<code>xelab</code> Parsing Design Files, xvhdl and xvlog
<code>-sourcelibfile <filename></code>	File name of a Verilog source file with uncompiled modules.	<code>xelab</code> Parsing Design Files, xvhdl and xvlog
<code>-stat</code>	Print tool CPU and memory usages, and design statistics.	<code>xelab</code>
<code>-sv</code>	Compile input files in SystemVerilog mode.	Parsing Design Files, xvhdl and xvlog
<code>-timescale</code>	Specify default timescale for Verilog modules. Default: <code>1ns/1ps</code> .	<code>xelab</code>
<code>-timeprecision_vhdl <arg></code>	Specify time precision for vhdl designs. Default: <code>1ps</code> .	<code>xelab</code>
<code>-transport_int_delays</code>	Use transport model for interconnect delays.	<code>xelab</code>
<code>-typdelay</code>	Compile Verilog design units with typical delays (Default).	<code>xelab</code>
<code>-v [verbose] [0 1 2]</code>	Specify verbosity level for printing messages. Default = 0.	<code>xelab</code> Parsing Design Files, xvhdl and xvlog Parsing Design Files, xvhdl and xvlog
<code>-version</code>	Print the compiler version to screen.	<code>xelab</code> <code>xsim Executable Options</code> Parsing Design Files, xvhdl and xvlog Parsing Design Files, xvhdl and xvlog

Table 15: **xelab, xvhd, and xvlog Command Options** (cont'd)

Command Option	Description	Used by Command
<code>-work <library_name> [<library_dir>]</code>	Specify the work library. The format of the argument is <code><name>[=<dir>]</code> where: <ul style="list-style-type: none"> <code><name></code> is the logical name of the library. <code><library_dir></code> is an optional physical directory of the library. 	Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog
<code>-sv_root <arg></code>	Root directory of which DPI libraries are to be found. Default: <code><current_directory>/xsim.dir/xsc</code>	xelab
<code>--sc_lib arg</code>	Shared library name for SystemC functions; (.dll/.so) without the file extension	xelab
<code>--sc_root <arg></code>	Root directory of which SystemC libraries are to be found. Default: <code><current_directory>/xsim.dir/work/xsc</code>	xelab
<code>-sv_lib <arg></code>	Shared library name for DPI imported functions (.dll/.so) without the file extension.	xelab
<code>-sv_liblist <arg></code>	Bootstrap file pointing to DPI shared libraries.	xelab
<code>-dpiheader <arg></code>	Header filename for the exported and imported functions.	xelab
<code>-driver_display_limit <arg></code>	Enable driver debugging for signals with maximum size (Default: n = 65536).	xelab
<code>-dpi_absolute</code>	Use absolute paths instead of <code>LD_LIBRARY_PATH</code> on Linux for DPI libraries that are formatted as <code>lib<libname>.so</code> .	xelab
<code>-incr</code>	Enable incremental analysis/elaboration in simulation.	Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog xelab
<code>-93_mode</code>	Compile VHDL in pure 93 mode.	Parsing Design Files, xvhd and xvlog xelab
<code>-2008</code>	Compile VHDL file in 2008 mode.	Parsing Design Files, xvhd and xvlog
<code>-nosignalhandlers</code>	Don't allow compiler to trap Antivirus, firewall signal.	Parsing Design Files, xvhd and xvlog Parsing Design Files, xvhd and xvlog xelab
<code>-dpi_stacksize <arg></code>	User defined stack size for DPI task.	xelab
<code>-transform_timing_checkers</code>	Transform timing checker to Verilog process.	xelab
<code>-a</code>	Generate a standalone non-interactive simulation executable that performs run-all. Always use with <code>-R</code> . To run the simulation faster without any debug capability, use <code>-standalone</code> with <code>-R</code> . It will invoke the Simulation standalone without invoking Vivado IDE. This option will save the license loading time.	xelab

Table 15: **xelab**, **xvhd**, and **xvlog** Command Options (cont'd)

Command Option	Description	Used by Command
<code>-ignore_assertions</code>	Ignore SystemVerilog concurrent assertions.	<code>xelab</code>
<code>-ignore_coverage</code>	Ignore SystemVerilog functional coverage.	<code>xelab</code>
<code>-cov_db_dir <arg></code>	Functional coverage database dump directory. The coverage data is present under <code><arg>/xsim.covdb/<cov_db_name></code> directory. Default is <code>./</code> .	<code>xelab</code>
<code>-cov_db_name <arg></code>	Functional coverage database name. The coverage data is present under <code><cov_db_dir>/xsim.covdb/<arg></code> directory. Default is a snapshot name.	<code>xelab</code>
<code>-uvm_version <arg></code>	Specify UVM version (default 1.2).	Parsing Design Files, xvhd1 and xvlogxelab
<code>-report_assertion_pass</code>	Report SystemVerilog Concurrent Assertions Pass, even if there is no pass action block.	<code>xelab</code>
<code>-dup_entity_as_module</code>	Enable support for hierarchical references inside the Verilog hierarchy in mixed language designs.	<code>xelab</code>
	 CAUTION! This may cause significant slow down of compilation.	
<code>-cc_celldefines</code>	Specify if code coverage information needs to be captured for libs/modules with cell define attribute set. OFF by default.	<code>xelab</code>
<code>-cc_libs</code>	Specify if code coverage information needs to be captured for all the libraries specified. OFF by default.	<code>xelab</code>
<code>-cc_type arg</code>	Specify options for generating Code Coverage Statistics -bcesfxt. (s)Statement Coverage, (b)Branch Coverage, (c)Condition Coverage Supported.	<code>xelab</code>
<code>-cc_db arg</code>	Code coverage database will be saved inside <code><cc_dir_argvalue>/xsim.codecov/<cc_db_argvalue></code> . Default is <code>SnapshotName</code> .	<code>xelab</code>
<code>-cc_dir arg</code>	Code coverage database will be saved under the dir <code><cc_dir_argvalue>/xsim.codecov/<cc_db_argvalue></code> . Default is <code>./xsim.codecov/</code> .	<code>xelab</code>
<code>-ignore_localparam_override</code>	Ignore localparam override	<code>xelab</code>

Simulating the Design Snapshot, xsim

The `xsim` command loads a simulation snapshot to effect a batch mode simulation or provides a workspace (GUI) and/or a Tcl-based interactive simulation environment.

xsim Executable Syntax

The command syntax is as follows:

```
xsim <options> <snapshot>
```

Where:

- `xsim` is the command.
- `<options>` are the options specified in the following table.
- `<snapshot>` is the simulation snapshot.

xsim Executable Options

Table 16: xsim Executable Command Options

xsim Option	Description
<code>-f [-file] <filename></code>	Load the command line options from a file.
<code>-g [-gui]</code>	Run with interactive workspace.
<code>-h [-help]</code>	Print help message to screen.
<code>-log <filename></code>	Specify the log file name.
<code>-maxdeltaid arg (= -1)</code>	Specify the maximum delta number. Report an error if it exceeds maximum simulation loops at the same time.
<code>-maxlogsize arg (= -1)</code>	Set the maximum size a log file can reach in MB. The default setting is unlimited.
<code>-ieeewarnings</code>	Enable warnings from VHDL IEEE functions.
<code>-nolog</code>	Suppresses log file generation.
<code>-nosignalhandlers</code>	<p>Disables the installation of OS-level signal handlers in the simulation. For performance reasons, the simulator does not check explicitly for certain conditions, such as an integer division by zero, that could generate an OS-level fatal run time error. Instead, the simulator installs signal handlers to catch those errors and generates a report.</p> <p>With the signal handlers disabled, the simulator can run in the presence of such security software, but OS-level fatal errors could crash the simulation abruptly with little indication of the nature of the failure.</p> <p> CAUTION! Use this option only if your security software prevents the simulator from running successfully.</p>
<code>-onfinish <quit stop></code>	Specify the behavior at end of simulation.
<code>-onerror <quit stop></code>	Specify the behavior upon simulation run time error.
<code>-R [-runall]</code>	Runs simulation till end (such as <code>do 'run all;quit'</code>).
<code>-stats</code>	Display memory and CPU stats upon exiting.
<code>-testplusarg <arg></code>	Specify <code>plusargs</code> to be used by <code>\$test\$plusargs</code> and <code>\$value\$plusargs</code> system functions.
<code>-t [-tclbatch] <filename></code>	Specify the Tcl file for batch mode execution.
<code>-tp</code>	Enable printing to screen of hierarchical names of process being executed.
<code>-tl</code>	Enable printing to screen of file name and line number of statements being executed.
<code>-wdb <filename.wdb></code>	Specify the waveform database output file.
<code>-version</code>	Print the compiler version to screen.
<code>-view <wavefile.wcfg></code>	Open a wave configuration file. Use this switch together with <code>-gui</code> switch.
<code>-protoinst</code>	Specify a .protoinst file for protocol analysis.

Table 16: xsim Executable Command Options (cont'd)

xsim Option	Description
-sv_seed	Seed for SystemVerilog constraint random.
-cov_db_dir	Functional coverage database dump directory. The coverage data is present under <arg>/xsim.covdb/<cov_db_name> directory. Default is ./ or inherits the value set in xelab.
-cov_db_name	Functional coverage database name. The coverage data will be present under <cov_db_dir>/xsim.covdb/<arg> directory. Default is snapshot name or inherits the value set in xelab.
- downgrade_error2info	Downgrade the severity level of the HDL messages from Error to Info.
- downgrade_error2warning	Downgrade the severity level of the HDL messages from Error to Warning.
- downgrade_fatal2info	Downgrade the severity level of the HDL messages from Fatal to Info.
- downgrade_fatal2warning	Downgrade the severity level of the HDL messages from Fatal to Warning.
- downgrade_severity	Downgrade the severity level of the HDL messages. Following are the choices: <ul style="list-style-type: none"> error2warning error2info fatal2warning fatal2info
- ignore_assertions	Ignore SystemVerilog concurrent assertions.
- ignore_coverage	Ignore SystemVerilog functional coverage.
- ignore_feature	Ignore the effect of a specific HDL feature or construct. Following are the choices: <ul style="list-style-type: none"> assertion coverage
- tempDir	Specify the temporary directory name.
- autoloadwcfg	Load already saved waveform configuration file.



TIP: When running the `xelab`, `xsc`, `xsim`, `xvhdl`, `xcrg`, or `xvlog` commands in batch files or scripts, it might also be necessary to define the `XILINX_VIVADO` environment variable to point to the installation hierarchy of the Vivado Design Suite. To set the `XILINX_VIVADO` variable, add one of the following to your script or batch file:

- On Windows: `set XILINX_VIVADO=<vivado_install_area>/Vivado/<version>`
- On Linux: `setenv XILINX_VIVADO vivado_install_area>/Vivado/<version>`
- Where <version> is the version of Vivado tools you are using: 2014.3, 2014.4, 2015.1, etc

Functional Coverage Report Generator

Vivado simulator provides a utility using which you can generate the functional coverage report either in text or html format. The Xilinx Coverage Report Generator (XCRG) can also be used for merging multiple coverage databases into a single database.

Table 17: xcrg Command Options and Description

xcrg Option	Description
-db_name arg	Name of the database inside xsim.covdb. If unspecified all databases present in the directory are used.
-dir arg	Path where the xsim.covdb database directory is located. Default is ./xsim.covdb.
-file arg	Specify file with location of the coverage databases to be restored.
-h	Print help message and exit.
-help	Print help message and exit.
-merge_db_name arg	Name of the merged database. Default is xcrg_mdb.
-merge_dir arg	Directory where the merged database is saved. Default is ./xsim.covdb.
-nolog	Suppresses the log file generation.
-report_dir arg	Directory where the coverage database and the report are saved. Default is ./xcrg_report.
-report_format arg	Specify the desired format of the coverage report html or text or all. Default is html.
-log arg	Specify the file name which has the log saved. Default is xcrg.log.
-version	Print the version of XCRG and exit.
-cc_db <arg>	Specify the DB Name (Snapshot Name) which is used to save the code coverage database. Code coverage database can be restored from <cc_dir_argvalue>/xsim.codeCov/<cc_db_argvalue>.
-cc_dir <arg>	Specify the directory where the code coverage information database is saved. Code coverage database can be restored from <cc_dir_argvalue>/xsim.codeCov/<cc_db_argvalue>. Default is ./xsim.CodeCov/.
-cc_fullfile	Show the entire file in the code coverage report. By default this is OFF for files more than 50000 lines and only the file's module contents are shown.
-cc_report <arg>	Directory where the code coverage HTML report is saved. Default is xcrg_code_cov_report.
-merge_cc	Merge the code coverage databases specified and create a output merged code coverage database.
-cc_instancescount <arg>	Specify the maximum number of instances shown in the code coverage report. Default is 100.

xcrg Example Syntax

```

xcrg -h
xcrg -file /path/to/file
xcrg -file /path/to/file -db_name work.top
xcrg -dir /path/to/abc
xcrg -dir ./abc -report_dir def -report_format html
xcrg -dir ./abc -db_name work.top -report_dir def -report_format html
xcrg -dir /path/to/abc -db_name work.top -report_dir def -report_format text
xcrg -merge_dir m
xcrg -merge_db_name xyz -report_dir def
xcrg -report_format html -nolog
xcrg -report_format html -log xcrgOutput.log
xcrg -cc_db a1 -cc_dir ../
xcrg -cc_report abc -cc_db work.testbench -cc_dir ./xsim.codeCov/

```

Example of Running Vivado Simulator in Standalone Mode

When running the Vivado simulator in standalone mode, you can execute commands to:

- Analyze the design file
- Elaborate the design and create a snapshot
- Open the Vivado simulator workspace and wave configuration file(s) and run simulation

Step 1: Analyzing the Design File

To begin, analyze your HDL source files by type, as shown in the table below. Each command can take multiple files.

Table 18: File Types and Associated Commands for Design File Analysis

File Type	Command
Verilog	<code>xvlog <VerilogFileName(s)></code>
SystemVerilog	<code>xvlog -sv <SystemVerilogFileName(s)></code>
VHDL	<code>xvhdl <VhdlFileName(s)></code>

Step 2: Elaborating and Creating a Snapshot

After analysis, elaborate the design and create a snapshot for simulation using the `xelab` command:

```
xelab <topDesignUnitName> -debug typical
```



IMPORTANT! You can provide multiple top-level design unit names with `xelab`. To use the Vivado simulator workspace for purposes similar to those used during `launch_simulation`, you must set debug level to `typical`.

Step 3: Running Simulation

After successful completion of the `xelab` phase, the Vivado simulator creates a snapshot used for running simulation.

To invoke the Vivado simulator workspace, use the following command:

```
xsim <SnapshotName> -gui
```

To open the wave config file:

```
xsim <SnapShotName> -view <wcfg FileName> -gui
```

You can provide multiple `wcfg` files using multiple `-view` flags. For example:

```
xsim <SnapShotName> -view <wcfg FileName> -view <wcfg FileName>
```

Project File (.prj) Syntax

Note: The project file discussed here is a Vivado simulator text-based project file. It is not the same as the project file (`.xpr`) created by the Vivado Design Suite.

To parse design files using a project file, create a text file called `<proj_name>.prj`, and use the syntax shown below inside the project file.

```
verilog <work_library> <file_names>... [-d <macro>]... [-i  
<include_path>]...  
vhdl <work_library> <file_name>  
sv <work_library> <file_name>  
vhdl2008 <work_library> <file_name>
```

Where:

`<work_library>`: Is the library into which the HDL files on the given line are to be compiled.

`<file_names>`: Are Verilog source files. You can specify multiple Verilog files per line.

`<file_name>`: Is a VHDL source file; specify only one VHDL file per line.

1. For Verilog or SystemVerilog: `[-d <macro>]` provides you the option to define one or more macros.
2. For Verilog or SystemVerilog: `[-i <include_path>]` provides you the option to define one or more `<include_path>` directories.

Predefined Macros

XILINX_SIMULATOR is a Verilog predefined-macro. The value of this macro is 1. Predefined macros perform tool-specific functions, or identify which tool to use in a design flow. The following is an example usage:

```
' ifdef VCS
    // VCS specific code
` endif
` ifdef INCA
    // NCSIM specific code
` endif
` ifdef MODEL_TECH
    // MODELSIM specific code
` endif
` ifdef XILINX_ISIM
    // ISE Simulator (ISim) specific code
` endif
` ifdef XILINX_SIMULATOR
    // Vivado Simulator (XSim) specific code
` endif
` ifdef _VCP
//Aldec specific code
` endif
```

Library Mapping File (xsim.ini)

The HDL compile programs, xvhdl, xvlog, and xelab, use the xsim.ini configuration file to find the definitions and physical locations of VHDL and Verilog logical libraries.

The compilers attempt to read xsim.ini from these locations in the following order:

1. xsim.ini in current working directory
2. User-file specified through the -initfile switch. If -initfile is not specified, the program searches for xsim.ini in the current working directory.
3. <Vivado_Install_Dir>/data/xsim

The xsim.ini file has the following syntax:

```
<logical_library1> = <physical_dir_path1>
<logical_library2> = <physical_dir_path2>
```

The following is an example `xsim.ini` file:

```
std=<Vivado_Install_Area>/xsim/vhdl/std
ieee=<Vivado_Install_Area>/xsim/vhdl/ieee
vl=<Vivado_Install_Area>/xsim/vhdl/vl
ieee_proposed=$RDI_DATADIR/xsim/vhdl/ieee_proposed
synopsys=<Vivado_Install_Area>/xsim/vhdl/synopsys
uvm=<Vivado_Install_Area>/xsim/system_verilog/uvm
unisim=<Vivado_Install_Area>/xsim/vhdl/unisim
unimacro=<Vivado_Install_Area>/xsim/vhdl/unimacro
unifast=<Vivado_Install_Area>/xsim/vhdl/unifast
simprims_ver=<Vivado_Install_Area>/xsim/verilog/simprims_ver
unisims_ver=<Vivado_Install_Area>/xsim/verilog/unisims_ver
unimacro_ver=<Vivado_Install_Area>/xsim/verilog/unimacro_ver
unifast_ver=<Vivado_Install_Area>/xsim/verilog/unifast_ver
secureip=<Vivado_Install_Area>/xsim/verilog/secureip
work=../work
```

The `xsim.ini` file has the following features and limitations:

- There must be no more than one library path per line inside the `xsim.ini` file.
- If the directory corresponding to the physical path does not exist, `xvhd` or `xvlog` creates it when the compiler first tries to write to that path.
- You can describe the physical path in terms of environment variables. The environment variable must start with the `$` character.
- The default physical directory for a logical library is `xsim/<language>/<logical_library_name>`, for example, a logical library name of:

```
<Vivado_Install_Area>/xsim/vhdl/unisim
```

- File comments must start with `--`.

Note: From 2018.2 release onwards, Xilinx provides two init files named as `xsim.ini` and `xsim_legacy.ini`. The `xsim_legacy.ini` file is similar to `xsim.ini` of older version. It contains mapping for UNISIM library while the new `xsim.ini` file contains mapping for all the files of UNISIM library along with the mapping for pre-compiled IP.

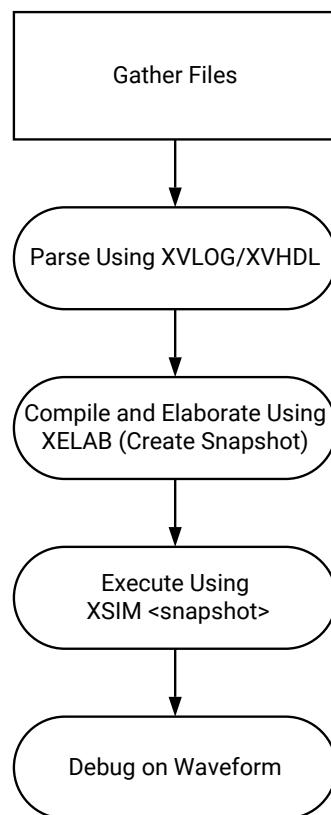
Running Simulation Modes

You can run any mode of simulation from the command line. The following subsections illustrate and describe the simulation modes when run from the command line.

Behavioral Simulation

The following figure illustrates the behavioral simulation process:

Figure 51: Behavioral Simulation Process



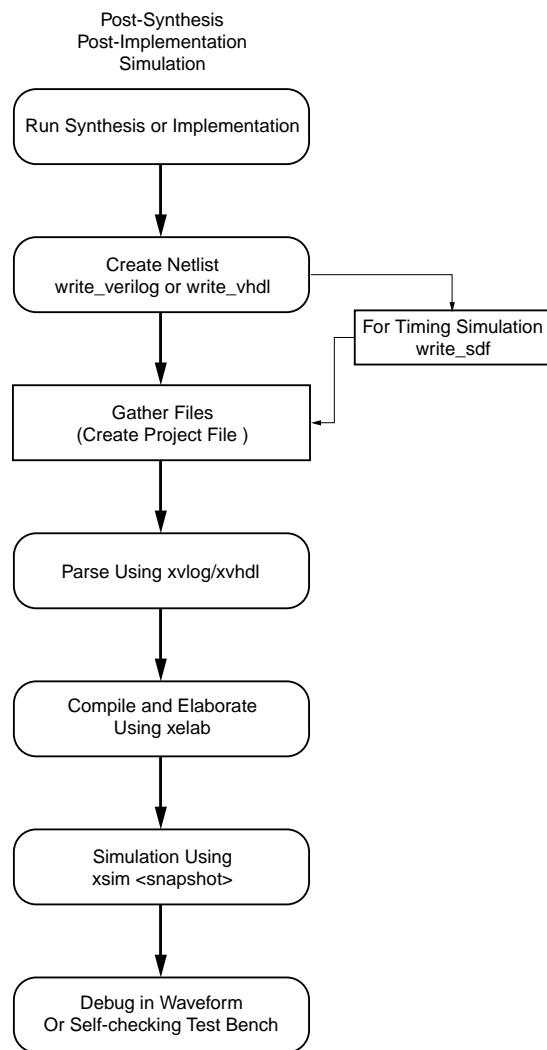
X23705-021420

To run behavioral simulation from within the Vivado Design Suite, use the Tcl command:
`launch_simulation -mode behavioral`.

Running Post-Synthesis and Post-Implementation Simulations

At post-synthesis and post-implementation, you can run a functional or a Verilog timing simulation. The following figure illustrates the post-synthesis and post-implementation simulation process:

Figure 52: Post-Synthesis and Post-Implementation Simulation



X12985

The following is an example of running a post-synthesis functional simulation from the command line:

```

synth_design -top top -part xc7k70tfbg676-2
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
launch_simulation
  
```



TIP: When you run a post-synthesis or post-implementation timing simulation, you must run the `write_sdf` command after the `write_verilog` command, and the appropriate `annotate` command is needed for elaboration and simulation.

Using Tcl Commands and Scripts

You can run Tcl commands on the Tcl Console individually, or batch the commands into a Tcl script to run simulation.

Using a `-tclbatch` File

You can type simulation commands into a Tcl file, and reference the Tcl file with the following command: `-tclbatch <filename>`

Use the `-tclbatch` option to contain commands within a file and execute those command as simulation starts. For example, you can have a file named `run.tcl` that contains the following:

```
run 20ns
id="ag415279">current_time
quit
```

Then launch simulation as follows:

```
xsim <snapshot> -tclbatch run.tcl
```

You can set a variable to represent a simulation command to quickly run frequently used simulation commands.

Launching Vivado Simulator from the Tcl Console

The following is an example of Tcl commands that create a project, read in source files, launch the Vivado simulator, do placing and routing, write out an SDF file, and re-launch simulation.

```
Vivado -mode Tcl
Vivado% create_project prj1
Vivado% read_verilog dut.v
Vivado% synth_design -top dut
Vivado% launch_simulation -simset sim_1 -mode post-synthesis -type
functional
Vivado% place_design
Vivado% route_design
Vivado% write_verilog -mode timesim -sdf_anno true -sdf_file postRoute.sdf
postRoute_netlist.v
Vivado% write_sdf postRoute.sdf
Vivado% launch_simulation -simset sim_1 -mode post-implementation -type
timing
Vivado% close_project
```

export_simulation

Export a simulation script file for the target simulator. The generated script will contain simulator commands for compiling, elaborating and simulating the design.

This command will retrieve the simulation compile order of specified objects, and export this information in a shell script with the compiler commands and default options for the target simulator. The specified object can be either a simulation fileset or an IP. If you want to run simulation outside Vivado IDE, use `export_simulation` in place of `launch_simulation --scripts-only` to generate scripts file.

```
export_simulation [-simulator <arg>] [-of_objects <arg>]
                  [-ip_user_files_dir <arg>] [-ipstatic_source_dir <arg>]
                  [-lib_map_path <arg>] [-script_name <arg>]
                  [-directory <arg>] [-runtime <arg>] [-define <arg>]
                  [-generic <arg>] [-include <arg>] [-use_ip_compiled_libs]
                  [-absolute_path] [-export_source_files]
                  [-generate_hier_access] [-32bit] [-force] [-quiet]
                  [-verbose] [-gcc_install_path <arg>] [-more_options <arg>]
```

Usage

Table 19: export_simulation Options

Name	Description
<code>[-simulator]</code>	Simulator for which the simulation script will be created. Allowed values are all, xsim, modelsim, questa, vcs, xcelium, riviera, and activehdl. Default: all
<code>[-of_objects]</code>	Export simulation script for the specified object. Default: None
<code>[-lib_map_path]</code>	Precompiled simulation library directory path. If not specified, follow the instructions in the generated script header to manually provide the simulation library mapping information. Default: Empty
<code>[-script_name]</code>	Output shell script filename. If not specified, then file with a default name will be created. Default: <code>top_module.sh</code>
<code>[-directory]</code>	Directory where the simulation script will be generated. Default: <code>export_sim</code>
<code>[-runtime]</code>	Run simulation for this time. Default: full simulation run or until a logical break or finish condition
<code>[-absolute_path]</code>	Make all file paths absolute with respect to the reference directory.
<code>[-export_source_files]</code>	Copy IP/BD design files to output directory.
<code>[-32bit]</code>	Perform 32-bit compilation.
<code>[-force]</code>	Overwrite previous files.
<code>[-quiet]</code>	Ignore command errors.
<code>[-verbose]</code>	Suspend message limits during command execution.

Table 19: **export_simulation Options** (cont'd)

Name	Description
<code>[-ip_user_files_dir]</code>	Directory path to exported IP/BD user files (for static, dynamic and data files). Default: Empty
<code>[-ipstatic_source_dir]</code>	Directory path to the exported IP/BD static files. Default: Empty
<code>[-define]</code>	Read Verilog defines from the list specified with this switch. Default: Empty
<code>[-generic]</code>	Read VHDL generics from the list specified with this switch. Default: Empty
<code>[-include]</code>	Read include directory paths from the list specified with this switch. Default: Empty
<code>[-use_ip_compiled_libs]</code>	Reference pre-compiled IP static library during compilation. This switch requires <code>-ip_user_files_dir</code> and <code>-ipstatic_source_dir</code> switches also for generating scripts using pre-compiled IP library.
<code>[-generate_hier_access]</code>	Extract path for hierarchical access simulation
<code>[-gcc_install_path]</code>	GNU compiler installation directory path for the g++/gcc executables. Default: Empty
<code>[-more_options]</code>	Pass specified options to the simulator tool. Default: Empty

Description

Export a simulation script file for the target simulator (please see the list of supported simulators below). The generated script will contain simulator commands for compiling, elaborating and simulating the design.

The command will retrieve the simulation compile order of specified objects, and export this information in a shell script with the compiler commands and default options for the target simulator. The specified object can be either a simulation fileset, IP or a BD (block design).

If the object is not specified, then this command will generate the script for the active simulation `top`. Any Verilog include directories or file paths for the files containing Verilog define statements will be added to the compiler command line.

By default, the design source file and include directory paths in the compiler command line will be set relative to the `reference_dir` variable that is set in the generated script. To make these paths absolute, specify the `-absolute_path` switch.

The command will also copy data files (if any) from the fileset, or from an IP, to the output directory. If the design contains Verilog sources, then the generated script will also copy the `glbl.v` file from the software installation path to the output directory.

A default `.do` file that is used in the compiler commands in the simulation script for the target simulator, will be written to the output directory.

Note: In order to perform simulation with the generated script, the simulation libraries must be compiled first using the `compile_simlib` Tcl command. The compiled library directory path must be specified when generating this script. The generated script will automatically include the setup files for the target simulator from the compiled library directory.

Supported Simulators

- Vivado simulator (xsim)
- ModelSim Simulator (modelsim)
- Questa Advanced Simulator (questa)
- Verilog Compiler Simulator (vcs)
- Riviera-PRO Simulator (riviera)
- Active-HDL Simulator (activehdl)
- Cadence Xcelium Parallel Simulator (xcelium)

Arguments

- **-of_objects:** (Optional) Specify the target object for which the simulation script file needs to be generated. The target object can be either a simulation fileset (simset) or an IP. If this option is not specified then this command will generate file for the current simulation fileset.
- **-lib_map_path:** (Optional) Specify path to the Xilinx pre-compiled simulation library for the selected simulator. The simulation library is compiled using `compile_simlib`. See the header section in the generated script for more information. If this switch is not specified, then the generated script will not reference the pre-compiled simulation library and the static IP files will be locally compiled.
- **-script_name:** (Optional) Specify name of the generated script. Default name is `<simulation_top>.sh`. If the `-of_objects` switch is specified, then the default syntax of the script will be as follows:

```
-of_objects [current_fileset -simset] .sh
-of_objects [get_ips ] .sh
-of_objects [get_files .xci] .sh
-of_objects [get_files .bd] .sh
```

- **-absolute_path:** (Optional) Specify this option to make source and include directory paths absolute. By default, all paths are set relative to the output directory specified with the `-directory` switch.
- **-32bit:** (Optional) Specify this option to perform 32-bit simulation. If this option is not specified then by default 64-bit option will be added to the simulation command line.
- **-force:** (Optional) Overwrite an existing script file of the same name. If the script file already exists, the tool returns an error unless the `-force` argument is specified.

- **-directory:** (Required) Specify the directory path where the script file will be exported.
- **-simulator:** (Required) Specify the target simulator name for the simulation script. The valid simulators names are `xsim`, `modelsim`, `questa`, and `vcs` (or `vcs_mx`).
- **-quiet:** (Optional) Execute the command quietly, ignoring any command line errors and returning no messages. The command also returns `TCL_OK` regardless of any errors encountered during execution.
- **-verbose:** (Optional) Temporarily override any message limits and return all messages from this command.
- **-generate_hier_access:** (Optional) Extract path for hierarchical access simulation.
- **-runtime:** (Optional) Specify simulation run-time.
- **-define:** (Optional) Specify the list of verilog defines used in the design.
- **-generic:** (Optional) Specify the list of VHDL generics used in the design.
- **-include:** (Optional) Specify the list of include directory paths for verilog include files in the design.
- **-export_source_files:** (Optional) Specify this option to copy the IP design files to the generated script directory in a sub-directory named `srcs`. The generated script will reference the design files from this `srcs` directory.

export_ip_user_files

Generate and export IP/IP integrator user files from a project. This can be scoped to work on one or more IPs.

Syntax

```
export_ip_user_files [-of_objects <arg>] [-ip_user_files_dir <arg>]
                     [-ipstatic_source_dir <arg>] [-lib_map_path <arg>]
                     [-no_script] [-sync] [-reset] [-force] [-quiet]
                     [-verbose]
```

Returns: List of files that were exported.

Usage

Table 20: **export_ip_user_files**

Name	Description
<code>[-of_objects]</code>	IP, IP integrator or a fileset. Default: None
<code>[-ip_user_files_dir]</code>	Directory path to simulation base directory (for static, dynamic, wrapper, netlist, script, and MEM files). Default: None
<code>[-ipstatic_source_dir]</code>	Directory path to the static IP files. Default: None
<code>[-lib_map_path]</code>	Compiled simulation library directory path. Default: Empty
<code>[-no_script]</code>	Do not export simulation scripts. Default: 1
<code>[-sync]</code>	Delete IP/IP integrator dynamic and simulation script files.
<code>[-reset]</code>	Delete all IP/IP integrator static, dynamic and simulation script files.
<code>[-force]</code>	Overwrite files.
<code>[-quiet]</code>	Ignore command errors.
<code>[-verbose]</code>	Suspend message limits during command execution.

Description

Export IP user files repository with static, dynamic, netlist, Verilog/VHDL stubs and memory initialization files.

Arguments

- `-of_objects`: (Optional) Specify the target object for which the IP static and dynamic files needs to be exported.
- `-ip_user_files_dir`: (Optional) Directory path to IP user files base directory (for dynamic and other IP non static files). By default, if this switch is not specified then this command will use the path specified with the `IP.USER_FILES_DIR` project property value.
- `-ipstatic_source_dir`: (Optional) Directory path to the static IP files. By default, if this switch is not specified then this command will use the path specified with the `SIM.IPSTATIC_SOURCE_DIR` project property value.

Note: If the `-ip_user_files_dir` switch is specified, by default the IP static files will be exported under the sub-directory with the name `ipstatic`. If this switch is specified with `-ipstatic_source_dir`, then the IP static files will be exported in the path specified with the `-ipstatic_source_dir` switch.

- `-clean_dir`: (Optional) Delete all files from central directory (including static, dynamic and other files)

Examples

The following command will export `char_fifo` IP dynamic files to `<project>/<project>.ip_user_files/ip/char_fifo` directory and `char_fifo` IP static files to `<project>/<project>.ip_user_files/ipstatic` directory:

```
% export_ip_user_files -of_objects [get_ips char_fifo]
```

Compilation, Elaboration, Simulation, Netlist, and Advanced Options

From the Vivado IDE Flow Navigator, you can right-click **Simulation**, and select **Simulation Settings** to open the simulation settings in the Settings dialog box. From the Simulation settings, you can set various compilation, elaboration, simulation, netlist, and advanced options.

Compilation Options

The Compilation tab defines and manages compiler directives, which are stored as properties on the simulation fileset and used by the xvlog and xvhd1 utilities to compile Verilog and VHDL source files for simulation.

Vivado Simulator Compilation Options

Table 21: Vivado Simulator Compilation Options

Option	Description
Verilog options	Browse to set Verilog include path and to define macro
Generics/Parameters options	Specify or browse to set the generic/parameter value
xsim.compile.tcl.pre	Tcl file containing set of commands that should be invoked before launch of compilation
xsim.compile.xvlog.nosort	Do not sort Verilog file during compilation
xsim.compile.xvhdl.nosort	Do not sort VHDL file during compilation
xsim.compile.xvlog.relax	Relax strict HDL language checking rules
xsim.compile.xvhdl.relax	Relax strict HDL language checking rules
xsim.compile.xvlog.more_options	More XVLOG compilation options
xsim.compile.xvhdl.more_options	More XVHDL compilation options
xsim.compile.xsc.more_options	More XSC compilation options

Questa Advanced Simulator Compilation Options

Table 22: Questa Advanced Simulator Compilation Options

Option	Description
Verilog options	Browse to set Verilog include path and to define macro
Generics/Parameters options	Specify or browse to set the generic/parameter value
questasim.compile.tcl.pre	TCL file containing set of commands that should be invoked before launch of compilation
questasim.compile.vhdl_syntax	Specify VHDL syntax
questasim.compile.use_explicit_decl	Log all signals
questasim.compile.load_glbl	Load GLBL module
questasim.compile.vlog.more_options	More VLOG compilation options
questasim.compile.vcom.more_options	More VCOM compilation options
questasim.compile.sccom.cores	Specify the number of process cores to run in parallel
questasim.compile.sccom.more_options	More SCCOM compilation options

ModelSim Simulator Compilation Options

Table 23: ModelSim Compilation Options

Option	Description
Verilog options	Browse to set Verilog include path and to define macro
Generics/Parameters options	Specify or browse to set the generic/parameter value
modelsim.compile.tcl.pre	TCL file containing set of commands that should be invoked before launch of compilation
modelsim.compile.vhdl_syntax	Specify VHDL syntax
modelsim.compile.use_explicit_decl	Log all signals
modelsim.compile.load_glbl	Load GLBL module
modelsim.compile.vlog.more_options	More VLOG compilation options
modelsim.compile.vcom.more_options	More VCOM compilation options

VCS Simulator Compilation Options

Table 24: VCS Simulator Compilation Options

Option	Description
Verilog options	Browse to set the Verilog include path and to define macro
Generics/Parameters options	Specify or browse to set the generic/parameter values
vcs.compile.tcl.pre	TCL file containing set of commands that should be invoked before launch of compilation
vcs.compile.load_glbl	Load GLBL module
vcs.compile.vhdlan.more_options	More VHDLAN compilation options

Table 24: VCS Simulator Compilation Options (cont'd)

Option	Description
vcs.compile.vlogan.more_options	Extra VLOGAN compilation options
vcs.compile.syscan.more_options	More SYSCAN compilation options
vcs.compile.g++.more_options	More G++ compilation options
vcs.compile.gcc.more_options	More GCC compilation options

Xcelium Simulator Compilation Options

Table 25: Xcelium Compilation Options

Options	Description
Verilog Options	Browse to set Verilog include path and to define macro
Generics/Parameters options	Specify or browse to set the generic/parameter value
xcelium.compile.tcl.pre	TCL file containing set of commands that should be invoked before the launch of a compilation
xcelium.compile.v93	Enable VHDL-93 features
xcelium.compile.relax	Enable relaxed VHDL interpretation
xcelium.compile.load_glbl	Load GLBL module
xcelium.compile.xmvhdl.more_options	More XMVHDL compilation options
xcelium.compile.xmilog.more_options	More XMVLOG compilation options
xcelium.compile.xmsc.more_option	More XMSC compilation option
xcelium.compile.g++.more_option	More G++ compilation option
xcelium.compile.gcc.more_option	More GCC compilation option

Elaboration Options

The Elaboration tab defines and manages elaboration directives, which are stored as properties on the simulation fileset and used by the xelab utility for elaborating and generating a simulation snapshot. Select a property in the table to display a description of the property and edit the value.

Vivado Simulator Elaboration Options

Table 26: Vivado Simulator Elaboration Options

Option	Description
xsim.elaborate.snapshot	Specifies the simulation snapshot name
xsim.elaborate.debug_level	Choose simulation debug visibility level. By default it is "typical"

Table 26: Vivado Simulator Elaboration Options (cont'd)

Option	Description
xsim.elaborate.relax	Relax strict HDL Language checking rules
xsim.elaborate.mt_level	Specify number of sub-compilation jobs to run in parallel
xsim.elaborate.load_glbl	Load GLBL module
xsim.elaborate.rangecheck	Enables run time value range check for VHDL
xsim.elaborate.sdf_delay	Specifies sdf timing delay type to be read for use in timing simulation
xsim.elaborate.xelab.more_option	More XELAB elaboration options
xsim.elaborate.xsc.more_option	More options for XSC during elaboration
xsim.elaborate.coverage.name	Specify coverage database name
xsim.elaborate.coverage.dir	Specify coverage database directory name
xsim.elaborate.coverage.type	Specify coverage type(s) (line branch condition or all)
xsim.elaborate.coverage.library	Track std/unisims/retarget libraries
xsim.elaborate.coverage.celldefine	Track modules with celldefine attributes
xsim.elaborate.link.sysc	Specify SystemC library to bind
xsim.elaborate.link.c	Specify C/C++ library to bind

Questa Advanced Simulator Elaboration Options

Table 27: Questa Advanced Simulator Elaboration Options

Option	Description
questasim.elaborate.acc	Enables access to simulation objects that might be optimized by default (default: npr)
questasim.elaborate.vopt.more_options	More VOPT elaboration options
questasim.elaborate.sccom.more_options	More options for sccom during elaboration
questasim.elaborate.link.sysc	Specify SystemC library to bind
questasim.elaborate.link.c	Specify C/C++ library to bind

ModelSim Simulator Elaboration Options

Table 28: ModelSim Elaboration Options

Option	Description
modelsim.elaborate.acc	Enables access to simulation objects that might be optimized by default
modelsim.elaborate.vopt.more_options	More VOPT elaboration options

VCS Simulator Elaboration Options

Table 29: VCS Elaboration Options

Option	Description
vcs.elaborate.debug_pp	Enable post-process debug access
vcs.elaborate.vcs.more_options	More VCS elaboration options
vcs.elaborate.link.sysc	Specify SystemC library to bind
vcs.elaborate.link.c	Specify C/C++ library to bind

Xcelium Simulator Elaboration Options

Table 30: Xcelium Elaboration Options

Option	Description
xcelium.elaborate.update	Checks if unit is up-to-date before writing
xcelium.elaborate.xmelab.more_options	More <code>xmelab</code> elaboration options
xcelium.elaborate.link.sysc	Specify SystemC library to bind
xcelium.elaborate.link.c	Specify C/C++ library to bind

Simulation Options

The Simulation tab defines and manages simulation directives, which are stored as properties on the simulation fileset and used by the `xsim` application for simulating the current project. Select a property in the table to display a description of the property and edit the value.

Vivado Simulator Simulation Options

Table 31: Vivado Simulator Simulation Options

Option	Description
<code>xsim.simulate.runtime</code>	Specifies simulation run time for the Vivado simulator. Enter blank to load just the simulation snapshot and wait for user input.
<code>xsim.simulate.tcl.post</code>	Tcl file containing set of commands that you want to invoke at end of simulation.
<code>xsim.simulate.log_all_signals</code>	Logs all object signals
<code>xsim.simulate.wdb</code>	Specifies simulation waveform database file
<code>xsim.simulate.saif</code>	Specifies SAIF file name
<code>xsim.simulate.saif_scope</code>	Specifies design hierarchy instance name for which power estimation is desired.
<code>xsim.simulate.saif_all_signals</code>	Logs all object signals for the design under test for SAIF file generation
<code>xsim.simulate.xsim.more_option</code>	More Vivado simulator simulation options

Table 31: Vivado Simulator Simulation Options (cont'd)

Option	Description
<code>xsim.simulate.custom_tcl</code>	Specify the name of a custom tcl file which will be the source during simulation in place of a regular Tcl file generated by Vivado
<code>xsim.simulate.add_positional</code>	Add positional parameter to XSIM for passing command line argument
<code>xsim.simulate.no_quit</code>	Do not quit simulation

Questa Advanced Simulator Simulation Options

Table 32: Questa Advanced Simulator Simulation Options

Option	Description
<code>questasim.simulate.runtime</code>	Specify simulation run time
<code>questasim.simulate.tcl.post</code>	TCL file containing set of commands that you want to invoke at end of simulation.
<code>questasim.simulate.log_all_signals</code>	Log all signals
<code>questasim.simulate.custom_do</code>	Specify the name of custom <code>do</code> file
<code>questasim.simulate.custom_udo</code>	Specify the name of custom user <code>do</code> file
<code>questa.simulate.ieee_warning</code>	Suppresses IEEE warnings
<code>questasim.simulate.sdf_delay</code>	Specify the delay type for <code>sdf</code> annotation
<code>questasim.simulate.saif</code>	Specify SAIF file
<code>questasim.simulate.saif_scope</code>	Specify design hierarchy instance name for which power estimation is desired
<code>questasim.simulate.vsim.more_option</code>	More VSIM simulation options
<code>questa.simulate.custom_wave_do</code>	Name of the custom <code>wave.do</code> file which is used in place of a regular Vivado generated <code>wave.do</code> file

ModelSim Simulator Simulation Options

Table 33: ModelSim Simulation Options

Option	Description
<code>modelsim.simulate.runtime</code>	Specify simulation run time
<code>modelsim.simulate.tcl.post</code>	TCL file containing set of commands that you want to invoke at end of simulation.
<code>modelsim.simulate.log_all_signals</code>	Log all signals
<code>modelsim.simulate.custom_do</code>	Specify the name of custom <code>do</code> file
<code>modelsim.simulate.custom_udo</code>	Specify the name of custom user <code>do</code> file
<code>modelsim.simulate.sdf_delay</code>	Specify the delay type for <code>sdf</code> annotation
<code>modelsim.simulate.ieee_warning</code>	Suppresses IEEE warnings
<code>modelsim.simulate.saif</code>	Specify SAIF file
<code>modelsim.simulate.saif_scope</code>	Specify design hierarchy instance name for which power estimation is desired

Table 33: ModelSim Simulation Options (cont'd)

Option	Description
modelsim.simulate.vsim.more_option	More VSIM simulation options
modelsim.simulate.custom_wave_do	Name of the custom wave.do file which is used in place of a regular Vivado generated wave.do file

VCS Simulator Simulation Options

Table 34: VCS Simulation Options

Option	Description
vcs.simulate.runtime	Specify simulation run time
vcs.simulate.tcl.post	TCL file containing set of commands that you want to invoke at end of simulation.
vcs.simulate.log_all_signals	Log all signals
vcs.simulate.saif	SAIF file name
vcs.simulate.saif_scope	Specify design hierarchy instance name for which power estimation is desired
vcs.simulate.vcs.more_option	More VCS simulation options

Xcelium Simulator Simulation Options

Table 35: Xcelium Simulator Simulation Options

Option	Description
xcelium.simulate.tcl.post	TCL file containing set of commands that you want to invoke at end of simulation
xcelium.simulate.runtime	Specify simulation run time
xcelium.simulate.log_all_signals	Log all signals
xcelium.simulate.update	Check if unit is up-to-date before writing
xcelium.simulate.ieee_warnings	Suppress IEEE warnings
xcelium.simulate.saif_scope	SAIF file name
xcelium.simulate.saif	Specify design hierarchy instance name for which power estimation is desired
xcelium.simulate.xmsim.more_options	More XMSIM simulation options

Netlist Options

The Netlist tab provides access to netlist configuration options related to SDF annotation of the Verilog netlist and the process corner captured by SDF delays. These options are stored as properties on the simulation fileset and are used while writing the netlist for simulation.

Vivado Simulator Netlist Options

Table 36: Vivado Simulator Netlist Options

Option	Description
<code>-sdf_anno</code>	A check box is available to select the <code>-sdf_anno</code> option. This option is enabled by default
<code>-process_corner</code>	You can specify the <code>-process_corner</code> as fast or slow

Note: The Netlist Options of all the third-party simulators (Questa Advanced Simulator, ModelSim Simulator, VCS and Xcelium Simulators) are similar to the options of Vivado simulator Netlist Options.

Advanced Simulation Options

Advanced tab contains two options.

- Enable incremental compilation** option: This option enables the incremental compilation and preserves the simulation files during successive run.
- Include all design sources for simulation** option: By default, this option is enabled. Selecting this option ensures that all the files from design sources along with the files from the current simulation set will be used for simulation. Even if you change the design sources, the same changes will be updated when you launch behavioral simulation.



IMPORTANT! *This is an advanced user feature. Unchecking the box could produce unexpected results. The **Include all design sources for simulation** check box is selected by default. As long as the check box is selected, the simulation set includes Out-of-Context (OOC) IP, IP Integrator files, and DCP.*

Unchecking the box gives you the flexibility to include only the files you want to simulate, but, as stated above, you might experience unexpected results.

Note: The Advanced Simulation Options are the same for all simulators.

SystemVerilog Support in Vivado Simulator

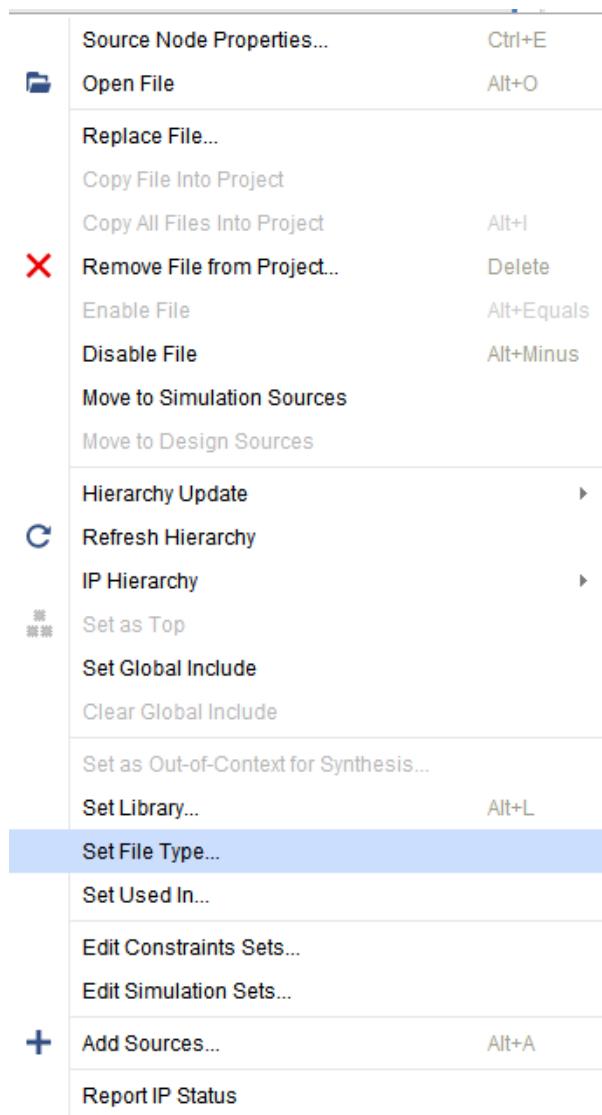
The Vivado simulator supports the subset of SystemVerilog. The synthesizable set of SystemVerilog is listed in the following table. The supported test bench features are listed in [Table 38: Supported Dynamic Type Constructs](#).

Targeting SystemVerilog for a Specific File

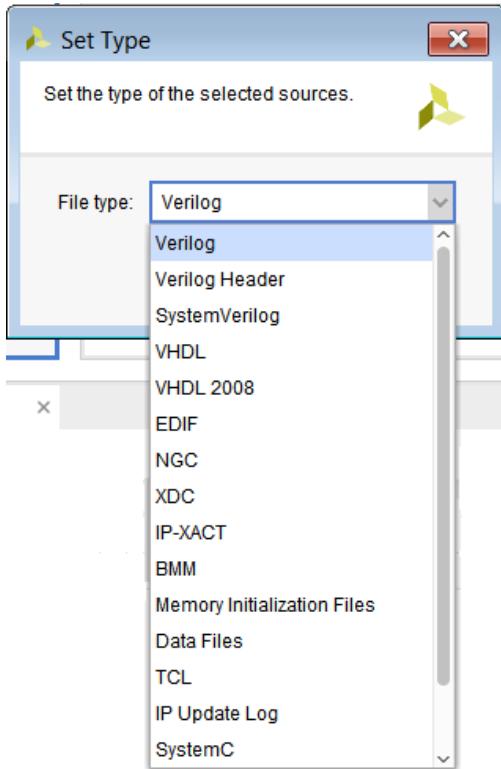
By default, the Vivado IDE compiles `.v` files with the Verilog 2001 syntax and `.sv` files with the SystemVerilog syntax.

To target SystemVerilog for a specific `.v` file in the Vivado IDE:

1. Right-click the file and select **Set file type** as shown in the figure below.



2. In the Set Type dialog box, shown in the figure below, change the file type from Verilog to **SystemVerilog** and click **OK**.



Alternatively, you can use the following command in the Tcl Console:

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

Running SystemVerilog in Standalone or prj Mode

Standalone Mode

A new `-sv` flag has been introduced to `xvlog`, so if you want to read any SystemVerilog file, you can use following command:

```
xvlog -sv <Design file list>
xvlog -sv -work <LibraryName> <Design File List>
xvlog -sv -f <FileName> [Where FileName contain path of test cases]
```

prj Mode

If you want to run the Vivado simulator in the `prj`-based flow, use `sv` as the file type, as you would `verilog` or `vhdl`.

```
xvlog -prj <prj File>
xelab -prj <prj File> <topModuleName> <other options>
```

Where the entry in `prj` file appears as follows:

```
verilog      library1 <FileName>
sv          library1 <FileName> [File parsed in SystemVerilog mode]
vhdl      library2 <FileName>
sv          library3 <FileName> [File parsed in SystemVerilog mode]
```

Table 37: Synthesizable Set of SystemVerilog 1800-2012

Primary construct	Secondary construct	LRM section	Status
Data type		6	
	Singular and aggregate types	6.4	Supported
	Nets and variables	6.5	Supported
	Variable declarations	6.8	Supported
	Vector declarations	6.9	Supported
	2-state (two-value) and 4-state (four-value) data types	6.11.2	Supported
	Signed and unsigned integer types	6.11.3	Supported
	Real, shortreal and realtime data types	6.12	Supported
	User-defined types	6.18	Supported
	Enumerations	6.19	Supported
	Defining new data types as enumerated types	6.19.1	Supported
	Enumerated type ranges	6.19.2	Supported
	Type checking	6.19.3	Supported
	Enumerated types in numerical expressions	6.19.4	Supported
	Enumerated type methods	6.19.5	Supported
	Type parameters	6.20.3	Supported
	Const constants	6.20.6	Supported
	Type operator	6.23	Supported
	Cast operator	6.24.1	Supported
	<code>\$cast</code> dynamic casting	6.24.2	Supported
	Bitstream casting	6.24.3	Supported
Aggregate data types		7	
	Structures	7.2	Supported
	Packed/Unpacked structures	7.2.1	Supported
	Assigning to structures	7.2.2	Supported
	Unions	7.3	Supported
	Packed/Unpacked unions	7.3.1	Supported
	Tagged unions	7.3.2	Not Supported
	Packed arrays	7.4.1	Supported
	Unpacked arrays	7.4.2	Supported
	Operations on arrays	7.4.3	Supported

Table 37: Synthesizable Set of SystemVerilog 1800-2012 (cont'd)

Primary construct	Secondary construct	LRM section	Status
	Multidimensional arrays	7.4.5	Supported
	Indexing and slicing of arrays	7.4.6	Supported
	Array assignments	7.6	Supported
	Arrays as arguments to subroutines	7.7	Supported
	Array querying functions	7.11	Supported
	Array manipulation methods	7.12	Supported
Processes		9	
	Combinational logic <code>always_comb</code> procedure	9.2.2	Supported
	Implicit <code>always_comb</code> sensitivities	9.2.2.1	Supported
	Latched logic <code>always_latch</code> procedure	9.2.2.3	Supported
	Sequential logic <code>always_ff</code> procedure	9.2.2.4	Supported
	Sequential blocks	9.3.1	Supported
	Parallel blocks	9.3.2	Supported
	Procedural timing controls	9.4	Supported
	Conditional event controls	9.4.2.3	Supported
	Sequence events	9.4.2.4	Not Supported
Assignment statement		10	
	The continuous assignment statement	10.3.2	Supported
	Variable declaration assignment (variable initialization)	10.5	Supported
	Assignment-like contexts	10.8	Supported
	Array assignment patterns	10.9.1	Supported
	Structure assignment patterns	10.9.2	Supported
	Unpacked array concatenation	10.10	Supported
	Net aliasing	10.11	Not Supported
Operators and expressions		11	
	Constant expressions	11.2.1	Supported
	Aggregate expressions	11.2.2	Supported
	Operators with real operands	11.3.1	Supported
	Operations on logic (4-state) and bit (2-state) types	11.3.4	Supported
	Assignment within an expression	11.3.6	Supported
	Assignment operators	11.4.1	Supported
	Increment and decrement operators	11.4.2	Supported
	Arithmetic expressions with unsigned and signed types	11.4.3.1	Supported
	Wildcard equality operators	11.4.6	Supported
	Concatenation operators	11.4.12	Supported
	Set membership operator	11.4.13	Supported

Table 37: Synthesizable Set of SystemVerilog 1800-2012 (cont'd)

Primary construct	Secondary construct	LRM section	Status
	Concatenation of <code>stream_expressions</code>	11.4.14.1	Supported
	Re-ordering of the generic stream	11.4.14.2	Supported
	Streaming concatenation as an assignment target (unpack)	11.4.14.3	Supported
	Streaming dynamically sized data	11.4.14.4	Supported
Procedural programming statement		12	
	<code>unique-if</code> , <code>unique0-if</code> and <code>priority-if</code>	12.4.2	Supported
	Violation reports generated by <code>unique-if</code> , <code>unique0-if</code> , and <code>priority-if</code> constructs	12.4.2.1	Supported
	If statement violation reports and multiple processes	12.4.2.2	Supported
	<code>unique-case</code> , <code>unique0-case</code> , and <code>priority-case</code>	12.5.3	Supported
	Violation reports generated by <code>unique-case</code> , <code>unique0-case</code> , and <code>priority-case</code> construct	12.5.3.1	Supported
	Case statement violation reports and multiple processes	12.5.3.2	Supported
	Set membership case statement	12.5.4	Supported
	Pattern matching conditional statements	12.6	Not Supported
	Loop statements	12.7	Supported
	Jump statement	12.8	Supported
Tasks		13.3	
	Static and Automatic task	13.3.1	Supported
	Tasks memory usage and concurrent activation	13.3.2	Supported
Function		13.4	
	Return values and void functions	13.4.1	Supported
	Static and Automatic function	13.4.2	Supported
	Constant function	13.4.3	Supported
	Background process spawned by function call	13.4.4	Supported
Subroutine calls and argument passing		13.5	
	Pass by value	13.5.1	Supported
	Pass by reference	13.5.2	Supported
	Default argument value	13.5.3	Supported
	Argument binding by name	13.5.4	Supported
	Optional argument list	13.5.5	Supported
	Import and Export function	13.6	Supported

Table 37: Synthesizable Set of SystemVerilog 1800-2012 (cont'd)

Primary construct	Secondary construct	LRM section	Status
	Task and function name	13.7	Supported
Utility system tasks and system functions (only synthesizable set)		20	Supported
I/O system tasks and system functions (only synthesizable set)		21	Supported
Compiler directives		22	Supported
Modules and hierarchy		23	
	Default port values	23.2.2.4	Supported
	Top-level modules and \$root	23.3.1	Supported
	Module instantiation syntax	23.3.2	Supported
	Nested modules	23.4	Supported
	Extern modules	23.5	Supported
	Hierarchical names	23.6	Supported
	Member selects and hierarchical names	23.7	Supported
	Upwards name referencing	23.8	Supported
	Overriding module parameters	23.10	Supported
	Binding auxiliary code to scopes or instances	23.11	Not Supported
Interfaces		25	
	Interface syntax	25.3	Supported
	Nested interface	25.3	Supported
	Ports in interfaces	25.4	Supported
	Example of named port bundle	25.5.1	Supported
	Example of connecting port bundle	25.5.2	Supported
	Example of connecting port bundle to generic interface	25.5.3	Supported
	Modport expressions	25.5.4	Supported
	Clocking blocks and modports	25.5.5	Supported
	Interfaces and specify blocks	25.6	Supported
	Example of using tasks in interface	25.7.1	Supported
	Example of using tasks in modports	25.7.2	Supported
	Example of exporting tasks and functions	25.7.3	Supported
	Example of multiple task exports	25.7.4	Supported
	Parameterized interfaces	25.8	Supported
	Virtual interfaces	25.9	Supported
Packages		26	
	Package declarations	26.2	Supported
	Referencing data in packages	26.3	Supported
	Using packages in module headers	26.4	Supported

Table 37: Synthesizable Set of SystemVerilog 1800-2012 (cont'd)

Primary construct	Secondary construct	LRM section	Status
	Exporting imported names from packages	26.6	Supported
	The std built-in package	26.7	Supported
Generate constructs		27	Supported

Testbench Feature

In Vivado simulator, support for some of the commonly used testbench features have been added, as shown in the table below.

Table 38: Supported Dynamic Type Constructs

Primary Construct	Secondary Construct	LRM Section	Status
String data type		6.16	Supported
	String operators (table 6-9 of IEEE 1800-2012)	6.16	Supported
	Len()	6.16.1	Supported
	Putc()	6.16.2	Supported
	Getc()	6.16.3	Supported
	Toupper()	6.16.4	Supported
	Tolower()	6.16.5	Supported
	Compare	6.16.6	Supported
	Icompare()	6.16.7	Supported
	Substr()	6.16.8	Supported
	Atoi(), atohex(), atooct(), atobin()	6.16.9	Supported
	Atoreal()	6.16.10	Supported
	Itoa()	6.16.11	Supported
	Hextoa()	6.16.12	Supported
	Octtoa()	6.16.13	Supported
	Bintoa()	6.16.14	Supported
	Realtoa()	6.16.15	Supported
Dynamic Array		7.5	Supported
	Dynamic array new	7.5.1	Supported
	Size	7.5.2	Supported
	Delete	7.5.3	Supported
Associative Array		7.8	Supported
	Wildcard index	7.8.1	Supported
	String index	7.8.2	Supported
	Class index	7.8.3	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Integral index	7.8.4	Supported
	Other user-defined types	7.8.5	Supported
	Accessing invalid index	7.8.6	Supported
	Associative array methods	7.9	Supported
	Num() and Size()	7.9.1	Supported
	Delete()	7.9.2	Supported
	Exists()	7.9.3	Supported
	First()	7.9.4	Supported
	Last()	7.9.5	Supported
	Next()	7.9.6	Supported
	Prev()	7.9.7	Supported
	Arguments to traversal Method	7.9.8	Supported
	Associative array assignment	7.9.9	Supported
	Associative array arguments	7.9.10	Supported
	Associative Array literals	7.9.11	Supported
Queue		7.10	Supported
	Queue operators	7.10.1	Supported
	Queue methods	7.10.2	Supported
	Size()	7.10.2.1	Supported
	Insert()	7.10.2.2	Supported
	Delete()	7.10.2.3	Supported
	Pop_front()	7.10.2.4	Supported
	Pop_back()	7.10.2.5	Supported
	Push_front()	7.10.2.6	Supported
	Push_back()	7.10.2.7	Supported
	Persistence of references to elements of a queue	7.10.3	Supported
	Updating a queue using assignment and unpacked array concatenation	7.10.4	Supported
	Bounded queues	7.10.5	Supported
Class		8	Supported
	Class General	8.1	Supported
	Overviews	8.2	Supported
	Syntax	8.3	Supported
	Objects(Class instance)	8.4	Supported
	Object properties and object parameter data	8.5	Supported
	Object methods	8.6	Supported
	Constructors	8.7	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Static class properties	8.8	Supported
	Static methods	8.9	Supported
	This	8.10	Supported
	Assignment, renaming, and copying	8.11	Supported
	Inheritance and subclasses	8.12	Supported
	Overridden members	8.13	Supported
	Super	8.14	Supported
	Casting	8.15	Supported
	Chaining constructors	8.16	Supported
	Data hiding and encapsulation	8.17	Supported
	Constant class properties	8.18	Supported
	Virtual methods	8.19	Supported
	Abstract classes and pure virtual methods	8.20	Supported
	Polymorphism: dynamic method lookup	8.21	Supported
	Class scope resolution operator ::	8.22	Supported
	Out-of-block declarations	8.23	Supported
	Parameterized classes	8.24	Supported
	Class resolution operator for parameterized classes	8.24.1	Supported
	Typedef class	8.25	Supported
	Classes and structures	8.26	Supported
	Memory management	8.27	Supported
Processes		9	Supported
	Parallel Process - Fork Join_Any and ForkJoin_None	9.3	Supported
	Wait fork	9.6.1	Supported
	Disable Fork	9.6.3	Supported
	Fine grain process control	9.7	Supported
Clocking Block		14	Supported
	General	14.1	Supported
	Overview	14.2	Supported
	Clocking block declaration	14.3	Supported
	Input and output Skew	14.4	Supported
	Hierarchical Expressions	14.5	Not Supported
	Signals in multiple clocking block	14.6	Supported
	Clocking block scope and lifetime	14.7	Supported
	Multiple clocking block example	14.8	Supported
	Interface and clocking block	14.9	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Clocking block event	14.10	Supported
	Cycle Delay	14.11	Supported
	Default clocking	14.12	Supported
	Input Sampling	14.13	Supported
	Global clocking	14.14	Not Supported
	Synchronous events	14.15	Supported
	Synchronous drives	14.16	Supported
	Drives and nonblocking assignments	14.16.1	Supported
	Driving clocking output signals	14.16.2	Supported
Semaphore		15.3	Supported
	Semaphore method new()	15.3.1	Supported
	Semaphore method put()	15.3.2	Supported
	Semaphore method get()	15.3.3	Supported
	Semaphore method try_get()	15.3.4	Supported
Mailbox		15.4	Supported
	Mailbox method new()	15.4.1	Supported
	Mailbox method num()	15.4.2	Supported
	Mailbox method put()	15.4.3	Supported
	Mailbox method try_put()	15.4.4	Supported
	Mailbox method get()	15.4.5	Supported
	Mailbox method try_get()	15.4.6	Supported
	Mailbox method peek()	15.4.7	Supported
	Mailbox method try_peek()	15.4.8	Supported
	Parameterized mailbox	15.4.9	Supported
Named Event		15.5	Supported
	Triggering an event	15.5.1	Supported
	Waiting on event	15.5.2	Supported
	Persistent trigger	15.5.3	Not Supported
	Event Sequence	15.5.4	Not Supported
	Operation on named event variable	15.5.5	Supported
	Merging Events	15.5.5.1	Supported
	Reclaiming event	15.5.5.2	Supported
	Event comparison	15.5.5.3	Supported
Assertion		16	Supported
	General	16.1	Supported
	Overview	16.2	Supported
	Assert	16.2	Supported
	Assume	16.2	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Cover	16.2	Not Supported
	Restrict	16.2	Not Supported
	Immediate assertion	16.3	Supported
	Deferred assertion	16.4	Not Supported
	Concurrent assertion overview	16.5	Supported
	Sampling	16.5.1	Supported
	Assertion clock	16.5.2	Supported
	Boolean expression	16.6	Supported
	Sequence	16.7	Supported
	Declaring sequence	16.8	Supported
	Typed formal argument in sequence declarations	16.8.1	Supported
	Local variable formal arguments in sequence declarations	16.8.2	Supported
	Sequence operations	16.9	Supported
	Operator precedence	16.9.1	Supported
	Repetition in sequences	16.9.2	Supported
	Sampled value functions	16.9.3	Supported
	Global clocking past and future sampled value functions	16.9.4	Not Supported
	AND operation	16.9.5	Supported
	Intersection (AND with length restriction)	16.9.6	Supported
	OR operation	16.9.7	Supported
	First_match operation	16.9.8	Supported
	Conditions over sequences	16.9.9	Supported
	Sequence contained within another sequence	16.9.10	Supported
	Composing sequences from simpler subsequences	16.9.11	Supported
	Local variables	16.10	Supported
	Calling subroutines on match of a sequence	16.11	Supported
	Declaring properties	16.12	Supported
	Sequence property	16.12.1	Supported
	Negation property	16.12.2	Supported
	Disjunction property	16.12.3	Supported
	Conjunction property	16.12.4	Supported
	If-else property	16.12.5	Supported
	Implication	16.12.6	Supported
	Implies and iff properties	16.12.7	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Property instantiation	16.12.8	Supported
	Followed-by property	16.12.9	Not Supported
	Next time property	16.12.10	Not Supported
	Always property	16.12.11	Not Supported
	Until property	16.12.12	Not Supported
	Eventually property	16.12.13	Not Supported
	Abort properties	16.12.14	Not Supported
	Weak and strong operators	16.12.15	Not Supported
	Case	16.12.16	Not Supported
	Recursive properties	16.12.17	Not Supported
	Typed formal arguments in property declarations	16.12.18	Supported
	Local variable formal arguments in property declarations	16.12.19	Supported
	Property examples	16.12.20	Supported
	Finite-length versus infinite-length behavior	16.12.21	Supported
	Nondegeneracy	16.12.22	Supported
	Multiclock support	16.13	Not Supported
	Concurrent assertions	16.14	Supported
	Assert statement	16.14.1	Supported
	Assume statement	16.14.2	Supported
	Cover statement	16.14.3	Not Supported
	Restrict statement	16.14.4	Not Supported
	Using concurrent assertion statements outside procedural code	16.14.5	Supported
	Embedding concurrent assertions in procedural code	16.14.6	Not Supported
	Inferred value functions	16.14.7	Not Supported
	Nonvacuous evaluations	16.14.8	Not Supported
	Disable iff resolution	16.15	Supported
	Clock resolution	16.16	Supported
	Semantic leading clocks for multiclocked sequence and properties	16.16.1	Supported
	Expect statement	16.17	Not Supported
	Clocking blocks and concurrent assertions	16.18	Supported
Random Constraint		18	Supported
	Concepts and Usage	18.3	Supported
	Random Variable	18.4	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Rand modifier	18.4.1	Supported
	Randc modifier	18.4.2	Supported
	Constraint block	18.5	Supported
	External constraint block	18.5.1	Supported
	Constraint inheritance	18.5.2	Supported
	Set membership	18.5.3	Supported
	Distribution	18.5.4	Supported
	Implication	18.5.6	Supported
	If-else constraint	18.5.7	Supported
	Iterative constraint	18.5.8	Supported
	foreach iterative constraint	18.5.8.1	Supported
	Array reduction iterative constraint	18.5.8.2	Supported
	Global constraint	18.5.9	Supported
	Variable Ordering	18.5.10	Supported
	Static constraint block	18.5.11	Supported
	Function in constraint	18.5.12	Supported
	Constraint Guards	18.5.13	Supported
	Soft constraint	18.5.14	Supported
	Method Randomize	18.6.1	Supported
	Pre_randomize and post_randomize	18.6.2	Supported
	Behavior of randomization method	18.6.3	Supported
	In-line constraints	18.7	Supported
	Local scope resolution	18.7.1	Supported
	Disabling random variable with rand_mode	18.8	Supported
	Controlling constraints with constraint_mode	18.9	Supported
	Dynamic constraint modification	18.10	Supported
	In-line random variable control	18.11	Supported
	In-line constraint checker	18.11.1	Supported
	Randomize of a scope variable std::randomize	18.12	Supported
	Adding constraint to scope variables std::randomize with	18.12.1	Supported
	Random number system functions and method	18.13	Supported
	\$urandom	18.13.1	Supported
	\$urandom_range	18.13.2	Supported
	srandom	18.13.3	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Get_randstate	18.13.4	Supported
	Set_randstate	18.13.5	Supported
	Random stability	18.14	Supported
	Manually seeding randomization	18.15	Supported
	Randcase	18.16	Supported
	Randsequence	18.17	Not Supported
Programs		24	Supported
	The Program construct	24.3	Supported
	Scheduling semantic of code in program construct	24.3.1	Supported
	Program port connection	24.3.2	Supported
	Eliminating test bench race	24.4	Supported
	Blocking task in cycle/event mode	24.5	Supported
	Anonymous Programs	24.6	Not Supported
	Program control task	24.7	Supported
Functional Coverage		19	Supported
	General	19.1	Supported
	Overview	19.2	Supported
	Defining coverage model: covergroup	19.3	Supported
	Using covergroup in classes	19.4	Supported
	Defining coverage points	19.5	Supported
	Specifying bins for values	19.5.1	Supported
	Coverpoint bin with covergroup expressions	19.5.1.1	Supported
	Coverpoint bin set covergroup expressions	19.5.1.2	Not supported
	Specifying bins for transitions	19.5.2	Supported
	Automatic bin creation for coverage points	19.5.3	Supported
	Wildcard specification of coverage point bins	19.5.4	Supported
	Excluding coverage point values or transitions	19.5.5	Supported
	Specifying Illegal coverage point values or transitions	19.5.6	Supported
	Value resolution	19.5.7	Supported
	Defining cross coverage	19.6	Supported
	Defining cross coverage bins	19.6.1	Supported
	Example of user-defined cross coverage and select expressions	19.6.1.1	Supported
	Cross bin with covergroup expressions	19.6.1.2	Supported

Table 38: Supported Dynamic Type Constructs (cont'd)

Primary Construct	Secondary Construct	LRM Section	Status
	Cross bin automatically defined types	19.6.1.3	Supported
	Cross bin set expression	19.6.1.4	Supported
	Excluding cross products	19.6.2	Supported
	Specifying illegal cross products	19.6.3	Supported
	Specifying coverage options	19.7	Supported
	Covergroup type options	19.7.1	Supported
	Predefined coverage methods	19.8	Supported
	Overriding the built-in sample method	19.8.1	Supported
	Predefined coverage system tasks and system functions	19.9	Supported
	Organization of option and type_option members	19.10	Supported

Note: Sensitivity on dynamic types such as Queue, Dynamic Array, Associative Array, and Class are not supported, therefore, block waiting on dynamic type update may not work correctly. For example:

```
module top();
int c[$];
event e1;
initial
begin
    c[0] = 10;
    for(int i = 0; i <= 10; i++)
    begin
        c = {i, c};
        -> e1;
        #5;
    end
end
always@(*) $display($time, " trying to read sensitivity on dynamic type : %d", c[0]);
// this won't work as sensitivity on dynamic type is not supported
always @(e1) $display($time, " coming from event sensitivity : %d",
c[0]); // this we
can do as WA
always_comb if(c.size() > 0) $display($time, " Coming from size
sensitivity : %d",
c[0]); // sensitivity on size works
```

Universal Verification Methodology Support

Vivado® integrated design environment supports universal verification methodology (UVM) in Vivado simulator (XSim). The UVM version 1.2 library is precompiled and is available with Vivado. If you are running your design through Vivado, you need not set anything. But if you are running standalone Vivado simulator, then you need to pass `-L uvm` to `xvlog` and `xelab` command.

By default, Vivado simulator supports UVM version 1.2. If you want to use UVM version 1.1, you need to pass `-uvm_version 1.1` to `xvlog` and `xelab` command. Set the following properties if you are using it through the Vivado integrated design environment:

```
set_property -name {xsim.compile.xvlog.more_options} -value {-uvm_version 1.1} -objects [get_filesets sim_1]
set_property -name {xsim.elaborate.xelab.more_options} -value {-uvm_version 1.1} -objects [get_filesets sim_1]
```

You can also set these properties from Vivado GUI using Compilation and Elaboration tab in simulation settings. For more information, see [Using Simulation Settings](#).

VHDL 2008 Support in Vivado Simulator

Introduction

The Vivado® simulator supports the subset of VHDL 2008 (IEEE 1076-2008). The complete list is given in Supported features of VHDL 2008 (IEEE1076-2008).

Compiling and Simulating

The Vivado simulator executable `xvhdl` is used to convert a VHDL design unit into parser dump (`.vdb`). By default, Vivado simulator uses mixed 93 and 2008 standard (STD) and IEEE packages to freely allow mixing of 93 and 2008 features. If you want to force only the VHDL-93 standard (STD) and IEEE package, pass `-93_mode` to `xvhdl`. To compile a file only with VHDL 2008 mode, you need to pass `-2008` switch to `xvhdl`.

For example, to compile a design called `top.vhdl` in VHDL-2008, following command line can be used:

```
xvhdl -2008 -work mywork top.vhdl
```

The Vivado simulator executable `xelab` is used to elaborate a design and produce an executable image for simulation.

`xelab` can do either of the following:

- Elaborate on parser dumps produced by `xvhdl`
- Directly use vhdl source files.

No switch is needed to elaborate on parser dumps produced by `xvhdl`. You can pass `-vhd12008` to `xelab` to directly use vhdl source files.

Example 1:

```
xelab top -s mysim; xsim mysim -R
```

Example 2:

```
xelab -vhdl2008 top.vhdl top -s mysim; xsim mysim -R
```

Instead of specifying VHDL files in the command line for `xvhdl` and `xelab`, a `.prj` file can be used. If you have two files for a design called `top.vhdl` (2008 mode) and `bot.vhdl` (93 mode), you can create a project file named `example.prj` as follows:

```
vhdl xil_defaultlib bot.vhdl
```

```
vhdl2008 xil_defaultlib top.vhdl
```

In the project file, each line starts with the language type of the file, followed by the library name such as `xil_defaultlib` and one or more file names with a space separator. For VHDL 93, one should use `vhdl` as the language type. For VHDL 2008, use `vhdl2008` instead.

A `.prj` file can be used as shown in the example below:

```
xelab -prj example.prj xil_defaultlib.top -s mysim; xsim mysim -R
```

Alternatively, to mix VHDL 93 and VHDL 2008 design units, compile the files separately with a proper language mode specified to `xvhdl`. Then, elaborate on top(s) of the design. For example, if we have a VHDL 93 module called `bot` in file `bot.vhdl`, and a VHDL-2008 module called `top` in file `top.vhdl`, you can compile them as shown in the example below:

```
xvhdl bot.vhdl
xvhdl -2008 top.vhdl
xelab -debug typical top -s mysim
```

Once the executable is produced by `xelab`, you can run the simulation as usual.

Example 1:

```
xsim mysim -gui
```

Example 2:

```
xsim mysim -R
```

Fixed and Floating Point Packages

Fixed and floating point packages used by the Vivado simulator are the new enhanced IEEE standard packages introduced in VHDL-2008. If you are using the VHDL-93 standard fixed or floating package, that may work in Vivado synthesis, however you must edit your VHDL source file for simulation.

For example, if you are using the following syntax for the fixed package in Vivado synthesis:

```
library ieee;
use ieee.fixed_pkg.all;
```

Change this to the following syntax in VHDL-2008 for use in the Vivado simulator:

```
library ieee_proposed;
use ieee_proposed.fixed_pkg.all;
```

See this [link](#) in the *Vivado Design Suite User Guide: Synthesis* ([UG901](#)) for more information about fixed and floating packages in Vivado Synthesis.

Similar changes will apply for floating package too.

Supported Features

Following are the supported features of VHDL 2008 (IEEE1076-2008):

- Matching relational operators.
- Maximum and minimum operators.
- Shift operators (rol, ror, sll, srl, sla, and sra).
- Unary logical reduction operators.
- Mixing array and scalar logical operators.
- If-else-if and case generate.
- Sequential assignments.
- Case? Statements.
- Select? Statements.
- Unconstrained element types.
- boolean_vector and integer_vector array types.
- Reading output ports.
- Expressions in port maps.

- Process (all) statement.
- Referencing generics in generic lists.
- Generic types in entities.
- Relaxed return rules for function return values.
- Extensions to globally static and locally static expressions.
- Static ranges and integer expressions in range bounds.
- Block comments.
- Context declaration.

Note: For detailed information about features, see Supported VHDL-2008 Features section in *Vivado Design Suite User Guide: Synthesis* ([UG901](#)).

Direct Programming Interface (DPI) in Vivado Simulator

Introduction

You can use the SystemVerilog Direct Programming Interface (DPI) to bind C code to SystemVerilog code. Using DPI, SystemVerilog code can call a C function, which in turn can call back a SystemVerilog task or function. Vivado® simulator supports all the constructs as DPI task/function, as described below.

Compiling C Code

A new compiler executable, `xsc`, is provided to convert C code into an object code file and to link multiple object code files into a shared library (`.a` on Windows and `.so` on Linux). The `xsc` compiler is available in the `<Vivado installation>/bin` directory. You can use `-sv_lib` to pass the shared library containing your C code to the Vivado simulator/elaborator executable. The `xsc` compiler works in the same way as a C compiler, such as `gcc`. The `xsc` compiler:

- Calls the LLVM clang compiler to convert C code into object code
- Calls the GNU linker to create a shared library (`.a` on Windows and `.so` on Linux) from one or more object files corresponding to the C files

The shared library generated by the `xsc` compiler is linked with the Vivado simulator kernel using one or more newly added switches in `xelab`, as described below. The simulation snapshot created by `xelab` thus has ability to connect the compiled C code with compiled SystemVerilog code and effect communication between C and SystemVerilog.

xsc Compiler

The xsc compiler helps you to create a shared library (.a on Windows or .so on Linux) from one or more C files. Use xelab to bind the shared library generated by xsc into the rest of your design. You can create a shared library using the following processes:

- **One-step process:** Pass all C files to xsc without using the -compile or -shared/shared_systemc/static switch.
- **Two-step process:**

```
xsc -compile <C files>
xsc --shared or -shared_systemc or -static <object files>
```

Usage

```
xsc [options] <files...>
```

Switches

You can use a double dash (--) or a single dash (-) for switches.

Table 39: XSC Compiler Switches

Switch	Description
-compile [c]	Generate the object files only from the source C files. The link stage is not run.
-f [-file] <arg>	Read additional options from the specified file.
-h [-help]	Print this help message.
-i [-input_file] <arg>	List of input files (one file per switch) for compiling or linking.
-mt <arg> (=auto)	Specifies the number of sub-compilation jobs that can be run in parallel. Choices are: <ul style="list-style-type: none"> • auto: automatic • n: where n is an integer greater than 1 • off: turn off multi-threading Default: auto
-o [-output] <arg>	Specify the name of output shared library. Works with --shared, --shared_systemc, --exe options only. Default for shared library is <current_directory>/xsim.dir/work/xsc/dpi.so.
-work <arg>	Specify the work directory in which to place the outputs (object files). Default: <current_directory>/xsim.dir/xsc
-v [-verbose] <arg>	Specify verbosity level for printing messages. Allowed values are: 0, 1 Default: 0

Table 39: XSC Compiler Switches (cont'd)

Switch	Description
<code>-gcc_compile_options <arg></code>	Supply an additional option to the compiler. You can use multiple <code>-gcc_compile_options</code> switches.
<code>-gcc_link_options <arg></code>	Supply an additional option to the linker. You can use multiple <code>-gcc_link_options</code> switches.
<code>-shared</code>	Run only the linking stage to generate the shared library (<code>.so</code>) from the object files.
<code>-gcc_version</code>	Print version of the C compiler used internally.
<code>-gcc_path</code>	Print path of the C compiler used internally.
<code>-lib <arg></code>	Specify the logical library directories that will be read. Default is <code><current_directory>/xsim.dir/xs</code> .
<code>-cppversion <arg></code>	Set the CPP version. Currently CPP 11 and 14 are supported. Default is 11.
<code>--shared_systemc</code>	Run only the linking stage to generate the shared library (<code>.dll</code>) for SystemC from the object files.
<code>--static</code>	Run only the linking stage to generate a static library (<code>.a</code>) for SystemC from the object files.
<code>--exe</code>	Create executable for standalone SystemC.
<code>--version</code>	Print version of the Vivado Simulator xsc being used.
<code>--debug</code>	Debug SystemC modules. This option is relevant only when used together with <code>-exe</code> option, otherwise is ignored.
<code>--print_gcc_version</code>	Print version of the C compiler used internally

Examples

```
xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi
xsc -compile function1.c function2.c -work abc
xsc -shared abc/function1.lnx64.o abc/function2.lnx64.o -work abc
```

Note: By default, Linux uses the `LD_LIBRARY_PATH` for searching the DPI libraries. Hence, provide `-dpi_absolute` flag to `xelab` on Linux if library name start with `lib*`.

Note: You can use `-additional_option` to the compiler to pass extra switch.

- **Example:**

```
xsc t1.c --additional_option "-I<path>"
```

- **Example to pass multiple path:**

```
xsc t1.c --additional_option "-I<path>" --additional_option "-I<path>"
```

Binding Compiled C Code to SystemVerilog Using xelab

The DPI-related switches for xelab that bind the compiled C code to SystemVerilog are as follows:

Table 40: DPI-Related Switches for xelab

Switch	Description
<code>-sv_root arg</code>	Root directory relative to which a DPI shared library should be searched. (Default: <code><current_directory>/xsim.dir/xsc</code>)
<code>-sv_lib arg</code>	Name of the DPI shared library without the file extension defining C function imported in SystemVerilog.
<code>-sv_liblist arg</code>	Bootstrap file pointing to DPI shared libraries.
<code>-dpiheader arg</code>	Generate a DPI C header file containing C declaration of imported and exported functions.
<code>-dpi_absolute</code>	Use absolute paths instead of <code>LD_LIBRARY_PATH</code> on Linux for DPI libraries that are formatted as <code>lib<libname>.so</code> .
<code>-dpi_stacksize arg</code>	User defined stack size for DPI tasks.

For more information on `-sv_liblist arg`, refer to the IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, Appendix J.4.1, page 1228.

Data Types Allowed on the Boundary of C and SystemVerilog

The IEEE Standard for SystemVerilog allows only subsets of C and SystemVerilog data types on the C and SystemVerilog boundary. Provided below are (1) details on data types supported in Vivado simulator and (2) descriptions of mapping between the C and SystemVerilog data types.

Supported Data Types

The following table describes data types allowed on the boundary of C and SystemVerilog, along with mapping of data types from SystemVerilog to C and vice versa.

Table 41: Data Types Allowed on the C-SystemVerilog Boundary

SystemVerilog	C	Supported	Comments
<code>byte</code>	<code>char</code>	Yes	None
<code>shortint</code>	<code>short int</code>	Yes	None

Table 41: Data Types Allowed on the C-SystemVerilog Boundary (cont'd)

SystemVerilog	C	Supported	Comments
int	int	Yes	None
longint	long long	Yes	None
real	double	Yes	None
shortreal	float	Yes	None
chandle	void *	Yes	None
string	const char*	Yes	None
bit	unsigned char	Yes	sv_0, sv_1
Available on C side using <code>svdpi.h</code>			
logic, reg	unsigned char	Yes	sv_0, sv_1, sv_z, sv_x:
Array (packed) of bits	svBitVecVal	Yes	Defined in <code>svdpi.h</code>
Array (packed) of logic/reg	svLogicVecVal	Yes	Defined in <code>svdpi.h</code>
enum	Underlying enum type	Yes	None
Packed struct, union	Passed as array	Yes	None
Unpacked arrays of bit, logic	Passed as array	Yes	C can call SystemVerilog
Unpacked struct	Passed as struct	Yes	None
Unpacked union	Passed as struct	No	None
Open arrays	svOpenArrayHandle	Yes	None

To generate a C header file that provides details on how SystemVerilog data types are mapped to C data types: pass the parameter `-dpiheader <file name>` to `xelab`. Additional details on data type mapping are available in the The IEEE Standard for SystemVerilog.

Mapping for User-Defined Types

Enum

You can define an enumerated type (`enum`) for conversion to the equivalent SystemVerilog types, `svLogicVecVal` or `svBitVecVal`, depending on the base type of `enum`. For enumerated arrays, equivalent SystemVerilog arrays are created.

Examples

- **SystemVerilog types::**

```
typedef enum reg [3:0] { a = 0, b = 1, c } eType;
eType e;
eType e1[4:3];
typedef enum bit { a = 0, b = 1} eTypeBit;
eTypeBit e3;
eTypeBit e4[3:1] ;
```

- **C types:**

```
svLogicVecVal e[SV_PACKED_DATA_NELEMS(4)];
svLogicVecVal e1[2][SV_PACKED_DATA_NELEMS(4)];
svBit e3;
svBit e4[3];
```



TIP: The C argument types depend on the base type of the *enum* and the direction.

Packed Struct/Union

When using a packed struct or union type, an equivalent SystemVerilog type, `svLogicVecVal` or `svBitVecVal`, is created on the DPI C side.

Examples

- **SystemVerilog type:**

```
typedef struct packed {
    int i;
    bit b;
    reg [3:0]r;
    logic [2:0] [4:8][9:1] l;
} sType;
sType c_obj;
sType [3:2] c_obj1[5];
```

- **C type:**

```
svLogicVecVal c_obj[SV_PACKED_DATA_NELEMS(172)];
svLogicVecVal c_obj1[5][SV_PACKED_DATA_NELEMS(344)];
```

Arrays, both packed and unpacked, are represented as arrays of `svLogicVecVal` or `svBitVecVal`.

Unpacked Struct

An equivalent unpacked type is created on the C side, in which all the members are converted to the equivalent C representation.

Examples

- **SystemVerilog type:**

```
typedef struct {
    int i;
    bit b;
    reg r[3:0];
    logic [2:0] l[4:8][9:1];
} sType;
```

- **C type:**

```
typedef struct {
    int i;
    svBit b;
    svLogic r[4];
    svLogicVecVal l[5][9][SV_PACKED_DATA_NELEMS(3)];
} sType;
```

Support for svdpi.h Functions

The `svdpi.h` header file is provided in this directory: `<vivado installation>/data/xsim/include`.

The following `svdpi.h` functions are supported:

```
svBit svGetBitselBit(const svBitVecVal* s, int i);
svLogic svGetBitselLogic(const svLogicVecVal* s, int i);
void svPutBitselBit(svBitVecVal* d, int i, svBit s);
void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);
void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int w);
void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int w);
const char* svDpiVersion();
svScope svGetScope();
svScope svSetScope(const svScope scope);
const char* svGetNameFromScope(const svScope);
int svPutUserData(const svScope scope, void*userKey, void* userData);
void* svGetUserData(const svScope scope, void* userKey);
```

Open Arrays in DPI

When declaring an import function in SystemVerilog, you may specify formal argument as open arrays. By specifying certain dimension(s) of formal array arguments as blank (open), it will allow passing actual arguments of different size, which facilitates more general C code. At C side, the open arrays are represented as `SVOpenArrayHandle`. By passing this handle to provided functions, you may query the information of open array, e.g. the size of opened dimension, and access the actual data.

Declaration

Open arrays may only appear in import function/task declaration in SystemVerilog code. By leaving the dimension(s) open, you must specify an open array and the size of blank dimension will be determined with respect to actual argument.

Examples

SystemVerilog function declaration:

```
import "DPI-C" function int myFunction1(input bit[] v);
import "DPI-C" function void myFunction2(input int v1[], input int v2[],
output int v3[]);
```

At C side, the open array(s) may only be accessed by the handle and provided APIs:

```
int myFunction1(const SVOpenArrayHandle v);
void myFunction2(const SVOpenArrayHandle v1, const SVOpenArrayHandle v2,
const SVOpenArrayHandle v3);
```

svdpi.h Support

The following open array related functions are supported in svdpi.h:

```
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svSize(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);
void *svGetArrayPtr(const svOpenArrayHandle);
int svSizeOfArray(const svOpenArrayHandle);
void *svGetArrElemPtr(const svOpenArrayHandle, int idx1, ...);
void *svGetArrElemPtr1(const svOpenArrayHandle, int idx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int idx1, int idx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int idx1, int idx2,
int idx3);
void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int idx1, ...);
void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int idx1);
```

```

void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int idx1, int idx2);
void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int idx1, int idx2, int idx3);
void svPutLogicArrElemVecVal(const svOpenArrayHandle d, const svLogicVecVal*
s, int idx1, ...);
void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int idx1);
void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int idx1, int idx2);
void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int idx1, int idx2, int idx3);
void svGetBitArrElemVecVal(svBitVecVal* d, const svOpenArrayHandle s,
int idx1, ...);
void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int idx1);
void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int idx1, int idx2);
void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int idx1, int idx2, int idx3);
void svGetLogicArrElemVecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int idx1, ...);
void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int
idx1);
void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int idx1, int idx2);
void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int idx1, int idx2, int idx3);
svBit svGetBitArrElem(const svOpenArrayHandle s, int idx1, ...);
svBit svGetBitArrElem1(const svOpenArrayHandle s, int idx1);
svBit svGetBitArrElem2(const svOpenArrayHandle s, int idx1, int idx2);
svBit svGetBitArrElem3(const svOpenArrayHandle s, int idx1, int idx2, int
idx3);
svLogic svGetLogicArrElem(const svOpenArrayHandle s, int idx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int idx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int idx1, int idx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int idx1, int idx2,
int
idx3);
void svPutLogicArrElem(const svOpenArrayHandle d, svLogic value, int
idx1, ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int
idx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int
idx1, int
idx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int idx1,
int idx2, int idx3);
void svPutBitArrElem(const svOpenArrayHandle d, svBit value, int
idx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int idx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int idx1,
int idx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int idx1,
int idx2, int idx3);

```

Usage Example - SystemVerilog code

```
module m();
  import "DPI-C" function void myFunction1(input int v[]);
  int arr[4];
  int dynArr[];
  initial begin
    arr = '{4, 5, 6, 7};
    myFunction1(arr);
    dynArr = new[6];
    dynArr = '{8, 9, 10, 11, 12, 13};
    myFunction1(dynArr);
  end
endmodule
C code:
#include "svdpi.h"
void myFunction1(const svOpenArrayHandle v)
{
  int l1 = svLow(v, 1);
  int h1 = svHigh(v, 1);
  for(int i = l1; i<= h1; i++) {
    printf("\t%d", *((char*)svGetArrElemPtr1(v, i)));
  }
  printf("\n");
}
```

Examples

Note: All the examples below print PASSED for a successful run.

Examples include:

- [Import Example Using -sv_lib, -sv_liblist, and -sv_root](#): A function import example that illustrates different ways to use the `-sv_lib`, `-sv_liblist` and `-sv_root` options.
- [Function with Output](#): A function that has output arguments.
- [Simple Import-Export Flow \(Illustrates xelab -dpiheader Flow\)](#): Shows a simple import>export flow (illustrates `xelab -dpiheader <filename>` flow).

Import Example Using -sv_lib, -sv_liblist, and -sv_root

Code

Assume that there are:

- Two files each containing a C function
- A SystemVerilog file that uses the following functions:
 - `function1.c`
 - `function2.c`
 - `file.sv`

function1.c

```
#include "svdpi.h"
DPI_DLLESPEC
int myFunction1()
{
    return 5;
}
```

function2.c

```
#include <svdpi.h>
DPI_DLLESPEC
int myFunction2()
{
    return 10;
}
```

file.sv

```
module m();
import "DPI-C" pure function int myFunction1();
import "DPI-C" pure function int myFunction2();
integer i, j;
initial
begin
#1;
    i = myFunction1();
    j = myFunction2();
    $display(i, j);
    if( i == 5 && j == 10)
        $display("PASSED");
    else
        $display("FAILED");
end
endmodule
```

Usage

Methods for compiling and linking the C files into the Vivado simulator are described below.

Single-step flow (simplest flow)

```
xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi
```

Flow description:

The xsc compiler compiles and links the C code to create the shared library `xsim.dir/xsc/dpi.so`, and xelab references the shared library through the switch `-sv_lib`.

Two-step flow

```
xsc -compile function1.c function2.c -work abc
xsc -shared/-shared_systemc abc/function1.lnx64.o abc/function2.lnx64.o -
work abc
xelab -svlog file.sv -sv_root abc -sv_lib dpi -R
```

Flow description:

- Compile the two C files into corresponding object code in the work directory `abc`.
- Link these two files together to create the shared library `dpi.so`.
- Make sure that this library is picked up from the work library `abc` via the `-sv_root` switch.



TIP: `-sv_root` specifies where to look for the shared library specified through the switch `-sv_lib`. On Linux, if `-sv_root` is not specified and the DPI library is named with the prefix `lib` and the suffix `.so`, then use the `LD_LIBRARY_PATH` environment variable for the location of shared library.

Two-step flow (same as above with few extra options)

```
xsc -compile function1.c function2.c -work "abc" -v 1
xsc -shared/-shared_systemc "abc/function1.lnx64.o" "abc/function2.lnx64.o" -
-work "abc" -o final -v 1
xelab -svlog file.sv -sv_root "abc" -sv_lib final -R
```

Flow description:

If you want to do your own compilation and linking, you can use the `-verbose` switch to see the path and the options with which the compiler was invoked. You can then tailor those to suit your needs. In the example above, a distinct shared library `final` is created. This example also demonstrates how spaces in file path work.

Function with Output

Code

file.sv

```
/*- - - - */
package pack1;
import "DPI-C" function int myFunction1(input int v, output int o);
import "DPI-C" function void myFunction2 (input int v1, input int v2,
output int o);
endpackage
/*- - - - */
module m();
int i, j;
int o1 ,o2 , o3;
initial
begin
#1;
j = 10;
```

```

o3 =pack1:: myFunction1(j, o1); //should be 10/2 = 5
pack1::myFunction2(j, 2+3, o2); // 5 += 10 + 2+3
$display(o1, o2);
if( o1 == 5 && o2 == 15)
$display( "PASSED" );
else
$display( "FAILED" );
end
endmodule

```

function.c

```

#include "svdpi.h"
DPI_DLLESPEC
int myFunction1(int j, int* o)
{
*o = j /2;
return 0;
}
DPI_DLLESPEC
void myFunction2(int i, int j, int* o)
{
*o = i+j;
return;
}

```

run.ksh

```

xsc function.c
xelab -vlog file.sv -sv -sv_lib dpi -R

```

Simple Import-Export Flow (Illustrates xelab -dpiheader Flow)

In this flow:

1. Run `xelab` with the `-dpiheader` switch to create the header file, `file.h`.
2. Your code in `file.c` then includes the `xelab`-generated header file (`file.h`), which is listed at the end.
3. Compile the code in `file.c` and `test.sv` as before to generate the simulation executable.

file.c

```

#include "file.h"
/* NOTE: This file is generated by xelab -dpiheader <filename> flow */
int cfunc (int a, int b) {
//Call the function exported from SV.
return c_exported_func (a,b);
}

```

test.sv

```

module m();
  export "DPI-C" c_exported_func = function func;
  import "DPI-C" pure function int cfunc (input int a ,b);
  /*This function can be called from both SV or C side. */
  function int func(input int x, y);
  begin
    func = x + y;
  end
  endfunction
  int z;
  initial
  begin
    #5;
    z = cfunc(2, 3);
    if(z == 5)
      $display("PASSED");
    else
      $display("FAILED");
  end
endmodule

```

run.ksh

```

xelab -dpiheader file.h -svlog test.sv
xsc file.c
xelab -svlog test.sv -sv_lib dpi -R
file.h
/****************************************************************************
 *----- */
/* / \ / \ */
/* /---\ \ / */
/* \ \ \ \ */
/* \ \ Copyright (c) 2003-2013 Xilinx, Inc. */
/* / / All Right Reserved. */
/* /---\ / \ */
/* \ \ / \ */
/* \---\ \---\ */
/* NOTE: DO NOT EDIT. AUTOMATICALLY GENERATED FILE. CHANGES WILL BE LOST. */
#ifndef DPI_H
#define DPI_H
#ifdef __cplusplus
#define DPI_LINKER_DECL extern "C"
#else
#define DPI_LINKER_DECL
#endif
#include "svdpi.h"
/* Exported (from SV) function */
DPI_LINKER_DECL DPI_DLLSPEC
int c_exported_func(
  int x, int y);
/* Imported (by SV) function */
DPI_LINKER_DECL DPI_DLLESPEC
int cfunc(
  int a, int b);
#endif

```

DPI Examples Shipped with the Vivado Design Suite

There are two examples shipped with the Vivado Design Suite that can help you understand how to use DPI in Vivado simulator. Locate these in your installation directory, <vivado installation dir>/examples/xsim/systemverilog/dpi. Each includes a README file that can help you get started. The examples include:

- `simple_import`: simple import of pure function
- `simple_export`: simple export of pure function



TIP: When the return value of a function is computed solely on the value of its inputs, it is called a "pure function."

SystemC Support in Vivado IDE

Vivado® Design Suite provides simulation models as a set of files and libraries. Simulation libraries contain the device and IP behavioral and timing models. The compiled libraries can be used by multiple design projects. You must compile these files prior to design simulation through a utility called `compile_simlib` to compile the simulation models for the target simulator. This utility can be invoked from the Vivado IDE or by executing it from the Tcl console.

For SystemC simulation verification, simulation models are provided in C/C++/SystemC. Vivado Design Suite provides two sets of simulation models:

- Protected models
- Unprotected models

Note: With Vivado simulator, there is no need to compile the simulation libraries. Libraries must generally be compiled or recompiled with a new software release to update simulation models and to support a new version of simulator and GCC.

Selecting Simulation Model Type

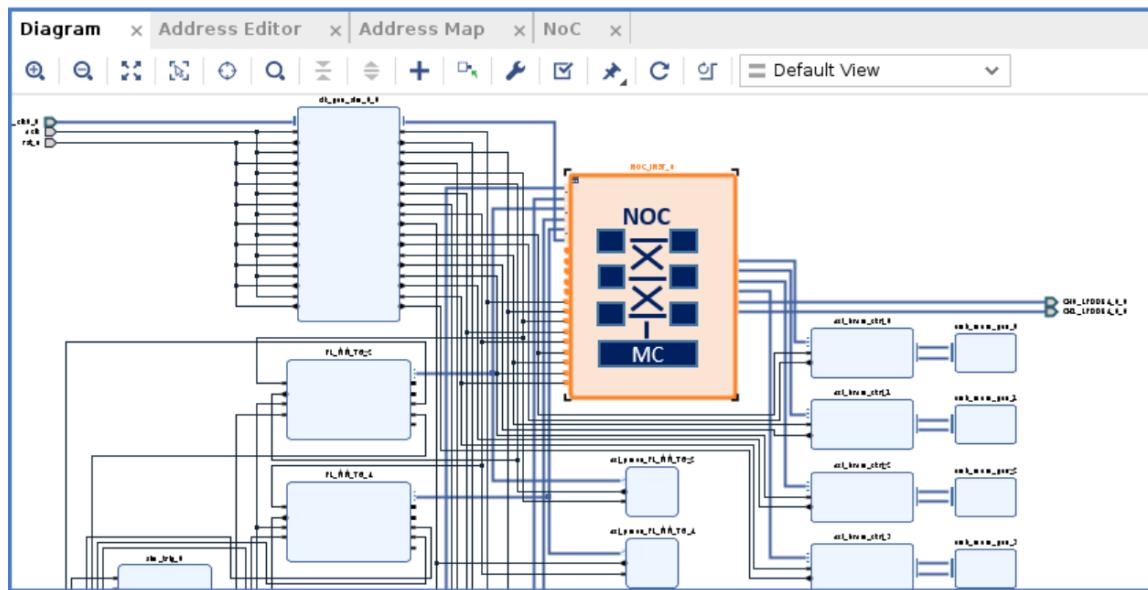
To speed up the simulation run time, Xilinx provides transaction level simulation models (tlm) for certain IPs like Control, Interfaces and Processing System, SmartConnect, NoC, and AIE. You can select one of the supported simulation models for your IP by using either project property (PREFERRED_SIM_MODEL) or an IP property (SELECTED_SIM_MODEL). Following are the supported simulation models properties:

- **ALLOWED_SIM_MODELS:** This is a read only property. It describes different simulation model types such as `rtl`, `tlm`, `tlm_dpi`, `dpi` which are available for a particular IP.
- **SELECTED_SIM_MODEL:** This is an IP level setting which allows you to select and set one of the simulation model from the `ALLOWED_SIM_MODELS`.
- **PREFERRED_SIM_MODEL:** This is a project level setting which allows you to set the default simulation model for the project. This is common across all IPs present in your project.

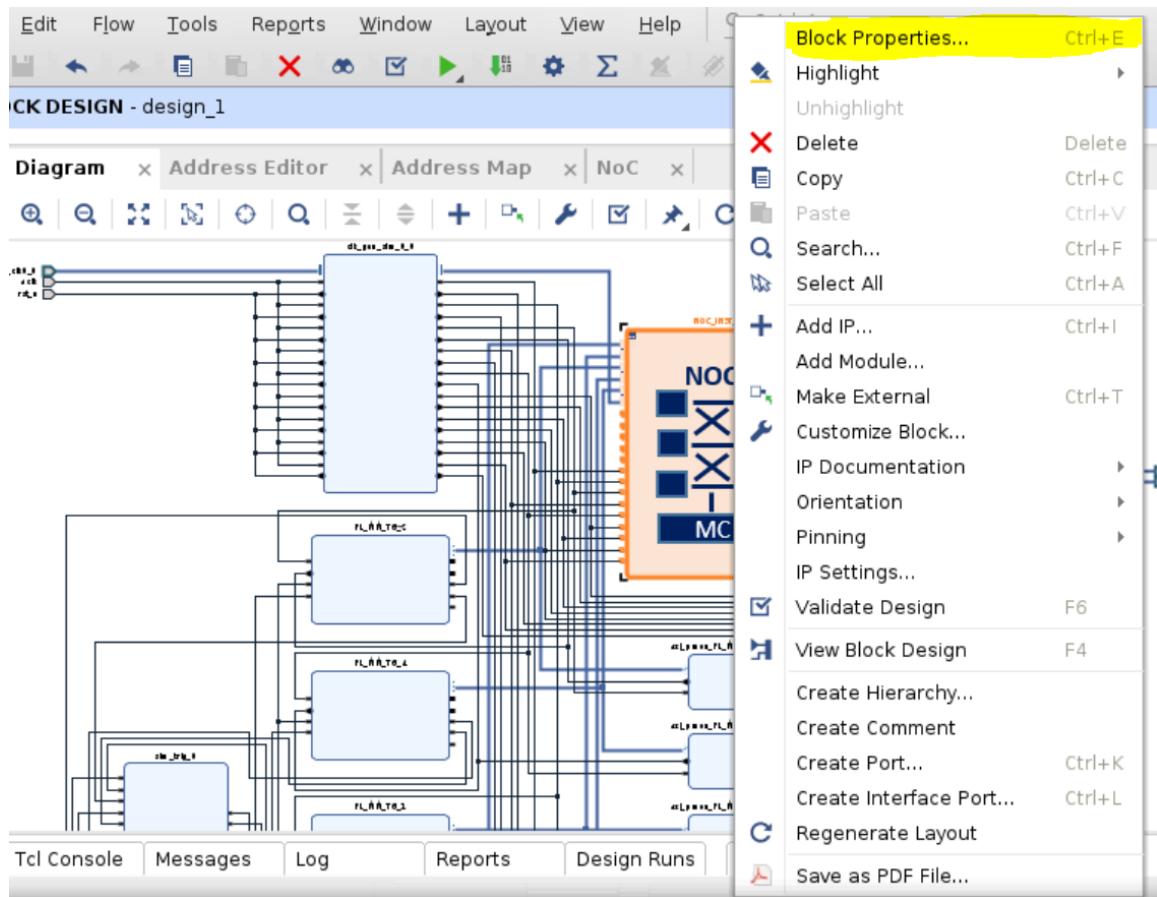
Using SELECTED_SIM_MODEL IP Property

Perform the following steps to change the simulation model of your IP using SELECTED_SIM_MODEL:

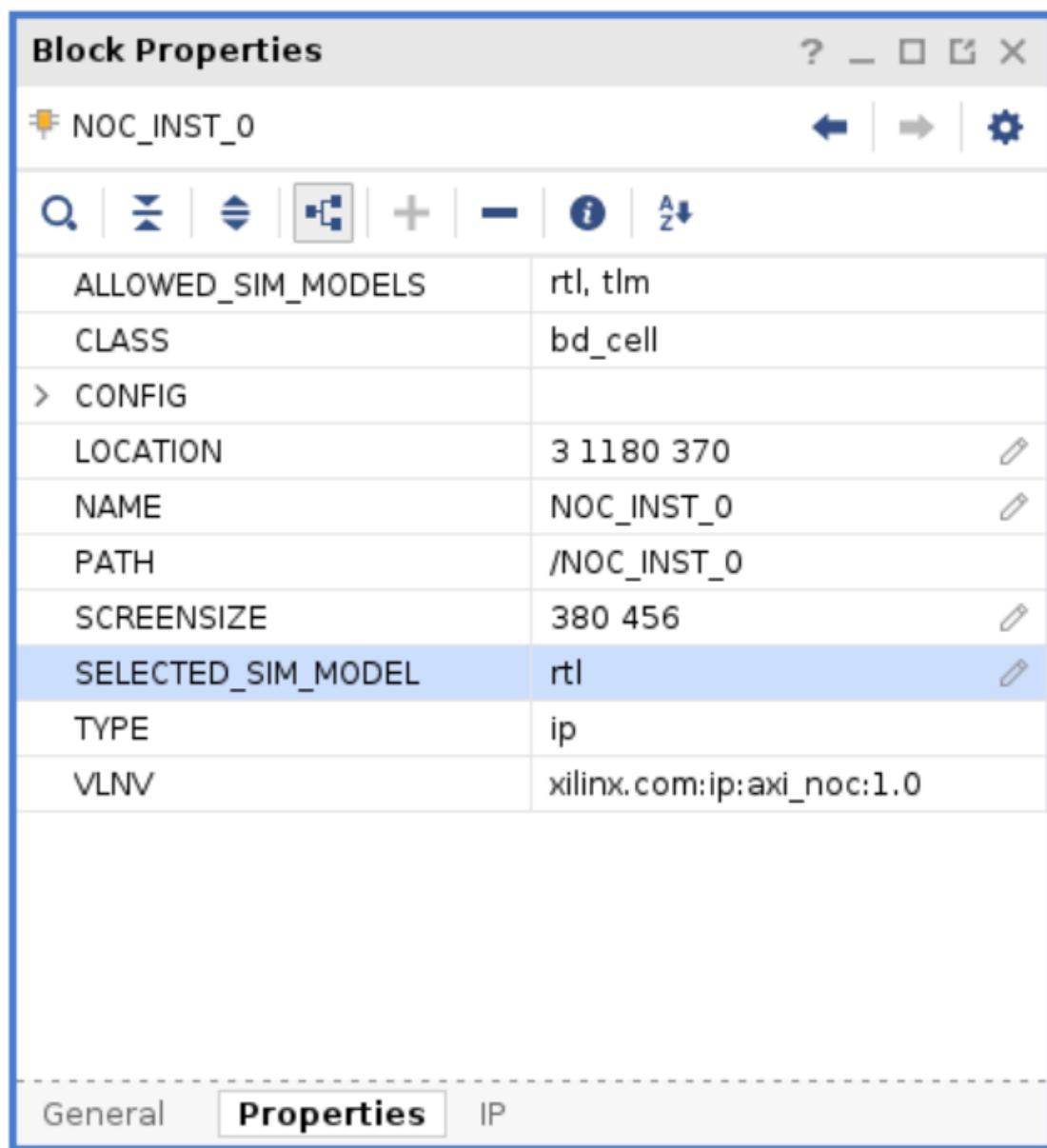
1. In the Flow Navigator, click **Open Block Design** to open a block design.
2. Select the desired IP from the block design.



3. Right-click and click **Block Properties** option.



4. Change the **SELECTED_SIM_MODEL** option from Block Properties window of your IP, for example from **rtl** to **tlm**.



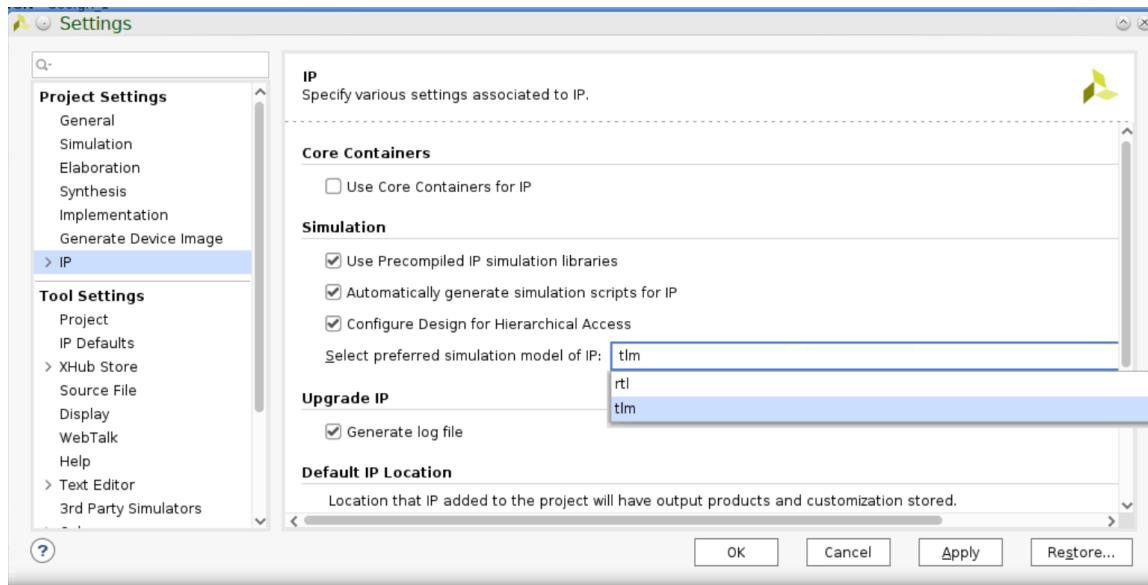
The following Tcl command is an equivalent to change the SELECTED_SIM_MODEL:

```
set_property SELECTED_SIM_MODEL tlm [get_bd_cells /NOC_INST_0]
```

Using PREFERRED_SIM_MODEL Project Property

Perform the following steps to change the simulation model of your IP using PREFERRED_SIM_MODEL:

1. Click IP option in the **Settings** dialog box.
2. Select **tlm** from **Select preferred simulation model of IP** drop-down menu.



The following Tcl command is an equivalent to change the PREFERRED_SIM_MODEL:

```
set_property preferred_sim_model tlm [current_project]
```

Note: Setting PREFERRED_SIM_MODEL to tlm sets all IPs SELECTED_SIM_MODEL to tlm except IPs which does not support tlm.

Protected Models

The protected models are pre-compiled and released in the form of a shared library that is built for the respective simulator. This shared library is packaged as part of the Vivado® install and based on the design configuration these models are bonded during elaboration. The following two protected models are delivered as part of Vivado install:

- AI Engine
- Network on chip (NoC)

These models are in the form of shared library present in the following installation path:

```
<Vivado-install-path>/data/simmodels/<simulator>/<simulator_version>/
<os_type>/<gcc_version>/systemc/protected
```

- **Vivado simulator:** <Vivado-install-path>/data/simmodels/xsim/2020.2/
 lnx64/6.2.0/systemc/protected
- **Xcelium simulator:** <Vivado-install-path>/data/simmodels/xcelium/
 20.03.005/lnx64/6.3/systemc/protected

- **Questa Advanced Simulator:** <Vivado-install-path>/data/simmodels/questa/2020.2/lnx64/5.3.0/systemc/protected
 - **VCS Simulator:** <Vivado-install-path>/data/simmodels/vcs/R-2020.12/lnx64/6.2.0/systemc/protected/
-

Unprotected Models

The unprotected models are released as a source code in the install. You need to compile the model for the target simulator using the `compile_simlib` utility. For Vivado® simulator, these unprotected models are pre-compiled in the standard <Vivado-install-path>/data/xsim folder where other libraries are compiled. For third party simulators, these models must be compiled using `compile_simlib`. The following un-protected models are delivered as part of Vivado installation:

- aie_xtlm
- axi_tg_sc
- axis_dwidth_converter_sc
- axis_switch_sc
- common_cpp
- common_rpc
- debug_tcp_server
- emu_perf_common
- noc_sc
- pl_fileio
- remote_port_c
- remote_port_sc
- rwd_tlmmodel
- sim_ddr
- sim_qdma_cpp
- sim_qdma_sc
- sim_xdma_cpp
- sim_xdma_sc
- tlm_ext
- xtlm

- `xtlm_ap_ctrl`
- `xtlm_ipc`
- `xtlm_simple_interconnect`
- `xtlm_trace_model`

These simulation model sources are present in the following installation path:

```
<Vivado-install-path>/data/systemc/
```

The following IP support SystemC simulation:

- `processing_system7_v5_5_6`
- `versal_cips_v2_1_0`
- `zynq_ultra_ps_e_v3_2_6`
- `zynq_ultra_ps_e_v3_3_3`

Note: GCC version to compile these models should be supported version as mentioned in this user guide.

SystemC Simulation Using Vivado

Running SystemC simulation design needs:

- Creating design sources
- Compiling simulation models using `compile_simlib`
- Specify tool/design settings needed

c/c++/SystemC sources can be compiled using GCC. Each simulator supports different versions of GCC. If design contains Xilinx provided SystemC models, GCC version used should be the supported version. Design needs to be re-compiled if GCC version changes.

Simulators Supported for SystemC Simulation

Following are the simulators supported for SystemC simulation in the Vivado® Design Suite:

Table 42: Simulators Supported for SystemC Simulation

Simulator	Version	Compatible GCC Version	SystemC Compiler
Vivado® simulator	2021.1	6.2.0	XSC
Siemens EDA Questa Advanced simulator	2020.4	5.3.0	SCCOM

Table 42: Simulators Supported for SystemC Simulation (cont'd)

Simulator	Version	Compatible GCC Version	SystemC Compiler
Cadence Xcelium Parallel simulator	20.09.006	6.3	XMSC
VCS	R-2020.12	6.2.0	SYSCAN

Simulator Settings for Third-Party Tools

Table 43: Simulator Settings for Third Party Tools

Simulator	Linux
Questa	<pre>setenv MODEL_TECH <tool installation path> setenv LM_LICENSE_FILE <license file> setenv PATH \${MODEL_TECH}/bin:\$PATH</pre>
Xcelium	<pre>setenv CDS_INST_DIR <xcelium_install_dir> setenv LD_LIBRARY_PATH \$CDS_INST_DIR/tools/xcelium/ lib:\$LD_LIBRARY_PATH setenv PATH \$CDS_INST_DIR/tools/xcelium/ bin:\$CDS_INST_DIR/tools/bin:\$PATH setenv CDS_LICENSE_DIR <tool_license></pre>
VCS	<pre>setenv VCS_HOME <tool_install_path> setenv SYSTEMC_HOME \$VCS_HOME/linux64/lib setenv LM_LICENSE_FILE <license file> setenv VG_GNU_PACKAGE /tools/installs/synopsys/vg_gnu/ 2020.12/linux setenv PATH \${VCS_HOME}/bin:\$PATH source \$VG_GNU_PACKAGE/source_me[.sh .csh]</pre>

Note: By default, GCC path is auto determined from the tool installation location for Questa and Xcelium.

GCC Path Settings

The following table describes GCC executable path settings for `compile_simlib` and `launch_simulation`:

Table 44: GCC Path Settings

Command	Settings
<code>compile_simlib</code>	<ul style="list-style-type: none"> Specify the GCC compiler install path using <code>-gcc_exec_path</code> switch. If not, set the environment variable <code>GCC_SIM_EXE_PATH <gcc_install_dir></code>.
<code>launch_simulation</code>	<ul style="list-style-type: none"> Specify the GCC compiler install path using <code>-gcc_install_path</code> switch. If not, set the property using <code>set_property simulator.<name>_gcc_install_dir <gcc_path> [current_project]</code> If not, set the environment variable <code>GCC_SIM_EXE_PATH <gcc_install_dir></code>.

Note: If these recommended settings are not found, Vivado would pick install path from PATH env variable. Also, it is always recommended to use tool native SystemC compilers.

Running SystemC Simulation Using Vivado Simulator

For step-by-step demonstration on how to run Vivado® simulation, see *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#)).

Note: If you are using the Vivado simulator, be sure to specify all appropriate project settings for your design before running simulation.

Automated Testbench Generation for Sub-Design

From 2021.2 onwards, new methodology is introduced in Vivado simulator (XSim) to create a realistic functional testbench for a language independent sub-design unit. This methodology currently works for Verilog/VHDL/System Verilog and mixed design of these language. To use this methodology, two new Tcl commands are introduced. Usage of this new methodology with a real design is explained in subsequent sections.

generate_vcd_ports

The `generate_vcd_port` command opens a VCD file handle for writing port activity of the given instance. This VCD file is read by `create_testbench` Tcl command to read port activity for writing the stimuli source. The instance can be selected from the scope window in the Vivado simulator IDE or by specifying the hierarchical path to the instance when executing this command from Tcl console. The command creates `dumpports.vcd` file that gets populated when running simulation for the selected instance scope.

Note: The Vivado simulator must be active to generate this file for the selected instance.

If running this command from Vivado IDE, then the `dumpports.vcd` file is created in the simulation run directory. If running this command from Vivado simulator standalone GUI, then the `dumpports.vcd` file is created in `vcd2tb` sub-directory of the current directory. Following are the `generate_vcd_port` options:

- **-scope <arg> (required):** Specify the instance scope hierarchy name.
- **-quiet (optional):** Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command-line while launching the command are returned. Only errors occurring inside the command are trapped.

- **-verbose (optional):** Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

Following example command creates a VCD file for `/top/DUT/fifo/buf_1` instance of type `buf` module, record waveform activity for 2000ns, and close the VCD file handle:

```
generate_vcd_ports {/top/DUT/fifo/buf_1}
run 2000ns
close_vcd -ports
```

create_testbench

Create testbench for a design unit instance. This command creates a functional system verilog based testbench for the scoped hierarchical instance. The testbench contains port/signal specification, parameter declaration, stimuli vector include file, and module instantiation of the selected instance as design under test (DUT). This command allows you to add the testbench to an existing or a new simulation fileset from which the simulation can be launched.

Note: The generated testbench is simulator independent.

Table 45: create_testbench Command Options

Option	Description
<code>-name <arg></code>	Specify the name of the testbench module name. Default name is <code>testbench</code> .
<code>-add_to_simset <arg></code>	Specify simulation fileset name to which the testbench needs to be added. If this switch is not specified, then the command adds testbench to the current active simulation fileset.
<code>-set_as_top</code>	Set the generated testbench module at the top in the simulation fileset where the testbench is added.
<code>-mode <arg></code>	Specifies simulation mode. Allowed values are behavioral, post-synthesis, or post-implementation. Default is behavioral.
<code>-type <arg></code>	Specifies simulation type. Allowed values are functional or timing (not applicable for behavioral mode).
<code>-force</code>	Overwrite existing testbench file.
<code>-quiet</code>	Execute the command quietly, returning no messages from the command. The command also returns <code>TCL_OK</code> regardless of any errors encountered during execution. Note: Any errors encountered on the command-line while launching the command are returned. Only errors occurring inside the command are trapped.
<code>-verbose</code>	Temporarily override any message limits and return all messages from this command. Note: Message limits can be defined with the <code>set_msg_config</code> command.

Notes:

1. All the arguments are optional as default value is set as explained for each flag.

The following example command creates a testbench for fifo module and adds it to the `sub_design_fifo` simulation filesset:

```
create_testbench -name fifo -add_to_simset sub_design_fifo
```

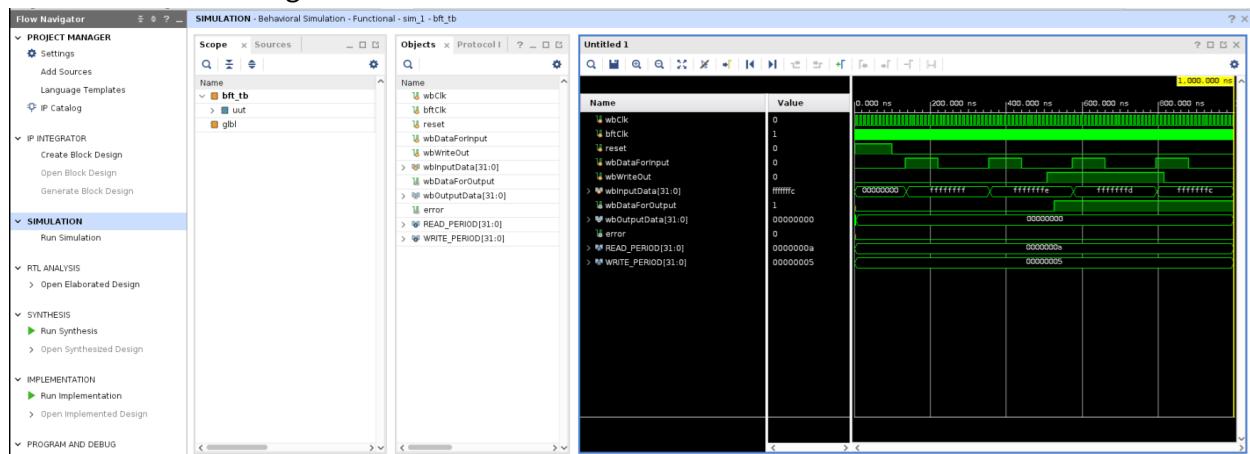
The following example command generates VCD file for `/top/DUT/fifo/buf_1` instance of type buf module, record the waveform activity in the VCD file for 2000ns, create a testbench with module named tb, add the testbench to the `test_buffer` simulation filesset and set tb as top module in this filesset:

```
generate_vcd_ports {/top/DUT/fifo/buf_1}
run 2000ns
close_vcd -ports
create_testbench -name tb -add_to_simset test_buffer -set_as_top
```

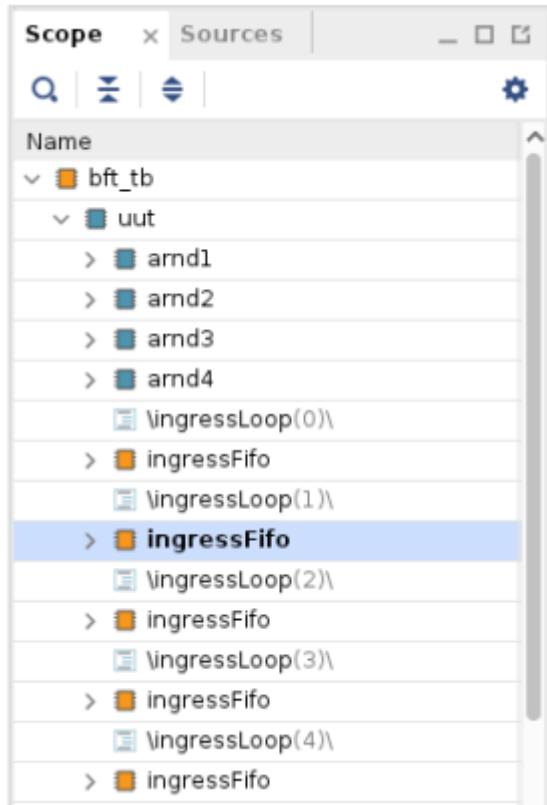
Using Automated Testbench Generation on Example Design

For demo purpose, let us use the BFT example design shipped with Vivado IDE.

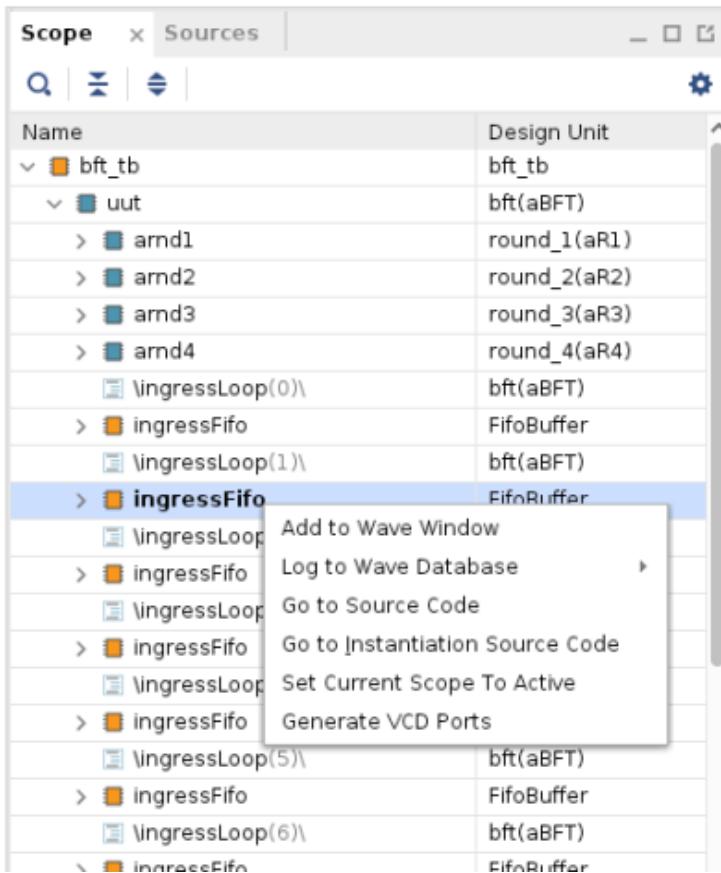
1. Open the BFT example design in Vivado IDE.
2. Call `launch_simulation` with Vivado as the selected simulator. You should see the ports shown in the following waveform.



3. Select the desired scope for which you want to generate testbench as shown in the following figure:



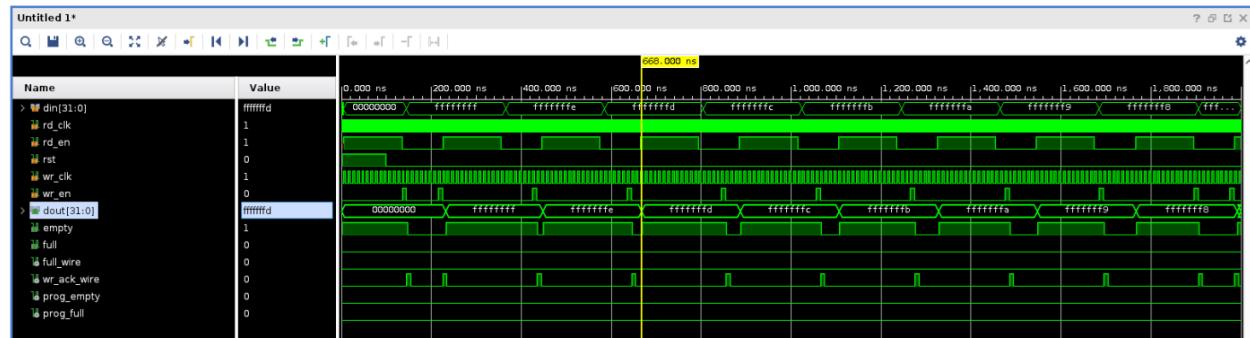
4. Right-click the selected scope and select **Generate VCD Port**.



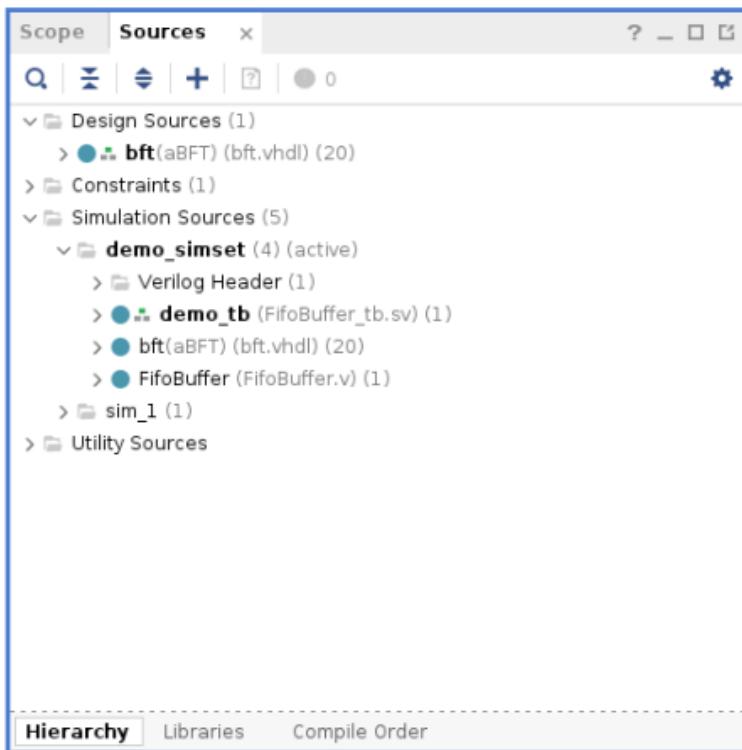
5. Delete all existing signals on the waveform and select **Add to Wave Window** for the selected scope.

Note: Step 5 will be used to demonstrate that the generated testbench is driving the design unit correctly.

6. Use `restart`, `run 2000 ns` and `close_vcd -ports` commands on Tcl console to dump the signal activity. This logs the signal from time 0 to 2000 ns on the waveform as shown in the following figure:



7. Use `create_testbench -name demo_tb -add_to_simset demo_simset -set_as_top` command on Tcl console to generate testbench. This creates your testbench with the module name `demo_tb` and creates a `demo_simset` with this testbench as top module.



8. Use `launch_simulation` command to run simulation with the newly generated testbench.
9. Compare the input/output of the waveform with waveform of your original design, notice that the input/output are same.

This is how you can create testbench for your sub-design and use the generated testbench independently with any standard simulator.

Handling Special Cases

Using Global Reset and 3-State

Xilinx devices have dedicated routing and circuitry that connect to every register in the device.

Global Set and Reset Net

During configuration, the dedicated Global Set/Reset (GSR) signal is asserted. The GSR signal is deasserted upon completion of device configuration. All the flip-flops and latches receive this reset, and are set or reset depending on how the registers are defined.

Although you can access the GSR net after configuration, avoid use of the GSR circuitry in place of a manual reset. This is because the FPGA devices offer high-speed backbone routing for high fanout signals such as a system reset. This backbone route is faster than the dedicated GSR circuitry, and is easier to analyze than the dedicated global routing that transports the GSR signal.

In post-synthesis and post-implementation simulations, the GSR signal is automatically asserted for the first 100 ns to simulate the reset that occurs after configuration.

A GSR pulse can optionally be supplied in pre-synthesis functional simulations, but is not necessary if the design has a local reset that resets all registers.



TIP: When you create a test bench, remember that the GSR pulse occurs automatically in the post-synthesis and post-implementation simulation. This holds all registers in reset for the first 100 ns of the simulation.

Note: If a design uses ICAP primitive, GSR will last for 1.281 us at that time.

Global 3-State Net

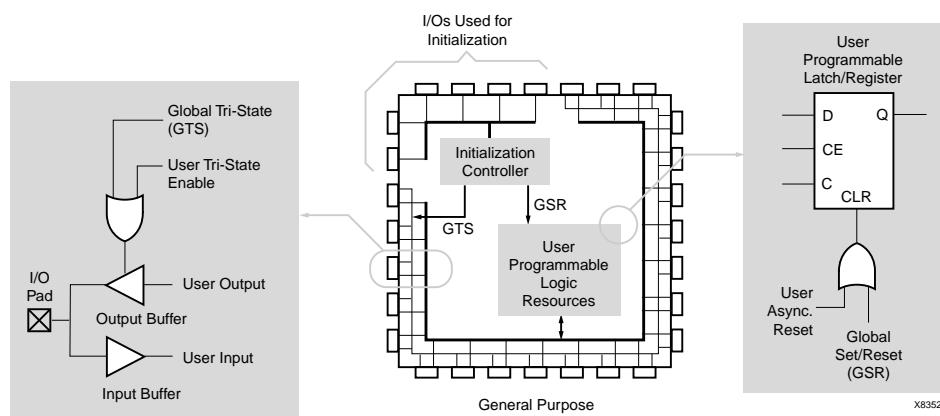
In addition to the dedicated global GSR, output buffers are set to a high impedance state during configuration mode with the dedicated Global 3-state (GTS) net. All general-purpose outputs are affected whether they are regular, 3-state, or bidirectional outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the FPGA is configured.

In simulation, the GTS signal is usually not driven. The circuitry for driving GTS is available in the post-synthesis and post-implementation simulations and can be optionally added for the pre-synthesis functional simulation, but the GTS pulse width is set to 0 by default.

Using Global 3-State and Global Set and Reset Signals

The following figure shows how Global 3-State (GTS) and Global Set/Reset (GSR) signals are used in an FPGA.

Figure 53: Built-in FPGA Initialization Circuitry Diagram



Global Set and Reset and Global 3-State Signals in Verilog

The GSR and GTS signals are defined in the `<Vivado_Install_Dir>/data/verilog/src/glbl.v` module.

In most cases, GSR and GTS need not be defined in the test bench.

The `lbl.v` file declares the global GSR and GTS signals and automatically pulses GSR for 100 ns.

Global Set and Reset and Global 3-State Signals in VHDL

The GSR and GTS signals are defined in the file: `<Vivado_Install_Dir>/data/vhdl/src/unisims/primitive/GLBL_VHD.vhd`.

To use the `GLBL_VHD` component you must instantiate it into the test bench.

The `GLBL_VHD` component declares the global GSR and GTS signals and automatically pulses GSR for 100 ns.

The following code snippet shows an example of instantiating the `GLBL_VHD` component in the test bench and changing the assertion pulse width of the Reset on Configuration (ROC) to 90 ns:

```
GLBL_VHD inst:GLBL_VHD generic map ( ROC_WIDTH => 90000 ) ;
```

Delta Cycles and Race Conditions

This user guide describes event-based simulators. Event-based simulators can process multiple events at a given simulation time. While these events are being processed, the simulator cannot advance the simulation time. This event processing time is commonly referred to as *delta cycles*. There can be multiple delta cycles in a given simulation time step.

Simulation time is advanced only when there are no more transactions to process for the current simulation time. For this reason, simulators can give unexpected results, depending on when the events are scheduled within a time step. The following VHDL coding example shows how an unexpected result can occur.

VHDL Coding Example With Unexpected Results

```
clk_b <= clk;
clk_prcs : process (clk)
begin
  if (clk'event and clk='1') then
    result <= data;
  end if;
end process;
clk_b_prcs : process (clk_b)
begin
  if (clk_b'event and clk_b='1') then
    result1 <= result;
  end if;
end process;
```

In this example, there are two synchronous processes:

- `clk_prcs`
- `clk_b_prcs`

The simulator performs the `clk_b <= clk` assignment before advancing the simulation time. As a result, events that should occur in two clock edges occur in one clock edge instead, causing a race condition.

Recommended ways to introduce causality in simulators for such cases include:

- Do not change clock and data at the same time. Insert a delay at every output.
- Use the same clock.
- Force a delta delay by using a temporary signal, as shown in the following example:

```
clk_b <= clk;
clk_prcs : process (clk)
begin
  if (clk'event and clk='1') then
    result <= data;
  end if;
end process;
result_temp <= result;
clk_b_prcs : process (clk_b)
begin
  if (clk_b'event and clk_b='1') then
    result1 <= result_temp;
  end if;
end process;
```

Most event-based simulators can display delta cycles. Use this to your advantage when debugging simulation issues.

Using the ASYNC_REG Constraint

The ASYNC_REG constraint:

- Identifies asynchronous registers in the design
- Disables X propagation for those registers

The ASYNC_REG constraint can be attached to a register in the front-end design by using either:

- An attribute in the HDL code
- A constraint in the Xilinx Design Constraints (XDC)

The registers to which ASYNC_REG are attached retain the previous value during timing simulation, and do not output an X to simulation. Use care; a new value might have been clocked in as well.

The ASYNC_REG constraint is applicable to CLB and Input Output Block (IOB) registers and latches only. For more information, see ASYNC_REG constraint at this [link](#) in the *Vivado Design Suite Properties Reference Guide (UG912)*.

If you cannot avoid clocking in asynchronous data, do so for IOB or CLB registers only. Clocking in asynchronous signals to RAM, Shift Register LUT (SRL), or other synchronous elements has less deterministic results; therefore, should be avoided. Xilinx highly recommends that you first properly synchronize any asynchronous signal in a register, latch, or FIFO before writing to a RAM, Shift Register LUT (SRL), or any other synchronous element. For more information, see the *Vivado Design Suite User Guide: Using Constraints* ([UG903](#)).

Disabling X Propagation for Synchronous Elements

When a timing violation occurs during a timing simulation, the default behavior of a latch, register, RAM, or other synchronous elements is to output an X to the simulator. This occurs because the actual output value is not known. The output of the register could:

- Retain its previous value
- Update to the new value
- Go metastable, in which a definite value is not settled upon until some time after the clocking of the synchronous element

Because this value cannot be determined, and accurate simulation results cannot be guaranteed, the element outputs an X to represent an unknown value. The X output remains until the next clock cycle in which the next clocked value updates the output if another violation does not occur.

The presence of an X output can significantly affect simulation. For example, an X generated by one register can be propagated to others on subsequent clock cycles. This can cause large portions of the design under test to become unknown.

To correct X-generation:

- On a synchronous path, analyze the path and fix any timing problems associated with this or other paths to ensure a properly operating circuit.
- On an asynchronous path, if you cannot otherwise avoid timing violations, disable the X propagation on synchronous elements during timing violations by using the `ASYNC_REG` property.

When X propagation is disabled, the previous value is retained at the output of the register. In the actual silicon, the register might have changed to the 'new' value. Disabling X propagation might yield simulation results that do not match the silicon behavior.



CAUTION! Exercise care when using this option. Use it only if you cannot otherwise avoid timing violations.

Simulating Configuration Interfaces

This section describes the simulation of the following configuration interfaces:

- JTAG simulation
- SelectMAP simulation

JTAG Simulation

BSCAN component simulation is supported on all devices.

The simulation supports the interaction of the JTAG ports and some of the JTAG operation commands. The JTAG interface, including interface to the scan chain, is not fully supported. To simulate this interface:

1. Instantiate the `BSCANE2` component and connect it to the design.
2. Instantiate the `JTAG_SIME2` component into the test bench (not the design).

This becomes:

- The interface to the external JTAG signals (such as TDI, TDO, and TCK)
- The communication channel to the `BSCAN` component

The communication between the components takes place in the `VPKG` VHDL package file or the `g1b1` Verilog global module. Accordingly, no implicit connections are necessary between the specific `JTAG_SIME2` component and the design, or the specific `BSCANE2` symbol.

Stimulus can be driven and viewed from the specific `JTAG_SIME2` component within the test bench to understand the operation of the JTAG/BSCAN function. Instantiation templates for both of these components are available in both the Vivado® Design Suite templates and the specific-device libraries guides.

SelectMAP Simulation

The configuration simulation models (`SIM_CONFIGE2` and `SIM_CONFIGE3`) with an instantiation template allow supported configuration interfaces to be simulated to ultimately show the `DONE` pin going HIGH. This is a model of how the supported devices react to stimulus on the supported configuration interface.

The following table lists the supported interfaces and devices.

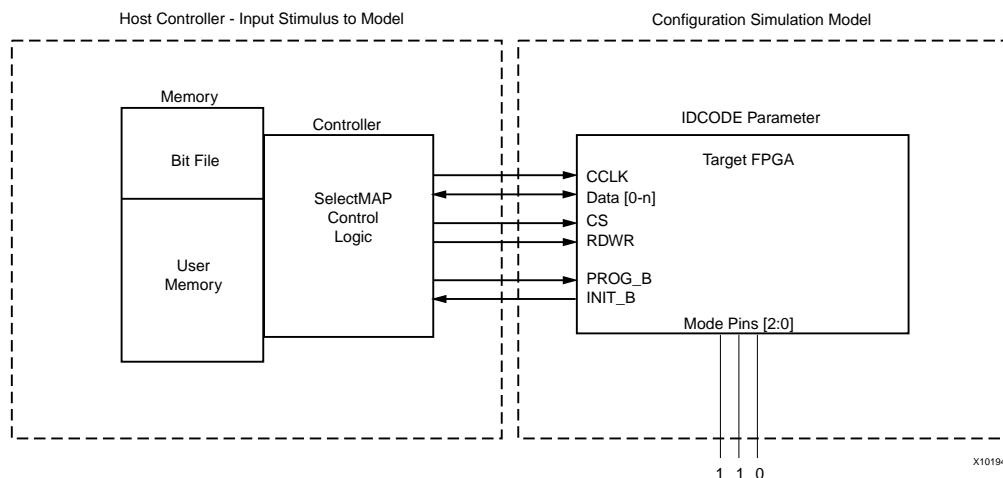
Table 46: Supported Configuration Devices and Modes

Devices	SelectMAP	Serial	SPI	BPI
7 series and Zynq®-7000 SoC Devices	Yes	Yes	No	No
UltraScale Devices	Yes	Yes	No	No
UltraScale+™ Devices	Yes	Yes	No	No

The models handle control signal activity as well as bit file downloading. Internal register settings such as the CRC, IDCODE, and status registers are included. You can monitor the Sync Word as it enters the device and the start-up sequence as it progresses. The following figure illustrates how the system should map from the hardware to the simulation environment.

The configuration process is specifically outlined in the configuration user guides for each device. These guides contain information on the configuration sequence, as well as the configuration interfaces.

Figure 54: Block Diagram of Model Interaction



System Level Description

The configuration models allow the configuration interface control logic to be tested before the hardware is available. It simulates the entire device, and is used at a system level for:

- Applications using a processor to control the configuration logic to ensure proper wiring, control signal handling, and data input alignment.
- Applications that control the data loading process with the CS (SelectMAP Chip Select) or CLK signal to ensure proper data alignment.
- Systems that need to perform a SelectMAP ABORT or Readback.

The `config_test_bench.zip` file has sample test benches that simulate a processor running the SelectMAP logic. These test benches have control logic to emulate a processor controlling the SelectMAP interface, and include features such as a full configuration, ABORT, and Readback of the IDCODE and status registers.

For the ZIP files associated with this model, see Xilinx Answer Record [53632](#).

The simulated host system must have a method for file delivery as well as control signal management. These control systems should be designed as set forth in the device configuration user guides.

The configuration models also demonstrate what is occurring inside the device during the configuration procedure when a BIT file is loaded into the device.

During the BIT file download, the model processes each command and changes registers settings that mirror the hardware changes.

You can monitor the CRC register as it actively accumulates a CRC value. The model also shows the Status Register bits being set as the device progresses through the different states of configuration.

Debugging with the Model

Each configuration model provides an example of a correct configuration. You can leverage this example to assist in the debug procedure if you encounter device programming issues.

You can read the Status Register through JTAG using the Vivado Device Programmer tool. This register contains information relating to the current status of the device and is a useful debugging resource. If you encounter issues on the board, reading the Status Register in Vivado Device Programmer is one of the first debugging steps to take.

After the status register is read, you can map it to the simulation to pinpoint the configuration stage of the device.

For example, the `GHIGH` bit is set HIGH after the data load process completes successfully; if this bit is not set, then the data loading operation did not complete. You can also monitor the `GTW`, `GWE`, and `DONE` signals set in BitGen that are released in the start-up sequence.

The configuration models also allow for error injection. The active CRC logic detects any issue if the data load is paused and started again with any problems. It also detects bit flips manually inserted in the BIT file, and handles them just as the device would handle this error.

Feature Support

Each device-specific configuration user guide outlines the supported methods of interacting with each configuration interface. The table below shows which features discussed in the configuration user guides are supported.

The SIM_CONFIGE2 model:

- Does not support Readback of configuration data.
- Does not store configuration data provided, although it does calculate a CRC value.
- Can perform Readback on specific registers only to ensure that a valid command sequence and signal handling is provided to the device.
- Is not intended to allow Readback data files to be produced.

Table 47: Model-Supported Slave SelectMAP and Serial Features

Slave SelectMAP and Serial Features	Supported
Master mode	No
Daisy chain - slave parallel daisy chains	No
SelectMAP data loading	Yes
Continuous SelectMAP data loading	Yes
Non-continuous SelectMAP data loading	Yes
SelectMAP ABORT	Yes
SelectMAP reconfiguration	No
SelectMAP data ordering	Yes
Reconfiguration and MultiBoot	No
Configuration CRC—CRC checking during configuration	Yes
Configuration CRC—post-configuration CRC	No

Disabling Block RAM Collision Checks for Simulation

Xilinx® block RAM memory is a true dual-port RAM where both ports can access any memory location at any time. Be sure that the same address space is not accessed for reading and writing at the same time. This causes a block RAM address collision. These are valid collisions, because the data that is being read from the read port is not valid.

In the hardware, the value that is read might be the old data, the new data, or a combination of the old data and the new data.

In simulation, this is modeled by outputting X because the value read is unknown. For more information on block RAM collisions, see the user guide for the device.

In certain applications, this situation cannot be avoided or designed around. In these cases, the block RAM can be configured not to look for these violations. This is controlled by the generic (VHDL) or parameter (Verilog) `SIM_COLLISION_CHECK` string in block RAM primitives.

The following table shows the string options you can use with `SIM_COLLISION_CHECK` to control simulation behavior in the event of a collision.

Table 48: SIM_COLLISION_CHECK Strings

String	Write Collision Messages	Write Xs on the Output
ALL	Yes	Yes
WARNING_ONLY	Yes	No. Applies only at the time of collision. Subsequent reads of the same address space could produce Xs on the output.
GENERATE_X_ONLY	No	Yes
None	No	No. Applies only at the time of collision. Subsequent reads of the same address space could produce Xs on the output.

Apply the `SIM_COLLISION_CHECK` at an instance level so you can change the setting for each block RAM instance.

Dumping the Switching Activity Interchange Format File for Power Analysis

- Vivado simulator: [Power Analysis Using Vivado Simulator](#)
- [Dumping SAIF for Power Analysis](#), and [Dumping SAIF in VCS](#) in [Chapter 3: Simulating with Third-Party Simulators](#)

Skipping Compilation or Simulation

Skipping Compilation

You can run simulation on an existing snapshot and skip the compilation (or recompilation) of the design by setting the `SKIP_COMPILATION` property on the simulation fileset:

```
set_property SKIP_COMPILATION 1 [get_filesets sim_1]
```

Note: Any change to design files after the last compilation is not reflected in simulation when you set this property.

Skipping Simulation

To perform a semantic check on the design HDL files, by elaborating and compiling the simulation snapshot without running simulation, you can set the SKIP_SIMULATION property on the simulation fileset:

```
set_property SKIP_SIMULATION true [get_filesets sim_1]
```



IMPORTANT! If you elect to use one of the properties above, disable the **Clean up simulation files** check box in the simulations settings, or if you are running in batch/Tcl mode, call `launch_simulation` with `-noclean_dir`.

Value Rules in Vivado Simulator Tcl Commands

This appendix contains the value rules that apply to both the `add_force` and the `set_value` Tcl commands.

String Value Interpretation

The interpretation of the value string is determined by the declared type of the HDL object and the `-radix` command line option. The `-radix` always overrides the default radix determined by the HDL object type.

- For HDL objects of type `logic`, the value is a one-dimensional array of the `logic` type or the value is a string of digits of the specified radix.
 - If the string specifies less bits than the type expects, the string is implicitly zero-extended (not sign-extended) to match the length of the type.
 - If the string specifies more bits than the type expects, the extra bits on the MSB side must be zero; otherwise the command generates a size mismatch error.

For example: The value 3F specifies 8 bits (4 per hex digit) with radix hex and a 6 bit `logic` array, equivalent to binary 0011 1111. But, because the upper two bits of 3 are zero, the value can be assigned to the HDL object. In contrast, the value 7F would generate an error, because the upper two bits are not zero.

- A scalar (not array or record) `logic` HDL object has an implicit length of 1 bit.
- For a `logic` array declared as a `[left:right]` (Verilog) or a `(left TO/DOWNT0 right)`, the left-most value bit (after extension/truncation) is assigned to `a[left]` and the right-most value bit is assigned to `a[right]`.

Vivado Design Suite Simulation Logic

The logic is not a concept defined in HDL but is a heuristic introduced by the Vivado® simulator.

- A Verilog object is considered to be of `logic` type if it is of the implicit Verilog bit type, which includes wire and reg objects, as well as integer and time.
- A VHDL object is considered to be of `logic` type if the objects type is bit, `std_logic`, or any enumeration type whose enumerators are a subset of those of `std_logic` and include at least 0 and 1, or type of the object is a one-dimensional array of such a type.
- For HDL objects, which are of VHDL enumeration type, the value can be one of the enumerator literals, without single quotes if the enumerator is a character literal. Radix is ignored.
- For VHDL objects, of integral type, the value can be a signed decimal integer in the range of the type. Radix is ignored.
- For VHDL and Verilog floating point types the value can be a floating point value. Radix is ignored.
- For all other types of HDL objects, the Tcl command set does not support setting values.

Vivado Simulator Mixed Language Support and Language Exceptions

The Vivado Integrated Design Environment (IDE) supports the following languages:

- VHDL, see [IEEE Standard VHDL Language Reference Manual \(IEEE-STD-1076-1993\)](#)
- Verilog, see [IEEE Standard Verilog Hardware Description Language \(IEEE-STD-1364-2001\)](#)
- SystemVerilog Synthesizable subset. See [IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language \(IEEE-STD-1800-2009\)](#)
- IEEE P1735 encryption, see [Recommended Practice for Encryption and Management of Electronic Design Intellectual Property \(IP\) \(IEEE-STD-P1735\)](#)

This appendix lists the application of Mixed Language in the Vivado simulator, and the exceptions to Verilog, SystemVerilog, and VHDL support.

Using Mixed Language Simulation

The Vivado simulator supports mixed language project files and mixed language simulation. This lets you include Verilog/SystemVerilog (SV) modules in a VHDL design, and vice versa.

Restrictions on Mixed Language in Simulation

- A VHDL design can instantiate Verilog/SystemVerilog (SV) modules and a Verilog/SV design can instantiate VHDL components. Component instantiation-based default binding is used for binding a Verilog/SV module to a VHDL component. Any other kind of mixed use of VHDL and Verilog, such as VHDL process calling a Verilog function, is not supported.
- A subset of VHDL types, generics, and ports are allowed on the boundary to a Verilog/SV module. Similarly, a subset of Verilog/SV types, parameters and ports are allowed on the boundary to VHDL components. See [Table 50: Supported VHDL and Verilog Data Types](#).



IMPORTANT! Connecting whole VHDL record object to a Verilog object is unsupported; however, VHDL record elements of a supported type can be connected to a compatible Verilog port.

- A Verilog/SV hierarchical reference cannot refer to a VHDL unit nor can a VHDL expanded or selected name refer to a Verilog/SV unit.

Key Steps in a Mixed Language Simulation

1. Optionally, specify the search order for VHDL components or Verilog/SV modules in the design libraries of a mixed language project.
2. Use `xelab -L` to specify the binding order of a VHDL component or a Verilog/SV module in the design libraries of a mixed language project.

Note: The library search order specified by `-L` is used for binding Verilog modules to other Verilog modules as well.

Mixed Language Binding and Searching

When you instantiate a VHDL component in a Verilog/SV module or a Verilog/SV module in a VHDL architecture, the `xelab` command:

- First searches for a unit of the same language as that of the instantiating design unit.
- If a unit of the same language is not found, `xelab` searches for a cross-language design unit in the libraries specified by the `-L` option.

The search order is the same as the order of appearance of libraries on the `xelab` command line.

Note: When using the Vivado IDE, the library search order is specified automatically. No user intervention is necessary or possible.

Related Information

[Verilog Search Order](#)

Instantiating Mixed Language Components

In a mixed language design, you can instantiate a Verilog/SV module in a VHDL architecture or a VHDL component in a Verilog/SV module as described in the following subsections.

To ensure that you are correctly matching port types, review the [Port Mapping and Supported Port Types](#).

Instantiating a Verilog Module in a VHDL Design Unit

1. Declare a VHDL component with the same name and in the same case as the Verilog module that you want to instantiate. For example:

```
COMPONENT MY_VHDL_UNIT PORT (
  Q : out STD_ULOGIC;
  D : in  STD_ULOGIC;
  C : in  STD_ULOGIC );
END COMPONENT;
```

2. Use named or positional association to instantiate the Verilog module. For example:

```
UUT : MY_VHDL_UNIT PORT MAP(
  Q => O,
  D => I,
  C => CLK);
```

Instantiating a VHDL Component in a Verilog/SV Design Unit

To instantiate a VHDL component in a Verilog/SV design unit, instantiate the VHDL component as if it were a Verilog/SV module.

For example:

```
module testbench ;
  wire in, clk;
  wire out;
  FD FD1(
    .Q(Q_OUT),
    .C(CLK),
    .D(A));
endmodule
```

Port Mapping and Supported Port Types

The following table lists the supported port types.

Table 49: Supported Port Types

VHDL ¹	Verilog/SV ²
IN	INPUT
OUT	OUTPUT
INOUT	INOUT

Notes:

1. Buffer and linkage ports of VHDL are not supported.
2. Connection to bi-directional pass switches in Verilog are not supported. Unnamed Verilog ports are not allowed on mixed design boundary.

The following table shows the supported VHDL and Verilog data types for ports on the mixed language design boundary.

Table 50: Supported VHDL and Verilog Data Types

VHDL Port	Verilog Port
bit	net
std_logic	net
bit_vector	vector net
signed	vector net

Table 50: Supported VHDL and Verilog Data Types (cont'd)

VHDL Port	Verilog Port
unsigned	vector net
std_ulogic_vector	vector net
std_logic_vector	vector net

Note: Verilog output port of type `reg` is supported on the mixed language boundary. On the boundary, an output `reg` port is treated as if it were an output net (wire) port. Any other type found on mixed language boundary is considered an error.

Note: The Vivado simulator supports the record element as an actual in the port map of a Verilog module that is instantiated in the mixed domain. All those types that are supported as VHDL port (listed in [Table 50: Supported VHDL and Verilog Data Types](#)) are also supported as a record element.

Table 51: Supported SV and VHDL Data Types

SV Data type	VHDL Data type
Int	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
byte	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
shortint	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
longint	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
integer	
	bit_vector

Table 51: Supported SV and VHDL Data Types (cont'd)

SV Data type	VHDL Data type
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
vector of bit(1D)	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
vector of logic(1D)	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
vector of reg(1D)	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
logic/bit	
	bit
	std_logic
	std_ulogic
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned

Note: VHDL entity instantiating Verilog Module having real port is supported.

Generics (Parameters) Mapping

The Vivado simulator supports the following VHDL generic types (and their Verilog/SV equivalents):

- integer
- real
- string
- boolean

Note: Any other generic type found on mixed language boundary is considered an error.

VHDL and Verilog Values Mapping

The following table lists the Verilog states mappings to `std_logic` and `bit`.

Table 52: Verilog States mapped to std_logic and bit

Verilog	std_logic	bit
Z	Z	0
0	0	0
1	1	1
X	X	0

Note: Verilog strength is ignored. There is no corresponding mapping to strength in VHDL.

The following table lists the VHDL type `bit` mapping to Verilog states.

Table 53: VHDL bit Mapping to Verilog States

bit	Verilog
0	0
1	1

The following table lists the VHDL type `std_logic` mappings to Verilog states.

Table 54: VHDL std_logic mapping to Verilog States

std_logic	Verilog
U	X
X	X
0	0
1	1
Z	Z
W	X
L	0
H	1
-	X

Because Verilog is case sensitive, named associations and the local port names that you use in the component declaration must match the case of the corresponding Verilog port names.

VHDL Language Support Exceptions

Certain language constructs are not supported by the Vivado simulator. The following table lists the VHDL language support exceptions.

Table 55: VHDL Language Support Exceptions

Supported VHDL Construct	Exceptions
abstract_literal	Floating point expressed as based literals are not supported.
alias_declaration	Alias to non-objects are in general not supported; particularly the following: <ul style="list-style-type: none"> • Alias of an alias • Alias declaration without subtype_indication • Signature on alias declarations • Operator symbol as alias_designator • Alias of an operator symbol • Character literals as alias designators
alias_designator	Operator_symbol as alias_designator Character_literal as alias_designator
association_element	Globally, locally static range is acceptable for taking slice of an actual in an association element.
attribute_name	Signature after prefix is not supported.
binding_indication	Binding_indication without use of entity_aspect is not supported.
bit_string_literal	Empty bit_string_literal (" ") is not supported.
block_statement	Guard_expression is not supported; for example, guarded blocks, guarded signals, guarded targets, and guarded assignments are not supported.
choice	Aggregate used as choice in case statement is not supported.
concurrent_assertion_statement	Postponed is not supported.
concurrent_signal_assignment_statement	Postponed is not supported.
concurrent_statement	Concurrent procedure call containing wait statement is not supported.
conditional_signal_assignment	Keyword guarded as part of options is not supported as there is no supported for guarded signal assignment.
configuration_declaration	Non locally static for generate index used in configuration is not supported.
entity_class	Literals, unit, file, and group as entity class are not supported.

Table 55: VHDL Language Support Exceptions (cont'd)

Supported VHDL Construct	Exceptions
entity_class_entry	Optional <> intended for use with group templates is not supported.
file_logical_name	Although file_logical_name is allowed to be any wild expression evaluating to a string value, only string literal and identifier is acceptable as file name.
function_call	Slicing, indexing, and selection of formals is not supported in a named parameter association within a function_call.
instantiated_unit	Direct configuration instantiation is not supported.
mode	Linkage and Buffer ports are not supported completely.
options	Guarded is not supported.
primary	At places where primary is used, allocator is expanded there.
procedure_call	Slicing, indexing, and selection of formals is not supported in a named parameter association within a procedure_call.
process_statement	Postponed processes are not supported.
selected_signal_assignment	The guarded keyword as part of options is not supported as there is no support for guarded signal assignment.
signal_declaration	The signal_kind is not supported. The signal_kind is used for declaring guarded signals, which are not supported.
subtype_indication	Resolved subtype of composites (arrays and records) is not supported.
waveform	Unaffected is not supported.
waveform_element	Null waveform element is not supported as it only has relevance in the context of guarded signals.

Verilog Language Support Exceptions

The following table lists the exceptions to supported Verilog language support.

Table 56: Verilog Language Support Exceptions

Verilog Construct	Exception
Compiler Directive Constructs	
`unconnected_drive	not supported
`nounconnected_drive	not supported
Attributes	
attribute_instance	not supported
attr_spec	not supported
attr_name	not supported
Primitive Gate and Switch Types	
cmos_switchtype	not supported
mos_switchtype	not supported

Table 56: Verilog Language Support Exceptions (cont'd)

Verilog Construct	Exception
pass_en_switchtype	not supported
Generated Instantiation	
generated_instantiation	<p>The <code>module_or_generate_item</code> alternative is not supported.</p> <p>Production from IEEE standard (see IEEE Standard Verilog Hardware Description Language (IEEE-STD-1364-2001) section 13.2):</p> <pre>generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_block module_or_generate_item</pre> <p>Production supported by Simulator:</p> <pre>generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_block generate_condition</pre>
genvar_assignment	<p>Partially supported.</p> <p>All generate blocks must be named.</p> <p>Production from standard (see IEEE Standard Verilog Hardware Description Language (IEEE-STD-1364-2001) section 13.2):</p> <pre>generate_block ::= begin [: generate_block_identifier] { generate_item } end</pre> <p>Production supported by Simulator:</p> <pre>generate_block ::= begin: generate_block_identifier { generate_item } end</pre>
Source Text Constructs	
Library Source Text	
library_text	not supported
library_descriptions	not supported
library_declaration	not supported
include_statement	This refers to include statements within library map files (See IEEE Standard Verilog Hardware Description Language (IEEE-STD-1364-2001) section 13.2. This does not refer to the `include compiler directive.
System Timing Check Commands	
\$skew_timing_check	not supported
\$timeskew_timing_check	not supported
\$fullskew_timing_check	not supported

Table 56: Verilog Language Support Exceptions (cont'd)

Verilog Construct	Exception
\$nochange_timing_check	not supported
System Timing Check Command Argument	
checktime_condition	not supported
PLA Modeling Tasks	
\$async\$nand\$array	not supported
\$async\$nor\$array	not supported
\$async\$or\$array	not supported
\$sync\$and\$array	not supported
\$sync\$nand\$array	not supported
\$sync\$nor\$array	not supported
\$sync\$or\$array	not supported
\$async\$and\$plane	not supported
\$async\$nand\$plane	not supported
\$async\$nor\$plane	not supported
\$async\$or\$plane	not supported
\$sync\$and\$plane	not supported
\$sync\$nand\$plane	not supported
\$sync\$nor\$plane	not supported
\$sync\$or\$plane	not supported
Value Change Dump (VCD) Files	
\$dumpportson \$dumpports \$dumpportsoff \$dumpportsflush \$dumpportslimit \$vcdplus	not supported

Vivado Simulator Quick Reference Guide

The following table provides a quick reference and examples for common Vivado® simulator commands.

Parsing HDL Files																					
Vivado simulator supports three HDL file types: Verilog, SystemVerilog and VHDL. You can parse the supported files using XVHDL and XVLOG commands.																					
Parsing VHDL files	<pre>xvhdl file1.vhd file2.vhd xvhdl -work worklib file1.vhd file2.vhd xvhdl -prj files.prj</pre>																				
Parsing Verilog files	<pre>xvlog file1.v file2.v xvlog -work worklib file1.v file2.v xvlog -prj files.prj</pre>																				
Parsing SystemVerilog files	<pre>xvlog -sv file1.v file2.v xvlog -work worklib -sv file1.v file2.v xvlog -prj files.prj</pre> <p>For information about the PRJ file format, see Project File (.prj) Syntax.</p>																				
Additional xvlog and xvhdl Options																					
xvlog and xvhdl Key Options	<p>See Table 15: xelab, xvhd, and xvlog Command Options for a complete list of command options. The following are key options for <code>xvlog</code> and <code>xvhdl</code>:</p> <table border="1"> <thead> <tr> <th>Key Option</th><th>Applies to:</th></tr> </thead> <tbody> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog, xvhdl</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog, xvhdl</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog, xvhdl</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog, xvhdl</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog, xvhdl</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvhdl, vlog</code></td></tr> <tr> <td>xelab, xvhd, and xvlog xsim Command Options</td><td><code>xvlog, xvhdl</code></td></tr> </tbody> </table>	Key Option	Applies to:	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvhdl, vlog</code>	xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>
Key Option	Applies to:																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvhdl, vlog</code>																				
xelab, xvhd, and xvlog xsim Command Options	<code>xvlog, xvhdl</code>																				

Elaborating and Generating an Executable Snapshot				
After parsing, you can elaborate the design in Vivado simulator using the <code>XELAB</code> command. <code>XELAB</code> generates an executable snapshot.				
You can skip the parser stage, directly invoke the <code>XELAB</code> command, and pass the PRJ file. <code>XELAB</code> calls <code>XVLOG</code> and <code>XVHDL</code> for parsing the files.				
Usage	<code>xelab top1 top2</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units are compiled in the <code>work</code> library.		
	<code>xelab lib1.top1 lib2.top2</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units have are compiled in <code>lib1</code> and <code>lib2</code> , respectively		
	<code>xelab top1 top2 -prj files.prj</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units are compiled in the <code>work</code> library. The file <code>files.prj</code> contains entries such as: <code>verilog <libraryName></code> <code><VerilogDesignFileName></code> <code>vhdl <libraryName> <VHLDDesignFileName></code> <code>sv <libraryName></code> <code><SystemVerilogDesignFileName></code>		
	<code>xelab top1 top2 -s top</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units are compiled in the <code>work</code> library. After compilation, <code>xelab</code> generates an executable snapshot with the name <code>top</code> . Without the <code>-s top</code> switch, <code>xelab</code> creates the snapshot by concatenating the unit names.		
Command Line Help and <code>xelab</code> Options	<code>xelab -help</code> Table 15: xelab, xvhd, and xvlog Command Options			
Running Simulation				
After parsing, elaboration and compilation stages are successful; <code>xsim</code> generates an executable snapshot to run simulation.				
Usage	<code>xsim top -R</code>	Simulates the design to through completion.		
	<code>xsim top -gui</code>	Opens the Vivado simulator workspace (GUI).		
	<code>xsim top</code>	Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as: <code>run -all</code> <code>run 100 ns</code>		
Important Shortcuts				
You can invoke the parsing, elaboration, and executable generation and simulation in one, two, or three stages.				
	Three Stage	<code>xvlog bot.v</code> <code>xvhdl top.vhd</code> <code>xelab work.top -s top</code> <code>xsim top -R</code>		
	Two Stage	<code>xelab -prj my_prj.prj work.top -s top</code> <code>xsim top -R</code> where <code>my_prj.prj</code> file contains: <code>verilog work bot.v</code> <code>vhdl work top.vhd</code>		
	Single Stage	<code>xelab -prj my_prj.prj work.top -s top -R</code> where <code>my_prj.prj</code> file contains: <code>verilog work bot.v vhdl work top.vhd</code>		

<p>Note: If your design contain UVM construct then you need to pass <code>-L uvm</code> to xvlog and xelab command</p>																															
Vivado Simulation Tcl Commands																															
<p>The following are commonly used Tcl commands. For a complete list, invoke following commands in the Tcl Console:</p> <pre>load_features simulator help -category simulation</pre> <p>For information on any Tcl Command, type: <code>-help < Tcl_command ></code></p>																															
<table border="1"> <tr> <td rowspan="14" style="vertical-align: top; padding-right: 10px;">Common Vivado Simulator Tcl Commands:</td><td><code>add_bp</code></td><td>Add break point at a line of HDL source.</td></tr> <tr> <td><code>add_force</code></td><td>Force the value of a signal, wire, or register to a specified value. Tcl command exampled are provided on Using Force Commands.</td></tr> <tr> <td><code>current_time now</code></td><td>Report current simulation time. See Using a -tclbatch File for an example of this command within a Tcl script.</td></tr> <tr> <td><code>current_scope</code></td><td>Report or set the current, working HDL scope. See Scope Window for more information.</td></tr> <tr> <td><code>get_objects</code></td><td>Get a list of HDL objects in one or more HDL scopes, per the specified pattern. For example command usage refer to: Example SAIF Tcl Commands.</td></tr> <tr> <td><code>get_scopes</code></td><td>Get a list of child HDL scopes. See Scope Window for more information.</td></tr> <tr> <td><code>get_value</code></td><td>Get the current value of the selected HDL object (variable, signal, wire, register). Type <code>get_value -help</code> in Tcl Console for more information.</td></tr> <tr> <td><code>launch_simulation</code></td><td>Launch simulation using the Vivado simulator.</td></tr> <tr> <td><code>remove_bps</code></td><td>Remove breakpoints from a simulation.</td></tr> <tr> <td><code>report_drivers</code></td><td>Print drivers along with current driving values for an HDL wire or signal object. Reference for more information: Using the report_drivers Tcl Command.</td></tr> <tr> <td><code>report_values</code></td><td>Print current simulated value of given HDL objects (variables, signals, wires, or registers). For example Tcl command usage, see Scope Window.</td></tr> <tr> <td><code>restart</code></td><td>Rewind simulation to post loading state (as though the design was reloaded); time is set to 0.</td></tr> <tr> <td><code>set_value</code></td><td>Set the HDL object (variable, signal, wire, or register) to a specified value. Reference for more information: Appendix I: Value Rules in Vivado Simulator Tcl Commands.</td></tr> <tr> <td><code>step</code></td><td>Step simulation to the next statement. See Stepping Through a Simulation.</td></tr> </table>			Common Vivado Simulator Tcl Commands:	<code>add_bp</code>	Add break point at a line of HDL source.	<code>add_force</code>	Force the value of a signal, wire, or register to a specified value. Tcl command exampled are provided on Using Force Commands .	<code>current_time now</code>	Report current simulation time. See Using a -tclbatch File for an example of this command within a Tcl script.	<code>current_scope</code>	Report or set the current, working HDL scope. See Scope Window for more information.	<code>get_objects</code>	Get a list of HDL objects in one or more HDL scopes, per the specified pattern. For example command usage refer to: Example SAIF Tcl Commands .	<code>get_scopes</code>	Get a list of child HDL scopes. See Scope Window for more information.	<code>get_value</code>	Get the current value of the selected HDL object (variable, signal, wire, register). Type <code>get_value -help</code> in Tcl Console for more information.	<code>launch_simulation</code>	Launch simulation using the Vivado simulator.	<code>remove_bps</code>	Remove breakpoints from a simulation.	<code>report_drivers</code>	Print drivers along with current driving values for an HDL wire or signal object. Reference for more information: Using the report_drivers Tcl Command .	<code>report_values</code>	Print current simulated value of given HDL objects (variables, signals, wires, or registers). For example Tcl command usage, see Scope Window .	<code>restart</code>	Rewind simulation to post loading state (as though the design was reloaded); time is set to 0.	<code>set_value</code>	Set the HDL object (variable, signal, wire, or register) to a specified value. Reference for more information: Appendix I: Value Rules in Vivado Simulator Tcl Commands .	<code>step</code>	Step simulation to the next statement. See Stepping Through a Simulation .
Common Vivado Simulator Tcl Commands:	<code>add_bp</code>	Add break point at a line of HDL source.																													
	<code>add_force</code>	Force the value of a signal, wire, or register to a specified value. Tcl command exampled are provided on Using Force Commands .																													
	<code>current_time now</code>	Report current simulation time. See Using a -tclbatch File for an example of this command within a Tcl script.																													
	<code>current_scope</code>	Report or set the current, working HDL scope. See Scope Window for more information.																													
	<code>get_objects</code>	Get a list of HDL objects in one or more HDL scopes, per the specified pattern. For example command usage refer to: Example SAIF Tcl Commands .																													
	<code>get_scopes</code>	Get a list of child HDL scopes. See Scope Window for more information.																													
	<code>get_value</code>	Get the current value of the selected HDL object (variable, signal, wire, register). Type <code>get_value -help</code> in Tcl Console for more information.																													
	<code>launch_simulation</code>	Launch simulation using the Vivado simulator.																													
	<code>remove_bps</code>	Remove breakpoints from a simulation.																													
	<code>report_drivers</code>	Print drivers along with current driving values for an HDL wire or signal object. Reference for more information: Using the report_drivers Tcl Command .																													
	<code>report_values</code>	Print current simulated value of given HDL objects (variables, signals, wires, or registers). For example Tcl command usage, see Scope Window .																													
	<code>restart</code>	Rewind simulation to post loading state (as though the design was reloaded); time is set to 0.																													
	<code>set_value</code>	Set the HDL object (variable, signal, wire, or register) to a specified value. Reference for more information: Appendix I: Value Rules in Vivado Simulator Tcl Commands .																													
	<code>step</code>	Step simulation to the next statement. See Stepping Through a Simulation .																													

Using Xilinx Simulator Interface

The Xilinx® Simulator Interface (XSI) is a C/C++ application programming interface (API) to the Xilinx Vivado simulator (xsim) that enables a C/C++ program to serve as the test bench for a HDL design. Using XSI, the C/C++ program controls the activity of the Vivado simulator which hosts the HDL design.

The C/C++ program controls the simulation in the following methods:

- Setting the values of the top-level input ports of the HDL design
- Instructing the Vivado simulator to run the simulation for a certain amount of simulation time

Additionally, the C/C++ program can read the values of the top-level output ports of the HDL design.

Perform the following steps to use XSI in your C/C++ program:

1. Prepare the XSI API functions to be called through dynamic linking
 2. Write your C/C++ test bench code using the API functions
 3. Compile and link your C/C++ program
 4. Package the Vivado simulator and the HDL design together into a shared library
-

Preparing the XSI Functions for Dynamic Linking

Xilinx recommends the usage of dynamic linking for indirectly calling the XSI functions. While this technique involves more steps than simply calling XSI functions directly, dynamic linking allows you to keep the compilation of your HDL design independent of the compilation of your C/C++ program. You can compile and load your HDL design at any time, even while your C/C++ program continues to run.

To call a function through dynamic linking requires your program to perform the following steps:

1. Open the shared library containing the function.
2. Look up the function by name to get a pointer to the function.

3. Call the function using the function pointer.
4. Close the shared library (optional).

Steps 1, 2, and 4 require the use of OS-specific library calls, as shown in the following table. See your operating system documentation for details about these functions.

Table 58: Operating System Specific Library Calls

Function	Linux	Windows
Open shared library	<pre>void *dlopen(const char *filename, int flag);</pre>	<pre>HMODULE WINAPI LoadLibrary(_In_ LPCTSTR lpFileName);</pre>
Look up function by name	<pre>void *dlsym(void *handle, const char *symbol);</pre>	<pre>FARPROC WINAPI GetProcAddress(_In_ HMODULE hModule, _In_ LPCSTR lpProcName);</pre>
Close shared library	<pre>int dlclose(void *handle);</pre>	<pre>BOOL WINAPI FreeLibrary(_In_ HMODULE hModule);</pre>

XSI requires you to call functions from two shared libraries: the kernel shared library and your design shared library. The kernel shared library ships with the Vivado simulator and is called `librdi_simulator_kernel.so` (Linux) or `librdi_simulator_kernel.dll` (Windows). It resides in the following directory:

```
<Vivado Installation Root>/lib/<platform>
```

where `<platform>` is `lnx64.o` or `win64.o`. Make sure to include this directory in your library path while running your program. On Linux, include the directory in the environment variable `LD_LIBRARY_PATH`, and on Windows, in the environment variable `PATH`.

Your design shared library, which the Vivado simulator creates in the course of compiling your HDL design, as described in [Preparing the Design Shared Library](#), is called `xsimk.so` (Linux) or `xsimk.dll` (Windows) and typically resides at the following location:

```
<HDL design directory>/xsim.dir/<snapshot name>
```

where `<HDL design directory>` is the directory from which your design shared library was created, and `<snapshot name>` is the name of the snapshot that you specify during the creation of the library.

Your C/C++ program will call the XSI function `xsi_open()` residing in your design shared library and all other XSI functions from the kernel shared library.

The XSI code examples that ship with the Vivado simulator consolidate the XSI functions into a C++ class called `Xsi::Loader`. The class accepts the names of the two shared libraries, internally executes the necessary dynamic linking steps, and exposes all the XSI functions as member functions of the class. Wrapping the XSI functions in this manner eliminates the necessity of calling the dynamic linking OS functions directly. You can find the source code for the class that can be copied into your own program at the following location under your Vivado installation:

```
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.h
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.cpp
```

To use `Xsi::Loader`, simply instantiate it by passing the names of the two shared libraries as shown in the following example:

```
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so",
"librdi_simulator_kernel.so");
```

Writing the Test Bench Code

A C/C++ test bench using XSI typically uses the following steps:

1. Open the design.
2. Fetch the IDs of each top-level port.
3. Repeat the following until the simulation is finished:
 - a. Set values on top-level input ports.
 - b. Run the simulation for a specific amount of time.
 - c. Fetch the values of top-level output ports.
4. Close the design.

The following table lists the XSI functions and their `Xsi::Loader` member function equivalents to use for each step. You can find the usage details for each XSI function in the [XSI Function Reference](#).

Table 59: Xsi::Loader member functions

Activity	XSI Function	Xsi::Loader Member Function
Open the design	<code>xsi_open</code>	<code>open</code>
Fetch a port ID	<code>xsi_get_port_number</code>	<code>get_port_number</code>
Set an input port value	<code>xsi_put_value</code>	<code>put_value</code>
Run the simulation	<code>xsi_run</code>	<code>run</code>

Table 59: `Xsi::Loader` member functions (cont'd)

Activity	XSI Function	<code>Xsi::Loader</code> Member Function
Fetch an output port value	<code>xsi_get_value</code>	<code>get_value</code>
Close the design	<code>xsi_close</code>	<code>close</code>

You can find the example C++ programs that use XSI in your Vivado simulator installation at the following location:

```
<Vivado Installation Root>/examples/xsim/<HDL language>/xsi
```

Compiling Your C/C++ Program

You can use the XSI example programs as a guideline. Each example supplies one or two scripts for compiling and running the example. Refer to your compiler's documentation for details on compiling a program. On Linux, compiling and running is a two-step process.

1. In a C shell, source `set_env.csh`
2. Invoke `run.csh`

On Windows, simply run the batch file `run.bat`.

Note the following from the scripts:

1. The compilation lines specify (via `-I`) the inclusion of the directory containing the `xsi.h` include file.
2. There is no mention of the design shared library or kernel shared library during the compilation of a C++ program.

The XSI include file resides at the following location:

```
<Vivado Installation Root>/data/xsim/include/xsi.h
```

Preparing the Design Shared Library

The last step for producing a working XSI-based C/C++ program involves the compilation of a HDL design and packaging it together with the Vivado simulator to become your design shared library. You may repeat this step whenever there is a change in HDL designs source code.



CAUTION! If you intend to rebuild the design shared library for your C/C++ program while your program continues to run, be sure to close the design in your program before executing this step.

Create your design shared library by invoking xelab on the HDL design and including the -dll switch to instruct xelab to produce a shared library instead of the usual snapshot for use with the Vivado simulator's user interface.

For example:

Type the following in the Linux command line to create a design shared library at `./xsim.dir/design/xsimk.so`:

```
xelab work.top1 work.top2 -dll -s design
```

where `work.top1` and `work.top2` are the top module names and `design` is the snapshot name.

See [xelab, xvhd1, and xvlog xsim Command Options](#) for more details on compiling an HDL design.

XSI Function Reference

This section presents each of the XSI API functions in plain (direct C call) and `Xsi::Loader` member function forms. The plain form functions take an `xsiHandle` argument, whereas the member functions do not take this argument. The `xsiHandle` contains state information about the opened HDL design. The plain form `xsi_open` produces the `xsiHandle`. `Xsi::Loader` contains an `xsiHandle` internally.

xsi_close

```
void xsi_close(xsiHandle design_handle);  
void Xsi::Loader::close();
```

This function closes an HDL design, freeing the memory associated with the design. Call this function to end the simulation.

xsi_get_error_info

```
const char* xsi_get_error_info(xsiHandle design_handle);  
const char* Xsi::Loader::get_error_info();
```

This function returns a string description of the last error encountered.

xsi_get_port_number

```
XSI_INT32 xsi_get_port_number(xsiHandle design_handle, const char*
port_name);
int Xsi::Loader::get_port_number(const char* port_name);
```

This function returns an integer ID for the requested top-level port of the HDL design. You may subsequently use the ID to specify the port in `xsi_get_value` and `xsi_put_value` calls. `port_name` is the name of the port and is case sensitive for Verilog and case insensitive for VHDL. The function returns -1 if no port of the specified name exists.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
```

xsi_get_status

```
XSI_INT32 xsi_get_status(xsiHandle design_handle);
int Xsi::Loader::get_status();
```

This function returns the status of the simulation. The status may be equal to one of the following identifiers:

Table 60: Xsi Simulation Status Identifiers

Status code Identifiers	Description
<code>xsiNormal</code>	No error
<code>xsiError</code>	The simulation has encountered an HDL run-time error
<code>xsiFatalError</code>	The simulation has encountered an error condition for which the Vivado simulator cannot continue.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
if (loader.get_status() == xsiError)
    printf("HDL run-time error encountered.\n");
```

xsi_get_value

```
void xsi_get_value(xsiHandle design_handle, XSI_INT32 port_number, void* value);
int Xsi::Loader::get_value(int port_number, void* value);
```

This function fetches the value of the port indicated by port ID `port_number`. The value is placed in the memory buffer to which `value` points. See [xsi_get_port_number](#) for information on obtaining an ID for a port.



IMPORTANT! Your program must allocate sufficient memory for the buffer before calling the function. See [Vivado Simulator VHDL Data Format](#) and [Vivado Simulator Verilog Data Format](#) to determine the necessary size of the buffer.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Buffer for value of port "count"
s_xsi_vlog_logicval count_val = {0X00000000, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
loader.get_value(count, &count_val);
```

xsi_open

```
typedef struct t_xsi_setup_info {
    char* logFileName;
    char* wdbFileName;
} s_xsi_setup_info, *p_xsi_setup_info;
xsiHandle xsi_open(p_xsi_setup_info setup_info);
void Xsi::Loader::open(p_xsi_setup_info setup_info);
bool Xsi::Loader::isopen() const;
```

This function opens an HDL design for simulation. To use this function, you must first initialize an `s_xsi_setup_info` struct to pass to the function. Use `logFileName` for the name of the simulation log file, or `NULL` to disable logging. If waveform tracing is on (see [xsi_trace_all](#)), `wdbFileName` is the name of the output WDB (waveform database) file. Use `NULL` for the default name of `xsim.wdb`. If the waveform tracing is off, the Vivado simulator ignores the `wdbFileName` field.



TIP: To protect your program from future changes to the XSI API, Xilinx recommends that you zero out the `s_xsi_setup_info` struct before filling in the fields, as shown in the [xsi_open](#).

The plain (non-loader) form of the function returns an `xsiHandle`, a C object containing process state information about the design, to be used with all other plain-form XSI functions. The loader form of the function has no return value. However, you may check whether the loader has opened a design by querying the `isopen` member function, which returns true if the `open` member function had been invoked.

Example

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
info.logFileName = NULL;
char wdbName[] = "test.wdb"; // make a buffer for holding the string
"test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
```

xsi_put_value

```
void xsi_put_value(xsiHandle design_handle, XSI_INT32 port_number, void*
value);
void Xsi::Loader::put_value(int port_number, const void* value);
```

This function deposits the value stored in `value` onto the port specified by port ID `port_number`. See [xsi_get_port_number](#) for information on obtaining an ID for a port. `value` is a pointer to a memory buffer that your program must allocate and fill. See the [Vivado Simulator VHDL Data Format](#) and [Vivado Simulator Verilog Data Format](#) for information on the proper format of `value`.



CAUTION! For maximum performance, the Vivado simulator performs no checking on the size or type of the value you pass to `xsi_put_value`. Passing a value to `xsi_put_value` which does not match the size and type of the port may result in unpredictable behavior of your program and the Vivado simulator.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Hard-coded Buffer for a 1-bit "1" Verilog 4-state value
const s_xsi_vlog_logicval one_val = {0X00000001, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

xsi_restart

```
void xsi_restart(xsiHandle design_handle);  
void Xsi::Loader:: restart();
```

This function resets the simulation to simulation time 0.

xsi_run

```
void xsi_run(xsiHandle design_handle, XSI_UINT64 time_ticks);  
void Xsi::Loader::run(XSI_INT64 step);
```

This function runs the simulation for the given amount of time specified in kernel precision units. A kernel precision unit is the smallest unit of time precision specified among all HDL source files of the design. For example, if a design has two source files, one of which specifies a precision of 1 ns and the other specifies a precision of 1 ps, the kernel precision unit becomes 1 ps, as that time unit is the smaller of the two.

A Verilog source file may specify the time precision using the ``timescale` directive.

Example:

```
`timescale 1ns/1ps
```

In this example, the time unit after the `/` (1 ps) is the time precision. VHDL has no equivalent of ``timescale`.

You may additionally adjust the kernel precision unit through the use of the `xelab` command-line options `--timescale`, `--override_timeprecision`, and `--timeprecision_vhdl`. See [xelab, xvhd1, and xvlog xsim Command Options](#) for information on the use of these command-line options.

Note: `xsi_run` blocks until the specified simulation run time has elapsed. Your program and the Vivado simulator share a single thread of execution.

xsi_trace_all

```
void xsi_trace_all(xsiHandle design_handle);  
void Xsi::Loader:: trace_all();
```

Call this function after `xsi_open` to turn on waveform tracing for all signals of the HDL design. Running the simulation with waveform tracing on causes the Vivado simulator to produce a waveform database (WDB) file containing all events for every signal in the design. The default name of the WDB file is `xsim.wdb`. To specify a different WDB file name, set the `wdbFileName` field of the `s_xsi_setup_info` struct when calling `xsi_open`, as shown in the example code.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so", "librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
char wdbName[] = "test.wdb"; // make a buffer for holding the string
"test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
loader.trace_all();
```

After the simulation completes, you can open the WDB file in Vivado to examine the waveforms of the signals. See [Opening a Previously Saved Simulation Run](#) for more information on how to view WDB files in Vivado.



IMPORTANT! When compiling the HDL design, you must specify `-debug all` or `-debug typical` on the `xelab` command line. The Vivado simulator will not record the waveform data without the `-debug` command line option.

Vivado Simulator VHDL Data Format

This section describes how to convert between VHDL values and the format of the memory buffers to use with the XSI functions `xsi_get_value` and `xsi_put_value`.

IEEE std_logic Type

A single bit of VHDL `std_logic` and `std_ulogic` is represented in C/C++ as a single byte (char or unsigned char). The following table shows the values of `std_logic`/`std_ulogic` and their C/C++ equivalents.

Table 61: std_logic/std_ulogic values and their C/C++ Equivalents

std_logic Value	C/C++ Byte Value (Decimal)
'U'	0
'X'	1
'0'	2
'1'	3
'Z'	4
'W'	5
'L'	6
'H'	7

Table 61: std_logic/std_ulogic values and their C/C++ Equivalents (cont'd)

std_logic Value	C/C++ Byte Value (Decimal)
'_'	8

Example code:

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : std_logic;
const char one_val = 3; // C encoding for std_logic '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

VHDL bit Type

A single bit of VHDL bit type is represented in C/C++ as a single byte. The following table shows the values of bit and their C/C++ equivalents.

Table 62: Values of bit and their C/C++ equivalents

bit Value	C/C++ Byte Value (Decimal)
'0'	0
'1'	1

Example code:

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : bit;
const char one_val = 1; // C encoding for bit '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

VHDL character Type

A single VHDL character value is represented in C/C++ as a single byte. VHDL character values are exactly identical to C/C++ char literals and are also equal to their ASCII numeric values. For example, the VHDL character value 'm' is equivalent to the C/C++ char literal 'm' or decimal value 109.

Example code:

```
// Put a 'T' on signal "myChar," where "myChar" is defined as
// signal myChar : character;
const char tVal = 'T';
int myChar = loader.get_port_number("myChar");
loader.put_value(myChar, &tVal);
```

VHDL integer Type

A single VHDL `integer` value is represented in C/C++ as an `int`.

Example code:

```
// Put 1234 (decimal) on signal "myInt," where "myInt" is defined as
// signal myInt : integer;
const int intValue = 1234;
int myInt = loader.get_port_number("myInt");
loader.put_value(myInt, &intValue);
```

VHDL real Type

A single VHDL `real` value is represented in C/C++ as a `double`.

Example code:

```
// Put 3.14 on signal "myReal," where "myReal" is defined as
// signal myReal : real;
const double doubleValue = 3.14;
int myReal = loader.get_port_number("myReal");
loader.put_value(myReal, &doubleValue);
```

VHDL Array Types

A VHDL array is represented in C/C++ as an array of whatever C/C++ type represents the element type of the VHDL array. The following table shows the examples of VHDL arrays and their C/C++ equivalent types.

Table 63: VHDL Arrays and their C/C++ Equivalent Types

VHDL Array Type	C/C++ Array Type
<code>std_logic_vector</code> (array of <code>std_logic</code>)	<code>char []</code>
<code>bit_vector</code> (array of <code>bit</code>)	<code>char []</code>
<code>string</code> (array of <code>character</code>)	<code>char []</code>
array of <code>integer</code>	<code>int []</code>
array of <code>real</code>	<code>double []</code>

VHDL arrays are organized in C/C++ with the left index of the VHDL array mapped to C/C++ array element 0 and the right index mapped to C/C++ element `<array size> - 1`.

C/C++ Array Index	0	1	2		<code><array size> - 1</code>
VHDL array(<i>left</i> TO <i>right</i>) Index	<i>left</i>	<i>left</i> + 1	<i>left</i> + 2		<i>right</i>
VHDL array(<i>left</i> DOWNTO <i>right</i>) Index	<i>left</i>	<i>left</i> - 1	<i>left</i> - 2		<i>right</i>

Example code:

```

// For the following VHDL definitions
// signal slv : std_logic_vector(7 downto 0);
// signal bv : bit_vector(3 downto 0);
// signal s : string(1 to 11);
// type IntArray is array(natural range <>) of integer;
// signal iv : IntArray(0 to 3);
// do the following assignments
//
// slv <= "11001010";
// bv <= B"1000";
// s <= "Hello world";
// iv <= (33, 44, 55, 66);
const unsigned char slvVal[] = {3, 3, 2, 2, 3, 2, 3, 2}; // 3 = '1', 2 = '0'
loader.put_value(slv, slvVal);
const unsigned char bvVal[] = {1, 0, 0, 0};
loader.put_value(bv, bvVal);
const char sVal[] = "Hello world"; // ends with extra '\0' that XSI ignores
loader.put_value(s, sVal);
const int ivVal[] = {33, 44, 55, 66};
loader.put_value(iv, ivVal);

```

Vivado Simulator Verilog Data Format

Verilog logic data is encoded in C/C++ using the following struct, defined in `xsi.h`:

```

typedef struct t_xsi_vlog_logicval {
    XSI_UINT32 aVal;
    XSI_UINT32 bVal;
} s_xsi_vlog_logicval, *p_xsi_vlog_logicval;

```

Each four-state bit of Verilog value occupies one bit position in `aVal` and the corresponding bit position in `bVal`.

Table 65: Verilog Value Mapping

Verilog Value	aVal Bit Value	bVal Bit Value
0	0	0
1	1	0
X	1	1
Z	0	1

For two-state SystemVerilog bit values, an `aVal` bit holds the bit value, and the corresponding `bVal` bit is unused. Xilinx recommends that you zero out `bVal` when composing two-state values for `xsi_put_value`.

Verilog vectors are organized in C/C++ with the right index of the Verilog vector mapped to aVal/bVal bit position 0 and the left index mapped to aVal/bVal bit position <vector size> - 1

aVal/bVal Bit Position	<vector size> to 31	<vector size> - 1	<vector size> - 2	...	1	0
Index of wire [left:right] vec (where left > right)	unused	left	left - 1	...	right + 1	right
Index of wire [left:right] vec (where left < right)	unused	left	left + 1	...	right - 1	right

For example, the following table shows the Verilog and C/C++ equivalents of the following Verilog vector.

```
wire [7:4] w = 4'bXX01;
```

Verilog Bit Index				7	6	5	4
Verilog Bit Value				X	X	0	1
C/C++ Bit Position	31	...	4	3	2	1	0
aVal Bit Value	unused	...	unused	1	1	0	1
bVal Bit Value	unused	...	unused	1	1	0	0

The C/C++ representation of a Verilog vector with more than 32 elements is an array of `s_xsi_vlog_logicval`, for which the right-most 32 bits of the Verilog vector maps to element 0 of the C/C++ array. The next 32 bits of the Verilog vector maps to element 1 of the C/C++ array, and so forth. For example, the following table shows the mapping of Verilog vector

```
wire [2:69] vec;
```

to the C/C++ array

```
s_xsi_vlog_logicval val[3];
```

Table 68: Verilog Index Range

Verilog Index Range	C/C++ Array Element
vec[38:69]	val[0]
vec[6:37]	val[1]
vec[2:5]	val[3]

Hence, `vec[2]` maps to `val[3]` bit position 3, and `vec[69]` maps to `val[0]` bit position 0.

A multi-dimensional Verilog array maps to the bits of a `s_xs_i_vlog_logicval` or `s_xs_i_vlog_logicval` array as if the Verilog array were flattened in row-major order before mapping to C/C++.

For example, the two-dimensional array

```
reg [7:0] mem[0:1];
```

is treated as if copied to a vector before mapping to C/C++:

```
reg [15:0] vec;
vec[7:0] = mem[1];
vec[8:15] = mem[0];
```

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help**→**Documentation and Tutorials**.
- On Windows, select **Start**→**All Programs**→**Xilinx Design Tools**→**DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing ([UG973](#))
2. Vivado Design Suite User Guide: System-Level Design Entry ([UG895](#))
3. Vivado Design Suite User Guide: Designing with IP ([UG896](#))
4. Vivado Design Suite User Guide: Using the Vivado IDE ([UG893](#))
5. Vivado Design Suite User Guide: Using Tcl Scripting ([UG894](#))
6. Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide ([UG953](#))
7. Vivado Design Suite Tcl Command Reference Guide ([UG835](#))
8. Vivado Design Suite User Guide: Power Analysis and Optimization ([UG907](#))
9. Vivado Design Suite User Guide: Using Constraints ([UG903](#))
10. Vivado Design Suite Tutorial: Logic Simulation ([UG937](#))
11. Vivado Design Suite User Guide: Design Flows Overview ([UG892](#))
12. Vivado Design Suite Properties Reference Guide ([UG912](#))
13. Vivado Design Suite User Guide: Synthesis ([UG901](#))
14. Writing Efficient Test Benches ([XAPP199](#))
15. IEEE Standard VHDL Language Reference Manual (IEEE-STD-1076-1993)
16. IEEE Standard Verilog Hardware Description Language(IEEE-STD-1364-2001)
17. IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language (IEEE-STD-1800-2009)
18. Standard Delay Format Specification (SDF) (IEEE-STD-1497-2004)
19. Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) (IEEE-STD-P1735)

Links to Additional Information on Third-Party Simulators

1. Questa Advanced Simulator/ModelSim simulators:
 - <http://www.mentor.com/products/fv/questa/>
 - <http://www.mentor.com/products/fv/modelsim/>
2. Synopsys VCS Simulators: <http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx>
3. Active-HDL Simulators: <https://www.aldec.com/en/support/resources/documentation/articles/1579>

-
4. Riviera PRO Simulators: <https://www.aldec.com/en/support/resources/documentation/articles/1525>

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. [Designing FPGAs Using the Vivado Design Suite 1 Training Course](#)
 2. [Designing FPGAs Using the Vivado Design Suite 2 Training Course](#)
 3. [Designing FPGAs Using the Vivado Design Suite 3 Training Course](#)
 4. [Vivado Design Suite Quick Take Video: How to use the Zynq-7000 Verification IP to verify and debug using simulation](#)
 5. [Vivado Design Suite Quick Take Video: Logic Simulation](#)
 6. [Vivado Design Suite QuickTake Video Tutorials](#)
-

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://>

www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2012-2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.