Vivado Design Suite User Guide

Logic Simulation

UG900 (v2014.3) October 1, 2014

This document applies to the following software versions: Vivado Design Suite 2014.3 and 2014.4.
# Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
</table>
| 10/01/2014 | 2014.3  | **Added:**  
Chapter 7, Compiling Simulation Libraries  
Appendix E, Direct Programming Interface (DPI) in Vivado Simulator.  
**Modified:**  
Chapter 8, Simulating with QuestaSim/ModelSim  
Chapter 9, Simulating with Cadence Incisive Enterprise Simulator (IES)  
Chapter 10, Simulating with Synopsys VCS |
| 06/04/2014 | 2014.2  | References to “early access” status of SystemVerilog in the Vivado® simulator have been removed; the feature is now fully supported.  
Added section Using the log_wave Tcl Command, page 128.  
Added Table B-3, “Supported SV and VHDL Data Types.”  
In Table D-1, noted that “Pass by reference” construct is now supported; added supported construct, “Nested interfaces.” |
| 04/23/2014 | 2014.1  | Coded example for running a post-synthesis functional simulation from the command line corrected, page 111.  
Corrected information about switches in the section Compiling Simulation Libraries for Synopsys VCS, page 162.  
Added Appendix D, SystemVerilog Constructs Supported by the Vivado Simulator (early access).  
IDS_lite is obsolete; references to it have been removed.  
General updates made to reflect changes in version 2014.1, including SystemVerilog information, updates to graphics.  
Revisions and enhancements throughout, including updates to cross references and citations; addition of instructions on locating and displaying GUI features. |

Send Feedback
# Table of Contents

## Chapter 1: Logic Simulation Overview
- Introduction ........................................................................................................... 7
- Simulation Flow ...................................................................................................... 7
- Supported Simulators ............................................................................................ 10
- Language and Encryption Support ....................................................................... 11
- OS Support and Release Changes ....................................................................... 11

## Chapter 2: Understanding Simulation Components in Vivado
- Introduction ........................................................................................................... 12
- Using Test Benches and Stimulus Files ................................................................. 13
- Using Xilinx Simulation Libraries ......................................................................... 14
- Compiling Simulation Libraries ........................................................................... 24
- Understanding the Simulator Language Option ..................................................... 25
- Using the launch_simulation Command ............................................................. 27
- Recommended Simulation Resolution ................................................................... 28
- Generating a Netlist ............................................................................................... 28
- Annotating the SDF File ....................................................................................... 30
- Using Global Reset and 3-State ............................................................................ 30
- Delta Cycles and Race Conditions ........................................................................ 32
- Using the ASYNC_REG Constraint ..................................................................... 33
- Simulating Configuration Interfaces ..................................................................... 35
- Disabling Block RAM Collision Checks for Simulation ........................................ 38
- Dumping the Switching Activity Interchange Format File for Power Analysis ..... 39

## Chapter 3: Using the Vivado Simulator from the Vivado IDE
- Introduction ........................................................................................................... 41
- Vivado Simulator Features ..................................................................................... 41
- Adding or Creating Simulation Source Files ........................................................ 42
- Running the Vivado Simulator .............................................................................. 50
- Running Post-Synthesis Simulation ...................................................................... 58
- Running Post-Implementation Simulations ............................................................ 60
- Identifying Between Multiple Simulation Runs ...................................................... 61
- Pausing a Simulation ............................................................................................. 61
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Introduction ................................................................. 65
Using Wave Configurations and Windows .......................... 65
Opening a Previously Saved Simulation Run ...................... 67
Understanding HDL Objects in Waveform Configurations ...... 68
Customizing the Waveform ............................................ 73
Controlling the Waveform Display .................................... 81
Organizing Waveforms .................................................. 83
Analyzing Waveforms ................................................... 85
Using Force Options ..................................................... 88

Chapter 5: Using Vivado Simulator Command Line and Tcl

Introduction ................................................................. 92
Compiling and Simulating a Design .................................. 92
Elaborating and Generating a Design Snapshot .................... 94
Simulating the Design Snapshot ...................................... 99
Example of Running Vivado Simulator in Standalone Mode ... 106
Project File (.prj) Syntax ............................................... 107
Predefined Macros ....................................................... 108
Library Mapping File (xsim.ini) ......................................... 108
Running Simulation Modes ............................................ 110
Using Tcl Commands and Scripts .................................... 112
Tcl Property Commands ............................................... 113

Chapter 6: Debugging a Design with Vivado Simulator

Introduction ................................................................. 117
Debugging at the Source Level ........................................ 117
Generating (forcing) Stimulus ........................................ 120
Power Analysis Using Vivado Simulator ......................... 125
Using the report_drivers Tcl Command ............................ 126
Using the Value Change Dump Feature ............................. 127
Using the log_wave Tcl Command ................................... 128
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

Introduction ................................................................. 198
Compiling C Code .......................................................... 198
Description of the xsc Compiler ........................................ 199
Binding Compiled C Code to SystemVerilog Using xelab. .......... 200
Data Types Allowed on the Boundary of C and SystemVerilog ....... 200
Mapping for User-Defined Types ........................................ 202
Support for svdpi.h functions ........................................... 204
Examples ................................................................. 204
DPI Examples Shipped with the Vivado Design Suite ................. 210

Appendix F: Additional Resources and Legal Notices

Xilinx Resources ............................................................. 211
Solution Centers ............................................................... 211
Documentation References ............................................... 211
Links to Language and Encryption Support Standards ............... 212
Training Resources ........................................................... 212
Please Read: Important Legal Notices .................................. 213
Chapter 1

Logic Simulation Overview

Introduction

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 Vivado Design Suite Integrated Design Environment (IDE) provides an integrated simulation environment when using the Vivado simulator.

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

- Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 2]
- Vivado Design Suite User Guide: Design Flows Overview (UG892) [Ref 9]

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 the 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.
Figure 1-1 illustrates the simulation flow for a typical design:

1. **RTL Design**
2. **Behavioral Simulation** (Verify Design Behaves as Intended)
3. **Synthesize**
4. **Post Synthesis Simulation**
5. **Implement (Place and Route)**
6. **Post Implementation Simulation** (Close to Emulating HW)
7. **Debug the Design**
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 at system level.

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.

... 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 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. See UNISIM Library, page 16 for more information.
Chapter 1: Logic Simulation Overview

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

Supported Simulators

The Vivado Design Suite supports the following simulators:

- Vivado simulator: Tightly integrated into the Vivado IDE, where each simulation launch appears as a framework of windows within the IDE. See Chapter 3, Using the Vivado Simulator from the Vivado IDE.
- Mentor Graphics QuestaSim/ModelSim: Integrated in the Vivado IDE. See Chapter 8, Simulating with QuestaSim/ModelSim for more information about integrated third-party simulators
- Cadence Incisive Enterprise Simulator (IES): Integrated in the Vivado IDE. See Chapter 9, Simulating with Cadence Incisive Enterprise Simulator (IES).
- Synopsys VCS and VCS MX: Integrated in the Vivado IDE. See Chapter 10, Simulating with Synopsys VCS.
- Aldec Active-HDL and Rivera-PRO*

**Note:** Aldec offers support for Aldec simulators.

Language and Encryption Support

The Vivado simulator supports:

- VHDL, Verilog, SystemVerilog (synthesizable subset) and Standard Delay Format (SDF) [Ref 15]
- IEEE standards for language and encryption. For links to related IEEE standards, see the Additional Resources section of this document, Links to Language and Encryption Support Standards, page 212.

OS Support and Release Changes

The Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973) [Ref 1] provides information about the most recent release changes, operating systems support and licensing requirements.
Chapter 2

Understanding Simulation Components in Vivado

Introduction

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

The process of simulation includes:

- Creating a test bench that reflects the simulation actions you want to run
- Selecting and declaring the libraries you need to use
- Compiling your libraries (if not using the Vivado simulator)
- Netlist generation (if performing post-synthesis or post-implementation simulation)
- Understanding the use of global reset and 3-state in Xilinx devices
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 100ns 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, see Using Global Reset and 3-State, page 30.

For more information about test benches, see Writing Efficient TestBenches (XAPP199) [Ref 4].

**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.
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 data input will be scheduled by the simulator 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.

Table 2-1 lists the Xilinx-provided simulation libraries:

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Description</th>
<th>VHDL Library Name</th>
<th>Verilog Library Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNISIM</td>
<td>Functional simulation of Xilinx primitives.</td>
<td>UNISIM</td>
<td>UNISIMS_VER</td>
</tr>
<tr>
<td>UNIMACRO</td>
<td>Functional simulation of Xilinx macros.</td>
<td>UNIMACRO</td>
<td>UNIMACRO_VER</td>
</tr>
<tr>
<td>UNIFAST</td>
<td>Fast simulation library.</td>
<td>UNIFAST</td>
<td>UNIFAST_VER</td>
</tr>
<tr>
<td>SIMPRIM</td>
<td>Timing simulation of Xilinx primitives.</td>
<td>N/A</td>
<td>SIMPRIMS_VER *</td>
</tr>
<tr>
<td>SECUREIP</td>
<td>Simulation library for both functional and timing simulation of Xilinx device features, such as the: • PCIe® IP • Gigabit Transceiver</td>
<td>SECUREIP</td>
<td>SECUREIP</td>
</tr>
</tbody>
</table>

a. The SIMPRIMS_VER is the logical library name to which the Verilog SIMPRIM physical library is mapped.

IMPORTANT:
- You must specify different simulation libraries according to the simulation points.
- There are different gate-level cells in pre- and post-implementation netlists.
Table 2-2 lists the required simulation libraries at each simulation point.

Table 2-2: Simulation Points and Relevant Libraries

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

**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.

Table 2-3 lists the library locations.

Table 2-3: Simulation Library Locations

<table>
<thead>
<tr>
<th>Library</th>
<th>HDL Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNISIM</td>
<td>Verilog</td>
<td>&lt;Vivado_Install_Dir&gt;/data/verilog/src/unisims</td>
</tr>
<tr>
<td></td>
<td>VHDL</td>
<td>&lt;Vivado_Install_Dir&gt;/data/vhdl/src/unisims</td>
</tr>
<tr>
<td>UNIFAST</td>
<td>Verilog</td>
<td>&lt;Vivado_Install_Dir&gt;/data/verilog/src/unifast</td>
</tr>
<tr>
<td></td>
<td>VHDL</td>
<td>&lt;Vivado_Install_Dir&gt;/data/vhdl/src/unifast</td>
</tr>
<tr>
<td>UNIMACRO</td>
<td>Verilog</td>
<td>&lt;Vivado_Install_Dir&gt;/data/verilog/src/unimacro</td>
</tr>
<tr>
<td></td>
<td>VHDL</td>
<td>&lt;Vivado_Install_Dir&gt;/data/vhdl/src/unimacro</td>
</tr>
<tr>
<td>SECUREIP</td>
<td>Verilog</td>
<td>&lt;Vivado_Install_Dir&gt;/data/secureip/secureip_cell.list.f.</td>
</tr>
</tbody>
</table>

The following subsections describe the libraries in more detail.
Chapter 2: Understanding Simulation Components in Vivado

**UNISIM Library**

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

**IMPORTANT:** The Vivado tools deliver IP simulation models as output products when you generate the IP. Consequently, they are not included in the precompiled libraries when you use the compile_simlib command.

**Encrypted Component Files**

Table 2-4 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. See Method 1: Using the complete UNIFAST library (Recommended), page 22 for more information.

<table>
<thead>
<tr>
<th>Component File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Vivado_Install_Dir&gt;/data/verilog/src/unisim_retarget_comp.vp</code></td>
<td>Encrypted Verilog file</td>
</tr>
<tr>
<td><code>&lt;Vivado_Install_Dir&gt;/data/vhdl/src/unisims/unisim_retarget_VCOMP.vhdp</code></td>
<td>Encrypted VHDL file</td>
</tr>
</tbody>
</table>

**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.vhdp`)
- Package files (`unisim_VPKG.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.
Chapter 2: Understanding Simulation Components in Vivado

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

See the following sections of this document for examples that use the -y switch:

- Using Verilog UNIFAST Library, page 21
- Chapter 9, Simulating with Cadence Incisive Enterprise Simulator (IES)
- Chapter 10, Simulating with Synopsys VCS

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

UNIMACRO Library

The UNIMACRO library is used during functional simulation and contains macro descriptions for selective 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 All Programmable SoC Libraries Guide (UG953) [Ref 5].

VHDL UNIMACRO Library

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

```
library UNIMacro;
use UNIMacro.Vcomponents.all;
```
Chapter 2: Understanding Simulation Components in Vivado

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`, and module `IBUF` is `IBUF.v`. Ensure that UNISIM 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) [Ref 6].

Specify this library as follows:

```
-L SIMPRIMS_VER
```

Where:

- `-L` is the library specification command.
- `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) [Ref 16]. 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.

Table 2-5 lists special considerations that must be arranged with your simulator vendor for using these libraries.

<table>
<thead>
<tr>
<th>Simulator Name</th>
<th>Vendor</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelSim SE</td>
<td>Mentor Graphics</td>
<td>If design entry is in VHDL, a mixed language license or a SECUREIP OP is required. Contact the vendor for more information.</td>
</tr>
<tr>
<td>ModelSim PE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ModelSim DE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuestaSim</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCS and VCS MX</td>
<td>Synopsys</td>
<td></td>
</tr>
<tr>
<td>Active-HDL</td>
<td>Aldec</td>
<td>If design entry is VHDL only, a SECUREIP language-neutral license is required. Contact the vendor for more information.</td>
</tr>
<tr>
<td>Riviera-PRO*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IMPORTANT:** See *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) [Ref 1] 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;
use 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
```

You can use the Verilog SECUREIP library at compile time by using `-f` switch. The file list is available in the `<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 run time.

**IMPORTANT:**
This model cannot be used for timing-driven simulations. 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 running 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.

**GTXE2_CHANNEL/GTXE2_COMMON**

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

• GTX links must be 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.

**Using Verilog UNIFAST Library**

There are two methods of simulating with the UNIFAST models.

• Method 1 is the recommended method whereby you simulate with all the UNIFAST models.
• Method 2 is for more advanced users to determine which modules to use with the UNIFAST models.

The following subsections describe these simulation methods.
Method 1: Using the complete UNIFAST library (Recommended)

To enable UNIFAST support in a Vivado project environment for the Vivado simulator or ModelSim, check the Enable fast simulation models box, as shown in Figure 2-1.

See the Encrypted Component Files, page 16 for more information regarding component files.

For IES and VCS simulators, point to the UNIFAST library using:

```bash
-y ../verilog/src/unifast
```

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

Method 2: Using specific UNIFAST modules

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;)

See the figures and diagrams for more details.
Chapter 2: Understanding Simulation Components in Vivado

- 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 gets added to hierarchy name. For example: `instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.MMCME2;` - VSIM

**Example config.v**

```vhd
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:

```vhdl
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 01 inst:DSP48E1;
    ........use entity unifast.DSP48E1;
    ........end for;
    ......end for;
    ..end for;
end cfg_xilinx;
```

Compiling Simulation Libraries

**IMPORTANT:**

*If you are using the Vivado simulator, there is no need to compile the simulation libraries. Compile the simulation libraries only if you are not using the Vivado simulator.*

*If you are not using the Vivado simulator are therefore compiling the simulation libraries: Because the Vivado simulator has precompiled libraries, it is not necessary for you to identify the library locations.*

Before you can simulate your design using a third-party simulation tool, you must compile the libraries and map the logical library names to the physical library location.

Xilinx provides the `compile_simlib` Tcl command to automate that task (see also, Chapter 7, Compiling Simulation Libraries). This Tcl command provides many options including: target simulation tool, language, architecture, and more. For complete details on this command, please see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 6] or use `-help` on the Tcl command line.

To compile Xilinx HDL-based simulation libraries for third-party simulation vendors, use the following Tcl command:

```tcl
compile_simlib
```

Examples of `compile_simlib` Tcl commands include:
compile_simlib -simulator modelsim

Libraries are typically compiled (or recompiled) anytime a new simulator version is installed, when you update to a new version of the Vivado IDE, or when any library source files are modified (either by you or by a software patch).

Changing compile_simlib Defaults

For a complete description of compiler defaults and settings, see Chapter 7, Compiling Simulation Libraries.

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 (Figure 3-3, page 45 shows the location at which you can set the simulator 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. Click Window > 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 Figure 2-2) opens.
Table 2-6 illustrates the function of the simulator_language property.

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

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]
Chapter 2: Understanding Simulation Components in Vivado

Using the launch_simulation Command

The launch_simulation command lets you run different simulators in script mode.

The syntax of launch_simulation is as follows:

```bash
[-scripts_only] [-of_objects <args>] [-absolute_path <arg>]
[-install_path <arg>] [-noclean_dir] [-quiet] [-verbose]
```

Table 2-7 describes the options of launch_simulation.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-simset]</td>
<td>Name of the simulation fileset.</td>
</tr>
<tr>
<td>[-type]</td>
<td>Netlist type. Values: functional, timing. This is only applicable when the mode is set to post-synthesis or post-implementation.</td>
</tr>
<tr>
<td>[-scripts_only]</td>
<td>Only generate scripts.</td>
</tr>
<tr>
<td>[-of_objects]</td>
<td>Generate compile order file for this object (applicable with -scripts_only option only)</td>
</tr>
<tr>
<td>[-absolute_path]</td>
<td>Make all file paths absolute with respect to the reference directory.</td>
</tr>
<tr>
<td>[-install_path]</td>
<td>Custom installation directory path.</td>
</tr>
<tr>
<td>[-noclean_dir]</td>
<td>Do not remove simulation run directory files.</td>
</tr>
<tr>
<td>[-quiet]</td>
<td>Ignore command errors.</td>
</tr>
<tr>
<td>[-verbose]</td>
<td>Suspend message limits during command execution.</td>
</tr>
</tbody>
</table>

Examples

- **Running behavioral simulation using vivado_simulator**
  ```bash
  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 QuestaSim.**
  ```bash
  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 ModelSim [current_project]
set_property compxlib.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.compiled_library_dir <compiled_library_location>
[current_project]
launch_simulation -mode post-synthesis -type functional

- Running post-implementation timing simulation using Cadence IUS

set_property target_simulator IES [current_project]
set_property compxlib.compiled_library_dir <compiled_library_location>
[current_project]
launch_simulation -mode post-implementation -type timing

Recommended Simulation Resolution

**IMPORTANT:** Run simulations using a time resolution of 1 ps. Some Xilinx primitive components, such as MMCM, require a 1 ps resolution to work properly in either functional or timing simulation.

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

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.

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
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.
Chapter 2: Understanding Simulation Components in Vivado

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

- **Tcl Command:**
  
  ```tcl
  write_verilog -mode timesim -sdf_anno true <Verilog_Netlist_Name>
  ```

## Annotating the SDF File

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

**RECOMMENDED:** 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

## 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.

**RECOMMENDED:** 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.

### 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

*Figure 2-3* shows how Global 3-State (GTS) and Global Set/Reset (GSR) signals are used in an FPGA.

![Built-in FPGA Initialization Circuitry Diagram](image)

**Figure 2-3:** 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 glbl.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**

```vhdl
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:

```vhdl
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.

**RECOMMENDED:** 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) [Ref 7].

### 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.

### Correcting X-Generation

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 the method described in Using the `ASYNC_REG` Constraint, page 33.
Chapter 2: Understanding Simulation Components in Vivado

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 glbl 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 model (SIM_CONFIGE2) with an instantiation template allows 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.

Table 2-8 lists the supported interfaces and devices.

**Table 2-8: Supported Configuration Devices and Modes**

<table>
<thead>
<tr>
<th>Devices</th>
<th>SelectMAP</th>
<th>Serial</th>
<th>SPI</th>
<th>BPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Series and Zynq®-7000 AP SoC Devices</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The model handles 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. Figure 2-4, below, 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 2-4: Block Diagram of Model Interaction**

### System Level Description

The SIM_CONFIGE2 model allows 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.
Chapter 2: **Understanding Simulation Components in Vivado**

- 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 ZIP file associated with this model is located at:

http://www.xilinx.com/txpatches/pub/documentation/misc/config_test_bench.zip

The 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.

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 `SIM_CONFIGE2` model also demonstrates 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**

The `SIM_CONFIGE2` 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 `SIM_CONFIGE2` model also allows 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>
<thead>
<tr>
<th>Slave SelectMAP and Serial Features</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master mode</td>
<td>No</td>
</tr>
<tr>
<td>Daisy chain - slave parallel daisy chains</td>
<td>No</td>
</tr>
<tr>
<td>SelectMAP data loading</td>
<td>Yes</td>
</tr>
<tr>
<td>Continuous SelectMAP data loading</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-continuous SelectMAP data loading</td>
<td>Yes</td>
</tr>
<tr>
<td>SelectMAP ABORT</td>
<td>Yes</td>
</tr>
<tr>
<td>SelectMAP reconfiguration</td>
<td>No</td>
</tr>
<tr>
<td>SelectMAP data ordering</td>
<td>Yes</td>
</tr>
<tr>
<td>Reconfiguration and MultiBoot</td>
<td>No</td>
</tr>
<tr>
<td>Configuration CRC – CRC checking during configuration</td>
<td>Yes</td>
</tr>
<tr>
<td>Configuration CRC – post-configuration CRC</td>
<td>No</td>
</tr>
</tbody>
</table>

**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.

Table 2-10 shows the string options you can use with `SIM_COLLISION_CHECK` to control simulation behavior in the event of a collision.

<table>
<thead>
<tr>
<th>String</th>
<th>Write Collision Messages</th>
<th>Write Xs on the Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>WARNING_ONLY</td>
<td>Yes</td>
<td>No. Applies only at the time of collision. Subsequent reads of the same address space could produce Xs on the output.</td>
</tr>
<tr>
<td>GENERATE_X_ONLY</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>None</td>
<td>No</td>
<td>No. Applies only at the time of collision. Subsequent reads of the same address space could produce Xs on the output.</td>
</tr>
</tbody>
</table>

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

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.

See the information about the respective simulator for more detail:

- Vivado simulator: [Power Analysis Using Vivado Simulator, page 125](#)
- Dumping SAIF in QuestaSim/ModelSim, page 148
Chapter 2: Understanding Simulation Components in Vivado

- Running Timing Simulation Using IES, page 158
- Dumping SAIF for Power Analysis for VCS, page 170
Chapter 3

Using the Vivado Simulator from the Vivado IDE

Introduction

This chapter describes the Vivado® simulator features, which are available in the Vivado Integrated Design Environment (IDE), allowing you to do pushbutton waveform tracing and debug.

The Vivado simulator is a Hardware Description Language (HDL) event-driven simulator that supports functional and timing simulations for VHDL, Verilog, and mixed VHDL/Verilog designs. See the “Important” note on page 59 for timing simulation support information.


Vivado Simulator Features

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) use
- Single-click simulation re-compile and re-launch
- One-click compilation and simulation
- Built-in support for Xilinx® simulation libraries
- Real-time waveform update
## 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**.

   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:

   - **Specify Simulation Set**: Enter the name of the simulation set in which to store simulation sources (the default is `sim_1`, `sim_2`, and so forth).

     You can select the Create Simulation Set command from the drop-down menu to define a new simulation set. When more than one simulation set is available, the Vivado simulator shows which simulation set is the **active** (currently used) set.

     For a demonstration of this feature, see the Vivado Design Suite Quick Take Video: Logic Simulation.

   - **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 the Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 2] 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 Selected File Up**: Moves the file up in the list order.
     - **Move Selected File 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.

## 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, select `Simulation Sources > Edit Simulation Sets`, as shown in `Figure 3-1`.

![Figure 3-1: Edit Simulation Sets Option](image)

The Add or Create Simulation Sources wizard opens.
2. From the Add or Create Simulation Sources wizard, select Add Files.

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.

**Using Simulation Settings**

The **Flow Navigator > Simulation > Simulation Settings** section lets you configure the simulation settings in Vivado IDE. The Flow Navigator Simulation section is shown in **Figure 3-2**.

- Simulation Settings: Opens the Simulation Settings dialog box where you can select and configure the Vivado simulator.

- Run Simulation: Sets up the command options to compile, elaborate, and simulate the design based on the simulation settings, then launches the Vivado simulator. When you run simulation prior to synthesizing the design, the Vivado simulator runs a behavioral simulation, and opens a waveform window, (see **Figure 3-13, page 55**) that shows the HDL objects with the signal and bus values in either digital or analog form.

  At each design step (both after you have successfully synthesized and after implementing the design) you can run a functional simulation and timing simulation.

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

---

**TIP:** *This command has a -scripts_only option that writes a script to run the Vivado simulator.*
Chapter 3: Using the Vivado Simulator from the Vivado IDE

Vivado Simulator Simulation Options

In the Flow Navigator, click **Simulation Settings** to open the Project Settings dialog box, shown in Figure 3-3.

![Project Settings Dialog Box, Simulation Options](image)

*Figure 3-3: Project Settings Dialog Box, Simulation Options*
Table 3-1 shows the simulation options.

**Table 3-1: Simulation Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Simulator</td>
<td>Displays the selected simulation tool</td>
</tr>
<tr>
<td>Simulator Language</td>
<td>Displays the simulator language used</td>
</tr>
<tr>
<td>Simulation set</td>
<td>Sets the simulation to be performed</td>
</tr>
<tr>
<td>Simulation top module name</td>
<td>Top design unit name</td>
</tr>
<tr>
<td>Clean up simulation files</td>
<td>By default checked. Cleans simulation files before re-run.</td>
</tr>
<tr>
<td>Generate scripts only</td>
<td>Do not run simulation. Generates scripts only.</td>
</tr>
<tr>
<td>Compiled library location</td>
<td>Path where library compiled using <code>compile_simlib</code> command has been kept.</td>
</tr>
</tbody>
</table>

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

**IMPORTANT:** Because the Vivado simulator has precompiled libraries, it is not necessary to identify the library locations

**Selecting Compilation Options**

The Compilation tab options are shown in Figure 3-4.

*Figure 3-4: Compilation Tab*
Table 3-2 describes the compilation options.

**Table 3-2: Compilation Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog option</td>
<td>Browse to set Verilog include path and to define macro</td>
</tr>
<tr>
<td>Generics/Parameters options</td>
<td>Sets the generic/parameter value</td>
</tr>
<tr>
<td>xsim.compile.xvlog.nosort</td>
<td>Do not sort Verilog file during compilation</td>
</tr>
<tr>
<td>xsim.compile.xvhdl.nosort</td>
<td>Do not sort VHDL file during compilation</td>
</tr>
<tr>
<td>xsim.compile.xvlog.more_options</td>
<td>Extra xvlog options if you prefer</td>
</tr>
<tr>
<td>xsim.compile.xvhdl.more_options</td>
<td>Extra xvhdl options if you prefer</td>
</tr>
</tbody>
</table>

**Selecting Elaboration Options**

Figure 3-5 shows the Elaboration tab.

**Figure 3-5: Elaboration Tab**

Table 3-3 describes the elaboration options.

**Table 3-3: Elaboration Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsim.elaborate.snapshot</td>
<td>Specifies the simulation snapshot name</td>
</tr>
<tr>
<td>xsim.elaborate.debug_level</td>
<td>Choose the debug level. By default it is “typical”</td>
</tr>
<tr>
<td>xsim.elaborate.relax</td>
<td>Do relax check on language semantic</td>
</tr>
<tr>
<td>xsim.elaborate.mt_level</td>
<td>Specify number of sub-compilation jobs to run in parallel</td>
</tr>
<tr>
<td>xsim.elaborate.load_glbl</td>
<td>Adds glbl with top module</td>
</tr>
<tr>
<td>xsim.elaborate.rangecheck</td>
<td>Enables run time value range check for VHDL</td>
</tr>
</tbody>
</table>
Chapter 3: Using the Vivado Simulator from the Vivado IDE

Table 3-3: Elaboration Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsim.elaborate.sdf_delay</td>
<td>Specifies sdf timing delay type to be read for use in timing simulation</td>
</tr>
<tr>
<td>xsim.elaborate.unifast</td>
<td>Enables fast simulation model</td>
</tr>
<tr>
<td>xsim.elaborate.xelab.more_option</td>
<td>Specifies extra option for xelab</td>
</tr>
</tbody>
</table>

Selecting Simulation Options

Figure 3-6 shows the Simulation tab options.

Table 3-4 describes the Simulation tab options.

Table 3-4: Simulation Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsim.simulate.runtime</td>
<td>Specifies simulation run time for the Vivado simulator. Enter blank to load just the simulation snapshot and wait for user input.</td>
</tr>
<tr>
<td>xsim.simulate.uut</td>
<td>Specifies the instance name for the design under test (default: /uut)</td>
</tr>
<tr>
<td>xsim.simulate.wdb</td>
<td>Specifies Waveform database file</td>
</tr>
<tr>
<td>xsim.simulate.saif</td>
<td>Specifies SAIF file name</td>
</tr>
<tr>
<td>xsim.simulate.xsim.more_option</td>
<td>More Vivado simulator simulation options</td>
</tr>
</tbody>
</table>

Figure 3-6: Simulation Tab
**Specify Netlist Options**

Table 3-7 shows the Netlist tab options.

![Netlist Tab](image)

**Figure 3-7: Netlist Tab**

Table 3-5 describes the Netlist options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sdf_anno</td>
<td>A check box is available to select the -sdf_anno command. This option is enabled by default</td>
</tr>
<tr>
<td>-process_corner</td>
<td>You can specify the -process_corner as fast or slow</td>
</tr>
</tbody>
</table>

**Selecting Advanced Simulation Options**

The Advanced view is shown in Figure 3-8.

**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.
Running the Vivado Simulator

From the Flow Navigator, select **Run Simulation** to invoke the Vivado simulator workspace, shown in **Figure 3-9**.

The main components of the Vivado simulator workspace are:

1. Main Toolbar
2. Run Menu
3. Objects Window
4. Simulation Toolbar
5. Wave Objects
6. Waveform Window
7. Scopes Window
8. Sources Window
Chapter 3: Using the Vivado Simulator from the Vivado IDE

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.
Chapter 3: Using the Vivado Simulator from the Vivado IDE

The Run menu for simulation is shown in Figure 3-10.

The Vivado simulator Run menu options:

- **Restart**: Lets you restart an existing simulation from 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>`

- **Step**: Runs the simulation up to the next HDL source line.

- **Break**: Lets you interrupt a running simulation.

- **Delete All Breakpoints**: Deletes all breakpoints.

- **Relaunch Simulation**: Recompiles the simulation files and relaunches the run.
Chapter 3: Using the Vivado Simulator from the Vivado IDE

Objects Window

The HDL Objects window displays the HDL objects in the design, as shown in Figure 3-11.

![HDL Objects Window](image)

**Figure 3-11:** HDL Objects Window

Buttons beside the HDL objects show the language or process type. This view lists the Name, Value, and Block Type of the simulation objects.

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

```
get_value <hdl_object>
```

Table 3-6 briefly describes the buttons at the top of the Objects window as follows:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Search" /></td>
<td>The <strong>Search</strong> button, when selected, opens a field in which you can enter an object name on which to search.</td>
</tr>
<tr>
<td><img src="image" alt="Input" /></td>
<td>Input signal.</td>
</tr>
<tr>
<td><img src="image" alt="Output" /></td>
<td>Output signal.</td>
</tr>
<tr>
<td><img src="image" alt="Input/Output" /></td>
<td>Input/Output signal.</td>
</tr>
<tr>
<td><img src="image" alt="Internal" /></td>
<td>Internal signal.</td>
</tr>
<tr>
<td><img src="image" alt="Constant" /></td>
<td>Constant signal.</td>
</tr>
<tr>
<td><img src="image" alt="Variable" /></td>
<td>Variable signal.</td>
</tr>
</tbody>
</table>

**TIP:** Hover over the HDL Object buttons for tool tip descriptions.
Simulation Toolbar

When you run the Vivado simulator, the simulation-specific toolbar opens along with the Vivado IDE toolbars, and displays simulation-specific buttons display for ease-of-use.

When you hover your mouse over the toolbar buttons, a tool tip describes the button function. The buttons are also labeled in Figure 3-10, page 52.

Wave Objects

The Vivado IDE waveform window is common across a number of Vivado Design Suite tools. An example of the wave objects in a waveform configuration is shown in Figure 3-12.

The waveform 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 wave objects. The waveform portion of the waveform window displays additional items for time measurement, that include: cursors, markers, and timescale rulers.

The Vivado IDE traces the HDL object in the waveform configuration during simulation, and you use the wave configuration to examine the simulation results.
The design hierarchy and the waveforms are not part of the wave configuration, and are stored in a separate WDB database file.

See Chapter 4, Analyzing with the Vivado Simulator Waveforms for more information about using the waveform.

Waveform Window

When you invoke the simulator, by default, it opens a waveform window that displays a new wave configuration consisting of the traceable top-module of HDL objects in the simulation as shown in Figure 3-13.

TIP: Displaying the Waveform Window: on closing and reopening a project, you must rerun simulation to invoke the Waveform Window.
To add an individual HDL object or set of objects to the waveform window, you can right-click the object and select the **Add to Wave Window** option from the context menu.

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 Scopes window.

### Saving a Waveform

The new wave configuration is not saved to disk automatically. Select **File > Save Waveform Configuration As** and supply a file name to produce 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.

### Creating and Using Multiple Waveform Configurations

In a simulation session, you can create and use multiple wave configurations, each in its own waveform window. When you have more than one waveform 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 waveform window upon which commands external to the window apply. For example: **HDL Objects > Add to Wave Window**.

You can set a different waveform window to be the **active window** by clicking the title of the window. See **Identifying Between Multiple Simulation Runs**, page 61 and **Creating a New Wave Configuration**, page 66 for more information.
Chapter 3: Using the Vivado Simulator from the Vivado IDE

Scopes Window

Figure 3-14 shows the Scopes Window, where you can view and filter HDL objects by type using the filter buttons at the top of the window. Hover over a button for a tool tip description of what object type the filter button represents.

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 Figure 3-15.

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.
Chapter 3: Using the Vivado Simulator from the Vivado IDE

You can also open a source file by right-clicking on the object and selecting the Go to Source Code option.

Additional Scopes and Sources Options

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

See Using the Scopes Window, page 69 for more detail on how to use the Scopes window in the Vivado simulator. As an equivalent to using the Scopes and Objects windows, you can navigate the HDL design by typing the following in the Tcl Console:

```
get_scopes
current_scope
report_scopes
report_values
```

TIP: To access source files for editing, you can open files from the Scopes or Objects window by selecting Go to Source Code, as shown in Figure 3-16.

![Objects Context Menu](image)

Figure 3-16: Objects Context Menu

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

Running Post-Synthesis Simulation

You can select which one of the following options Post-Synthesis and Post-Implementation timing simulation includes:

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

Post-Synthesis timing simulation uses the estimated timing numbers after synthesis.

You use Post-Implementation timing simulation after your design is completely through the implementation (Place and Route) process in Vivado IDE. You can now begin to observe how your design behaves in the actual circuit.
You defined the overall functionality of the design in the beginning; when the design is implemented, accurate timing information is available.

The Vivado IDE creates the netlist and SDF by calling the netlist writer (write_verilog with the -mode timesim switch and the SDF annotator (write_sdf)), then sends the generated netlist to the target simulator.

You control these options using Simulation Settings as described in Using Simulation Settings, page 44.

**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) [Ref 6].

**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

After successfully run synthesis on your design, you can run a Post-Synthesis simulation (Functional or Timing).

**Running Post-Synthesis Functional Simulation**

When synthesis is run successfully, the **Run Simulation > Post-Synthesis Functional Simulation** option becomes available, as shown in **Figure 3-17**.

![Figure 3-17: Run Post-Synthesis Functional Simulation](image)

After synthesis, the simulation information is much more complete, so you can get a better perspective on how the functionality of your design is meeting your requirements. After you select a post-synthesis functional simulation, the functional netlist is generated and the UNISIM libraries are used for simulation.
Running Post-Synthesis Timing Simulation

When synthesis is run successfully, the Run Simulation > Post-Synthesis Timing Simulation option becomes available, as shown in Figure 3-18.

![Figure 3-18: Run Post-Synthesis Timing Simulation](image)

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.

Running Post-Implementation Simulations

After you have run implementation on your design you can run a post-implementation functional or timing simulation.

Running Post-Implementation Functional Simulations

When implementation is successful, the Run Simulation > Post-Implementation Functional Simulation option is available, as shown in Figure 3-19.

![Figure 3-19: Run Post-Implementation Functional Simulation](image)

After implementation, the simulation information is much more complete, so you can get a better perspective on how the functionality of your design is meeting your requirements.

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

When post-implementation is successful, the **Run Simulation > Post-Implementation Timing Simulation** option is available, as shown in Figure 3-20.

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.

Identifying 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 Figure 3-21.

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**, page 118 for more information.
Chapter 3: Using the Vivado Simulator from the Vivado IDE

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/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.

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 4, Analyzing with the Vivado Simulator Waveforms.

Closing 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.

To close a simulation from the Tcl Console, type:

```
close_sim
```

The command first checks for unsaved wave configurations. If any exist, the command issues an error.

Adding a post.tcl Batch File

You can add additional commands to be run after you have created a project and simulated a design. To do so:

1. Create a Tcl file with the simulation commands you want to add to the simulation source files. For example, create a file that adds more time to a simulation originally run for 1,000 ns:
   ```
   run 5us
   ```

2. Name the file post.tcl, and place it in an available location.

3. Use the Add Sources button 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 Figure 3-22.

![Figure 3-22: Using the post.tcl File in a Design](image)

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 and the resulting simulation runs for the additional time.

---

**Skipping Compilation or Simulation**

You can skip the compilation through `xelab` and or simulation through the Vivado simulator, as follows:

```
set_property skip_compilation 1 [get_filesets sim_1]
```

The Vivado tools skip the compilation step of Vivado simulator and runs simulation with existing compiled result.

**Note:** Any change to design files after the last compilation is not reflected in simulation when you set this property.

```
set_property skip_simulation 1 [get_filesets sim_1]
```

The Vivado tools skip the execution of simulation step.
Chapter 3: Using the Vivado Simulator from the Vivado IDE

Viewing Simulation Messages

The Vivado IDE contains a message area where you can view informational, warning, and error messages. As shown in Figure 3-23, messages from the Vivado simulator contain an issue description and a suggested resolution.

![Figure 3-23: Simulator Message Description and Resolution Information](image)

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

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

An example of such a command is as follows:

```tcl
help -message {simulator 43-3120}
```
Chapter 4

Analyzing with the Vivado Simulator Waveforms

Introduction

In the Vivado® simulator workspace, you can work with the waveform to analyze your design and debug your code. The simulator populates design data in other areas of the workspace, such as the Objects and the Scopes 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 Testbenches (XAPP199) [Ref 4].

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 Scopes 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

Although both a wave configuration and a WCFG file refer to the customization of lists of waveforms, there is a conceptual difference between them:

- The wave configuration is an object that is loaded into memory with which you can work.
- The WCFG file is the saved form of a wave configuration on disk.

A wave configuration can have a name or be untitled. The name shows on the title bar of the wave configuration window.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Creating a New Wave Configuration

Create a new waveform configuration for displaying waveforms as follows:

1. Select File > New Waveform Configuration.

   A new waveform window opens and displays a new, untitled waveform configuration.

   add_wave <HDL_Object>.

2. Add HDL objects to the waveform configuration using the steps listed in Understanding HDL Objects in Waveform Configurations, page 68.

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

   See Chapter 3, Using the Vivado Simulator from the Vivado IDE for more information about creating new waveform configurations. Also see Creating and Using Multiple Waveform Configurations, page 56 for information on multiple waveforms.

Opening a WCFG File

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

1. Select File > Open Waveform Configuration.

   The Specify Simulation Results 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 waveform 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

To save a wave configuration to a WCFG file, select File > Save Waveform Configuration As, and type a name for the waveform configuration.

   Tcl command: save_wave_config <waveform_name>
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Opening a Previously Saved Simulation Run

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

Interactive Method

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

When you run a simulation and display HDL objects in a waveform 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.

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.

**IMPORTANT:** Currently, you cannot open a WDB file created in a previous version of the Vivado Design Suite. However, starting in version 2014.3, cross-operating system compatibility has been enabled. That is, a WDB file created in one operating system will work in other operating systems. WCFG files are both backward and cross-OS compatible.

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. As the simulator creates no waveform configuration by default, you must create a new waveform configuration or open an existing WCFG file.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

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 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 background of the icon indicates whether the object is a scalar or a compound such as a Verilog vector or VHDL record.

Figure 4-1 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, for which you can specify the text to display.
- Value: Displays the value of the object at the time indicated in the main cursor of the waveform window. You can change the formatting 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.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Using the Scopes Window

Figure 4-2 shows the Vivado simulator Scopes window.

You can filter scopes within the Scopes window using one of the following methods:
Chapter 4: Analyzing with the Vivado Simulator Waveforms

- To hide certain types of scope from display, click one or more scope-filtering buttons.
- To limit the display to scopes containing a specified string, click the Search button and type the string in the text box.

You can filter object within the Scopes window by clicking a scope. When you have selected an scope, the Scopes popup menu provides the following options:

- Add to Wave Window: Adds all viewable HDL objects of the selected scope to the waveform configuration.
  
  Alternately, you can drag and drop the objects from the Objects window to the Name column of the waveform window.

**IMPORTANT:** Waveforms for an object show only from the simulation time when the object was added to the window. 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 module and VHDL entity instances, opens the source code at the point of instantiation for the selected instance.

In the source code text editor, you can hover over an identifier in a file to get the value, as shown in Figure 4-3.

**IMPORTANT:** You must have the correct scope in the Scopes window selected to use this feature.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Figure 4-3: Source Code with Identifier Value Displayed
Using the Objects Window

Figure 4-4 shows the Vivado simulator Objects window.

![Objects Window](image)

You can hide certain types of HDL object from display by clicking one or more object-filtering buttons. Hover over the button for a tool tip describing what object type it represents.

When you have selected an object, the Objects popup menu provides the following options:

- **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 waveform window.

- **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.

- **Go To Source Code**: Open the source code at the definition of the selected object.

**TIP**: Some HDL objects cannot be viewed as a waveform, such as: Verilog-named events, Verilog parameters, VHDL constants, and objects with more elements than the max traceable size. The default max traceable size is 65536 and can be overridden by setting the `trace_limit` property to the current simulation run: `set_property TRACE_LIMIT <value> [current_sim]` Note that the property must be applied to the current simulation runs, so the simulation session must be open.
Customizing the Waveform

The following subsections describe the options available to customize a waveform.

Using Analog Waveforms

The following subsections describe the features and requirements around 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 radix causes 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. Open the Real Settings dialog box, shown in Figure 4-5.
2. In the waveform configuration window, select an HDL object, and right-click to open the popup menu.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

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 Options Dialog Box, page 80, 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:

1. In the name area of a waveform window, right-click a bus to open the popup menu.
2. Select Waveform Style >:
   - Analog: Sets a Digital waveform to Analog.
   - Digital: Sets an Analog waveform object to Digital.
   - Analog Settings: Figure 4-6 shows the Analog Settings dialog box with the settings for analog waveform drawing.
The Analog Settings dialog box options are:

- **Row Height**: Specifies how tall to make the selected 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).

- **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 radix of the wave object radix is integral, 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 waveform window itself.

• Horizontal Line: Specifies whether to draw a horizontal rule at the given value. If the check-box 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 integral.

IMPORTANT: Zoom settings are not saved with the wave configuration.

About Radixes

Understanding the type of data on your bus is important. You need to recognize the relationship between the radix setting and the data type to use the waveform options of Digital and Analog effectively. See Displaying Waveforms as Analog, page 75 for more information about the radix setting and its effect on Analog waveform analysis.

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 default waveform radix defaults to binary.

To change the default waveform radix:

1. In the waveform window sidebar, click the Waveform Options button to open the waveform options view.

2. On the General page, click the Default Radix drop-down menu.

3. From the drop-down list, select a radix.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Changing the Radix on Individual Wave Objects

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

1. Select a bus in the Objects window.
2. Select Radix and the format you want from the drop-down menu:
   - Binary (default)
   - Hexadecimal
   - Unsigned Decimal
   - Signed Decimal
   - Octal
   - ASCII

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

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

Where <radix> is one the following: bin, unsigned, hex, dec, ascii, or oct.

**IMPORTANT:** Changes to the radix of an item in the Objects window do not apply to values in the waveform window or the Tcl Console. To change the radix of an individual waveform object in the waveform window, use the waveform window popup menu.

Waveform Object Naming Styles

There are options for renaming objects, viewing object names, and change 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. Select Rename from the popup menu.

   The Rename dialog box opens.

3. Type the new name in the Rename dialog box, and click OK.

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


**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 in the next section.

---

**Changing the Object Name Display**

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. Select **Name >:**
   - **Long** to display the full hierarchical name.
   - **Short** to display the name of the signal or bus only.
   - **Custom** to display the custom name given to the signal when renamed. See **Renaming Objects**, page 78.

**Note:** 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.
Using the Waveform Options Dialog Box

Select the **Waveforms Options** button to open the Waveform Options dialog box, shown in **Figure 4-7**.

**Figure 4-7: Waveform Options Dialog Box**

The General Waveform Options are:

- **Default Radix**: Sets the numerical format to use for newly-created design wave objects.
- **Draw Waveform Shadow**: Creates a shaded representation of the waveform.
- **Show signal indices**: check box 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 trigger markers**: Has not effect on Vivado Simulator waveforms.
- **The Colors page lets you set colors of items within the waveform.**
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Controlling the Waveform Display

You can control the waveform display using:

- Zoom feature buttons in the HDL Objects window sidebar
- Zoom combinations with the mouse wheel
- Vivado IDE Y-Axis zoom gestures

Note: In contrast to other Vivado Design Suite graphic windows, zooming in a waveform 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.

Using the Zoom Feature Button

You have zoom functions as sidebar buttons to zoom in and out of a wave configuration as needed.

Zooming with the Mouse Wheel

After clicking within the waveform, you can use the mouse wheel with the Ctrl key in combination to zoom in and out, emulating the operation of the dials on an oscilloscope.
**Y-Axis Zoom Gestures**

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 Figure 4-8.

![Diagram of Analog Zoom Options](image)

**Figure 4-8: 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 waveform 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.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

Be aware of the following limitations:

- Maximum bus width of 64 bits on real numbers
- Verilog real and VHDL real are not supported as an analog waveform
- Floating point supports only 32- and 64-bit arrays

---

Organizing Waveforms

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

Using Groups

A Group is an expandable and collapsible container to which you can add wave objects in the wave configuration to organize 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:

1. In a waveform window, select one or more wave objects to add to a group.
   
   **Note:** A group can include dividers, virtual buses, and other groups.

2. Select **Edit > New Group**, or 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 and selecting **Edit > Wave Objects > Ungroup**, or 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 **Renaming Objects, page 78**.
Using Dividers

Dividers create a visual separator between HDL objects. 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 waveform window, click a signal to add a divider below that signal.
2. From the context menu, select Edit > New Divider, or 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 Renaming Objects, page 78.

Using Virtual Buses

You can add a virtual bus to your 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 you to add to a virtual bus.
2. Select Edit > New Virtual Bus, or 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.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

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 Renaming Objects, page 78.

To remove a virtual bus, and ungroup its contents, highlight the virtual bus, and select Edit > Wave Objects > Ungroup or 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 let you analyze the data within the waveform.

Using Cursors

Cursors are temporary indicators of time and are expected to be moved frequently, as in the case when you are measuring the time between two waveform edges.

**TIP:** WCFG files do not record cursor positions. For more permanent indicators, used in situations such as establishing a time-base for multiple measurements, and indicating notable events in the simulation, add markers to the waveform window instead. See Using Markers, page 87 for more information.

**Placing Main and Secondary Cursors**

You can place the main cursor with a single click in the waveform 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.
Chapter 4: Analyzing with the Vivado Simulator Waveforms

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 waveform window, you see a hollow or filled-in circle if the Snap to Transition button is selected, which is the default behavior.

- A hollow circle \( \bigcirc \) under the mouse indicates that you are between transitions in the waveform of the selected signal.
- A filled-in circle \( \bullet \) 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 waveform window where there is no cursor, marker, or floating ruler.

Finding the Next or Previous Transition on a Waveform

The waveform window sidebar 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 sidebar button, 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 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 waveform window.

You can display (or hide) the floating ruler and drag it to change the vertical position in the waveform 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 button and the floating ruler itself are 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. Click the Floating Ruler button.

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.

Select the command again to hide the floating ruler.

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.
  a. Place the main cursor at the time where you want to add the marker by clicking in the waveform window at the time or on the transition.
  b. Select Edit > Markers > Add Marker, or click the Add Marker button.

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 <-filename> <-line_number>
```
• You can move the marker to another location in the waveform 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.
  
  ° The drag symbol indicates that the marker can be moved. As you drag the marker in the waveform window, you see a hollow or filled-in circle if the Snap to Transition button is selected, 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.

  Note: 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) [Ref 6] for command usage.

Using Force Options

The Vivado simulator provides an interactive mechanism to specify (or force) stimulus.

You can use this capability instead of creating a test bench to drive stimulus. The available types of Force are:

• Force Constant

• Force Clock

• Remove Force

Force Constant

The Force Constant option lets you assign a new constant force that overrides the assignments made within HDL code or previously applied constant or clock force.
**Force Constant** is an option on the Objects window context menu, as shown in Figure 4-9:

![Figure 4-9: Force Options](image)

When you select the Force Constant option, the Force Constant dialog box opens where you can enter the relevant values, as shown in Figure 4-10.

![Figure 4-10: Force Selected Signal Dialog Box](image)

The Force Constant options descriptions:

- **Signal name**: Displays the default signal name, that is, the full path name of the selected item.
- **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, 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.
• Starting at 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.

• 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 commands:

```
add_force /testbench/TENSOUT 1 200 -cancel_after 500

add_force /testbench/TENSOUT -radix binary {0} {1} -repeat_every 10ns -cancel_after 3us
```

**Force Clock**

When you select the **Force Clock** option in the Objects window menu, the Force Clock dialog box opens, as shown in **Figure 4-11**:

![Force Clock Dialog Box](image)

**Figure 4-11: Force Clock Dialog Box**

The options in the Force Clock dialog box are:

• Signal name: Displays the default signal name; the full path name of the item selected in the Objects panel or waveform.

  *Note:* Running the **restart** command cancels all the Vivado simulation force commands.
• 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, 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 at 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.

• Period: Specifies the length of the clock pulse, defined as a time value. Time can be a string, such as 10 or 10 ns.

**Remove Force**

To remove any specified force from an object use the following Tcl command:

    remove_force <force object>
Using Vivado Simulator Command Line and Tcl

Introduction

This chapter describes the command line compilation and simulation process. The Vivado® Design Suite simulator executables and their corresponding switch options are listed, as well as Tcl commands for running simulation.

For a list of Vivado simulator Tcl commands, type the following:

```
help -category sim
```

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 6] for Tcl command usage.

Compiling and Simulating a Design

Running a simulation from the command line for either a behavioral or a timing simulation requires the following steps:

1. Parsing design files
2. Elaboration and generation of a simulation snapshot
3. Simulating the design

The following subsections describe these steps.

There are additional requirements for a timing simulation, described in the following document areas:

- Generating a Timing Netlist in Chapter 2
- Running Post-Synthesis and Post-Implementation Simulations, page 111
Parsing Design Files

The `xvhdl` and `xvlog` commands parse VHDL and Verilog files, respectively. Descriptions for each option are available in Table 5-2, page 102.

In PDF reader, you can turn on the Previous View and Next View buttons to navigate back and forth.

**xvhdl**

The `xvhdl` command is the VHDL analyzer (parser).

**xvhdl Syntax**

```
xvhdl
    [-encryptdumps]
    [-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>]
```

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>]]
[-encryptdumps]
[-f [file] <filename>]
[-h [-help]]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=<library_dir>]]
[-log <filename>]
[-noarg]
[-noarg_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>]]
```

### xvlog Examples

```
xvlog file1.v file2.v
xvlog -work worklib file1.v file2.v
xvlog -prj files.prj
```

---

**Elaborating and Generating a Design Snapshot**

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 simulates the model by loading the snapshot and executing it (using the `xsim` command). 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.
Chapter 5: Using Vivado Simulator Command Line and Tcl

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, page 107 for more information about project files.

xelab Command Syntax Options

Descriptions for each option are available in Table 5-2, page 102.

TIP: In Adobe Acrobat Reader, you can turn on Previous View and Next View buttons to navigate back and forth between the option descriptions and the list below.

```
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]
```
Chapter 5: Using Vivado Simulator Command Line and Tcl

[-notimingchecks]
[-nosdfinterconnectdelays]
[-nospecify]
[-O arg]
[-Odisable_acceleration arg]
[-Odisable_process_opt]
[-Oenable_cdfg]
[-Odisable_cdfg]
[-Oenable_unused_removal]
[-Odisable_unused_removal]
[-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>]
[-stat]
[-timescale]
[-timeprecision_vhdl arg]
[-trace_limit arg]
[-transport_int_delays]
[-v [verbose] [0|1|2]]
[-version]

**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
Chapter 5: Using Vivado Simulator Command Line and Tcl

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:

   ```verilog
   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 Appendix B, 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>
<thead>
<tr>
<th>File half_adder.v compiled into logical library named adder_lib</th>
<th>File full_adder.v compiled into logical library named work</th>
</tr>
</thead>
<tbody>
<tr>
<td>module half_adder(a,b,c,s); input a,b; output c,s; s = a ^ b; c = a &amp; b; endmodule</td>
<td>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</td>
</tr>
</tbody>
</table>

Simulating the Design Snapshot

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.
Chapter 5: Using Vivado Simulator Command Line and Tcl

**xsim Executable Syntax**

The command syntax is as follows:

```
xsim <options> <snapshot>
```

Where:

- `xsim` is the command.
- `<options>` are the options specified in Table 5-1.
- `<snapshot>` is the simulation snapshot.

**xsim Executable Options**

*Table 5-1: xsim Executable Command Options*

<table>
<thead>
<tr>
<th>xsim Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-f [ -file] &lt;filename&gt;</code></td>
<td>Load the command line options from a file.</td>
</tr>
<tr>
<td><code>-g [ -gui]</code></td>
<td>Run with interactive workspace.</td>
</tr>
<tr>
<td><code>-h [ -help]</code></td>
<td>Print help message to screen.</td>
</tr>
<tr>
<td><code>-log &lt;filename&gt;</code></td>
<td>Specify the log file name.</td>
</tr>
<tr>
<td><code>-maxdeltaid arg (-1)</code></td>
<td>Specify the maximum delta number. Report an error if it exceeds maximum</td>
</tr>
<tr>
<td></td>
<td>simulation loops at the same time.</td>
</tr>
<tr>
<td><code>-maxlogsize arg (-1)</code></td>
<td>Set the maximum size a log file can reach in MB. The default setting is</td>
</tr>
<tr>
<td></td>
<td>unlimited.</td>
</tr>
<tr>
<td><code>-noieeeewarnings</code></td>
<td>Disable warnings from VHDL IEEE functions.</td>
</tr>
<tr>
<td><code>-nolog</code></td>
<td>Suppresses log file generation.</td>
</tr>
<tr>
<td><code>-nosignalhandlers</code></td>
<td>Disables the installation of OS-level signal handlers in the simulation.</td>
</tr>
<tr>
<td></td>
<td>For performance reasons, the simulator does not check explicitly for certain</td>
</tr>
<tr>
<td></td>
<td>conditions, such as an integer division by zero, that could generate an</td>
</tr>
<tr>
<td></td>
<td>OS-level fatal run time error. Instead, the simulator installs signal</td>
</tr>
<tr>
<td></td>
<td>handlers to catch those errors and generates a report.</td>
</tr>
<tr>
<td></td>
<td>With the signal handlers disabled, the simulator can run in the presence of</td>
</tr>
<tr>
<td></td>
<td>such security software, but OS-level fatal errors could crash the simulation</td>
</tr>
<tr>
<td></td>
<td>abruptly with little indication of the nature of the failure.</td>
</tr>
<tr>
<td></td>
<td><strong>CAUTION!</strong> Use this option only if your security software prevents the</td>
</tr>
<tr>
<td></td>
<td>simulator from running successfully.</td>
</tr>
<tr>
<td>`-onfinish &lt;quit</td>
<td>stop&gt;`</td>
</tr>
<tr>
<td>`-onerror &lt;quit</td>
<td>stop&gt;`</td>
</tr>
<tr>
<td><code>-R [ -runall]</code></td>
<td>Runs simulation till end (such as <code>do 'run all; quit'</code>).</td>
</tr>
</tbody>
</table>
Table 5-1:  xsim Executable Command Options *(Cont’d)*

<table>
<thead>
<tr>
<th>xsim Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-testplusarg &lt;arg&gt;</td>
<td>Specify plusargs to be used by $test$plusargs and $value$plusargs system functions.</td>
</tr>
<tr>
<td>-t [-tclbatch] &lt;filename&gt;</td>
<td>Specify the Tcl file for batch mode execution.</td>
</tr>
<tr>
<td>-tp</td>
<td>Enable printing to screen of hierarchical names of process being executed.</td>
</tr>
<tr>
<td>-tl</td>
<td>Enable printing to screen of file name and line number of statements being executed.</td>
</tr>
<tr>
<td>-wdb &lt;filename.wdb&gt;</td>
<td>Specify the waveform database output file.</td>
</tr>
<tr>
<td>-version</td>
<td>Print the compiler version to screen.</td>
</tr>
<tr>
<td>-view &lt;wavefile.wcfg&gt;</td>
<td>Open a wave configuration file. Use this switch together with -gui switch.</td>
</tr>
</tbody>
</table>

**TIP:** When running the xelab, xsc, xsim, xvhd1, 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.)
### xelab, xvhd, and xvlog Command Options

Table 5-2 lists the command options for the `xelab`, `xvhdl`, and `xvlog` commands.

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
<th>Used by Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d [define] &lt;name&gt;[=&lt;val&gt;]</code></td>
<td>Define Verilog macros. Use `-d</td>
<td>--define<code>for each Verilog macro. The format of the macro is</code>&lt;name&gt;[=&lt;val&gt;]<code>where</code>&lt;name&gt;<code>is name of the macro and</code>&lt;value&gt;` is an optional value of the macro.</td>
</tr>
<tr>
<td><code>-debug &lt;kind&gt;</code></td>
<td>Compile with specified debugging ability turned on. The <code>&lt;kind&gt;</code> options are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>typical</code>: Most commonly used abilities, including: <code>line</code> and <code>wave</code>.</td>
<td>xelab</td>
</tr>
<tr>
<td></td>
<td>• <code>line</code>: HDL breakpoint.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>wave</code>: Waveform generation, conditional execution, force value.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>xlibs</code>: Visibility into Xilinx® precompiled libraries. This option is only available on the command line.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>off</code>: Turn off all debugging abilities (Default).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>all</code>: Uses all the debug options.</td>
<td></td>
</tr>
<tr>
<td><code>-encryptdumps</code></td>
<td>Encrypt parsed dump of design units being compiled.</td>
<td>xvhd, xvlog</td>
</tr>
<tr>
<td><code>-f [-file] &lt;filename&gt;</code></td>
<td>Read additional options from the specified file.</td>
<td>xelab, xsim, xvhd, xvlog</td>
</tr>
<tr>
<td><code>-generic_top &lt;value&gt;</code></td>
<td>Override generic or parameter of a top-level design unit with specified value. Example: <code>-generic_top &quot;P1=10&quot;</code></td>
<td>xelab</td>
</tr>
<tr>
<td><code>-h [-help]</code></td>
<td>Print this help message.</td>
<td>xelab, xsim, xvhd, xvlog</td>
</tr>
<tr>
<td><code>-i [include] &lt;directory_name&gt;</code></td>
<td>Specify directories to be searched for files included using Verilog <code>include</code>. Use `-i</td>
<td>--include` for each specified search directory.</td>
</tr>
<tr>
<td><code>-initfile &lt;init_filename&gt;</code></td>
<td>User-defined simulator initialization file to add to or override settings provided by the default <code>xsim.ini</code> file.</td>
<td>xelab, xvhd, xvlog</td>
</tr>
</tbody>
</table>
Table 5-2: xelab, xvhd, and xvlog Command Options (Cont’d)

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
<th>Used by Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-L [-lib] &lt;library_name&gt; [-=library_dir&gt;]</code></td>
<td>Specify search libraries for the instantiated non-VHDL design units; for example, a Verilog design unit. Use `-L</td>
<td>--lib<code>for each search library. The format of the argument is</code>&lt;name&gt;[-=dir]<code>where</code>&lt;name&gt;<code>is the logical name of the library and</code>&lt;library_dir&gt;` is an optional physical directory of the library.</td>
</tr>
<tr>
<td><code>-log &lt;filename&gt;</code></td>
<td>Specify the log file name. Default: `&lt;xvlog</td>
<td>xvhdl</td>
</tr>
<tr>
<td><code>-maxarraysize arg</code></td>
<td>Set maximum vhdl array size to be $2^n$ (Default: $n = 28$, which is $2^{28}$)</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-maxdelay</code></td>
<td>Compile Verilog design units with maximum delays.</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-maxdesigndepth arg</code></td>
<td>Override maximum design hierarchy depth allowed by the elaborator (Default: 5000).</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-maxlogsize arg (-1)</code></td>
<td>Set the maximum size a log file can reach in MB. The default setting is unlimited.</td>
<td>xsim</td>
</tr>
<tr>
<td><code>-mindelay</code></td>
<td>Compile Verilog design units with minimum delays.</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-mt arg</code></td>
<td>Specifies the number of sub-compilation jobs which can be run in parallel. Possible values are auto, off, or an integer greater than 1. If auto is specified, xelab selects the number of parallel jobs based on the number of CPUs on the host machine. (Default = auto). To further control the <code>-mt</code> option, an advanced user can set the Tcl property as follows: `set_property XELAB.MT_LEVEL off</td>
<td>N [get_filesets sim_1]`</td>
</tr>
<tr>
<td><code>-nolog</code></td>
<td>Suppress log file generation.</td>
<td>xelab xsim xvhd xvlog</td>
</tr>
<tr>
<td><code>-noieeewarnings</code></td>
<td>Disable warnings from VHDL IEEE functions.</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-noname_unnamed_generate</code></td>
<td>Do not generate name for an unnamed generate block.</td>
<td>xelab xvlog</td>
</tr>
<tr>
<td><code>-notimingchecks</code></td>
<td>Ignore timing check constructs in Verilog specify block(s).</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-nosdfinterconnectdelays</code></td>
<td>Ignore SDF port and interconnect delay constructs in SDF.</td>
<td>xelab</td>
</tr>
<tr>
<td><code>-nospecify</code></td>
<td>Ignore Verilog path delays and timing checks.</td>
<td>xelab</td>
</tr>
</tbody>
</table>
### Chapter 5: Using Vivado Simulator Command Line and Tcl

#### Table 5-2: xelab, xvhd, and xvlog Command Options (Cont’d)

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
<th>Used by Command</th>
</tr>
</thead>
</table>
| -O arg                                | Enable or disable optimizations.  
  -O0 = Disable optimizations  
  -O1 = Enable basic optimizations  
  -O2 = Enable most commonly desired optimizations (Default)  
  -O3 = Enable advanced optimizations  
  **Note:** A lower value speeds compilation at expense of slower simulation: a higher value slows compilation but simulation runs faster. | xelab           |
| -O disable_acceleration arg           | Turn off acceleration for the specified HDL package. Choices are: all, math_real, math_complex, numeric_std, std_logic_signed, std_logic_unsigned (default: acceleration is on) | xelab           |
| -O disable_process_opt                | Turn off the process-level optimization (default on)                                                                                       | xelab           |
| -O enable_cdfg                        | Turn on (enable) or off (disable) the building of the control+data flow graph (default: on)                                                  | xelab           |
| -O disable_cdfg                       |                                                                                                                                             | xelab           |
| -O enable_unused_removal              | Turn on (enable or off (disable) the optimization to remove unused signals and statements (default: on)                                      | xelab           |
| -O disable_unused_removal             |                                                                                                                                             | 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                    | Specify when error about pulse being shorter than module path delay should be handled. Choices are: ondetect: report error right when violation is detected  
  onevent: report error after the module path delay.  
  Default: onevent                                                          | xelab           |
### Table 5-2: `xelab`, `xvhdl`, and `xvlog` Command Options (Cont’d)

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
<th>Used by Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-prj &lt;filename&gt;</code></td>
<td>Specify the Vivado simulator project file containing one or more entries of vhdl</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td>verilog &lt;work lib&gt; &lt;HDL filename&gt;.</td>
<td><code>xvhdl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xvlog</code></td>
</tr>
<tr>
<td><code>-r [-run]</code></td>
<td>Run the generated executable snapshot in command-line interactive mode.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-rangecheck</code></td>
<td>Enable run time value range check for VHDL.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-R [-runall]</code></td>
<td>Run the generated executable snapshot until the end of simulation.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xsim</code></td>
</tr>
<tr>
<td><code>-relax</code></td>
<td>Relax strict language rules.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xvhdl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xvlog</code></td>
</tr>
<tr>
<td><code>-s [-snapshot] arg</code></td>
<td>Specify the name of output simulation snapshot. Default is &lt;worklib&gt;.&lt;unit&gt;;</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td>for example: work.top. Additional unit names are concatenated using #: for example: work.t1#work.t2.</td>
<td></td>
</tr>
<tr>
<td><code>-sdfnowarn</code></td>
<td>Do not emit SDF warnings.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sdfnoerror</code></td>
<td>Treat errors found in SDF file as warning.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sdfmin arg</code></td>
<td>&lt;root=file&gt; SDF annotate &lt;file&gt; at &lt;root&gt; with minimum delay.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sdftyp arg</code></td>
<td>&lt;root=file&gt; SDF annotate &lt;file&gt; at &lt;root&gt; with typical delay.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sdfmax arg</code></td>
<td>&lt;root=file&gt; SDF annotate &lt;file&gt; at &lt;root&gt; with maximum delay.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sdfroot &lt;root_path&gt;</code></td>
<td>Default design hierarchy at which SDF annotation is applied.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sourcelibdir &lt;sourcelib_dirname&gt;</code></td>
<td>Directory for Verilog source files of uncompiled modules. Use -</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td>-sourcelibdir &lt;sourcelib_dirname&gt; for each source directory.</td>
<td><code>xvhdl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xvlog</code></td>
</tr>
<tr>
<td><code>-sourcelibext &lt;file_extension&gt;</code></td>
<td>File extension for Verilog source files of uncompiled modules. Use</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td>-sourcelibext &lt;file extension&gt; for source file extension</td>
<td><code>xvhdl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xvlog</code></td>
</tr>
<tr>
<td><code>-sourcelibfile &lt;filename&gt;</code></td>
<td>File name of a Verilog source file with uncompiled modules. Use -</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td></td>
<td>-sourcelibfile &lt;filename&gt;</td>
<td><code>xvhdl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xvlog</code></td>
</tr>
<tr>
<td><code>-stat</code></td>
<td>Print tool CPU and memory usages, and design statistics.</td>
<td><code>xelab</code></td>
</tr>
<tr>
<td><code>-sv</code></td>
<td>Compile input files in System Verilog mode.</td>
<td><code>xvlog</code></td>
</tr>
</tbody>
</table>
Chapter 5: Using Vivado Simulator Command Line and Tcl

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

**Step1: 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>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
<th>Used by Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>-timescale</td>
<td>Specify default timescale for Verilog modules. Default: 1ns/1ps.</td>
<td>xelab</td>
</tr>
<tr>
<td>-timeprecision_vhdl arg</td>
<td>Specify time precision for vhdl designs. Default: 1ps.</td>
<td>xelab</td>
</tr>
<tr>
<td>-trace_limit arg (=65536)</td>
<td>Maximum allowable signal width that can be traced for waveform viewing</td>
<td>xelab</td>
</tr>
<tr>
<td>-transport_int_delays</td>
<td>Use transport model for interconnect delays.</td>
<td>xelab</td>
</tr>
<tr>
<td>-typdelay</td>
<td>Compile Verilog design units with typical delays (Default).</td>
<td>xelab</td>
</tr>
<tr>
<td>-v [verbose] [0</td>
<td>1</td>
<td>2]</td>
</tr>
<tr>
<td>-version</td>
<td>Print the compiler version to screen.</td>
<td>xelab</td>
</tr>
<tr>
<td>-work &lt;library_name&gt; [=&lt;library_dir&gt;]</td>
<td>Specify the work library. The format of the argument is &lt;name&gt;[=&lt;dir&gt;] where: • &lt;name&gt; is the logical name of the library. • &lt;library_dir&gt; is an optional physical directory of the library.</td>
<td>xvhdl xvlog</td>
</tr>
</tbody>
</table>
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_simulator`, 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 FileNa me> -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

To parse design files using a project file, create a file called `<proj_name>.prj`, and use the following syntax inside the project file:

```
verilog <work_library> <file_names>... [-d <macro>]...[-i <include_path>]...
vhdl <work_library> <file_name>
sv <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.

- For Verilog or System Verilog: [-d <macro>] provides you the option to define one or more macros.
- For Verilog or System Verilog: [-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:

```vhdl
'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
```

### 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. <Vivado_Install_Dir>/data/xsim
2. User-file specified through the -initfile switch. If -initfile is not specified, the program searches for xsim.ini in the current working directory.

The xsim.ini file has the following syntax:
Chapter 5: Using Vivado Simulator Command Line and Tcl

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
synopsys=<Vivado_Install_Area>/xsim/vhdl/synopsys
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 --
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

Figure 5-1 illustrates the behavioral simulation process:

To run behavioral simulation, use the Tcl command: `launch_simulator -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. Figure 5-2 illustrates the post-synthesis and post-implementation simulation process:

```
synth_design -top top -part xc7k70tfbg676-2
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
exec xvlog -work work test_synth.v
exec xvlog -work work testbench.v
exec xelab -L unisims_ver testbench glbl -s test
xsim test -gui
```

Figure 5-2: Post-Synthesis and Post-Implementation Simulation

The following is an example of running a post-synthesis functional simulation from the command line:
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 commands as simulation starts. For example, you can have a file named `run.tcl` that contains the following:

```tcl
run 20ns
current_time
quit
```

Then launch simulation as follows:

```bash
xsim <snapshot> -tclbatch run.tcl
```

You can set a variable to represent a simulation command to quickly run frequently used simulation commands.
Chapter 5: Using Vivado Simulator Command Line and Tcl

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_simulator -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
Vivado% write_sdf postRoute.sdf
Vivado% launch_simulator -simset sim_1 -mode post-implementation -type timing
Vivado% close_project

Tcl Property Commands

You can use Tcl *_property commands to find the properties associated with a design. The following subsections briefly describe some of the property commands that are useful during simulation.

See the Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 5] for more information regarding the property commands.

report_property Command

Tcl Command:

    [-regexp] [-quiet] [-verbose] [<object>] [<pattern>]

Related commands are as follows:

- create_property
- get_cells
- list_property Command
- list_property_value
- reset_property
- set_property
- get_property
Chapter 5: Using Vivado Simulator Command Line and Tcl

list_property Command

The `list_property` Tcl command lists the properties of a specified object:

```tcl
list_property [-class <arg>] [-regexp] [-quiet] [-verbose] [<object>] [<pattern>]
```

The command returns a list of property names.

get_property

The `get_property` Tcl command lets you modify the value of a specified property.

```tcl
get_property [-quiet] [-verbose] <name> <object>
```

Example:

The following example gets the NAME property from the specified cell:

```tcl
get_property NAME [lindex [get_cells] 3]
```

Property Command Options

The table below lists the Property options.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Read-only</th>
<th>Visible</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>string</td>
<td>TRUE</td>
<td>TRUE</td>
<td>fileset</td>
</tr>
<tr>
<td>FILESET_TYPE</td>
<td>string</td>
<td>TRUE</td>
<td>TRUE</td>
<td>SimulationSrcs</td>
</tr>
<tr>
<td>GENERIC</td>
<td>string*</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>INCLUDE_DIRS</td>
<td>string*</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>IS_AUTO_DISABLED</td>
<td>bool</td>
<td>TRUE</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>NAME</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td>sim_1</td>
</tr>
<tr>
<td>NEEDS_REFRESH</td>
<td>bool</td>
<td>TRUE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>NL.CELL</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>NL.INCL_UNISIM_MODELS</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>NL.MODE</td>
<td>string</td>
<td>FALSE</td>
<td>FALSE</td>
<td>timesim</td>
</tr>
<tr>
<td>NL.NOLIB</td>
<td>bool</td>
<td>FALSE</td>
<td>FALSE</td>
<td>1</td>
</tr>
<tr>
<td>NL.PROCESS_CORNER</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td>slow</td>
</tr>
<tr>
<td>NL.RENAME_TOP</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>NL.SDF_ANNO</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>NL.WRITE_ALL_OVERRIDES</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1000ns</td>
</tr>
<tr>
<td>SIM_MODE</td>
<td>string</td>
<td>FALSE</td>
<td>FALSE</td>
<td>behavioral</td>
</tr>
</tbody>
</table>

Send Feedback
Table 5-4: Property Options (Cont’d)

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Read-only</th>
<th>Visible</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIP_COMPILATION</td>
<td>bool</td>
<td>FALSE</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>SKIP_SIMULATION</td>
<td>bool</td>
<td>FALSE</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>SOURCE_SET</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td>sources_1</td>
</tr>
<tr>
<td>TOP</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td>bft_tb</td>
</tr>
<tr>
<td>TOP_ARCH</td>
<td>string</td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>TOP_AUTO_SET</td>
<td>bool</td>
<td>FALSE</td>
<td>FALSE</td>
<td>1</td>
</tr>
<tr>
<td>TOP_FILE</td>
<td>string</td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>TOP_LIB</td>
<td>string</td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>UNIT_UNDER_TEST</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>VERILOG_DEFINE</td>
<td>string*</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>VERILOG_DIR</td>
<td>string*</td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>VERILOG_UPPERCASE</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>VHDL_GENERIC</td>
<td>string*</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XELAB_DEBUG_LEVEL</td>
<td>enum</td>
<td>FALSE</td>
<td>TRUE</td>
<td>typical</td>
</tr>
<tr>
<td>XELAB.DLL</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XELAB_LOAD_GLBL</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>XELAB more OPTIONS</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XELAB_MT_LEVEL</td>
<td>enum</td>
<td>FALSE</td>
<td>TRUE</td>
<td>auto</td>
</tr>
<tr>
<td>XELAB_NOSORT</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XELAB_RANGECHECK</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XELAB_RELAX</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>XELAB.SDF_DELAY</td>
<td>enum</td>
<td>FALSE</td>
<td>TRUE</td>
<td>sdfmax</td>
</tr>
<tr>
<td>XELAB_SNAPSHOT</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XELAB.UNIFAST</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XSIM.compile.XVHDL.More_Option</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.compile.XVHDL.Nosort</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XSIM.compile.XVLOG.More_Option</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.compile.XVLOG.Nosort</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XSIM.elaborate.debug_level</td>
<td>enum</td>
<td>FALSE</td>
<td>TRUE</td>
<td>typical</td>
</tr>
<tr>
<td>XSIM.elaborate.load_glbl</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>XSIM.elaborate.MT_Level</td>
<td>enum</td>
<td>FALSE</td>
<td>TRUE</td>
<td>auto</td>
</tr>
<tr>
<td>XSIM.elaborate.rangecheck</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XSIM.elaborate.relax</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>XSIM.elaborate.SDF_Delay</td>
<td>enum</td>
<td>FALSE</td>
<td>TRUE</td>
<td>sdfmax</td>
</tr>
</tbody>
</table>
Table 5-4: Property Options (Cont’d)

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Read-only</th>
<th>Visible</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSIM.ELABORATE.SNAPSHOT</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.ELABORATE.UNIFAST</td>
<td>bool</td>
<td>FALSE</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>XSIM.ELABORATE.XELABMOREOPTIONS</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.MORE_OPTIONS</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.SAIF</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.SIMULATE.RUNTIME</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td>1000ns</td>
</tr>
<tr>
<td>XSIM.SIMULATE.SAIF</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.SIMULATE.UUT</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.SIMULATE.WDB</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.SIMULATE.XSIMMOREOPTIONS</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.TCLBATCH</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.VIEW</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>XSIM.WDB</td>
<td>string</td>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6

Debugging a Design with Vivado Simulator

Introduction

You can debug a design in the Vivado® Design Suite simulator from the source code or by setting breakpoints and running simulation until a breakpoint is reached.

This chapter describes debugging methods, and includes Tcl commands that are valuable in the debug process. There is also a flow diagram that illustrates the process of debugging in third-party simulators.

You can also set conditions by which the tools searches for and displays the source code that matches the specified condition and executes a command. Use the Tcl command:

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

See Adding Conditions and Outputting Diagnostic Messages for Debugging, page 120 for more information.

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.

**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.

A yellow 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 waveform 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 3, Using the Vivado Simulator from the Vivado IDE.

2. Select **Window > Tile Horizontally** (or **Window > Tile Vertically**) to simultaneously see the waveform and the HDL code.

3. Repeat the **Step** action until debugging is complete.

As each line is executed, you can see the yellow arrow moving down the code. If the simulator is executing lines in another file, the new file opens, and the yellow 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 source code line with the breakpoint is reached.
**Chapter 6: Debugging a Design with Vivado Simulator**

**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.

**To set a breakpoint in the workspace (GUI):**

1. To set a breakpoint, 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.

**To set a breakpoint in the Tcl Console:**

1. Type the Tcl Command: `add_bp <line_number> <file_name>`
   
   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) [Ref 5] for command usage.

**To debug a design using breakpoints:**

1. Open the HDL source file.
2. Set breakpoints on executable lines in the HDL source file.
3. Repeat steps 1 and 2 until all breakpoints are set.
4. 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 Specified Time** command.

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

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

5. 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 > Breakpoint > Delete All Breakpoints** or click the **Delete All Breakpoints** button. ✗
To delete all breakpoints:

- Type the Tcl command `remove_bp`

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

- Type the Tcl command `report_bp <list>`

### Adding Conditions and Outputting Diagnostic Messages for Debugging

To add set 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 at the 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, then stop when reset goes to 1 and wbClk goes to 1:

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

A run all breaks simulation at 5 ns when the condition is met and "Reset went to high" is printed to the console.

---

### Generating (forcing) Stimulus

You can use the `add_force` Tcl command to force a signal, wire, or reg to a specified value:

```
add_force
```

Command syntax:

```
add_force [\-radix <arg>] [\-repeat_every <arg>] [\-cancel_after <arg>] [\-quiet]
[\-verbose] <hdl_object> <values>...
```

**Figure 6-1** illustrates how the `add_force` functionality is applied given the following command:

```
add_force mySig (0 t_1) {1 t_2} {0 t_3} {1 t_4} {0 t_5} -repeat_every tr -cancel_after tc
```
You can get more detail on the command by typing the following in the Tcl Console:

```
add_force -help
```

## Adding Force in Verilog Code

The following code snippet is a Verilog example of adding force:

```verilog
module bot(input in1, in2, output out1);
reg sel;
assign out1 = sel? in1: in2;
endmodule

module top;
reg in1, in2;
wire out1;
bot II(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
```

*Figure 6-1: Illustration of -add_force Functionality*
Command Examples

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
```
Using add_force with remove_force

The following is an example Verilog file, top.v, which instantiates a counter. You can use this file in the following command example.

```verilog
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
```
Running add_force and remove_force in a Tcl Batch File

1. Create a file called add_force.tcl with following command:
   ```
   create_project add_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 batch mode, and source the add_force.tcl file.

3. In the Tcl Console, type:
   ```
   add_force clk {0 1} {1 2} -repeat_every 3 -cancel_after 500
   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 Waveform window.
   Observe the value of updown signal in the Waveform window.

4. In the Tcl Console, type:
   ```
   remove_force force2
   ```
   Observe that the value of signal updown is now the default value present in design.

5. In the Tcl Console, type:
   ```
   run 100
   ```
   Observe that only the value of out1 increments.

6. In the Tcl Console, type:
   ```
   remove_force 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 SAIF dumping is optimized for Xilinx® Power tools and dumps the following HDL types:

- **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.

Generating SAIF Dumping

Before you use the `log_saif` command, you must call `open_saif`. The `log_saif` does not return any object or value. The switches are the same as those used in the `log_wave` command.

1. Compile your RTL code with the `-debug typical` option to enable SAIF dumping:
   ```
   xelab -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, the Tcl command:
   ```
   log_saif [get_objects -r *]
   ```

4. Import simulation data into an SAIF format using the following Tcl command:
   ```
   read_saif
   ```

5. Run the simulation (use any of the run commands).
6. Close the SAIF file, by typing the Tcl command:
   close_saif

**Example SAIF Tcl Commands**

To log SAIF for:

- **All signals in the scope** /tb:  
  
  ```tcl
  log_saif /tb/*
  ```

- **All the ports of the scope** /tb/UUT:
  
  ```tcl
  log_saif /tb/UUT/*
  ```

- **Those objects having names that start with a and end in b and have digits in between:**
  
  ```tcl
  log_saif [get_objects -regexp {^a[0-9]+b$}]
  ```

- **The objects in the current_scope and children_scope:**
  
  ```tcl
  log_saif [get_objects -r *]
  ```

- **The objects in the current_scope:**
  
  ```tcl
  log_saif * or log_saif [get_objects]
  ```

- **Only the ports of the scope /tb/UUT, use the command:**
  
  ```tcl
  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:**
  
  ```tcl
  log_saif [get_objects -filter {type == signal} /tb/UUT/* ]
  ```

**Dumping SAIF using a Tcl Simulation Batch File**

```tcl
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:

```tcl
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.
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 6-1: Tcl Commands for VCD

<table>
<thead>
<tr>
<th>Tcl Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_vcd</td>
<td>Opens a VCD file for capturing simulation output. This Tcl command models the behavior of <code>$dumpfile</code> Verilog system task.</td>
</tr>
<tr>
<td>checkpoint_vcd</td>
<td>Models the behavior of the <code>$dumpall</code> Verilog system task.</td>
</tr>
<tr>
<td>start_vcd</td>
<td>Models the behavior of the <code>$dumpon</code> Verilog system task.</td>
</tr>
<tr>
<td>log_vcd</td>
<td>Logs VCD for the specified HDL objects. This command models behavior of the <code>$dumpvars</code> Verilog system task.</td>
</tr>
<tr>
<td>flush_vcd</td>
<td>Models behavior of the <code>$dumpflush</code> Verilog system task.</td>
</tr>
<tr>
<td>limit_vcd</td>
<td>Models behavior of the <code>$dumplimit</code> Verilog system task.</td>
</tr>
<tr>
<td>stop_vcd</td>
<td>Models behavior of the <code>$dumpoff</code> Verilog system task.</td>
</tr>
<tr>
<td>close_vcd</td>
<td>Closes the VCD generation.</td>
</tr>
</tbody>
</table>

See the Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 5], or type the following in the Tcl Console:

```
<command> -help
```

See Verilog Language Support Exceptions in Appendix B for more information.
Using the log_wave Tcl Command

The log_wave command logs simulation output for viewing specified wires, signals, or registers with the Vivado simulator waveform viewer. Unlike add_wave, the log_wave command does not add the waveform object to the waveform viewer (that is, the Waveform Configuration). It simply enables the logging of output to the Vivado Simulator Waveform Database (WDB).

Syntax:

    log_wave [-recursive] [-r] [-quiet] [-verbose] <hdl_objects>...

Example log_wave TCL Command Usage

To log the waveform output for:

- All signal 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$]

- The objects in the current_scope and children_scope:
  
  log_wave [get_objects -r *] or log_wave -r *

- The objects in the current_scope:
  
  log_wave * or log_wave [get_objects]

- 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/*]
Chapter 7

Compiling Simulation Libraries

Introduction

Before you begin working with a simulator other than the Vivado® simulator, you must run compile_simlib.

Running compile_simlib compiles the Xilinx® simulation libraries for the target simulator. In the Vivado tools, you can achieve this in GUI or Tcl mode.

IMPORTANT: You must compile libraries whenever tool versions or Vivado tool versions change.

GUI Mode

Starting in 2014.3, there is a feature in the GUI that you can use to generate the Xilinx® simulation libraries for the target simulator.

1. Open the Vivado tools in GUI mode.
2. Select Tools > Compile Simulation Libraries to open the dialog box shown in Figure 7-1.
Dialog Box Options

- **Simulator**: From the Simulator drop-down menu, select a simulator, as shown in Figure 7-2.

- **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. See Figure 7-3.
Chapter 7: Compiling Simulation Libraries

- **Library**: Specifies the simulation library to compile. By default, the `compile_simlib` command compiles all simulation libraries. See Figure 7-4.

  ![Figure 7-3: Language Selection](image)

  **Figure 7-3: Language Selection**

- **Family**: Compiles selected libraries to the specified device family. All device families are generated by default. See Figure 7-5.

  ![Figure 7-4: Simulation Libraries](image)

  **Figure 7-4: Simulation Libraries**

  - **Family**: Compiles selected libraries to the specified device family. All device families are generated by default. See Figure 7-5.

  ![Figure 7-5: Family Options](image)

  **Figure 7-5: Family Options**

- **Compiled library location**: 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. Refer to the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 9] for more information on Project and Non-Project modes.

- **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.

- **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.

**TIP**: At the bottom of the Compile Simulation Libraries dialog box, there is a field labeled Command (shown in Figure 7-1, above). The value of the Command field changes based on the options you select. You can use the value of the Command field to generate a simulation library in Tcl/non-project mode.

---

## Tcl Mode

### Syntax:

```
 [-library <arg>] [-print_library_info <arg>] -simulator <arg>
 [-simulator_exec_path <arg>] [-source_library_path <arg>]
 [-32bit] [-quiet] [-verbose]
```

### Table 7-1: Tcl Mode Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-directory</code></td>
<td>Directory path for saving the compiled results.</td>
</tr>
<tr>
<td><code>-family</code></td>
<td>Selects device architecture. Default: all</td>
</tr>
<tr>
<td><code>-force</code></td>
<td>Overwrites the pre-compiled libraries</td>
</tr>
<tr>
<td><code>-language</code></td>
<td>Compiles libraries for this language. Default: all</td>
</tr>
<tr>
<td><code>-library</code></td>
<td>Selects library to compile. Default: all</td>
</tr>
<tr>
<td><code>-print_library_info</code></td>
<td>Prints pre-compiled library information</td>
</tr>
<tr>
<td><code>-simulator</code></td>
<td>Compiles libraries for this simulator</td>
</tr>
<tr>
<td><code>-simulator_exec_path</code></td>
<td>Uses simulator executables from this directory</td>
</tr>
<tr>
<td><code>-source_library_path</code></td>
<td>If specified, this directory is searched for the library source files before searching the default path(s) found in the environment variable XILINX_VIVADO for Vivado tools.</td>
</tr>
</tbody>
</table>
Table 7-1: Tcl Mode Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-32bit]</td>
<td>Performs the 32-bit compilation</td>
</tr>
<tr>
<td>[-quiet]</td>
<td>Ignores command errors</td>
</tr>
<tr>
<td>[-verbose]</td>
<td>Suspends message limits during command execution</td>
</tr>
</tbody>
</table>

For more details, type `compile_simlib -help` on the Vivado Design Suite Tcl console.

**Example commands:**

- Generating a simulation library for Questa for all languages and for all libraries and all families in the current directory.
  ```tcl```
  ```
  compile_simlib -language all -simulator questa -library all -family all
  ```
  ```

- Generating a simulation library for IES for the Verilog language, for the UNISIM library at /a/b/c.
  ```tcl```
  ```
  compile_simlib -language verilog -dir {/a/b/c} -simulator ies -library unisim -family all
  ```
  ```

- Generating a simulation library for VCS for the Verilog language, for the UNISIM library at /a/b/c.
  ```tcl```
  ```
  compile_simlib -language verilog -dir {/a/b/c} -simulator vcs_mx -library unisim -family all
  ```
  ```

- Generating simulation library for ModelSim at /a/b/c, where the ModelSim executable path is `<simulator_installation_path>`.
  ```tcl```
  ```
  compile_simlib -language all -dir {/a/b/c} -simulator modelsim -simulator_exec_path {<simulator_installation_path>} -library all -family all
  ```
  ```

Chapter 8

Simulating with QuestaSim/ModelSim

Introduction

This chapter provides an overview of running simulation using QuestaSim/ModelSim in the Vivado® Design Suite.

IMPORTANT: Use 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) [Ref 1].

Simulating Xilinx Designs using QuestaSim/ModelSim

Mentor Graphics QuestaSim/ModelSim is supported through the Vivado Integrated Design Environment (IDE). You can launch these simulators directly.

The Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 2] describes the use of the Vivado IDE.

For more information on the QuestaSim/ModelSim simulators, see the following websites:

- www.mentor.com/products/fv/questa/
- www.mentor.com/products/fv/modelsim/


Setting QuestaSim/ModelSim for Use in the Vivado IDE

The following subsections describe two vital steps required for invoking Questasim/Modelsim from the Vivado IDE: Pointing to the QuestaSim/ModelSim Simulator Install Location and Compiling Simulation Libraries for QuestaSim/ModelSim.
Pointing to the QuestaSim/ModelSim Simulator Install Location

To define the ModelSim/QuestaSim installation path:

1. Select **Tools > Options > General**
2. In the Vivado Options, General dialog box, *scroll down* to the **QuestaSim/ModelSim install path** field, shown in **Figure 8-1**, and browse to the appropriate installation path.

![Vivado General Options, Install Path](image)

Figure 8-1:  Vivado General Options, Install Path

Compiling Simulation Libraries for QuestaSim/ModelSim

Before you begin simulation, run the `compile_simlib` Tcl command to compile the Xilinx simulation libraries for the target simulator. This is one time job. For details, refer to **Chapter 7, Compiling Simulation Libraries**.

**IMPORTANT:** *Any change to the Vivado tools or the simulator versions requires that libraries be recompiled.*

**Note:** For information on Xilinx libraries, see **Chapter 2, Using Xilinx Simulation Libraries**.
After the libraries are compiled, the simulator references these compiled libraries using the modelsim.ini file. The compile_simlib command copies the modelsim.ini file to the <library_output_directory>.

The modelsim.ini file is the default initialization file and contains control variables that specify reference library paths, optimization, compiler, and simulator settings. If the correct modelsim.ini file is not found in the path, you cannot run simulation on designs that include Xilinx primitives.

Running QuestaSim/ModelSim Simulation

The Flow Navigator > Simulation Settings section lets you configure the simulation settings in Vivado IDE. The Flow Navigator Simulation menu is shown in Figure 8-2.

![Simulation Settings](image)

**Figure 8-2: Simulation Settings**

Simulation Settings opens the Simulation Settings dialog box where you can select and configure the simulator.
QuestaSim/ModelSim Simulation Options

In the Flow Navigator, click **Simulation Settings** to open the Project Settings dialog box, shown in the figure below.

![Project Settings Dialog Box, ModelSim/QuestaSim](image)

**Figure 8-3:** Project Settings Dialog Box, ModelSim/QuestaSim

**Table 8-1** describes each of the simulation menu options.

**Table 8-1: Simulation Menu Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Simulator</td>
<td>Displays the selected simulation tool.</td>
</tr>
<tr>
<td>Simulator Language</td>
<td>Displays the simulator language used.</td>
</tr>
<tr>
<td>Simulation set</td>
<td>Selects the simulation to be performed.</td>
</tr>
<tr>
<td>Simulation top module name</td>
<td>Top design unit name.</td>
</tr>
<tr>
<td>Clean up simulation files</td>
<td>Enabled by default. Cleans the simulation files before re-run.</td>
</tr>
</tbody>
</table>
Chapter 8: Simulating with QuestaSim/ModelSim

Selecting Compilation Options

The available compilation options are shown in Figure 8-4.

Table 8-2: Compilation Options

<table>
<thead>
<tr>
<th>Compilation Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog option</td>
<td>Browse to set Verilog include path and to define macro</td>
</tr>
<tr>
<td>Generics/Parameters options</td>
<td>Set the generic/parameter value</td>
</tr>
<tr>
<td>modelsim.compile.vhdl_syntax</td>
<td>Specify VHDL syntax</td>
</tr>
<tr>
<td>modelsim.compile.use_explicit_decl</td>
<td>Log all signals</td>
</tr>
<tr>
<td>modelsim.compile.load_glbl</td>
<td>Load GLBL module</td>
</tr>
<tr>
<td>modelsim.compile.incremental</td>
<td>Perform incremental compilation</td>
</tr>
<tr>
<td>modelsim.compile.unifast</td>
<td>Use UNIFAST in place of UNISIM library</td>
</tr>
</tbody>
</table>
Table 8-2:  Compilation Options (Cont’d)

<table>
<thead>
<tr>
<th>Compilation Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelsim.compile.vlog.more_options</td>
<td>More VLOG compilation options</td>
</tr>
<tr>
<td>modelsim.compile.vcom.more_options</td>
<td>More VCOM compilation options</td>
</tr>
</tbody>
</table>

Selecting Elaboration Options

Figure 8-5 shows the elaboration options.

Table 8-3:  Elaboration Options

<table>
<thead>
<tr>
<th>Elaboration Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelsim.elaborate.acc</td>
<td>Enables access to certain objects which might be optimized anyway</td>
</tr>
<tr>
<td>modelsim.elaborate.unifast</td>
<td>Enable fast simulation models</td>
</tr>
<tr>
<td>modelsim.elaborate.vopt.more_options</td>
<td>More VOPT elaboration options</td>
</tr>
</tbody>
</table>
Selecting Simulation Options

Figure 8-6 shows the simulation options.

![Simulation Options Tab, ModelSim/QuestaSim](image)

**Figure 8-6:** Simulation Options Tab, ModelSim/QuestaSim

Table 8-4 describes the simulation options.

**Table 8-4:** Simulation Options

<table>
<thead>
<tr>
<th>Simulation Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelsim.simulate.runtime</td>
<td>Specify simulation run time</td>
</tr>
<tr>
<td>modelsim.simulate.log_all_signals</td>
<td>Log all signals</td>
</tr>
<tr>
<td>modelsim.simulate.uut</td>
<td>Specify instance name for design under test (default :/uut)</td>
</tr>
<tr>
<td>modelsim.simulate.custom_do</td>
<td>Specify the name of custom do file</td>
</tr>
<tr>
<td>modelsim.simulate.custom_udo</td>
<td>Specify the name of custom user do file</td>
</tr>
<tr>
<td>modelsim.simulate.sdf_delay</td>
<td>Specify the delay type for sdf annotation</td>
</tr>
<tr>
<td>modelsim.simulate.saif</td>
<td>Specify SAIF file</td>
</tr>
<tr>
<td>modelsim.simulate.64bit</td>
<td>Call 64bit VSIM compiler</td>
</tr>
<tr>
<td>modelsim.simulate.vsim.more_option</td>
<td>More VSIM simulation options</td>
</tr>
</tbody>
</table>
Specify Netlist Options

Figure 8-7 shows the Netlist option tab.

![Netlist Options Tab, ModelSim/QuestaSim](image)

Table 8-5 describes the netlist options.

<table>
<thead>
<tr>
<th>Netlist Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sdf_anno</td>
<td>Specify if sdf_annotate system task statement is generated</td>
</tr>
<tr>
<td>-process_corner</td>
<td>Specify the -process_corner for which SDF delays are required (slow or fast)</td>
</tr>
</tbody>
</table>

See Annotating an SDF File in QuestaSim/ModelSim, page 148 for more information about SDF.
Selecting Advanced Simulation Options

The Advanced view is shown in Figure 8-8.

![Advanced Option Tab](image)

Figure 8-8: Advanced Option Tab

This view provides an option to include all design sources for simulation. Unchecking the box gives you the flexibility to include only the files you want to simulate.

**RECOMMENDED:** It is typically best not to disable this option.

Adding or Creating Simulation Source Files

To add simulation sources to a project:

1. Select **File > Add Sources**, or click the **Add Sources** button.

   The Add Sources wizard opens.

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

   The Add or Create Simulation Sources dialog box options are:

   - Specify Simulation Set: Enter the name of the simulation set in which to store test bench files and directories (the default is sim_1, sim_2, and so forth).
Chapter 8: Simulating with QuestaSim/ModelSim

To define a new simulation set, select the **Create Simulation Set** command from the drop-down menu. When more than one simulation set is available, the Vivado simulator shows which simulation set is the *active* (currently used) set.

- **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.

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 Selected File Up**: Moves the file up in the list order.
- **Move Selected File 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 selected 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.

**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 simulation 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 files to the project, you can specify which simulation source set into which to add files.

To edit a simulation set:

1. In the Sources window popup menu, select **Simulation Sources > Edit Simulation Sets**, as shown in **Figure 8-9**.

![Edit Simulation Sets Option](image)

**Figure 8-9: Edit Simulation Sets Option**

The Add or Create Simulation Sources wizard opens.

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

---

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

---

Additionally, you must write a Verilog or VHDL netlist of the design to export to the simulator, and simulate using the third-party simulation libraries as provided by the vendor. See the third-party simulator documentation for more information on setting up and running simulation in that tool.

---

**IMPORTANT:** Confirm the library compilation order before running a third-party simulation.
Running QuestaSim/ModelSim Simulation

The Run Simulation button sets up the command options to compile, elaborate, and simulate the design based on the simulation settings, then launches the QuestaSim/ModelSim simulator in a separate window.

When you run simulation prior to synthesizing the design, the QuestaSim/ModelSim 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, as shown in Figure 8-10.

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 file. Use the DO file to run QuestaSim/ModelSim simulations outside of the IDE.

Using a Custom DO File during a ModelSim Run

The Vivado IDE deletes the simulation directory after every re-launch and creates a new directory.

To use a custom DO or UDO file, specify its location using the appropriate command below:

```
set_property MODELSIM.CUSTOM_DO <Customized do file name> [get_filesets sim_1]
set_property MODELSIM.CUSTOM_UDO <Customized udo file name> [get_filesets sim_1]
```

Running Post-Synthesis Functional Simulation in QuestaSim/ModelSim

When synthesis is run successfully, the Run Simulation > Post-Synthesis Functional Simulation option becomes available, as shown in Figure 8-11.
Chapter 8: Simulating with QuestaSim/ModelSim

After synthesis, the simulation information is more complete and provides a better perspective on how the functionality of your design is meeting your requirements.

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

Running Post-Synthesis Timing Simulation in QuestaSim/ModelSim

When synthesis is run successfully, the Run Simulation > Post-Synthesis Timing Simulation option becomes available, as shown in Figure 8-12.

![Figure 8-12: Run Post-Synthesis Timing Simulation](image)

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.
Using ModelSim in Command Line Mode

The following subsections describe how to run QuestaSim/ModelSim outside the Vivado IDE.

Running RTL/Behavioral Simulation in QuestaSim/ModelSim

The following are the steps involved in simulating a Xilinx design.

1. Compile simulation libraries.
2. Collect source files and create the test bench.
   - If you are using Verilog compile glbl.v, see Using Global Reset and 3-State in Chapter 2.
   - If you have SECUREIP in your design, use the precompiled libraries and point to the library using the -L switch in VSIM. For example:

   ```
   vsim -t ps -L secureip -L unisims_ver work.<testbench> work.glbl
   ```

Running Timing Simulation in QuestaSim/ModelSim

Timing simulation uses the SIMPRIM library. Ensure that you are referencing the correct libraries during the timing simulation process.

**IMPORTANT:** UNIMACRO, UNIFAST, and UNISIM libraries are not necessary for timing simulation.

Timing simulation requires that you pass in additional switches for correct pulse handling in the simulator. Add the following switches to your simulator commands:

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

Running Netlist Simulation in QuestaSim/ModelSim

The netlist simulation process involved the same steps as described in the section Running Timing Simulation in QuestaSim/ModelSim, page 147.

1. Compile simulation libraries.
2. Gather files for simulation:
   a. You can reuse the RTL simulation test bench for the majority of designs.
   b. Generate the simulation netlist (as described in the section Generating a Netlist, page 28).
      - If you are using Verilog, compile glbl.v. See Using Global Reset and 3-State in Chapter 2.
If you have **SECUREIP** in your design, use the precompiled libraries and point to the library using the `-L` switch in **VSIM**. For example:

```
vsim -t ps -L secureip -L unisims_ver work.<testbench> work.glbl
```

3. Compile and simulate the design. Refer to the QuestaSim/ModelSim user guide of the simulator you are using.

**Note:** Make sure the **UNISIM**, **SECUREIP**, and **UNIFAST** libraries are referenced correctly for proper simulation. See Using Xilinx Simulation Libraries in Chapter 2.

**Dumping SAIF in QuestaSim/ModelSim**

See Dumping the Switching Activity Interchange Format File for Power Analysis, page 39 for more information about Switching Activity Interchange Format (SAIF).

QuestaSim/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 User Guide**.

**Example do File**

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

**Annotating an SDF File in QuestaSim/ModelSim**

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

**RECOMMENDED:** Run two separate simulations to check for setup and hold violations.
To run a setup check, create an SDF with \texttt{--process} corner slow, and use the max column from the SDF, then specify:

\begin{verbatim}
-sdfmax
\end{verbatim}

To run a hold check, create an SDF with \texttt{--process} corner fast, and use the min column from the SDF. To do so, specify:

\begin{verbatim}
-sdfmin
\end{verbatim}

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

\begin{itemize}
  \item Slow corner: \texttt{SDFMIN} and \texttt{SDFMAX}
  \item Fast corner: \texttt{SDFMIN} and \texttt{SDFMAX}
\end{itemize}

\textbf{Using Verilog UNIFAST Library with QuestaSim/ModelSim}

There are two methods of simulating with the UNIFAST models.

\textbf{Method 1}

Recommended method for simulating with all the UNIFAST models.

Select the \texttt{Simulation Settings > Enable fast simulation models} check box to enable UNIFAST support in a Vivado project environment for ModelSim. See \texttt{UNIFAST Library, page 20} for more information.

\textbf{Method 2}

Recommended for more advanced who want to determine which modules to use with the UNIFAST models. See \texttt{Method 2: Using specific UNIFAST modules, page 22} for the description of this method.
Simulating with Cadence Incisive Enterprise Simulator (IES)

Introduction

This chapter provides an overview of running simulation using Cadence IES in the Vivado® Design Suite.

**IMPORTANT:** Use 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) [Ref 1].

Simulating Xilinx Designs Using Cadence IES

Cadence IES is supported through the Vivado Integrated Design Environment (IDE). You can launch those simulators directly.

The Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 2] describes the use of the Vivado IDE.

For more information on the Cadence IES simulators, see the following websites:


Setting Cadence IES for Use in the Vivado IDE

The following subsections describe two vital steps required for invoking Cadence IES from the Vivado IDE: Pointing to the Cadence IES Simulator Install Location and Compiling Simulation Libraries for Cadence/IES.
**Pointing to the Cadence IES Simulator Install Location**

To define the Cadence IES installation path:

1. Select **Tools > Options > General**
2. In the **Vivado Options > General** dialog box, scroll down the **IES Simulator install path** field, shown in **Figure 9-1**, and browse to the appropriate installation path.

![Vivado Options General Options Dialog Box](image)

**Figure 9-1: General Options Dialog Box**

**Compiling Simulation Libraries for Cadence/IES**

Before you begin simulation, run the `compile_simlib` Tcl command to compile the Xilinx simulation libraries for the target simulator. This is one time job. For details, refer to **Chapter 7, Compiling Simulation Libraries**.

**IMPORTANT:** Any change to the Vivado tools or the simulator versions requires that libraries be recompiled.

**Note:** For information on Xilinx libraries, see **Using Xilinx Simulation Libraries in Chapter 2**.
After the libraries compile, the simulator references these compiled libraries using the 
cds.lib file. The compile_simlib command copies the cds.lib file to the
<library_output_directory>.

The cds.lib file is the default initialization file and contains control variables that specify
reference library paths, optimization, compiler, and simulator settings. If the correct
cds.lib file is not found in the path, you cannot run simulation on designs that include
Xilinx primitives.

Running Cadence IES Simulation

The Flow Navigator > Simulation Settings section lets you configure the simulation
settings in Vivado IDE. The Flow Navigator Simulation menu is shown in Figure 9-2.

Figure 9-2: Simulation Settings

Simulation Settings opens the Simulation Settings dialog box where you can select and
configure the simulator.
Chapter 9: Simulating with Cadence Incisive Enterprise Simulator (IES)

**Cadence IES Simulation Options**

In the Flow Navigator, click **Simulation Settings** to open the Project Settings dialog box, shown in **Figure 9-3**.

![Project Settings Dialog Box, IES](image)

**Option Description**

**Table 9-1: Simulator Options**

<table>
<thead>
<tr>
<th>Simulator Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Simulator</td>
<td>Displays the selected simulation tool</td>
</tr>
<tr>
<td>Simulator Language</td>
<td>Displays the simulator language used</td>
</tr>
<tr>
<td>Simulation set</td>
<td>Selects which simulation to be performed</td>
</tr>
</tbody>
</table>
Selecting Compilation Options

The available compilation options are shown in Figure 9-4:

![Compilation Options Tab, IES](image)

The options are described in Table 9-2.

Table 9-2: Compile Options

<table>
<thead>
<tr>
<th>Compile Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog option</td>
<td>Browse to set Verilog include path and to define macro</td>
</tr>
<tr>
<td>Generics/Parameters options</td>
<td>Set the generic/parameter value</td>
</tr>
<tr>
<td>ies.compile.v93</td>
<td>Enable VHDL-93 features</td>
</tr>
<tr>
<td>ies.compile.32bit</td>
<td>Invoke 32-bit executable</td>
</tr>
<tr>
<td>ies.compile.relax</td>
<td>Enable relaxed VHDL interpretation</td>
</tr>
</tbody>
</table>
Chapter 9: Simulating with Cadence Incisive Enterprise Simulator (IES)

Selecting Elaboration Options

See Figure 9-5 for elaboration options.

Table 9-2: Compile Options

<table>
<thead>
<tr>
<th>Compile Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ies.compile.load_glbl</td>
<td>Load GLBL module</td>
</tr>
<tr>
<td>ies.compile.update</td>
<td>Check if unit is up-to-date before writing</td>
</tr>
<tr>
<td>ies.compile.ncvhdmore_options</td>
<td>More NCVHDL compilation options</td>
</tr>
<tr>
<td>ies.compile.ncvlog.more_options</td>
<td>More NCVLOG compilation options</td>
</tr>
</tbody>
</table>

Table 9-2: Compile Options

Table 9-3 shows elaboration options.

Table 9-3: Elaboration Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ies.elaborate.update</td>
<td>Checks if unit is up-to-date before writing</td>
</tr>
<tr>
<td>ies.elaborate.unifast</td>
<td>Enable fast simulation models</td>
</tr>
<tr>
<td>ies.elaborate.ncelab.more_options</td>
<td>More ncelab elaboration options</td>
</tr>
</tbody>
</table>
Selecting Simulation Options

Figure 9-6 shows the simulation options.

![Simulation Options Tab, IES](image)

Table 9-4 describes the simulation options.

**Table 9-4: Simulation Options**

<table>
<thead>
<tr>
<th>Simulation Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ies.simulate.runtime</td>
<td>Specify simulation run time</td>
</tr>
<tr>
<td>ies.simulate.uut</td>
<td>Specify instance name for design under test (default :/uut)</td>
</tr>
<tr>
<td>ies.simulate.update</td>
<td>Check if unit is up-to-date before writing</td>
</tr>
<tr>
<td>ies.simulate.ieee_warning</td>
<td>Suppress IEEE warnings</td>
</tr>
<tr>
<td>ies.simulate.saif</td>
<td>SAIF file name</td>
</tr>
<tr>
<td>ies.simulate.ncsim.more_option</td>
<td>More NCSIM simulation option</td>
</tr>
</tbody>
</table>
**Specify Netlist Option**

Figure 9-7 shows the Netlist tab.

![Netlist Tab, IES](image)

**Table 9-5** shows the netlist options.

**Table 9-5: Netlist Options**

<table>
<thead>
<tr>
<th>Netlist Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sdf_anno</td>
<td>Specify if sdf_annotate system task statement is generated</td>
</tr>
<tr>
<td>-process_corner</td>
<td>Specify process corner for which SDF delays are required (slow or fast)</td>
</tr>
</tbody>
</table>
Chapter 9: Simulating with Cadence Incisive Enterprise Simulator (IES)

Selecting Advanced Simulation Options

The Advanced view is shown in Figure 9-8.

![Advanced Option Tab]

This view provides an option to include all design sources for simulation. Unchecking the box gives you the flexibility to include only the files you want to simulate.

**RECOMMENDED:** It is typically best not to disable this option!

Running Timing Simulation Using IES

Timing simulation includes the following steps:

1. Generating the simulation netlist (timesim.v generation).
2. Annotating timing information to the netlist (SDF file generation).
3. Analyzing, elaborating, and simulating the timing netlist and SDF using IES.
Generating a Timing Netlist and SDF Generation in the Vivado Design Suite

Use the following Tcl commands to generate a netlist and an SDF file:

```
write_verilog -mode timesim -sdf_anno -sdf_file <sdf_file>.sdf <sim_netlist>.v
write_sdf <sdf_file>.sdf
```

IES Timing run Simulation Tcl commands:

```
irun -sdf_file <sdf_file>.sdf -y $XILINX/verilog/src/unisims \ 
$XILINX/verilog/src/glbl.v \ 
-f $XILINX_VIVADO/data/secureip/secureip_cell.list.f\ 
<testfixture>.v <sim_netlist>.v
```

Dumping SAIF for Power Analysis in IES

IES provides power commands to generate SAIF with specific requirements.

1. Specify the scope to be dumped and the output SAIF file name, using the following Tcl command:

   ```
dumpsaif -scope hdl_objects -output filename.saif
```

2. Run the simulation.

3. End the SAIF dump by typing the following Tcl command:

   ```
dumpsaif -end
```

For more detailed usage or information on IES commands, see the Cadence IES documentation [Ref 10].

Running Xilinx IP

In this use model, the `accum_0.xci` file is the IP you generated from the Vivado IP catalog. You can now run them using following commands:

```
set_property target_simulator IES [current_project]
set_property compxlib.compiled_library_dir <compiled_library_location>
launch_simulation -noclean_dir -of_objects [get_files accum_0.xci]
```
Simulating a Design with AXI Bus Functional Models

If an AXI Bus Functional Model (BFM) exists in your design, perform the following additional steps:

1. Set the `LD_LIBRARY_PATH` environment variable, using the following syntax:
   
   ```sh
   setenv LD_LIBRARY_PATH $XILINX_VIVADO/lib/lnx64.o:/$LD_LIBRARY_PATH
   ```

2. Add the following switch to VCS more options file:
   
   ```sh
   "-load $XILINX_VIVADO/lib/lnx64.o/libxil_vcs.so:xilinx_register_systf"
   ```
Chapter 10

Simulating with Synopsys VCS

Introduction

This chapter provides an overview of running simulation using Synopsys VCS in the Vivado® Design Suite.

RECOMMENDED: Use 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) [Ref 1].

Simulating Xilinx Designs using Synopsys VCS

Synopsys VCS is supported through the Vivado Integrated Design Environment (IDE). You can launch these simulators directly.

The Vivado Design Suite User Guide: Using the Vivado IDE (UG893) [Ref 2] describes the use of the Vivado IDE.

For more information on the Synopsys VCS simulators, see [Ref 11].


Setting Synopsys VCS for Use in the Vivado IDE

The following subsections describe two vital steps required for invoking Synopsys VCS from the Vivado IDE: Pointing to the Synopsys VCS Simulator Install Location and Compiling Simulation Libraries for Synopsys VCS.

Pointing to the Synopsys VCS Simulator Install Location

To define the Synopsys VCS installation path:

1. Select Tools > Options > General
2. In the Vivado Options, General dialog box, scroll down to the **VCS Simulator install path field**, shown in **Figure 10-1**, and browse to the appropriate installation path.

![Figure 10-1: Simulation General Options](image)

**Compiling Simulation Libraries for Synopsys VCS**

Before you begin simulation, run the `compile_simlib` Tcl command to compile the Xilinx simulation libraries for the target simulator. This is a one time job. For details, refer to Chapter 7, Compiling Simulation Libraries.

**IMPORTANT:** Any change to the Vivado tools or the simulator versions requires that libraries be recompiled.

**Note:** For information on Xilinx libraries, see Using Xilinx Simulation Libraries in Chapter 2.

After the libraries are compiled, the simulator references these compiled libraries using the `synopsys_sim.setup` file. The `compile_simlib` command copies the `synopsys_sim.setup` file to the `<library_output_directory>`. 
The `synopsys_sim.setup` file is the default initialization file and contains control variables that specify reference library paths, optimization, compiler, and simulator settings. If the correct `synopsys_sim.setup` file is not found in the path, you cannot run simulation on designs that include Xilinx primitives.

Running Synopsys VCS Simulation

The Flow Navigator > Simulation Settings section lets you configure the simulation settings in Vivado IDE. The Flow Navigator Simulation section is shown in Figure 10-2.

![Simulation Settings](image)

**Figure 10-2: Simulation Settings**

**Simulation Settings** opens the Simulation Settings dialog box where you can select and configure the simulator.
Synopsys VCS Simulation Options

In the Flow Navigator, click Simulation Settings to open the Project Settings dialog box, shown in Figure 10-3.

![Figure 10-3: Project Settings Dialog Box, VCS](image)

The option descriptions are shown in Table 10-1.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target simulator</td>
<td>Displays the selected simulation tool</td>
</tr>
<tr>
<td>Simulator language</td>
<td>Displays the simulator language used</td>
</tr>
<tr>
<td>Simulation set</td>
<td>Sets the simulation to be performed</td>
</tr>
</tbody>
</table>
Chapter 10: Simulating with Synopsys VCS

Selecting Compilation Options

The available Compilation tab options are shown in Figure 10-4.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation top module name</td>
<td>Top design unit name</td>
</tr>
<tr>
<td>Clean up simulation files</td>
<td>Enabled by default. Cleans the simulation files before re-run</td>
</tr>
<tr>
<td>Generate scripts only</td>
<td>Do not run simulation. Generate scripts only</td>
</tr>
<tr>
<td>Compiled library location</td>
<td>Path where library compiled using compile_simlib command has been kept</td>
</tr>
</tbody>
</table>

Table 10-2 describes the compilation options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog option</td>
<td>Browse to set the Verilog include path and to define macro</td>
</tr>
<tr>
<td>Generics/Parameters options</td>
<td>Set the generic/parameter values</td>
</tr>
<tr>
<td>vcs.compile.32bit</td>
<td>Invoke 32-bit executable</td>
</tr>
<tr>
<td>vcs.compile.load_glbl</td>
<td>Load GLBL module</td>
</tr>
</tbody>
</table>

Figure 10-4: Compilation Options Tab, VCS
Table 10-2: **Compilation Options (Cont’d)**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcs.compile.vhdlan.more_options</td>
<td>More VHDLAN compilation options</td>
</tr>
<tr>
<td>vcs.compile.vlogan.more_options</td>
<td>Extra VLOGAN compilation options</td>
</tr>
</tbody>
</table>

**Selecting Elaboration Options**

Figure 10-5 shows the Elaboration tab options.

![Figure 10-5: Elaboration Options Tab, VCS](image)

Table 10-3 describes the elaboration options.

Table 10-3: **Elaboration Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcs.elaborate.debug_pp</td>
<td>Enable post-process debug access</td>
</tr>
<tr>
<td>vcs.elaborate.unifast</td>
<td>Enable fast simulation models</td>
</tr>
<tr>
<td>vcs.elaborate.vcs.more_options</td>
<td>More VCS elaboration options</td>
</tr>
</tbody>
</table>
**Selecting Simulation Options**

Figure 10-6 shows the Simulation tab and options.

![Simulation Options Tab, VCS](image)

**Figure 10-6**: Simulation Options Tab, VCS

Table 10-4 describes the simulation tab options.

**Table 10-4: Simulation Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcs.simulate.runtime</td>
<td>Specify simulation run time</td>
</tr>
<tr>
<td>vcs.simulate.uut</td>
<td>Specifies instance name for design under test (default: /uut)</td>
</tr>
<tr>
<td>vcs.simulate.saif</td>
<td>SAIF file name</td>
</tr>
<tr>
<td>vcs.simulate.vcs.more_option</td>
<td>More VCS simulation options</td>
</tr>
</tbody>
</table>
Specify Netlist Options

Figure 10-7 shows the Netlist tab.

![Netlist Options Tab, VCS](image)

**Figure 10-7:** Netlist Options Tab, VCS

Table 10-5 describes the Netlist tab options.

**Table 10-5: Netlist Tab Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sdf_anno</td>
<td>Specify if sdf_annotate system task statement is generated.</td>
</tr>
<tr>
<td>-process_corner</td>
<td>Specify the process corner for which SDF delays are required (slow or fast)</td>
</tr>
</tbody>
</table>
Selecting Advanced Simulation Options

The Advanced tab view is shown in Figure 10-8.

![Advanced Option Tab](image)

This view provides an option to include all design sources for simulation. Unchecking the box gives you the flexibility to include only the files you want to simulate.

**RECOMMENDED:** It is typically best not to disable this option.

VCS Timing Simulation

VCS Timing simulation consists of the following steps:

1. Generating the simulation netlist (*timesim.v* generation).
2. Annotating timing information to the netlist (SDF file generation).
3. Analyzing, elaborating, and simulating the timing netlist and SDF using VCS.
Timing Netlist/SDF Generation in the Vivado Design Suite

Use the following Tcl commands to generate a netlist and an SDF file:

- **Tcl Command:**
  ```tcl
  write_verilog -mode timesim -sdf_anno -sdf_file <sdf_file>.sdf <sim_netlist>.v
  write_sdf <sdf_file>.sdf
  ```

VCS Timing Simulation Command

```
vcs +compsdf -y $XILINX/verilog/src/unisims\ $XILINX/verilog/src/glbl.v\ -f <Vivado Install>/data/secureip/secureip_cell.list.f\ +libext+.v +transport_int_delays +pulse_int_e/0 +pulse_int_r/0\ -Mupdate -R <testfixture>.v <sim_netlist>.v
```

Dumping SAIF for Power Analysis for 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.

Running Xilinx IP

In this use model, the accum_0.xci file is the IP you generated from the Vivado IP catalog. Now you can run them using following commands:

```
set_property target_simulator VCS [current_project]
set_property compxlib.compiled_library_dir <compiled_library_location>
launch_simulation -noclean_dir -of_objects [get_files accum_0.xci]
```
Simulating a Design with AXI Bus Functional Models

If an AXI Bus Functional Model (BFM) exists in your design, perform the following additional steps:

1. Set the LD_LIBRARY_PATH environment variable, using the following syntax:
   
   ```
   setenv LD_LIBRARY_PATH $XILINX_VIVADO/lib/lnx64.o/:$LD_LIBRARY_PATH
   ```

2. Add the following switch to VCS more options file:
   
   ```
   "-load $XILINX_VIVADO/lib/lnx64.o/libxil_vcs.so:xilinx_register_systf"
   ```
Introduction

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 or 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, with radix hex and a 6 bit `logic` array, the value `3F` specifies 8 bits (4 per hex digit), 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/DOWNTO 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.
Appendix B

Vivado Simulator Mixed Language Support and Language Exceptions

Introduction

The Vivado® Integrated Design Environment (IDE) supports the following languages:

- Verilog, see IEEE Standard Verilog Hardware Description Language (IEEE-STD-1364-2001) [Ref 13]
- Verilog with IP, see Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) (IEEE-STD-P1735) [Ref 16]

This appendix lists the application of Mixed Language in the Vivado simulator, and the exceptions to Verilog and VHDL, System Verilog and VHDL support.

Using Mixed Language Simulation

The Vivado simulator supports mixed language project files and mixed language simulation. This lets you include Verilog modules in a VHDL design, and vice versa.

Restrictions on Mixed Language in Simulation

- A VHDL design can instantiate Verilog/System Verilog (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. Specifically, configuration specification and direct instantiation are not supported for a Verilog/SV module instantiated inside 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 B-2, page 177.

**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.

However, Verilog/SV units can traverse through an intermediate VHDL instance to go into another Verilog/SV unit using a Verilog hierarchical reference.

In the following code snippet, the `I1.const1` is a VHDL constant referred in the Verilog/SV module, `top`. This type of Verilog/SV hierarchical reference is not allowed in the Vivado simulator. However, `I1.I2.data` is allowed inside the Verilog/SV module `top`, where `I2` is a Verilog/SV instance and `I1` is a VHDL instance:

```vhdl
-- Bottom Verilog Module
module bot;
  wire data;
endmodule

// Intermediate VHDL Entity
entity mid is
end entity mid;

architecture arch of mid is
  constant const1 : natural := 10;
begin
  bot I2();
end architecture arch;

-- Top Verilog Module
module top(input in1,output reg out1);
  mid I1();
  always@(in1)
  begin
    // This hierarchical reference into a VHDL instance is not allowed
    if(I1.const1 >= 10) out1 = in1;
    // This hierarchical reference into a Verilog instance traversing through a VHDL instance is allowed
    if (I1.I2.data == 1)out1 = ~in1;
  end
endmodule
```
Appendix B: Vivado Simulator Mixed Language Support and Language Exceptions

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. See Method 1: Using the complete UNIFAST library (Recommended), page 22 for more information.

**Note:** When using the Vivado IDE, the library search order is specified automatically. No user intervention is necessary or possible.

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, page 177.

**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:

   ```vhdl
   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:

   ```vhdl
   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:

```verilog
module testbench ;
wire in, clk;
wire out;
FD FD1(
    .Q(Q_OUT),
    .C(CLK);
    .D(A);
);
```

**Port Mapping and Supported Port Types**

Table B-1 lists the supported port types.

<table>
<thead>
<tr>
<th>VHDL 1</th>
<th>Verilog/SV 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>INPUT</td>
</tr>
<tr>
<td>OUT</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>INOUT</td>
<td>INOUT</td>
</tr>
</tbody>
</table>

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 table below shows the supported VHDL and Verilog data types for ports on the mixed language design boundary.

**Table B-2: Supported VHDL and Verilog Data Types**

<table>
<thead>
<tr>
<th>VHDL Port</th>
<th>Verilog Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>net</td>
</tr>
<tr>
<td>std_logic</td>
<td>net</td>
</tr>
<tr>
<td>bit_vector</td>
<td>vector net</td>
</tr>
<tr>
<td>signed</td>
<td>vector net</td>
</tr>
<tr>
<td>unsigned</td>
<td>vector net</td>
</tr>
<tr>
<td>std_ulogic_vector</td>
<td>vector net</td>
</tr>
<tr>
<td>std_logic_vector</td>
<td>vector net</td>
</tr>
</tbody>
</table>
Appendix B: Vivado Simulator Mixed Language Support and Language Exceptions

**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 B-2) are also supported as a record element.

**Table B-3: Supported SV and VHDL Data Types**

<table>
<thead>
<tr>
<th>SV Data type</th>
<th>VHDL Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>byte</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>shortint</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>longint</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>integer</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
</tbody>
</table>
## Table B-3: Supported SV and VHDL Data Types (Cont’d)

<table>
<thead>
<tr>
<th>SV Data type</th>
<th>VHDL Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector of bit(1D)</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>vector of logic(1D)</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>vector of reg(1D)</td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td>logic/bit</td>
<td>bit</td>
</tr>
<tr>
<td></td>
<td>std_logic</td>
</tr>
<tr>
<td></td>
<td>std_ulogic</td>
</tr>
<tr>
<td></td>
<td>bit_vector</td>
</tr>
<tr>
<td></td>
<td>std_logic_Vector</td>
</tr>
<tr>
<td></td>
<td>std_ulogic_vector</td>
</tr>
<tr>
<td></td>
<td>signed</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
</tr>
</tbody>
</table>
Generics (Parameters) Mapping

The Vivado simulator supports the following VHDL generic types (and their Verilog equivalents):

- integer
- real
- string
- boolean

*Note:* Any other generic type found on mixed language boundary is considered an error.

VHDL and Verilog Values Mapping

*Table B-4* lists the Verilog states mappings to `std_logic` and `bit`.

<table>
<thead>
<tr>
<th>Verilog</th>
<th>std_logic</th>
<th>bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

*Note:* Verilog strength is ignored. There is no corresponding mapping to strength in VHDL.

*Table B-5* lists the VHDL type `bit` mapping to Verilog states.

<table>
<thead>
<tr>
<th>bit</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table B-6* lists the VHDL type `std_logic` mappings to Verilog states.

<table>
<thead>
<tr>
<th>std_logic</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
</tr>
</tbody>
</table>
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. Table B-7 lists the VHDL language support exceptions.

<table>
<thead>
<tr>
<th>Supported VHDL Construct</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract_literal</td>
<td>Floating point expressed as based literals are not supported.</td>
</tr>
<tr>
<td>alias_declaration</td>
<td>Alias to non-objects are in general not supported; particularly the following: 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</td>
</tr>
<tr>
<td>alias_designator</td>
<td>Operator_symbol as alias_designator Character_literal as alias_designator</td>
</tr>
<tr>
<td>association_element</td>
<td>Globally, locally static range is acceptable for taking slice of an actual in an association element.</td>
</tr>
<tr>
<td>attribute_name</td>
<td>Signature after prefix is not supported.</td>
</tr>
<tr>
<td>binding_indication</td>
<td>Binding_indication without use of entity_aspect is not supported.</td>
</tr>
<tr>
<td>bit_string_literal.</td>
<td>Empty bit_string_literal (&quot; &quot;) is not supported.</td>
</tr>
<tr>
<td>block_statement</td>
<td>Guard_expression is not supported; for example, guarded blocks, guarded signals, guarded targets, and guarded assignments are not supported.</td>
</tr>
<tr>
<td>choice</td>
<td>Aggregate used as choice in case statement is not supported.</td>
</tr>
<tr>
<td>concurrent_assertion_statement</td>
<td>Postponed is not supported.</td>
</tr>
<tr>
<td>concurrent_signal_assignment_statement</td>
<td>Postponed is not supported.</td>
</tr>
</tbody>
</table>
### Supported VHDL Construct

<table>
<thead>
<tr>
<th>Construct</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrent_statement</td>
<td>Concurrent procedure call containing wait statement is not supported.</td>
</tr>
<tr>
<td>conditional_signal_assignment</td>
<td>Keyword guarded as part of options is not supported as there is no support for guarded signal assignment.</td>
</tr>
<tr>
<td>configuration_declaration</td>
<td>Non locally static for generate index used in configuration is not supported.</td>
</tr>
<tr>
<td>entity_class</td>
<td>Literals, unit, file, and group as entity class are not supported.</td>
</tr>
<tr>
<td>entity_class_entry</td>
<td>Optional &lt;&gt; intended for use with group templates is not supported.</td>
</tr>
<tr>
<td>file_logical_name</td>
<td>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.</td>
</tr>
<tr>
<td>function_call</td>
<td>Slicing, indexing, and selection of formals is not supported in a named parameter association within a function_call.</td>
</tr>
<tr>
<td>instantiated_unit</td>
<td>Direct configuration instantiation is not supported.</td>
</tr>
<tr>
<td>mode</td>
<td>Linkage and Buffer ports are not supported completely.</td>
</tr>
<tr>
<td>options</td>
<td>Guarded is not supported.</td>
</tr>
<tr>
<td>primary</td>
<td>At places where primary is used, allocator is expanded there.</td>
</tr>
<tr>
<td>procedure_call</td>
<td>Slicing, indexing, and selection of formals is not supported in a named parameter association within a procedure_call.</td>
</tr>
<tr>
<td>process_statement</td>
<td>Postponed processes are not supported.</td>
</tr>
<tr>
<td>selected_signal_assignment</td>
<td>The guarded keyword as part of options is not supported as there is no support for guarded signal assignment.</td>
</tr>
<tr>
<td>signal_declaration</td>
<td>The signal_kind is not supported. The signal_kind is used for declaring guarded signals, which are not supported.</td>
</tr>
<tr>
<td>subtype_indication</td>
<td>Resolved subtype of composites (arrays and records) is not supported</td>
</tr>
<tr>
<td>waveform</td>
<td>Unaffected is not supported.</td>
</tr>
<tr>
<td>waveform_element</td>
<td>Null waveform element is not supported as it only has relevance in the context of guarded signals.</td>
</tr>
</tbody>
</table>
# Verilog Language Support Exceptions

Table B-8 lists the exceptions to supported Verilog language support.

## Table B-8: Verilog Language Support Exceptions

<table>
<thead>
<tr>
<th>Verilog Construct</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compiler Directive Constructs</strong></td>
<td></td>
</tr>
<tr>
<td>`unconnected_drive</td>
<td>not supported</td>
</tr>
<tr>
<td>`nounconnected_drive</td>
<td>not supported</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td></td>
</tr>
<tr>
<td>attribute_instance</td>
<td>not supported</td>
</tr>
<tr>
<td>attr_spec</td>
<td>not supported</td>
</tr>
<tr>
<td>attr_name</td>
<td>not supported</td>
</tr>
<tr>
<td><strong>Primitive Gate and Switch Types</strong></td>
<td></td>
</tr>
<tr>
<td>cmos_switchtype</td>
<td>not supported</td>
</tr>
<tr>
<td>mos_switchtype</td>
<td>not supported</td>
</tr>
<tr>
<td>pass_en_switchtype</td>
<td>not supported</td>
</tr>
<tr>
<td><strong>Generated Instantiation</strong></td>
<td></td>
</tr>
</tbody>
</table>
| generated_instantiation    | The module_or_generate_item alternative is not supported. Production from standard (see IEEE Standard Verilog Hardware Description Language (IEEE 1364-2001) section 13.2 [Ref 13]:
\[
generate_item_or_null ::= 
generate_conditional_statement | 
generate_case_statement | 
generate_loop_statement | 
generate_block | 
module_or_generate_item
\]
Production supported by Simulator:
\[
generate_item_or_null ::= 
generate_conditional_statement | 
generate_case_statement | 
generate_loop_statement | 
generate_block | 
generate_condition
\]
### Table B-8: Verilog Language Support Exceptions (Cont’d)

<table>
<thead>
<tr>
<th>Verilog Construct</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>genvar_assignment</td>
<td>Partially supported. All generate blocks must be named.</td>
</tr>
<tr>
<td></td>
<td>Production from standard (see IEEE Standard Verilog Hardware Description</td>
</tr>
<tr>
<td></td>
<td>Language (IEEE 1364-2001) section 13.2 [Ref 13]:</td>
</tr>
<tr>
<td></td>
<td>generate_block ::=</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>[ : generate_block_identifier ]</td>
</tr>
<tr>
<td></td>
<td>{ generate_item }</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>Production supported by Simulator:</td>
</tr>
<tr>
<td></td>
<td>generate_block ::=</td>
</tr>
<tr>
<td></td>
<td>begin:</td>
</tr>
<tr>
<td></td>
<td>generate_block_identifier {</td>
</tr>
<tr>
<td></td>
<td>generate_item }</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

**Source Text Constructs**

- **Library Source Text**: not supported
- **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 1364-2001) section 13.2 [Ref 13]. This does not refer to the `include compiler directive.

**System Timing Check Commands**

- **$skew_timing_check**: not supported
- **$timeskew_timing_check**: not supported
- **$fullskew_timing_check**: not supported
- **$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
### Table B-8: Verilog Language Support Exceptions (Cont’d)

<table>
<thead>
<tr>
<th>Verilog Construct</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sync$or$array</td>
<td>not supported</td>
</tr>
<tr>
<td>$async$and$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$async$nand$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$async$nor$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$async$or$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$sync$and$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$sync$nand$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$sync$nor$plane</td>
<td>not supported</td>
</tr>
<tr>
<td>$sync$or$plane</td>
<td>not supported</td>
</tr>
</tbody>
</table>

#### Value Change Dump (VCD) Files

<table>
<thead>
<tr>
<th>Dump Command</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dumpportson</td>
<td>not supported</td>
</tr>
<tr>
<td>$dumpports</td>
<td></td>
</tr>
<tr>
<td>$dumpportsoff</td>
<td></td>
</tr>
<tr>
<td>$dumpportsflush</td>
<td></td>
</tr>
<tr>
<td>$dumpportslimit</td>
<td></td>
</tr>
<tr>
<td>$vcdplus</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

Vivado Simulator Quick Reference Guide

Introduction

Table C-1 provides a quick reference and examples for common Vivado® simulator commands.

Table C-1: Standalone Mode: Parsing, Elaborating, and Running Simulation from a Command Line

<table>
<thead>
<tr>
<th>Parsing HDL Files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vivado Simulator</strong> supports three HDL file types: Verilog, SystemVerilog and VHDL. You can parse the supported files using XVHDL and XVLOG commands.</td>
</tr>
<tr>
<td><strong>Parsing VHDL files</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Parsing Verilog files</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Parsing SystemVerilog files</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Note:** For information about the PRJ file format, see Project File (.prj) Syntax in Chapter 5.

Additional xvlog and xvhd1 Options
## Table C-1: Standalone Mode: Parsing, Elaborating, and Running Simulation from a Command Line (Cont’d)

<table>
<thead>
<tr>
<th>Key Option</th>
<th>Applies to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d [define] &lt;name&gt;[=&lt;val&gt;]</td>
<td>xvlog</td>
</tr>
<tr>
<td>-h [-help]</td>
<td>xvlog, xvhd</td>
</tr>
<tr>
<td>-i [include] &lt;directory_name&gt;</td>
<td>xvlog</td>
</tr>
<tr>
<td>-initfile &lt;init_filename&gt;</td>
<td>xvlog, xvhd</td>
</tr>
<tr>
<td>-L [-lib] &lt;library_name&gt;[=&lt;library_dir&gt;]</td>
<td>xvlog, xvhd</td>
</tr>
<tr>
<td>-log &lt;filename&gt;</td>
<td>xvlog, xvhd</td>
</tr>
<tr>
<td>-prj &lt;filename&gt;</td>
<td>xvlog, xvhd</td>
</tr>
<tr>
<td>-relax</td>
<td>xvhd, vlog</td>
</tr>
<tr>
<td>-work &lt;library_name&gt;[=&lt;library_dir&gt;]</td>
<td>xvlog, xvhd</td>
</tr>
</tbody>
</table>

### Elaborating and Generating an Executable Snapshot

After parsing, you can elaborate the design in Vivado simulator using the XELAB command. XELAB generates an executable snapshot.

*Note:* You can skip the parser stage, directly invoke the XELAB command, and pass the PRJ file. XELAB calls XVLOG and XVHDL for parsing the files.

**Usage**

- **xelab top1 top2**
  Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units are compiled in the `/work` library.

- **xelab lib1.top1 lib2.top2**
  Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units are compiled in `lib1` and `lib2`, respectively.

- **xelab top1 top2 -prj files.prj**
  Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units are compiled in the `/work` library. The file `files.prj` contains entries such as:
  - `verilog <libraryName> <VerilogDesignFileName>`
  - `vhdl <libraryName> <VHDLDesignFileName>`
  - `sv <libraryName> <SystemVerilogDesignFileName>`

- **xelab top1 top2 -s top**
  Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units are compiled in the `/work` library. After compilation, `xelab` generates an executable snapshot with the name `top`. Without the `-s top` switch, `xelab` creates the snapshot by concatenating the unit names.

**Command Line Help and xelab Options**

- **xelab -help**
- **xelab, xvhd, and xvlog Command Options, page 102**
## Running Simulation

After parsing, elaboration and compilation stages are successful; xsim generates an executable snapshot to run simulation.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsim top -R</td>
<td>Simulates the design to through completion.</td>
</tr>
<tr>
<td>xsim top -gui</td>
<td>Opens the Vivado simulator workspace (GUI).</td>
</tr>
</tbody>
</table>
| xsim top       | Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as:  
                 | run -all                                              |
|                | run 100 ns                                            |

### Important Shortcuts

You can invoke the parsing, elaboration, and executable generation and simulation in one, two, or three stages.

<table>
<thead>
<tr>
<th>Three Stage</th>
<th>xsim top -R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simulates the design to through completion.</td>
</tr>
<tr>
<td></td>
<td>xsim top -gui Opens the Vivado simulator workspace (GUI).</td>
</tr>
</tbody>
</table>
|             | xsim top Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as:  
             | run -all                                              |
|             | run 100 ns                                            |

<table>
<thead>
<tr>
<th>Two Stage</th>
<th>xsim top -R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simulates the design to through completion.</td>
</tr>
<tr>
<td></td>
<td>xsim top -gui Opens the Vivado simulator workspace (GUI).</td>
</tr>
</tbody>
</table>
|              | xsim top Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as:  
              | run -all                                              |
|              | run 100 ns                                            |

<table>
<thead>
<tr>
<th>Single Stage</th>
<th>xsim top -R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simulates the design to through completion.</td>
</tr>
<tr>
<td></td>
<td>xsim top -gui Opens the Vivado simulator workspace (GUI).</td>
</tr>
</tbody>
</table>
|              | xsim top Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as:  
              | run -all                                              |
|              | run 100 ns                                            |

### Vivado Simulation Tcl Commands

The following are commonly used Tcl commands. For a complete list, invoke following commands in the Tcl Console:

- `load_features simulator`
- `help -category simulation`

For information on any Tcl Command, type: `-help <Tcl_command>`
<table>
<thead>
<tr>
<th><strong>Common Vivado Simulator Tcl Commands:</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>add_bp</td>
<td>Add break point at a line of HDL source. See <a href="#">Tcl commands; page 90</a> for more information</td>
</tr>
<tr>
<td>add_force</td>
<td>Force the value of a signal, wire, or register to a specified value.</td>
</tr>
<tr>
<td>current_time now</td>
<td>Report current simulation time. See <a href="#">current_time, page 112</a> for an example of this command within a Tcl script.</td>
</tr>
<tr>
<td>current_scope</td>
<td>Report or set the current, working HDL scope. See <a href="#">Additional Scopes and Sources Options, page 58</a> for more information.</td>
</tr>
<tr>
<td>get_objects</td>
<td>Get a list of HDL objects in one or more HDL scopes, per the specified pattern. For example command usage refer to: `log_saif [get_objects -filter {type == in_port</td>
</tr>
<tr>
<td>get_scopes</td>
<td>Get a list of child HDL scopes. See <a href="#">Additional Scopes and Sources Options, page 58</a> for more information.</td>
</tr>
<tr>
<td>get_value</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>launch_simulation</td>
<td>Launch simulation using the Vivado simulator.</td>
</tr>
<tr>
<td>remove_bps</td>
<td>Remove breakpoints from a simulation. Reference for more information: Type the Tcl command remove_bp, page 120.</td>
</tr>
<tr>
<td>report_drivers</td>
<td>Print drivers along with current driving values for an HDL wire or signal object. Reference for more information: <a href="#">Using the report_drivers Tcl Command, page 126</a>.</td>
</tr>
<tr>
<td>report_values</td>
<td>Print current simulated value of given HDL objects (variables, signals, wires, or registers). Reference for more information: <code>report_drivers &lt;hdl_object&gt;</code>, page 126.</td>
</tr>
<tr>
<td>restart</td>
<td>Rewind simulation to post loading state (as though the design was reloaded); time is set to 0. Reference for more information: Restart: Lets you restart an existing simulation from 0. Tcl Command: restart, page 52.</td>
</tr>
<tr>
<td>set_value</td>
<td>Set the HDL object (variable, signal, wire, or register) to a specified value. Reference for more information: <a href="#">Appendix A, Value Rules in Vivado Simulator Tcl Commands</a>.</td>
</tr>
<tr>
<td>step</td>
<td>Step simulation to the next statement. See <a href="#">Simulation Toolbar, page 54</a>.</td>
</tr>
</tbody>
</table>
SystemVerilog Constructs Supported by the Vivado Simulator

Introduction

The Vivado® simulator supports the subset of SystemVerilog RTL that can be synthesized. The complete list is given in Table D-1.

Targeting SystemVerilog for a Specific File

By default, the Vivado simulator tool 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 Source Node Properties.
2. In the Source Node Properties window, change the Set File Type from Verilog to SystemVerilog and click **Apply**. See **Figure D-1**.

![Source Node Properties window](image)

*Figure D-1: Source Node Properties > Set File Type*

Alternatively, you can use the following Tcl command in the Tcl Console:

```
set_property file_type SystemVerilog [get_files <filename>.v]
```
Appendix D: SystemVerilog Constructs Supported by the Vivado Simulator

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:

```markdown
xvlog  -sv  <Design file list>
```

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`.

```markdown
xvlog  -prj  <prj File>
```

Where the entry in `prj` file appears as follows:

```markdown
verilog  library1  <FileName>
sv       library1  <FileName>  [File parsed in SystemVerilog mode]
vhdl     library2  <FileName>
sv       library3  <FileName>  [File parsed in SystemVerilog mode]
```

**Table D-1: Synthesizable Set of System Verilog 1800-2009**

<table>
<thead>
<tr>
<th>Primary construct</th>
<th>Secondary construct</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td></td>
<td>6</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Singular and aggregate types</td>
<td>6.4</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Nets and variables</td>
<td>6.5</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Variable declarations</td>
<td>6.8</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Vector declarations</td>
<td>6.9</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>2-state (two-value) and 4-state (four-value) data types</td>
<td>6.11.2</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Signed and unsigned integer types</td>
<td>6.11.3</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Real, shortreal and realtime data types</td>
<td>6.12</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>User-defined types</td>
<td>6.18</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Enumerations</td>
<td>6.19</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Defining new data types as enumerated types</td>
<td>6.19.1</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Enumerated type ranges</td>
<td>6.19.2</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td>Type checking</td>
<td>6.19.3</td>
<td>Supported</td>
</tr>
</tbody>
</table>
### Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont’d)

<table>
<thead>
<tr>
<th>Primary construct</th>
<th>Secondary construct</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumerated types in numerical expressions</td>
<td></td>
<td>6.19.4</td>
<td>Supported</td>
</tr>
<tr>
<td>Enumerated type methods</td>
<td></td>
<td>6.19.5</td>
<td>Supported</td>
</tr>
<tr>
<td>Type parameters</td>
<td></td>
<td>6.20.3</td>
<td>Supported</td>
</tr>
<tr>
<td>Const constants</td>
<td></td>
<td>6.20.6</td>
<td>Supported</td>
</tr>
<tr>
<td>Type operator</td>
<td></td>
<td>6.23</td>
<td>Supported</td>
</tr>
<tr>
<td>Cast operator</td>
<td></td>
<td>6.24.1</td>
<td>Supported</td>
</tr>
<tr>
<td>$\text{cast}$ dynamic casting</td>
<td></td>
<td>6.24.2</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Bitstream casting</td>
<td></td>
<td>6.24.3</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Aggregate data types</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structures</td>
<td></td>
<td>7</td>
<td>Supported</td>
</tr>
<tr>
<td>Packed/Unpacked structures</td>
<td></td>
<td>7.2.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Assigning to structures</td>
<td></td>
<td>7.2.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Unions</td>
<td></td>
<td>7.3</td>
<td>Supported</td>
</tr>
<tr>
<td>Packed/Unpacked unions</td>
<td></td>
<td>7.3.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Tagged unions</td>
<td></td>
<td>7.3.2</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Packed arrays</td>
<td></td>
<td>7.4.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Unpacked arrays</td>
<td></td>
<td>7.4.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Operations on arrays</td>
<td></td>
<td>7.4.3</td>
<td>Supported</td>
</tr>
<tr>
<td>Multidimensional arrays</td>
<td></td>
<td>7.4.5</td>
<td>Supported</td>
</tr>
<tr>
<td>Indexing and slicing of arrays</td>
<td></td>
<td>7.4.6</td>
<td>Supported</td>
</tr>
<tr>
<td>Array assignments</td>
<td></td>
<td>7.6</td>
<td>Supported</td>
</tr>
<tr>
<td>Arrays as arguments to subroutines</td>
<td></td>
<td>7.7</td>
<td>Supported</td>
</tr>
<tr>
<td>Array querying functions</td>
<td></td>
<td>7.11</td>
<td>Supported</td>
</tr>
<tr>
<td>Array manipulation methods (those that do not return queue type)</td>
<td></td>
<td>7.12</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Processes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinational logic always_comb procedure</td>
<td></td>
<td>9.2.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Implicit always_comb sensitivities</td>
<td></td>
<td>9.2.2.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Latched logic always_latch procedure</td>
<td></td>
<td>9.2.2.3</td>
<td>Supported</td>
</tr>
<tr>
<td>Sequential logic always_ff procedure</td>
<td></td>
<td>9.2.2.4</td>
<td>Supported</td>
</tr>
<tr>
<td>Sequential blocks</td>
<td></td>
<td>9.3.1</td>
<td>Supported</td>
</tr>
</tbody>
</table>
### Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont’d)

<table>
<thead>
<tr>
<th>Primary construct</th>
<th>Secondary construct</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel blocks</td>
<td>9.3.2</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Procedural timing controls</td>
<td>9.4</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Conditional event controls</td>
<td>9.4.2.3</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Sequence events</td>
<td>9.4.2.4</td>
<td></td>
<td>Not Supported</td>
</tr>
</tbody>
</table>

**Assignment statement**

<table>
<thead>
<tr>
<th>Assignment statement</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>The continuous assignment statement</td>
<td>10.3.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Variable declaration assignment (variable initialization)</td>
<td>10.5</td>
<td>Supported</td>
</tr>
<tr>
<td>Assignment-like contexts</td>
<td>10.8</td>
<td>Supported</td>
</tr>
<tr>
<td>Array assignment patterns</td>
<td>10.9.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Structure assignment patterns</td>
<td>10.9.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Unpacked array concatenation</td>
<td>10.10</td>
<td>Supported</td>
</tr>
<tr>
<td>Net aliasing</td>
<td>10.11</td>
<td>Not Supported</td>
</tr>
</tbody>
</table>

**Operators and expressions**

<table>
<thead>
<tr>
<th>Operators and expressions</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant expressions</td>
<td>11.2.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Aggregate expressions</td>
<td>11.2.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Operators with real operands</td>
<td>11.3.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Operations on logic (4-state) and bit (2-state) types</td>
<td>11.3.4</td>
<td>Supported</td>
</tr>
<tr>
<td>Assignment within an expression</td>
<td>11.3.6</td>
<td>Supported</td>
</tr>
<tr>
<td>Assignment operators</td>
<td>11.4.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Increment and decrement operators</td>
<td>11.4.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Arithmetic expressions with unsigned and signed types</td>
<td>11.4.3.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Wildcard equality operators</td>
<td>11.4.6</td>
<td>Supported</td>
</tr>
<tr>
<td>Concatenation operators</td>
<td>11.4.12</td>
<td>Supported</td>
</tr>
<tr>
<td>Set membership operator</td>
<td>11.4.13</td>
<td>Supported</td>
</tr>
<tr>
<td>Concatenation of stream_expressions</td>
<td>11.4.14.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Re-ordering of the generic stream</td>
<td>11.4.14.2</td>
<td>Supported</td>
</tr>
</tbody>
</table>
### Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont’d)

<table>
<thead>
<tr>
<th>Primary construct</th>
<th>Secondary construct</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming concatenation as an assignment target (unpack)</td>
<td></td>
<td>11.4.14.3</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Streaming dynamically sized data</td>
<td></td>
<td>11.4.14.4</td>
<td>Not Supported</td>
</tr>
<tr>
<td><strong>Procedural programming statement</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Unique-if, unique0-if and priority-if</strong></td>
<td>12.4.2</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Violation reports generated by unique-case, unique0-case, and priority-case</strong></td>
<td>12.4.2.1</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>If statement violation reports and multiple processes</strong></td>
<td>12.4.2.2</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>unique-case, unique0-case, and priority-case</strong></td>
<td>12.5.3</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Violation reports generated by unique-case, unique0-case, and priority-case</strong></td>
<td>12.5.3.1</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Case statement violation reports and multiple processes</strong></td>
<td>12.5.3.2</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Set membership case statement</strong></td>
<td>12.5.4</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Pattern matching conditional statements</strong></td>
<td>12.6</td>
<td>Not Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Loop statements</strong></td>
<td>12.7</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Jump statement</strong></td>
<td>12.8</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Tasks</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Static and Automatic task</strong></td>
<td>13.3.1</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Tasks memory usage and concurrent activation</strong></td>
<td>13.3.2</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Return values and void functions</strong></td>
<td>13.4.1</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td><strong>Static and Automatic function</strong></td>
<td>13.4.2</td>
<td>Supported</td>
</tr>
</tbody>
</table>
### Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont’d)

<table>
<thead>
<tr>
<th>Primary construct</th>
<th>Secondary construct</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant function</td>
<td></td>
<td>13.4.3</td>
<td>Supported</td>
</tr>
<tr>
<td>Background process spawned by function call</td>
<td></td>
<td>13.4.4</td>
<td>Not Supported</td>
</tr>
<tr>
<td><strong>Subroutine calls and argument passing</strong></td>
<td></td>
<td>13.5</td>
<td></td>
</tr>
<tr>
<td>Pass by value</td>
<td></td>
<td>13.5.1</td>
<td>Supported</td>
</tr>
<tr>
<td>Pass by reference</td>
<td></td>
<td>13.5.2</td>
<td>Supported</td>
</tr>
<tr>
<td>Default argument value</td>
<td></td>
<td>13.5.3</td>
<td>Supported</td>
</tr>
<tr>
<td>Argument binding by name</td>
<td></td>
<td>13.5.4</td>
<td>Supported</td>
</tr>
<tr>
<td>Optional argument list</td>
<td></td>
<td>13.5.5</td>
<td>Supported</td>
</tr>
<tr>
<td>Import and Export function</td>
<td></td>
<td>13.6</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Task and function name</td>
<td></td>
<td>13.7</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Utility system tasks and system functions (only synthesizable set)</strong></td>
<td>20</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td><strong>I/O system tasks and system functions (only synthesizable set)</strong></td>
<td>21</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler directives</strong></td>
<td>22</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td><strong>Modules and hierarchy</strong></td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default port values</td>
<td>23.2.2.4</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Top-level modules and $root</td>
<td>23.3.1</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Module instantiation syntax</td>
<td>23.3.2</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Nested modules</td>
<td>23.4</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Extern modules</td>
<td>23.5</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Hierarchical names</td>
<td>23.6</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Member selects and hierarchical names</td>
<td>23.7</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Upwards name referencing</td>
<td>23.8</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Overriding module parameters</td>
<td>23.10</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Binding auxiliary code to scopes or instances</td>
<td>23.11</td>
<td>Not Supported</td>
<td></td>
</tr>
<tr>
<td><strong>Interfaces</strong></td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface syntax</td>
<td>25.3</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>Nested interface</td>
<td>25.3</td>
<td>Supported</td>
<td></td>
</tr>
</tbody>
</table>
## Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont’d)

<table>
<thead>
<tr>
<th>Primary construct</th>
<th>Secondary construct</th>
<th>LRM section</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports in interfaces</td>
<td>25.4</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Example of named port bundle</td>
<td>25.5.1</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Example of connecting port bundle</td>
<td>25.5.2</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Example of connecting port bundle to generic interface</td>
<td>25.5.3</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Modport expressions</td>
<td>25.5.4</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Clocking blocks and modports</td>
<td>25.5.5</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Interfaces and specify blocks</td>
<td>25.6</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Example of using tasks in interface</td>
<td>25.7.1</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Example of using tasks in modports</td>
<td>25.7.2</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Example of exporting tasks and functions</td>
<td>25.7.3</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Example of multiple task exports</td>
<td>25.7.4</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Parameterized interfaces</td>
<td>25.8</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Virtual interfaces</td>
<td>25.9</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Packages</td>
<td></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Package declarations</td>
<td>26.2</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Referencing data in packages</td>
<td>26.3</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Using packages in module headers</td>
<td>26.4</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Exporting imported names from packages</td>
<td>26.6</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>The std built-in package</td>
<td>26.7</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Generate constructs</td>
<td></td>
<td>27</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Appendix E

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 currently supports a subset of DPI features, as described below.

Compiling C Code

A new compiler executable, xsc, is provided to convert C code into object an ode 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 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.
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

Description of the xsc Compiler

The xsc compiler helps create a shared library (.a on Windows or .so on Linux) from one or more C files. You use xelab to bind the shared library generated by xsc into the rest of your design. You can create a shared library using a one- or two-step process:

One-step process:

Pass all C files to xsc without using the -compile or -link switch.

Two-step process:

  xsc -compile <C files>
  xsc -link <object files>

Usage:

xsc [options] <files...>

Switches

You can use a double dash (--) or a single dash (-) for switches

-compile          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.
-link              Run only the linking stage to generate the shared library (.a or .so) from the object files.
-mt arg (=auto)    Specifies the number of sub-compilation jobs that can be run in parallel. Choices are:
                   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 the -link option only.
-work arg          Specify the work directory in which to place the outputs. (Default: <current_directory>/xsim.dir/xsc)
-v [ -verbose ] arg Specify verbosity level for printing messages.
                   Allowed values are: 0, 1
                   (Default: 0)
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

Examples

xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi

Example

xsc -compile function1.c function2.c -work abc
xsc -link abc/function1.lnx64.o abc/function2.lnx64.o -work abc

BINDING COMPILRED C CODE TO SYSTEMVERILOG USING XELAB

The DPI-related switches for xelab that bind the compiled C code to SystemVerilog are as follows:

- **-sv_root arg** Root directory relative to which a DPI shared library should be searched. (Default: `<current_directory>/xsim.dir/xsc`

- **-sv_lib arg** Name of the DPI shared library without the file extension defining C function imported in SystemVerilog.

- **-sv_liblist arg** Bootstrap file pointing to DPI shared libraries.

- **-dpiheader arg** Generate a DPI C header file containing C declaration of imported and exported functions.


DATA TYPES ALLOWED ON THE BOUNDARY OF C AND SYSTEMVERILOG

The IEEE Standard for SystemVerilog [Ref 14] 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 E-1: Data Types Allowed on the C-SystemVerilog Boundary

<table>
<thead>
<tr>
<th>SystemVerilog</th>
<th>C</th>
<th>Supported</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>char</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>shortint</td>
<td>short int</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>longint</td>
<td>long long</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>real</td>
<td>double</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>shortreal</td>
<td>float</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>chandle</td>
<td>void *</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>string</td>
<td>const char*</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>bit</td>
<td>unsigned char</td>
<td>Yes</td>
<td>sv_0, sv_1, sv_z, sv_x</td>
</tr>
</tbody>
</table>

Available on C side using svdpi.h

| logic, reg | unsigned char | Yes | sv_0, sv_1, sv_z, sv_x: |

Available on C side using svdpi.h

| Array (packed) of bits | svBitVecVal | Yes | Defined in svdpi.h |
| Array (packed) of logic/reg | svLogicVecVal | Yes | Defined in svdpi.h |
| enum | Underlying enum type | Import Only | SystemVerilog to C call only |
| Packed arrays of bit, logic | Passed as array | Yes | None |
| Packed structs, unions | Passed as array | Import Only | SystemVerilog to C call only |
| Unpacked arrays of bit, logic | Passed as array | Export Only | C can call SystemVerilog |
| Unpacked structs, unions | Passed as struct | No | None |
| Open arrays | svOpenArrayHandle | No | 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 [Ref 14].
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

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:

```systemverilog
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:

```c
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:

```systemverilog
typedef struct packed {
    int i;
    bit b;
    reg [3:0]r;
    logic [2:0] [4:8] [9:1] l;
} sType;
sType c_obj;
sType c_obj1[5];
```
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

C type:

```c
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/Union

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:

```systemverilog
typedef struct {
    int i;
    bit b;
    reg r[3:0];
    logic [2:0] l[4:8][9:1];
} sType;
```

C type:

```c
typedef struct {
    int i;
    svBit b;
    svLogic r[4];
    svLogic l[5][9];
    svLogicVecVal l[5][9][SV_PACKED_DATA_NELEMS(3)];
} sType;
```
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

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:

```c
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);
```

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 these functions
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

```
cat function1.c

#include "svdpi.h"

DPI_DLLESPEC
int myFunction1()
{
    return 5;
}

cat function2.c

#include <stdio.h>
DPI_DLLESPEC
int myFunction2()
{
    return 10;
}

cat 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 -link 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`.

**Two-step flow (same as above with few extra options)**

```
xsc -compile function1.c function2.c -work "abc" -v 1
xsc -link "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.
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

Function with Output

Code

cat file.sv

/*- - - -*/
package pack1;
import "DPI-C" pure function int myFunction1(input int v, output int o);
import "DPI-C" pure 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

demodule


cat 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;
}
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

cat 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:

- You run xelab with the -dpiheader switch to create the header file, file.h.
- Your code in file.c then includes the xelab-generated header file (file.h), which is listed at the end.
- Compile the code in file.c and test.sv as before to generate the simulation executable.

cat 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);
}

cat 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
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator

cat run.ksh

xelab -dpiheader file.h -svlog test.sv
xsc file.c
xelab -svlog test.sv -sv_lib dpi -R

cat file.h

﻿#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_DLLISPEC
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.”
Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation References

4. Writing Efficient Testbenches (XAPP199)
8. Vivado Design Suite Tutorial: Simulation (UG937)


---

Links to Language and Encryption Support Standards


---

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. [Vivado Design Suite Tool Flow Training Course](https://www.xilinx.com/products/designHello-vivado-trainingcourse.html)


3. [Vivado Design Suite Quick Take Video: Logic Simulation](https://www.xilinx.com/products/designHello-vivado-trainingcourse.html)

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 http://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 http://www.xilinx.com/legal.htm#tos.

© Copyright 2012-2014 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, UltraScale, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.