Vivado Design Suite User Guide

Logic Simulation

UG900 (v2016.2) June 8, 2016

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/08/2016	2016.2	• Updated the Tcl command in Using the complete UNIFAST library section of Chapter 2, Preparing for Simulation. • Added a note regarding export_simulation script in export_simulation section in Chapter 7, Simulating in Batch or Scripted Mode.
04/06/2016	2016.1	Chapter 3, Understanding Vivado Simulatorr ◦ Added a Note in Closing a Simulation section. Chapter 7, Simulating in Batch or Scripted Mode ◦ Added -dpi_absolute command option in Table 7-2. Chapter 8, Using Third-Party Simulators ◦ Added a note regarding VCS simulator in Running Simulation Using Third-Party Tools section. ◦ Updated Simulation Step Control Constructs for ModelSim and Questa section. Updated Appendix F, Direct Programming Interface (DPI) in Vivado Simulator Updated Appendix G, Using Xilinx Simulator Interface

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: Preparing for Simulation

Introduction 12

Using Test Benches and Stimulus Files 12

Adding or Creating Simulation Source Files 13

Using Xilinx Simulation Libraries. 15

Using Simulation Settings 25

Understanding the Simulator Language Option 27

Recommended Simulation Resolution 29

Generating a Netlist 29

Chapter 3: Understanding Vivado Simulator

Introduction 31

Vivado Simulator Features 31

Running the Vivado Simulator 32

Running Functional and Timing Simulation 45

Saving Simulation Results 49

Distinguishing Between Multiple Simulation Runs 49

Closing a Simulation. 49

Adding a Simulation Start-up Script File. 50

Viewing Simulation Messages. 51

Using the launch_simulation Command 52

Re-running the Simulation After Design Changes (relaunch) 53

Chapter 4: Analyzing Simulation Waveforms

Introduction 55

Using Wave Configurations and Windows 55

Opening a Previously Saved Simulation Run 56
Understanding HDL Objects in Waveform Configurations 58
Customizing the Waveform. 61
Controlling the Waveform Display 67
Organizing Waveforms 69
Analyzing Waveforms 71

Chapter 5: Debugging a Design with Vivado Simulator

Introduction 75
Debugging at the Source Level 75
Forcing Objects to Specific Values 79
Power Analysis Using Vivado Simulator. 88
Using the report_drivers Tcl Command 89
Using the Value Change Dump Feature 90
Using the log_wave Tcl Command 91
Cross Probing Signals in the Object, Wave, and Text Editor Windows 92

Chapter 6: Handling Special Cases

Using Global Reset and 3-State. 95
Delta Cycles and Race Conditions. 97
Using the ASYNC_REG Constraint. 98
Simulating Configuration Interfaces. 99
Disabling Block RAM Collision Checks for Simulation. 103
Dumping the Switching Activity Interchange Format File for Power Analysis 104
Simulating a Design with AXI Bus Functional Models. 104
Skipping Compilation or Simulation. 104

Chapter 7: Simulating in Batch or Scripted Mode

Introduction 106
Exporting Simulation Files and Scripts 106
Running Third-Party Simulators in Batch Mode 112
Running the Vivado Simulator in Batch Mode. 112
Elaborating and Generating a Design Snapshot, xelab 115
Simulating the Design Snapshot, xsim 125
Example of Running Vivado Simulator in Standalone Mode 127
Project File (.prj) Syntax 128
Predefined Macros. 128
Library Mapping File (xsim.ini) 129
Running Simulation Modes. 130
Using Tcl Commands and Scripts 132

export_simulation 133
export_ip_user_files 135

Chapter 8: Using Third-Party Simulators

Introduction 137
Preparing for Simulation Using Third-Party Tools 137
Tcl Mode 142
Running Simulation with Third-Party Tools 144
After Running Simulation with Third-Party Tools 152
Simulating IP. 155
Using the Verilog UNIFAST Library 155
Simulating a Design with AXI Bus Functional Models 155
Using a Custom DO File During an Integrated Simulation Run 156
Generating a Simulator Specific Run Directory 156

Appendix A: Value Rules in Vivado Simulator Tcl Commands

Introduction 158
String Value Interpretation 158
Vivado Design Suite Simulation Logic. 159

Appendix B: Vivado Simulator Mixed Language Support and Language Exceptions

Introduction 160
Using Mixed Language Simulation 160
VHDL Language Support Exceptions 167
Verilog Language Support Exceptions 169

Appendix C: Vivado Simulator Quick Reference Guide

Introduction 172

Appendix D: System Verilog Constructs Supported by the Vivado Simulator

Introduction 176
Dynamic Types 185

Appendix E: VHDL 2008 Support for Vivado Simulator

Introduction 188
Compiling and Simulating 188
Supported Features 189

Appendix F: Direct Programming Interface (DPI) in Vivado Simulator

Introduction 191
Compiling C Code 191
xsc Compiler 192
Binding Compiled C Code to SystemVerilog Using xelab. 193
Data Types Allowed on the Boundary of C and SystemVerilog 194
Mapping for User-Defined Types 195
Support for svdpi.h functions 197
DPI Examples Shipped with the Vivado Design Suite 203

Appendix G: Using Xilinx Simulator Interface

Introduction 204
Preparing the XSI Functions for Dynamic Linking 204
Writing the Test Bench Code. 206
Compiling Your C/C++ Program. 207
Preparing the Design Shared Library 207
XSI Function Reference 208
Vivado Simulator VHDL Data Format 213
Vivado Simulator Verilog Data Format. 216

Appendix H: Additional Resources and Legal Notices

Xilinx Resources 219
Solution Centers. 219
Documentation References. 219
Links to Additional Information on Third-Party Simulators 220
Links to Language and Encryption Support Standards 220
Training Resources. 221
Please Read: Important Legal Notices 221

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 4]
- *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 12]

Simulation Flow

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

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

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

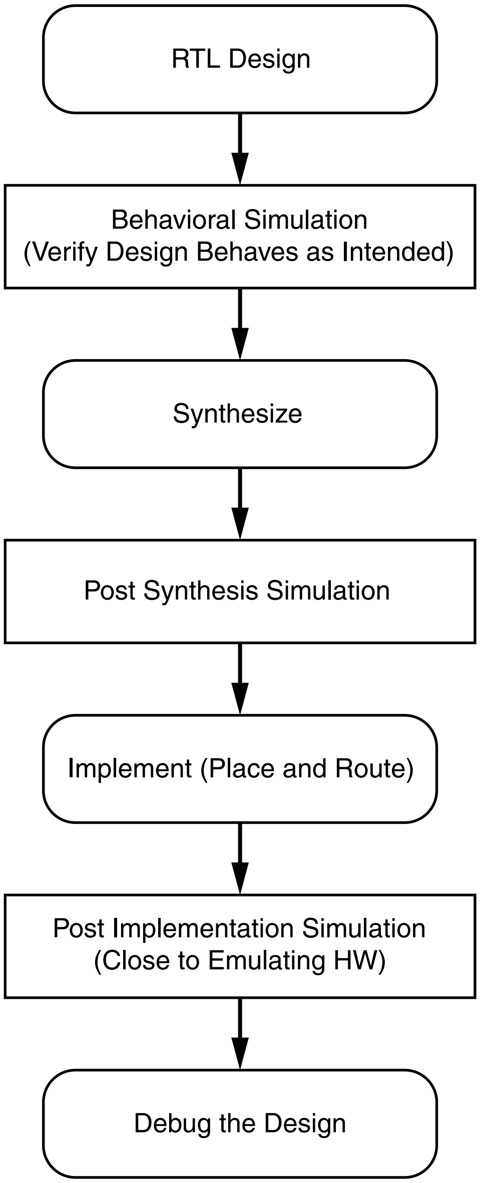

RTL Design
Behavioral Simulation
(Verify Design Behaves as Intended)
Synthesize
Post Synthesis Simulation
Implement (Place and Route)
Post Implementation Simulation
(Close to Emulating HW)
Debug the Design

Figure 1-1: **Simulation Flow**

Behavioral Simulation at the Register Transfer Level

Register Transfer Level (RTL) behavioral simulation can include:

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

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

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

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

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

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

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

Post-Synthesis Simulation

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

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

These primitives are contained in the `UNISIMS_VER` library for Verilog, and the `UNISIM` library for VHDL. See UNISIM Library, page 18 for more information.

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, Understanding Vivado Simulator.
- Xilinx supports the following third-party simulators:
    - Mentor Graphics Questa Advanced Simulator/ModelSim: Integrated in the Vivado IDE.
    - Cadence Incisive Enterprise Simulator (IES): Integrated in the Vivado IDE.
    - Synopsys VCS and VCS MX: Integrated in the Vivado IDE.
    - Aldec Active-HDL and Rivera-PRO
      Aldec offers support for these simulators.

  Note: For more information, see Chapter 8, Using Third-Party Simulators.

See the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) [Ref 1] for the supported versions of third-party simulators.

Language and Encryption Support

The Vivado simulator supports:

- VHDL, see *IEEE Standard VHDL Language Reference Manual* (IEEE-STD-1076-1993) [Ref 15]
- Verilog, see *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1364-2001)[Ref 16]
- System Verilog Synthesizable subset. See *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1800-2009) [Ref 17]
- IEEE P1735 encryption, see *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)* (IEEE-STD-P1735) [Ref 19]

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

Preparing for Simulation

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 95.

For more information about test benches, see *Writing Efficient TestBenches (XAPP199)* [Ref 6].

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.*

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.

VIDEO: *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 this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [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 2-1.

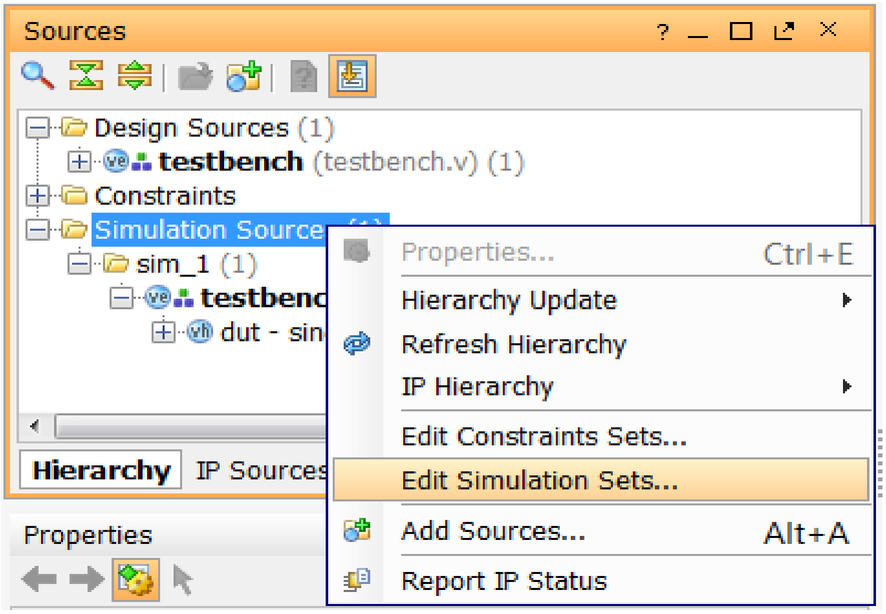


Figure 2-1: **Edit Simulation Sets Option**

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.

Using Xilinx Simulation Libraries

IMPORTANT: *With Vivado simulator, there is no need to compile the simulation libraries. However, you must compile the libraries when using a third-party simulator. Please refer to Chapter 8, Using Third-Party Simulators for more information.*

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

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

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

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

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

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

Table 2-1: **Simulation Libraries**

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

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**

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

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

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

Table 2-3 lists the library locations.

Table 2-3: **Simulation Library Locations**

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

The following subsections describe the libraries in more detail.

UNISIM Library

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

IMPORTANT: *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.

Table 2-4: **Component Files**

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

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

VHDL UNISIM Library

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

- The component declarations (`unisim_VCOMP.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.

Verilog UNISIM Library

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

The Verilog UNISIM library cannot be specified in the HDL file prior to using the module. To use the library module, specify the module name using all uppercase letters. The following example shows the instantiated module name as well as the file name associated with that module:

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

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

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

```
-L unisims_ver
Where:
-L is the library specification command.
```

UNIMACRO Library

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

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

VHDL UNIMACRO Library

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

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

Verilog UNIMACRO Library

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

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

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

SIMPRIM Library

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

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

TIP: *If you are a VHDL user, you can run post synthesis and post implementation functional simulation (in which case no standard default format (SDF) annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the* write_vhdl *Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide (UG835)* [Ref 8].

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 19]. 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 2-5: **Special Considerations for Using SECUREIP Libraries**

Simulator Name	Vendor	Requirements
ModelSim SE	Mentor Graphics	If design entry is in VHDL, a mixed language license or a SECUREIP OP is required. Contact the vendor for more information.
ModelSim PE		
ModelSim DE		
Questa Advanced Simulator		
VCS and VCS MX	Synopsys	
Active-HDL	Aldec	If design entry is VHDL only, a SECUREIP language-neutral license is required. Contact the vendor for more information.
Riviera-PRO*		

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
```

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

UNIFAST Library

The `UNIFAST` library is an optional library that you can use during RTL behavioral simulation to speed up simulation 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 run a complete verification using the UNISIM library.*

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

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

MMCME2

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

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

   Note: The output clock frequency is not generated from input-to-VCO clock.
4. The standard and the fast `MMCME2` simulation model `LOCKED` signal assertion times differ.
   - Standard Model `LOCKED` assertion time depends on the `M` and `D` setting. For large `M` and `D` values, the lock time is relatively long for a standard `MMCME2` simulation model.
   - In the fast simulation model, the `LOCKED` assertion time is shortened.

DSP48E1

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

- Pattern Detection
- OverFlow/UnderFlow
- DRP interface support

GTHE2_CHANNEL/GTHE2_COMMON

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

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

GTXE2_CHANNEL/GTXE2_COMMON

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

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

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 (fast simulation models) in a Vivado project environment for the Vivado simulator, ModelSim, IES, or VCS.

Use the following TCL command in TCL console:

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

See the Encrypted Component Files, page 18 for more information regarding component files.

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;`)
- The library search order for cells or instances that are not explicitly called out: (for example: `default liblist unisims_ver unifast_ver;`)
- The map for a particular `CELL` or `INSTANCE` to a particular library. (For example: `instance testbench.inst.O1 use unifast_ver.MMCME2;`)

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

Example config.v

```
config cfg_xilinx;
design testbench;
default liblist unisims_ver unifast_ver;
//Use fast MMCM for all MMCM blocks in design
cell MMCME2 use unifast_ver.MMCME2;
//use fast dSO48E1for only this specific instance in the design
instance testbench.inst.O1 use unifast_ver.DSP48E1;
//If using ModelSim or Questa, add in the genblk to the name
(instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.DSP48E1)
endconfig
```

Using VHDL UNIFAST Library

The VHDL `UNIFAST` library has the same basic structure as Verilog and can be used with architectures or libraries. You can include the library in the test bench file. The following example uses a *drill-down* hierarchy with a `for` call:

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

```
.........end for;
.......for O1 inst:DSP48E1;
.........use entity unifast.DSP48E1;
.......end for;
...end for;
..end for;
end for;
end cfg_xilinx;
```

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

Using Simulation Settings

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

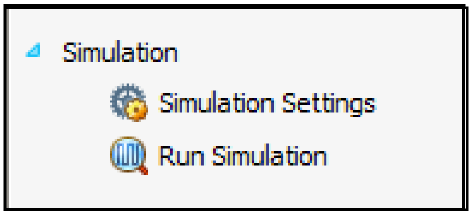


Figure 2-2: **Flow Navigator Simulation Options**

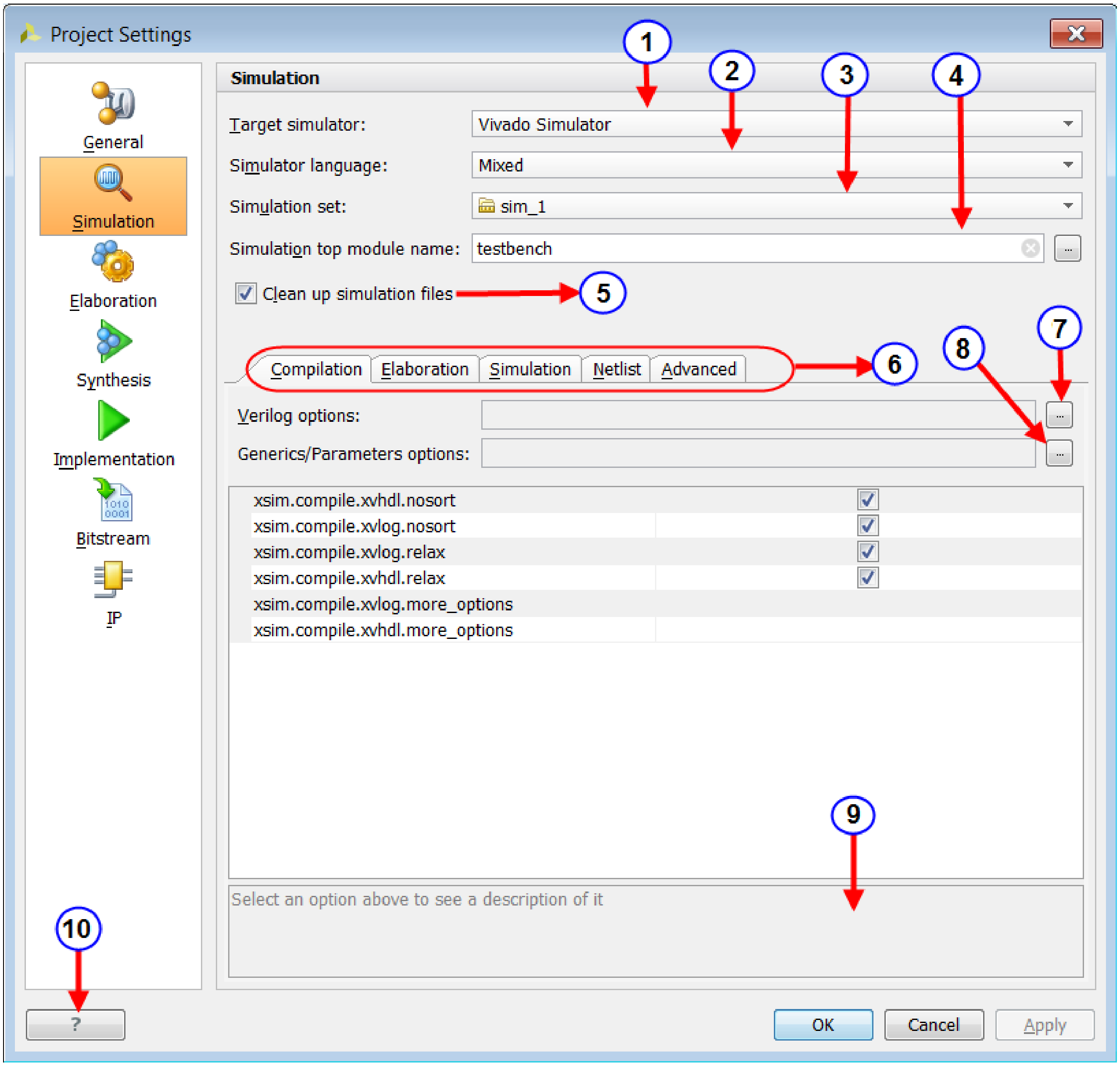
- Simulation Settings: Opens the Project Settings dialog box where you can select and configure the Vivado simulator. See Vivado Simulator Project Settings, page 26.
- Run Simulation: Sets up the command options to compile, elaborate, and simulate the design based on the simulation settings, then launches the Vivado simulator. When you run simulation prior to synthesizing the design, the Vivado simulator runs a behavioral simulation, and opens a Wave window, (see Figure 3-11, page 43) 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`.

Vivado Simulator Project Settings

In the Flow Navigator, click **Simulation Settings** to open the Project Settings dialog box, shown in Figure 2-3.



Project Settings

Simulation

Target simulator: Vivado Simulator

Simulator language: Mixed

Simulation set: sim_1

Simulation top module name: testbench

☑ Clean up simulation files

Verilog options:

Generics/Parameters options:

xsim.compile.xvhdl.nosort	☑
xsim.compile.xvlog.nosort	☑
xsim.compile.xvlog.relax	☑
xsim.compile.xvhdl.relax	☑
xsim.compile.xvlog.more_options	
xsim.compile.xvhdl.more_options	

Select an option above to see a description of it

1 Selects the target simulator.
2 Selects the simulator language.
3 Selects the simulation set.
4 Browses to the simulation top-level design name.
5 Cleans simulation files before re-run. Keeping the option enabled is recommended.
6 Select tabs to set options in the respective categories.
7 Compilation tab only. Browse to set include path or to define macros.
8 Compilation tab only. Browse to select generics/parameters location.
9 For each tab, an option list appears in the window, and when selected, an option description displays.
10 Click to see the information regarding each field in the Project Settings dialog box.

Figure 2-3: **Project Settings Dialog Box, Vivado Simulator Options**

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

TIP: *Because the Vivado simulator has precompiled libraries, it is not necessary to identify the library location.*

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

Understanding the Simulator Language Option

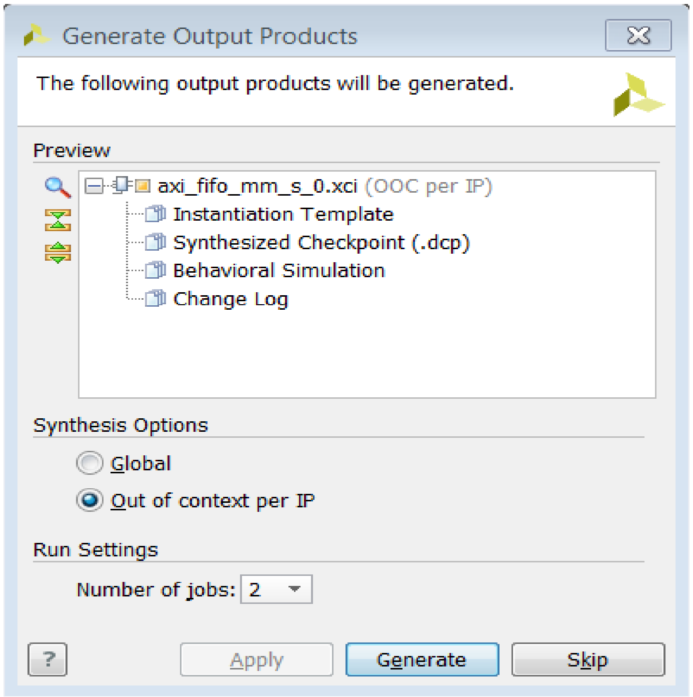
Most Xilinx IP deliver behavioral simulation models for a single language only, effectively disabling simulation for language-locked simulators if you are not licensed for the appropriate language. The `simulator_language` property ensures that an IP delivers a simulation model for any given language. (Figure 2-3, above, 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 2-4) opens.



Generate Output Products

The following output products will be generated.

Preview

- axi_fifo_mm_s_0.xci (OOC per IP)
  - Instantiation Template
  - Synthesized Checkpoint (.dcp)
  - Behavioral Simulation
  - Change Log

Synthesis Options

○ Global

● Out of context per IP

Run Settings

Number of jobs: 2

Figure 2-4: **Dialog Box Showing Generate Synthesized Checkpoint (.dcp) Option**

Table 2-6 illustrates the function of the `simulator_language` property.

Table 2-6: **Function of simulator_language Property**

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

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]`

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.

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

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

- **Tcl Commands**:
  - `write_verilog`: Verilog netlist
  - `write_vhdl`: VHDL netlist
  - `write_sdf`: SDF generation

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

Generating a Functional Netlist

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

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

These primitives are contained in the following libraries:

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

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

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

```
write_verilog -mode funcsim <Verilog_Netlist_Name.v>
```

```
write_vhdl -mode funcsim <VHDL_Netlist_Name.vhd>
```

Generating a Timing Netlist

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

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

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

There are two steps to generating a timing simulation netlist:

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

IMPORTANT: *Vivado IDE supports Verilog timing simulation only.*

Refer to Tip in page 20 for VHDL Timing Simulation support and workaround.

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

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

Chapter 3

Understanding Vivado Simulator

Introduction

This chapter describes the Vivado® simulator features available in the Vivado Integrated Design Environment (IDE), which include pushbutton waveform tracing and debug capability.

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

See the *Vivado Design Suite Tutorial: Logic Simulation* (UG937) [Ref 11] for a step-by-step demonstration of how to run Vivado simulation.

Vivado Simulator Features

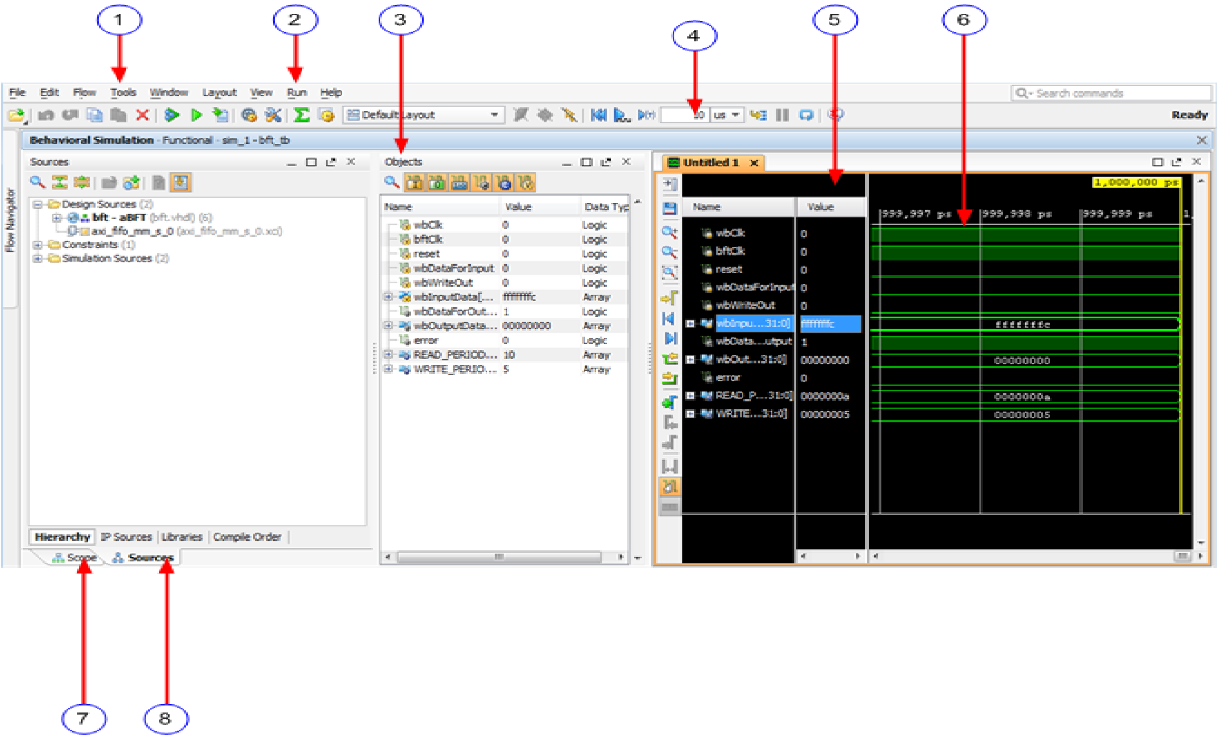
The Vivado simulator supports the following features:

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

Running the Vivado Simulator

IMPORTANT: *Before running simulation, be sure you have specified all appropriate project settings for your design. If you are using the Vivado simulator, these are described in Vivado Simulator Project Settings in Chapter 2. For supported third-party simulators, see Chapter 8, Using Third-Party Simulators.*

From the Flow Navigator, select **Run Simulation** to invoke the Vivado simulator workspace, shown in the figure below.



Name	Value	Data Typ
wbClk	0	Logic
bftClk	0	Logic
reset	0	Logic
wbDataForInput	0	Logic
wbWriteOut	0	Logic
wbInputData[...	fffffffc	Array
wbDataForOut...	1	Logic
wbOutputData...	00000000	Array
error	0	Logic
READ_PERIOD...	10	Array
WRITE_PERIO...	5	Array

Name	Value
wbClk	0
bftClk	0
reset	0
wbDataForInput	0
wbWriteOut	0
wbInpu...31:0]	fffffffc
wbData...utput	1
wbOut...31:0]	00000000
error	0
READ_P...31:0]	0000000a
WRITE...31:0]	00000005

1 Main Toolbar
2 Run Menu
3 Objects Window
4 Simulation Toolbar

5 Wave Objects
6 Wave window
7 Scopes Window
8 Sources Window

Figure 3-1: **Vivado Simulator Workspace**

Main Toolbar

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

Run Menu

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

The Run menu for simulation is shown in Figure 3-2.

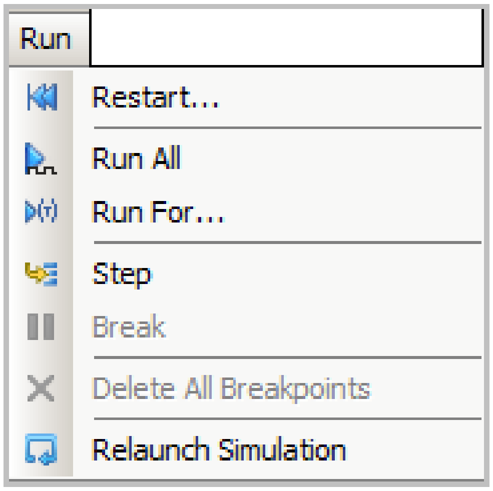


Figure 3-2: **Simulation Run Menu Options**

The Vivado simulator Run menu options:

- Restart: Lets you restart an existing simulation from time 0.
  Tcl Command: `restart`
- Run All: Lets you run an open simulation to completion.
  Tcl Command: `run all`
- Run For: Lets you specify a time for the simulation to run.
  Tcl Command: `run <time>`
- Step: Runs the simulation up to the next HDL source line.
- Break: Lets you pause a running simulation.
- Delete All Breakpoints: Deletes all breakpoints.
- Relaunch Simulation: Recompiles the simulation files and restarts the simulation. See Re-running the Simulation After Design Changes (relaunch) for more information.

Simulation Toolbar

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

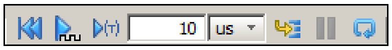


Figure 3-3: **Simulation Toolbar**

These are the same buttons labeled in Figure 3-2, page 33, above (without the Delete All Breakpoints option), and they are provided for ease of use.

Simulation Toolbar Button Descriptions

Hover over the toolbar buttons for tool-tip descriptions.

- **Restart:** resets the simulation time to zero.
- **Run all:** runs the simulation until it completes all events or until an HDL statement indicates that the simulation should stop.
- **Run For:** runs for a specified period of time.
- **Step:** runs the simulation until the next HDL statement.
- **Break:** pauses the current simulation.
- **Relaunch:** recompiles the simulation sources and restarts the simulation (after making code changes, for example). See Re-running the Simulation After Design Changes (relaunch) for more information.

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 3-4.

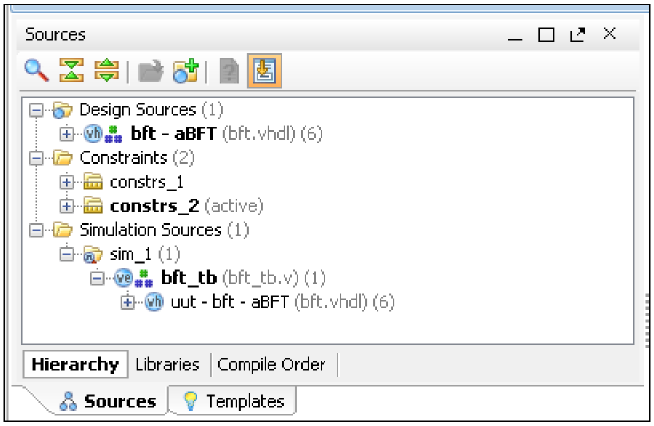


Figure 3-4: **Sources Window**

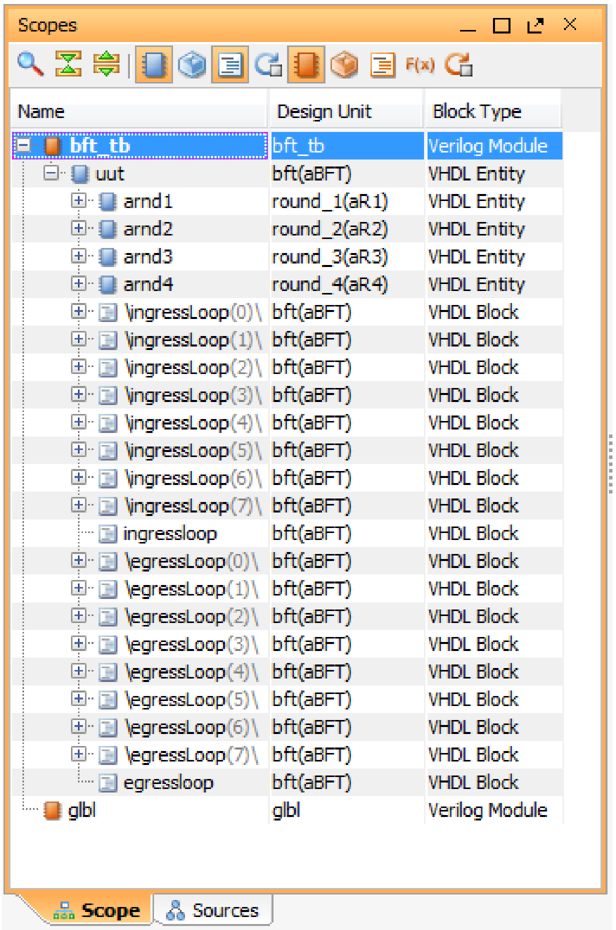
The Sources buttons are described by tool tips when you hover the mouse over them. The buttons let you examine, expand, collapse, add to, open, filter and scroll through files.

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

Scopes Window

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

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



Name	Design Unit	Block Type
bft_tb	bft_tb	Verilog Module
uut	bft(aBFT)	VHDL Entity
arnd1	round_1(aR1)	VHDL Entity
arnd2	round_2(aR2)	VHDL Entity
arnd3	round_3(aR3)	VHDL Entity
arnd4	round_4(aR4)	VHDL Entity
\ingressLoop(0)\	bft(aBFT)	VHDL Block
\ingressLoop(1)\	bft(aBFT)	VHDL Block
\ingressLoop(2)\	bft(aBFT)	VHDL Block
\ingressLoop(3)\	bft(aBFT)	VHDL Block
\ingressLoop(4)\	bft(aBFT)	VHDL Block
\ingressLoop(5)\	bft(aBFT)	VHDL Block
\ingressLoop(6)\	bft(aBFT)	VHDL Block
\ingressLoop(7)\	bft(aBFT)	VHDL Block
ingressloop	bft(aBFT)	VHDL Block
\egressLoop(0)\	bft(aBFT)	VHDL Block
\egressLoop(1)\	bft(aBFT)	VHDL Block
\egressLoop(2)\	bft(aBFT)	VHDL Block
\egressLoop(3)\	bft(aBFT)	VHDL Block
\egressLoop(4)\	bft(aBFT)	VHDL Block
\egressLoop(5)\	bft(aBFT)	VHDL Block
\egressLoop(6)\	bft(aBFT)	VHDL Block
\egressLoop(7)\	bft(aBFT)	VHDL Block
egressloop	bft(aBFT)	VHDL Block
glbl	glbl	Verilog Module

Figure 3-5: **Scopes Window**

Filtering Scopes

- Click a filter button to toggle between showing or hiding the corresponding scope type.

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

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

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

When you right-click a scope, a popup menu (shown in Figure 3-6) provides the following options:

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

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

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

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

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

TIP: *To display object values prior to the time of insertion, the simulation must be restarted. To avoid having to restart the simulation because of missing value changes: issue the log_wave -r / Tcl command at the start of a simulation run to capture value changes for all display-able HDL objects in your design. For more information, see Using the log_wave Tcl Command, page 91.*

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

- **Go To Source Code**: Opens the source code at the definition of the selected scope.
- **Go To Instantiation Source Code**: For Verilog modules and VHDL entity instances, opens the source code at the point of instantiation for the selected instance.

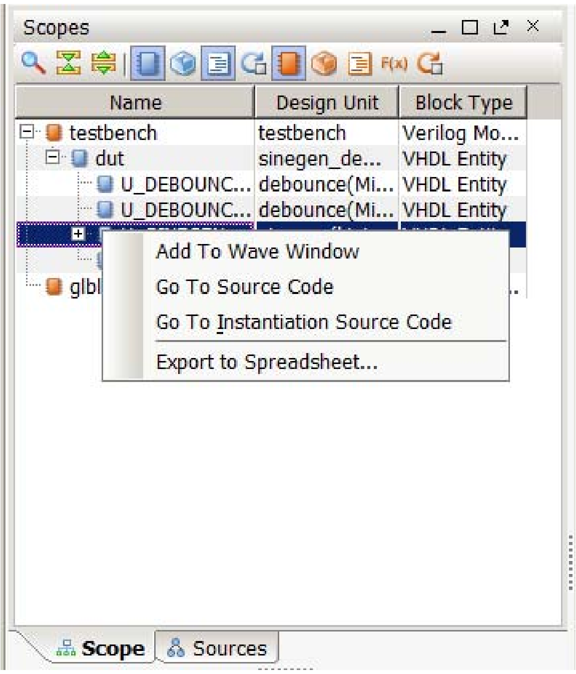
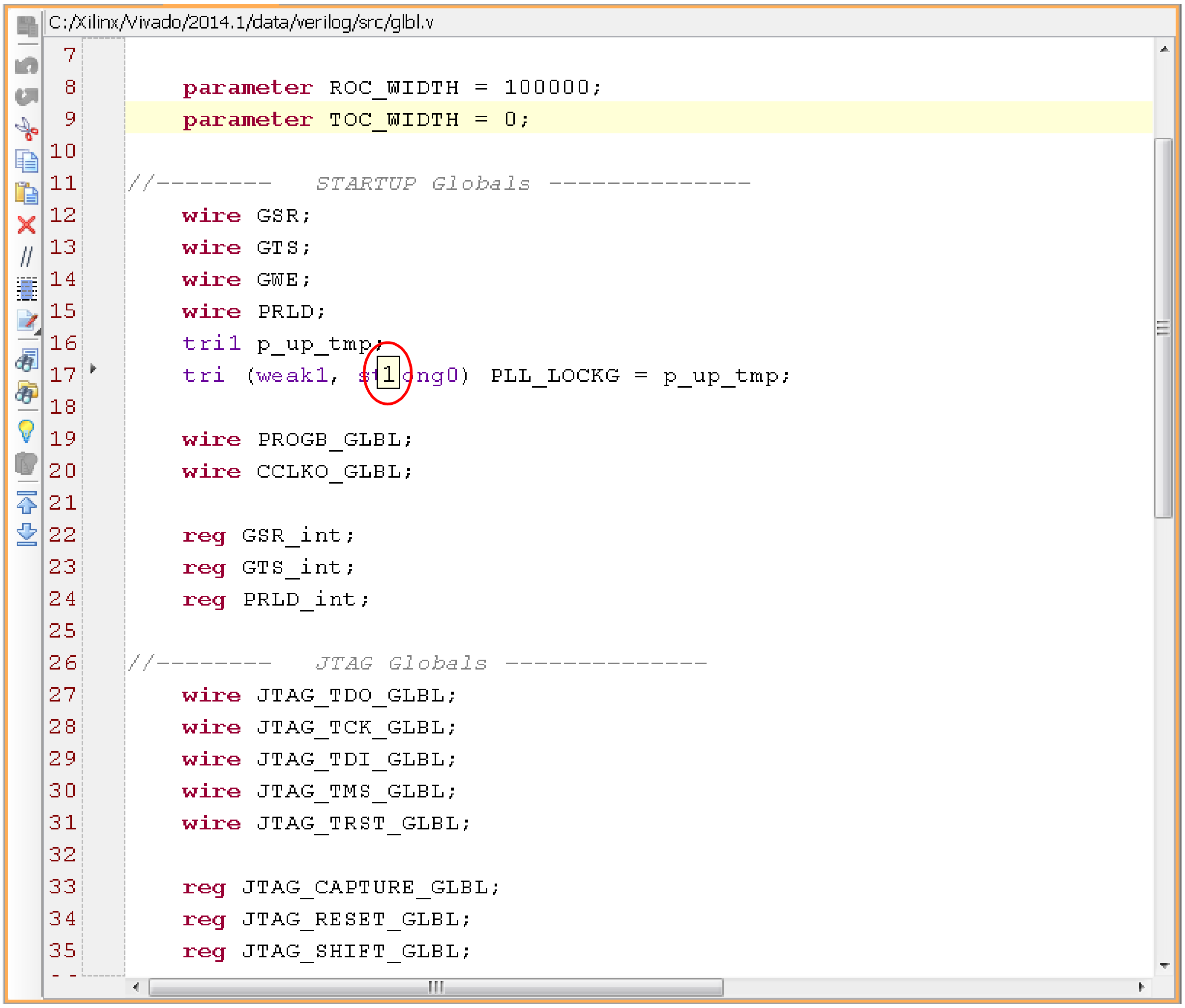


Figure 3-6: **Scopes Window Showing Right-Click Options**

In the source code text editor, you can hover over an identifier in the code get the value, as shown in Figure 3-7.

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

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


C:/Xilinx/Vivado/2014.1/data/verilog/src/glbl.v

```

    parameter ROC_WIDTH = 100000;
    parameter TOC_WIDTH = 0;

//--------   STARTUP Globals --------------
    wire GSR;
    wire GTS;
    wire GWE;
    wire PRLD;
    tri1 p_up_tmp;
    tri (weak1, strong0) PLL_LOCKG = p_up_tmp;

    wire PROGB_GLBL;
    wire CCLKO_GLBL;

    reg GSR_int;
    reg GTS_int;
    reg PRLD_int;

//--------   JTAG Globals --------------
    wire JTAG_TDO_GLBL;
    wire JTAG_TCK_GLBL;
    wire JTAG_TDI_GLBL;
    wire JTAG_TMS_GLBL;
    wire JTAG_TRST_GLBL;

    reg JTAG_CAPTURE_GLBL;
    reg JTAG_RESET_GLBL;
    reg JTAG_SHIFT_GLBL;
```

Figure 3-7: **Source Code with Identifier Value Displayed**

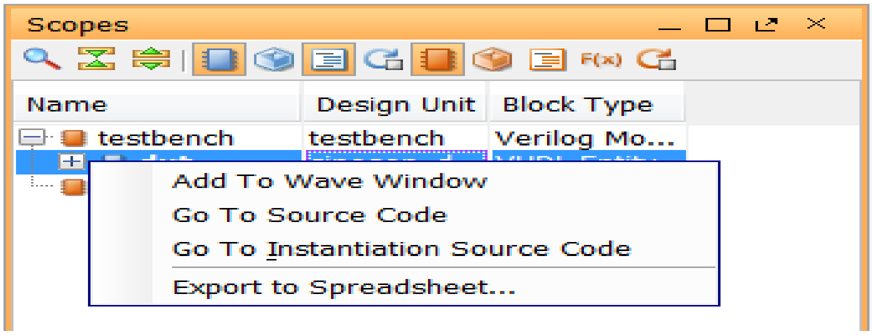
Additional Scopes and Sources Options

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

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 3-8.*

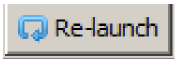


Scopes

Name	Design Unit	Block Type
testbench	testbench	Verilog Mo...

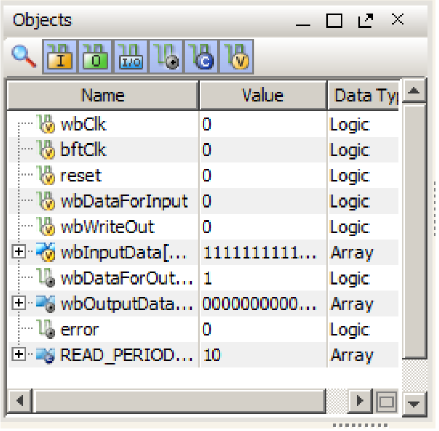
Figure 3-8: **Context Menu in Scopes Window**

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



Objects Window

The HDL Objects window displays the HDL simulation objects associated with the scope selected in the Scopes window, as shown in Figure 3-9.



Objects

Name	Value	Data Ty
wbClk	0	Logic
bftClk	0	Logic
reset	0	Logic
wbDataForInput	0	Logic
wbWriteOut	0	Logic
wbInputData[...	1111111111...	Array
wbDataForOut...	1	Logic
wbOutputData...	0000000000...	Array
error	0	Logic
READ_PERIOD...	10	Array

Figure 3-9: **HDL Objects Window**

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

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

```
get_value <hdl_object>
```

Table 3-1 briefly describes the buttons at the top of the Objects window. The HDL objects buttons display the selected objects in the Object window. Use this to filter or limit the contents of the Objects window.

Table 3-1: **HDL Object Buttons**

Button	Description
🔍	The **Search** button, when selected, opens a field in which you can enter an object name on which to search.
I	Input signals
0	Output signals
I/O	Input/Output signals
●	Internal signals
C	Constant signals
V	Variable signals

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

You can hide certain types of HDL object from display by clicking I 0 I/O ● C V one or more object-filtering buttons. Hover over the button for a tool tip describing what object type it represents.

Objects Context Menu

When you right-click an object in the Objects window, a context menu (shown in Figure 3-10) appears. The options in the context menu are described below.

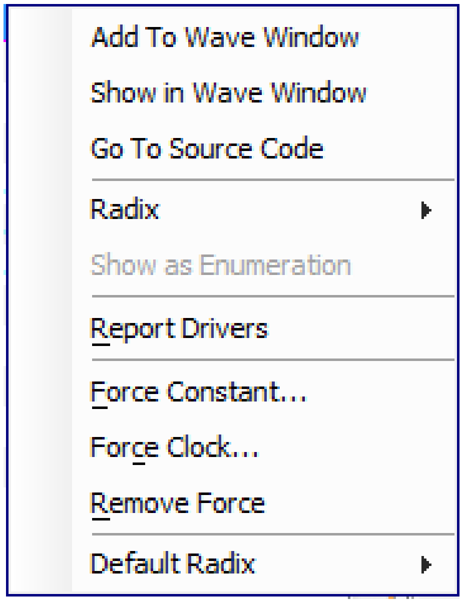


Figure 3-10: **Context Menu in Objects Window**

- **Add to Wave Window:** Add the selected object to the waveform configuration. Alternately, you can drag and drop the objects from the Objects window to the Name column of the Wave window.
- **Show in Wave Window:** Highlights the selected object in the Wave window.
- **Default Radix:** Set the default radix for all objects in the objects window and text editor. The default radix is Hexadecimal. You can change this option from the context menu.

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

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

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

  a. Right-click an item in the Objects window.

  b. From the context menu, select **Radix** and the format you want to use:

     - Default
     - Binary
     - Hexadecimal
     - Octal
     - ASCII
     - Unsigned Decimal
     - Signed Decimal

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

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

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

- **Report Drivers:** Display in the Tcl Console a report of the HDL processes that assign values to the selected object.
- **Go To Source Code:** Open the source code at the definition of the selected object.
- **Force Constant:** Forces the selected object to a constant value. For more information on forcing objects, see the section Force Constant in Chapter 5.
- **Force Clock:** Forces the selected object to an oscillating value. For more information, see the section Force Clock in Chapter 5.

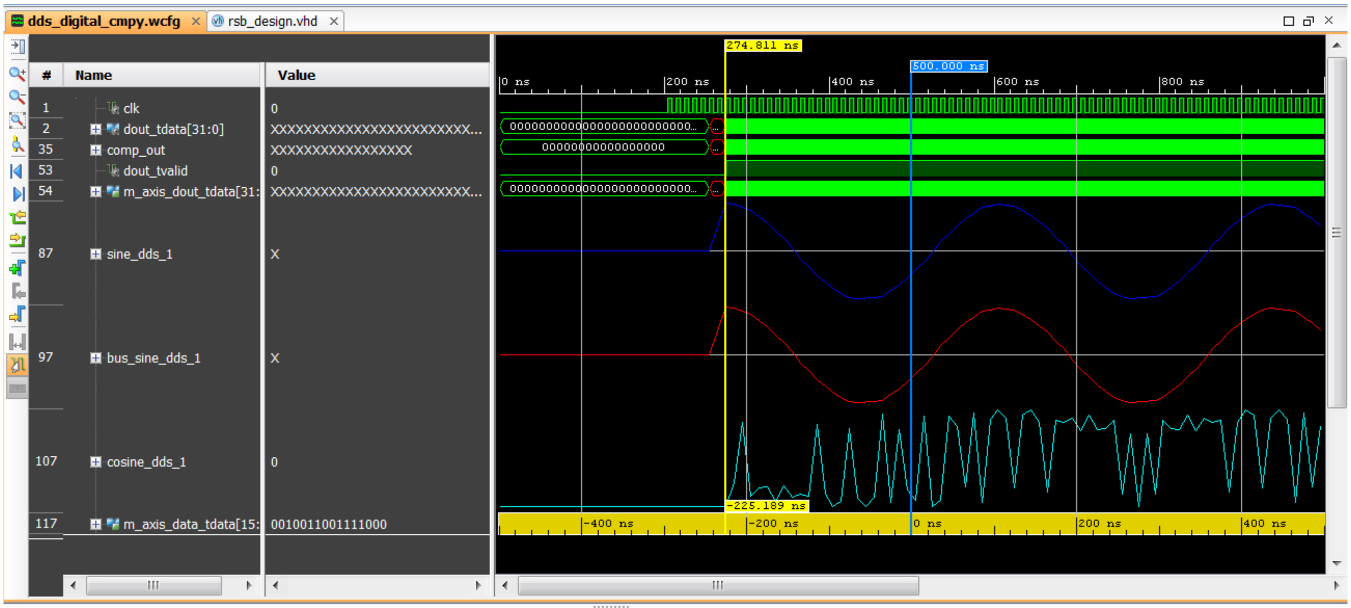
- **Remove Force:** Removes any force on the selected object. For more information, see the section Remove Force in Chapter 5.

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

Wave Window

When you invoke the simulator it opens a Wave window by default. The Wave window displays a new wave configuration consisting of the traceable HDL objects from the top module of the simulation, as shown in Figure 3-11.

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



#	Name	Value
1	clk	0
2	dout_tdata[31:0]	XXXXXXXXXXXXXXXXXXXXXXXX...
35	comp_out	XXXXXXXXXXXXXXXX
53	dout_tvalid	0
54	m_axis_dout_tdata[31:	XXXXXXXXXXXXXXXXXXXXXXXX...
87	sine_dds_1	X
97	bus_sine_dds_1	X
107	cosine_dds_1	0
117	m_axis_data_tdata[15:	0010011001111000

Figure 3-11: **Wave Window**

To add an individual HDL object or set of objects to the Wave window: in the Objects window, right-click an object or objects and select the **Add to Wave Window** option from the context menu (shown in Figure 3-9, page 40).

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`.

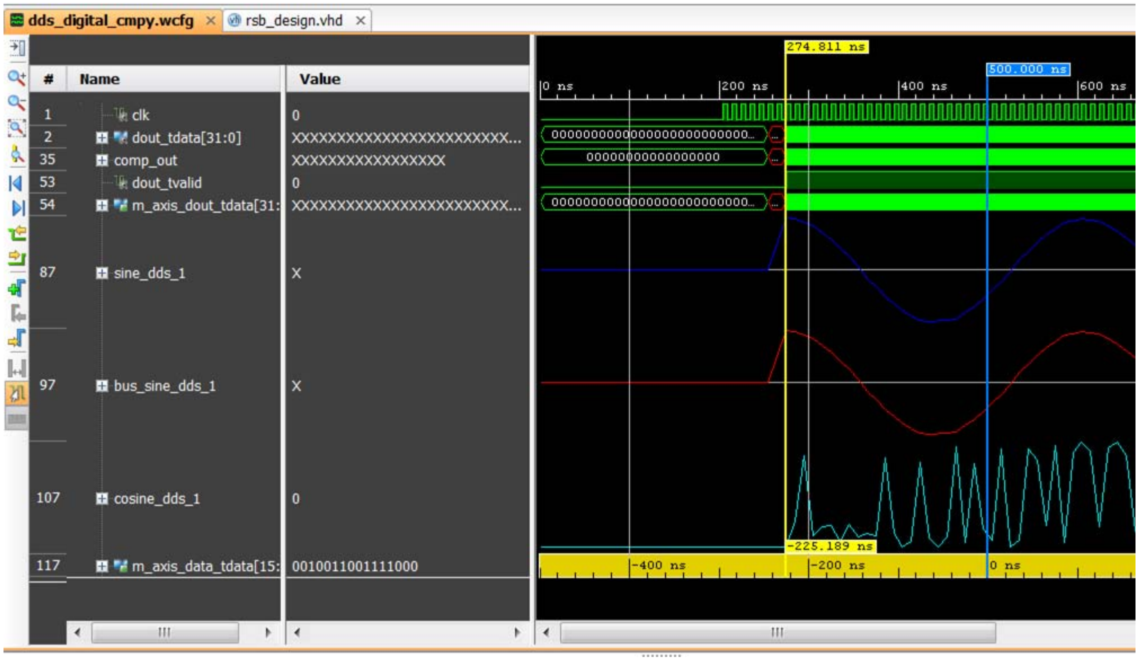
TIPS:

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.

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

Wave Objects

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



#	Name	Value
1	clk	0
2	dout_tdata[31:0]	XXXXXXXXXXXXXXXXXXXXXXXXX...
35	comp_out	XXXXXXXXXXXXXXXXX
53	dout_tvalid	0
54	m_axis_dout_tdata[31:	XXXXXXXXXXXXXXXXXXXXXXXXX...
87	sine_dds_1	X
97	bus_sine_dds_1	X
107	cosine_dds_1	0
117	m_axis_data_tdata[15:	0010011001111000

Figure 3-12: **HDL Objects in Waveform**

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

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

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

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

See Chapter 4, Analyzing Simulation Waveforms for more information about using the Wave window.

Saving a Waveform Configuration

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.

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

Creating and Using Multiple Waveform Configurations

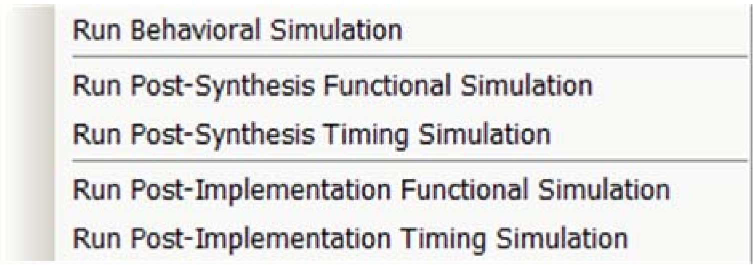
In a simulation session you can create and use multiple wave configurations, each in its own Wave window. When you have more than one Wave window displayed, the most recently-created or recently-used window is the *active window*. The active window, in addition to being the window currently visible,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

You can set a different Wave window to be the *active* window by clicking the title of the window. See Distinguishing Between Multiple Simulation Runs, page 49 and Creating a New Wave Configuration, page 55 for more information.

Running Functional and Timing Simulation

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

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



Run Behavioral Simulation

Run Post-Synthesis Functional Simulation

Run Post-Synthesis Timing Simulation

Run Post-Implementation Functional Simulation

Run Post-Implementation Timing Simulation

Figure 3-13: **Simulation Run Options**

Running Functional Simulation

Post-Synthesis Functional Simulation

When synthesis runs successfully, the **Run Simulation > Post-Synthesis Functional Simulation** option (shown in Figure 3-13) becomes available once you open a synthesized design.

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

Post-Implementation Functional Simulations

When implementation is successful, the **Run Simulation > Post-Implementation Functional Simulation** option (shown in Figure 3-13) becomes available once you open a synthesized design.

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

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

Running Timing Simulation

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

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

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

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

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

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

You control these options using Simulation Settings Simulation Settings as described in Using Simulation Settings, page 25.

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 8].*

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

Post-Synthesis Timing Simulation

When synthesis runs successfully, the **Run Simulation > Post-Synthesis Timing Simulation** option (shown in Figure 3-13) becomes available once you open a synthesized design.

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

Post-Implementation Timing Simulations

When post-implementation is successful, the **Run Simulation > Post-Implementation Timing Simulation** option (shown in Figure 3-13) becomes available once you open a synthesized design.

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

Annotating the SDF File for Timing Simulation

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

TIP: *To find the SDF file optional settings, in the Vivado IDE Flow Navigator, select* ***Simulation Settings****. In the* ***Project Settings*** *dialog box, select the* ***Netlist*** *tab. (See also, Vivado Simulator Project Settings in Chapter 2).*

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`

Saving Simulation Results

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

If you add objects to the Wave window and run the simulation, the design hierarchy for the complete design and the transitions for the added objects are automatically saved to the WDB file. You can also add objects to the waveform database that are not displayed in the Wave window using the `log_wave` command. For information abut the `log_wave` command, see Using the log_wave Tcl Command in Chapter 5.

Distinguishing Between Multiple Simulation Runs

When you have run several simulations against a design, the Vivado simulator displays named tabs at the top of the workspace with the simulation type that is currently in the window highlighted, as shown in Figure 3-14.

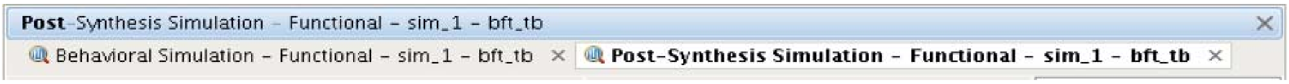

Post-Synthesis Simulation - Functional - sim_1 - bft_tb
Behavioral Simulation - Functional - sim_1 - bft_tb
Post-Synthesis Simulation - Functional - sim_1 - bft_tb

Figure 3-14: **Active Simulation Type**

Closing a Simulation

To close a simulation, in the Vivado IDE:

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

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

To close a simulation from the Tcl Console, type:

```
close_sim
```

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

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

Adding a Simulation Start-up Script File

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

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

   ```
   run 5us
   ```

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

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

2. Name the file `post.tcl` and save it.
3. Use the **Add Sources** 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 3-15.

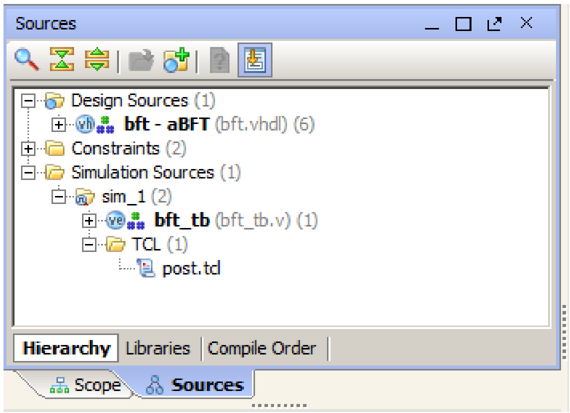


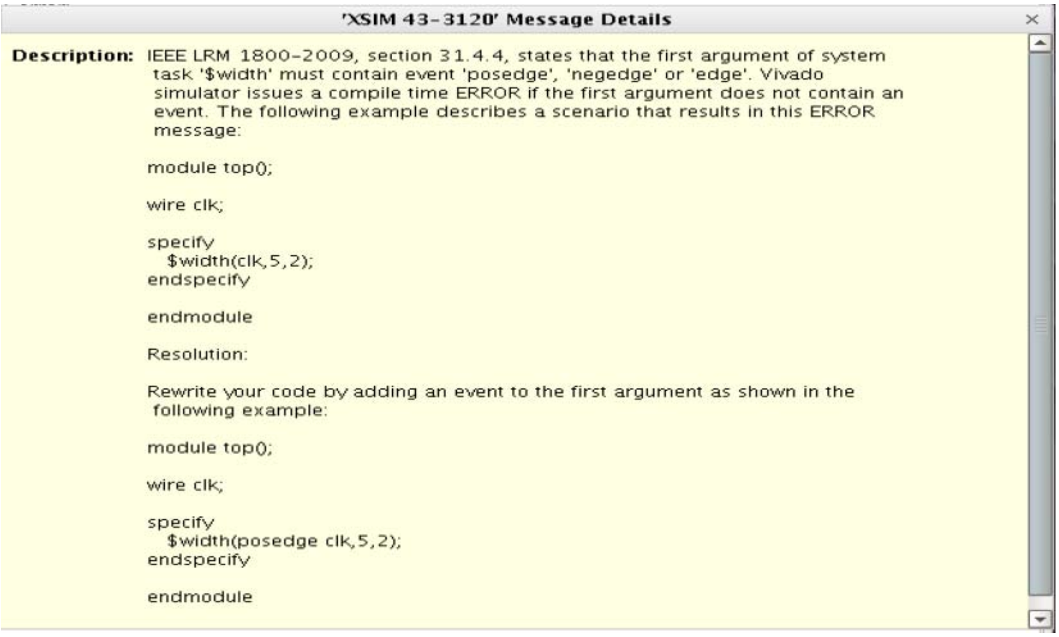
Figure 3-15: **Using the post.tcl File in a Design**

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

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

Viewing Simulation Messages

The Vivado IDE contains a message area where you can view informational, warning, and error messages. As shown in Figure 3-16, some messages from the Vivado simulator contain an issue description and a suggested resolution.



Description: IEEE LRM 1800-2009, section 31.4.4, states that the first argument of system task '$width' must contain event 'posedge', 'negedge' or 'edge'. Vivado simulator issues a compile time ERROR if the first argument does not contain an event. The following example describes a scenario that results in this ERROR message:

```
module top();

wire clk;

specify
   $width(clk,5,2);
endspecify

endmodule
```

Resolution:

Rewrite your code by adding an event to the first argument as shown in the following example:

```
module top();

wire clk;

specify
   $width(posedge clk,5,2);
endspecify

endmodule
```

Figure 3-16: **Simulator Message Description and Resolution Information**

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

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

An example of such a command is as follows:

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

Managing Message Output

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

1. In the Flow Navigator, open **Simulation Settings**.
2. In the Project Settings dialog box:
   a. Click the **Simulation** category.
   b. Select the **Simulation** tab.
   c. Next to *`xsim.simulate.xsim.more_options`* add `-maxlogsize <size>` where `<size>` is the maximum amount of text output in megabytes.

Using the launch_simulation Command

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

The syntax of `launch_simulation` is as follows:

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

Table 3-2 describes the options of `launch_simulation`.

Table 3-2: **launch_simulation Options**

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

Note: The `-scripts_only` switch has been deprecated and scheduled to be removed from future versions of Vivado. Xilinx recommends you to use `export_simulation` Tcl command.

Examples

- Running behavioral simulation using `vivado_simulator`

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

- Generating script for behavioral simulation with Questa Advanced Simulator.

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

Re-running the Simulation After Design Changes (relaunch)

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

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

1. Use the Vivado code editor or other text editor to update and save any necessary source code changes.
2. Use the Relaunch button on the Vivado IDE toolbar to re-compile and re-launch the simulation as shown in Figure 3-17. You may alternatively use the relaunch_sim Tcl command to re-compile and re-launch the simulation.

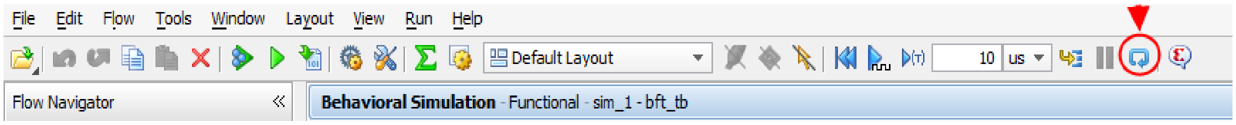


Figure 3-17: **relaunch sim option**

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

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

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

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

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

Chapter 4

Analyzing Simulation Waveforms

Introduction

In the Vivado® simulator, you can use the waveform to analyze your design and debug your code. The simulator populates design signal data in other areas of the workspace, such as the Objects and the 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 6].

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

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

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

Creating a New Wave Configuration

Create a new waveform configuration for displaying waveforms as follows:

1. Select **File > New Waveform Configuration**.

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

2. Add HDL objects to the waveform configuration using the steps listed in Understanding HDL Objects in Waveform Configurations, page 58.

See Chapter 3, Understanding Vivado Simulator for more information about creating new waveform configurations. Also see Creating and Using Multiple Waveform Configurations, page 45 for information on multiple waveforms.

Opening a WCFG File

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

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

   The Open Waveform Configuration dialog box opens.

2. Locate and select a WCFG file.

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

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

   Tcl command: `open_wave_config <waveform_name>`

Saving a Wave Configuration

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

Tcl command: `save_wave_config <waveform_name>`

Opening a Previously Saved Simulation Run

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

Standalone mode

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

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

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

Interactive Method

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

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

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

Programmatic Method

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

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

Then run it as:

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

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

When you run a simulation and display HDL objects in a Wave window, the running simulation produces a waveform database (WDB) file containing the waveform activity of the displayed HDL objects.

The WDB file also stores information about all the HDL scopes and objects in the simulated design. In this mode you cannot use commands that control or monitor a simulation, such as run commands, as there is no underlying "live" simulation model to control.

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

Understanding HDL Objects in Waveform Configurations

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

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

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

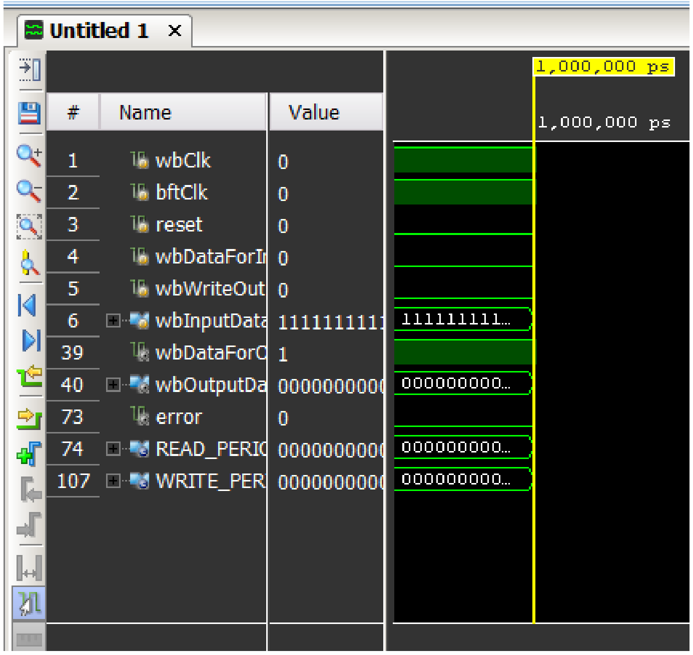
There are other kinds of wave objects available for display in a waveform configuration, such as: dividers, groups, and virtual buses.

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

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

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.
- Value: Displays the value of the object at the time indicated in the main cursor of the Wave window. You can change the formatting, or radix, of the value independent of the formatting of other design wave objects linked to the same HDL object and independent of the formatting of values displayed in the Objects window and source code window.



#	Name	Value	
1	wbClk	0	
2	bftClk	0	
3	reset	0	
4	wbDataForI	0	
5	wbWriteOut	0	
6	wbInputData	111111111	111111111...
39	wbDataForC	1	
40	wbOutputDa	000000000	000000000...
73	error	0	
74	READ_PERIC	000000000	000000000...
107	WRITE_PER	000000000	000000000...

Figure 4-1: **Waveform HDL Objects**

The Scopes window provides the ability to add all viewable HDL objects for a selected scope to the Wave window. For information on using the Scopes window, see Chapter 3, Scopes Window, page 35.

About Radixes

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

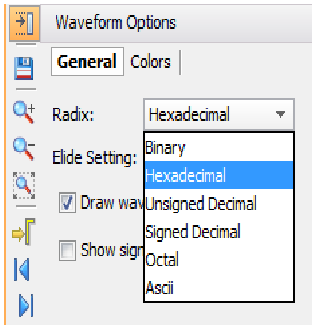
IMPORTANT: *Make a change to the radix setting in the window in which you wish to see the change. A change to the radix of an item in the Objects window does not apply to values in the Wave window or the Tcl Console. For example, the item wbOutputData[31:0] can be set to Signed Decimal in the objects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

Changing the Default Radix

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

To change the default waveform radix:

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



Radix: Hexadecimal

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

3. From the drop-down list, select a radix.

Changing the Radix on Individual Objects

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

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

Note: For a description of the usage for Real and Real Settings see Using Radixes and Analog Waveforms, page 61

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

```
set_property radix <radix> <wave_object>
```

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

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

Customizing the Waveform

Using Analog Waveforms

Using Radixes and Analog Waveforms

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

- Binary, octal, hexadecimal, ASCII, and unsigned decimal radixes 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.

Displaying Waveforms as Analog

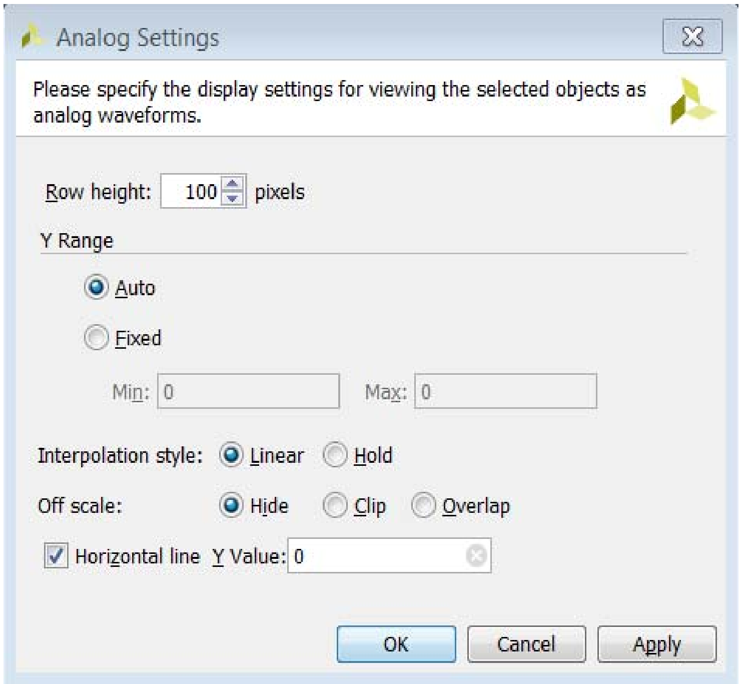
IMPORTANT: *When viewing an HDL bus object as an analog waveform—to produce the expected waveform, select a radix that matches the nature of the data in the HDL object.*
For example:
- If the data encoded on the bus is a 2's-compliment signed integer, you must choose a signed radix.
- If the data is 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. Right-click an HDL object in the Name column of the waveform configuration window and select **Waveform Style** from the drop-down menu. A popup menu appears, showing the following options:
   - Analog: Sets the waveform to Analog.
   - Digital: Sets the waveform object to Digital.
   - Analog Settings: Opens the Analog Settings dialog box (shown in Figure 4-2), which provides options for the analog waveform display.

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



Analog Settings

Please specify the display settings for viewing the selected objects as analog waveforms.

Row height: 100 pixels

Y Range

● Auto

○ Fixed

Min: 0 Max: 0

Interpolation style: ● Linear ○ Hold

Off scale: ● Hide ○ Clip ○ Overlap

☑ Horizontal line Y Value: 0

Figure 4-2: **Analog Settings Dialog Box**

Analog Settings Dialog Box Option Descriptions

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

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

TIP: *If the row indices separator lines are not visible, enable the checkbox in the Waveform Options dialog box to turn them on. Using the Waveform Options Dialog Box, page 66 for information on how to change the options settings. You can also change the row height by dragging the row index separator line to the left and below the waveform name.*

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

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

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

Waveform Object Naming Styles

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

Renaming Objects

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

1. Select the object name in the **Name** column.
2. Right-click and select **Rename** from the popup menu.

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

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

Changing the Object 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. Right-click and select **Name** from the drop-down menu. A popup menu appears, showing the following options:
   - **Long** to display the full hierarchical name of the design object.
   - **Short** to display the name of the signal or bus only.
   - **Custom** to display the custom name given to the object when renamed. See Renaming Objects, page 63.

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

Reversing the Bus Bit Order

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

To reverse the bit order:

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

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

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

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

Changing the Format of System Verilog Enumerations

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

To display enumerations using labels:

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

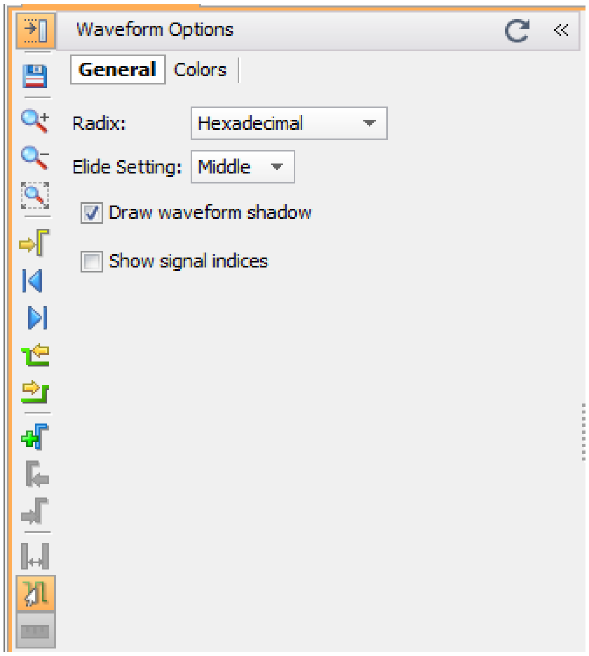
To display enumerations numerically:

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

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

Using the Waveform Options Dialog Box

Select the **Waveforms Options** button ⇥▯ to open the Waveform Options dialog box, shown in Figure 4-3.



Waveform Options

General | Colors

Radix: Hexadecimal

Elide Setting: Middle

☑ Draw waveform shadow

☐ Show signal indices

Figure 4-3: **Waveform Options Dialog Box**

The General Waveform Options are:

- Default Radix: Sets the numerical format to use for newly-created design wave objects.
- Elide Setting: Controls truncation of signal names that are too long for the Wave window.
    - **Left** truncates the left end of long names.
    - **Right** truncates the right end of long names.
    - **Middle** preserves both the left and right ends, omitting the middle part of long names.
- Draw Waveform Shadow: Creates a shaded representation of the waveform.
- Show signal indices: 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.
- From the Colors tab, you can set colors of items within the waveform.

Controlling the Waveform Display

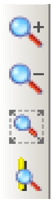
You can control the waveform display using:

- Zoom feature buttons in the Wave window sidebar
- Zoom combinations with the mouse wheel
- Vivado IDE Y-Axis zoom gestures
- Vivado simulation X-Axis zoom gestures. See the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 4] for more information about using the mouse to pan and zoom.

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

Using the Zoom Feature Button

There are zoom functions as sidebar buttons in the Wave window that let you 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 for Analog Waveforms

In addition to the zoom gestures supported for zooming in the X dimension, when over an analog waveform, additional zoom gestures are available, as shown in Figure 4-4.

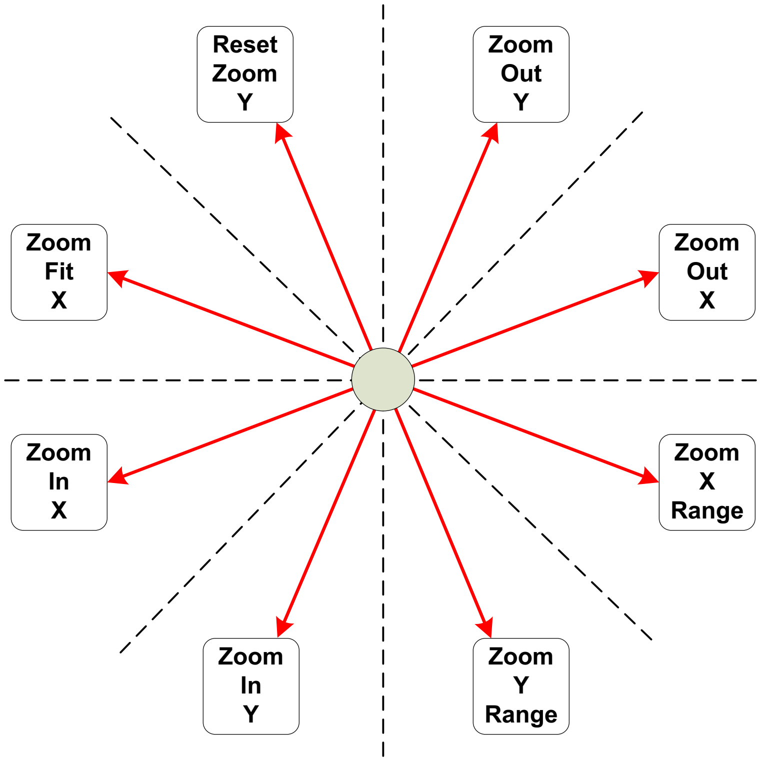


Figure 4-4: **Analog Zoom Options**

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

The additional zoom gestures are:

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

- Reset Zoom Y: Resets the Y range to that of the values currently displayed in the Wave window and sets the Y Range mode to Auto.

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

Organizing Waveforms

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

Grouping Signals and Objects

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

To add a Group:

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

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

2. 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 63.

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

Using Dividers

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

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

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

   Tcl command: `add_wave_divider`

You can move or delete Dividers as follows:

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

Dividers can be renamed also; see Renaming Objects, page 63.

Defining Virtual Buses

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

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

To add a virtual bus:

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

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

   Tcl Command: `add_wave_virtual_bus`

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

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

You can rename a virtual bus; see Renaming Objects, page 63.

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

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

Analyzing Waveforms

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

Using Cursors

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

TIP: *Waveform Configuration (WCFG) files do not record cursor positions. To save to the waveform configuration file, used in situations such as establishing a time-base for multiple measurements and indicating notable events in the simulation, add markers to the Wave window instead. See Using Markers, page 72 for more information.*

Placing Main and Secondary Cursors

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

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

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

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

Moving Cursors

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

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

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

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

Finding the Next or Previous Transition on a Waveform

The Wave 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 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 Wave window at the time or on the transition.
  b. Right-click **Marker > Add Marker**.

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

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

```
add_wave_marker <-filename> <-line_number>
```

- You can move the marker to another location in the Wave window using the drag and drop method. Click the marker label (at the top of the marker or marker line) and drag it to the location.
    - The drag symbol indicates that the marker can be moved. As you drag the marker in the Wave 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 8] for command usage.

Using the Floating Ruler

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

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

The floating ruler 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.

Chapter 5

Debugging a Design with Vivado Simulator

Introduction

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

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

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

Debugging at the Source Level

You can debug your HDL source code to track down unexpected behavior in the design. Debugging is accomplished through controlled execution of the source code to determine where issues might be occurring. Available strategies for debugging are:

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

- Set conditions. The tools evaluate each condition and execute Tcl commands when the condition is true. Use the Tcl command:

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

  See Adding Conditions, page 78 for more information.

Stepping Through a Simulation

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

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

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

1. To step through a simulation:
   - From the current running time, select **Run > Step**, or click the **Step** button.

     The HDL associated with the top design unit opens as a new view in the Wave window.
   - From the start (0 ns), restart the simulation. Use the **Restart** command to reset time to the beginning of the test bench. See Chapter 3, Understanding Vivado Simulator.
2. In the waveform configuration window, right-click the waveform or HDL tab and select **Tile Horizontally** see the waveform and the HDL code simultaneously.
3. Repeat the **Step** action until debugging is complete.

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

Using Breakpoints

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

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

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

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

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 <`file_name`> <`line_number`>

   This command adds a breakpoint at <`line_number`>of <`file_name`>. See the Vivado Design Suite help or the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 7] 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 an arrow, 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_bps -all`

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

- Type the Tcl command `report_bps`

Adding Conditions

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

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

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

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

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

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 76 for more information.

Forcing Objects to Specific Values

Using Force Commands

The Vivado simulator provides an interactive mechanism to force a signal, wire, or register to a specified value at a specified time or period of time. You can also force values on objects to change over a period of time.

TIP: *A "force" is both an action (that is, the overriding of HDL-defined behavior on a signal) and also a Tcl first-class object, something you can hold in a Tcl variable.*

You can use force commands on an HDL signal to override the behavior for that signal as defined in your HDL design. You might, for example, choose to override the behavior of a signal to:

- Supply a stimulus to a test bench signal that the HDL test bench itself is not driving
- Correct a bad value temporarily during debugging (allowing you to continue analyzing a problem)

The available force commands are:

- Force Constant
- Force Clock
- Remove Force

IMPORTANT: *Running the `restart` command preserves all forces that have not been cleared with the `remove_force` command. When the simulation runs again, the preserved forces take effect at the same absolute simulation time as in the previous simulation run.*

Figure 5-1 illustrates how the add_force functionality is applied given the following command:

```
add_force mySig {0 t1} {1 t2} {0 t3} {1 t4} {0 t5} -repeat_every tr -cancel_after tc
```

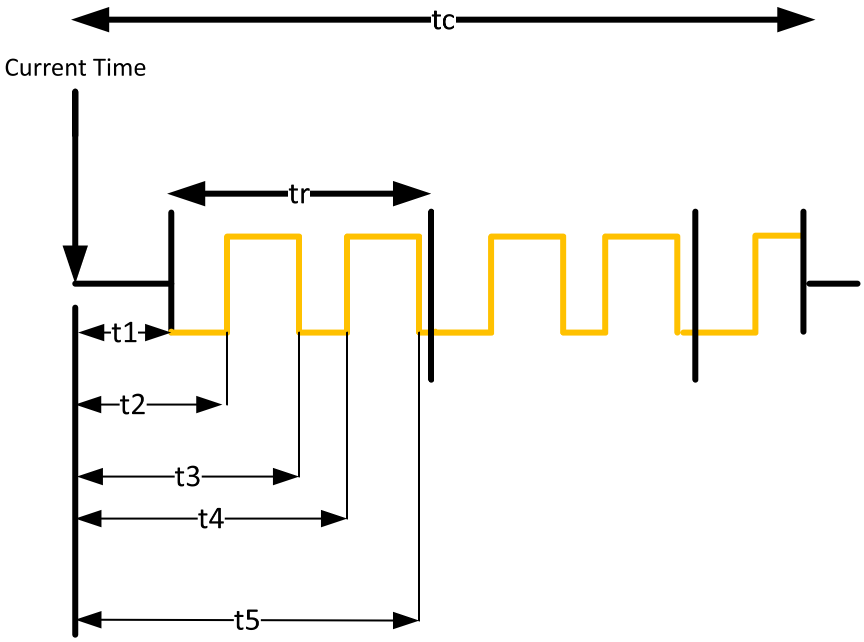


Figure 5-1: **Illustration of -add_force Functionality**

You can get more detail on the command by typing the following in the Tcl Console:

```
add_force -help
```

Force Constant

The Force Constant option lets you fix a signal to a constant value, overriding the assignments made within the HDL code or another previously applied constant or clock force.

Force Constant and **Force Clock** are options in the Objects or Wave window right-click menu (as shown in Figure 5-2), or in the text editor (source code).

TIP: *Double-click an item in the Objects, Sources, or Scopes window to open it in the text editor. For additional information about the text editor, see the Vivado Design Suite User Guide: Using the Vivado IDE [Ref 4].*

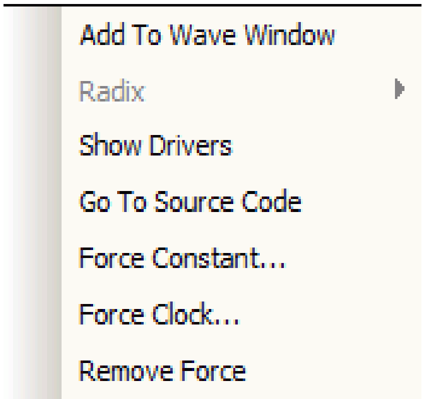
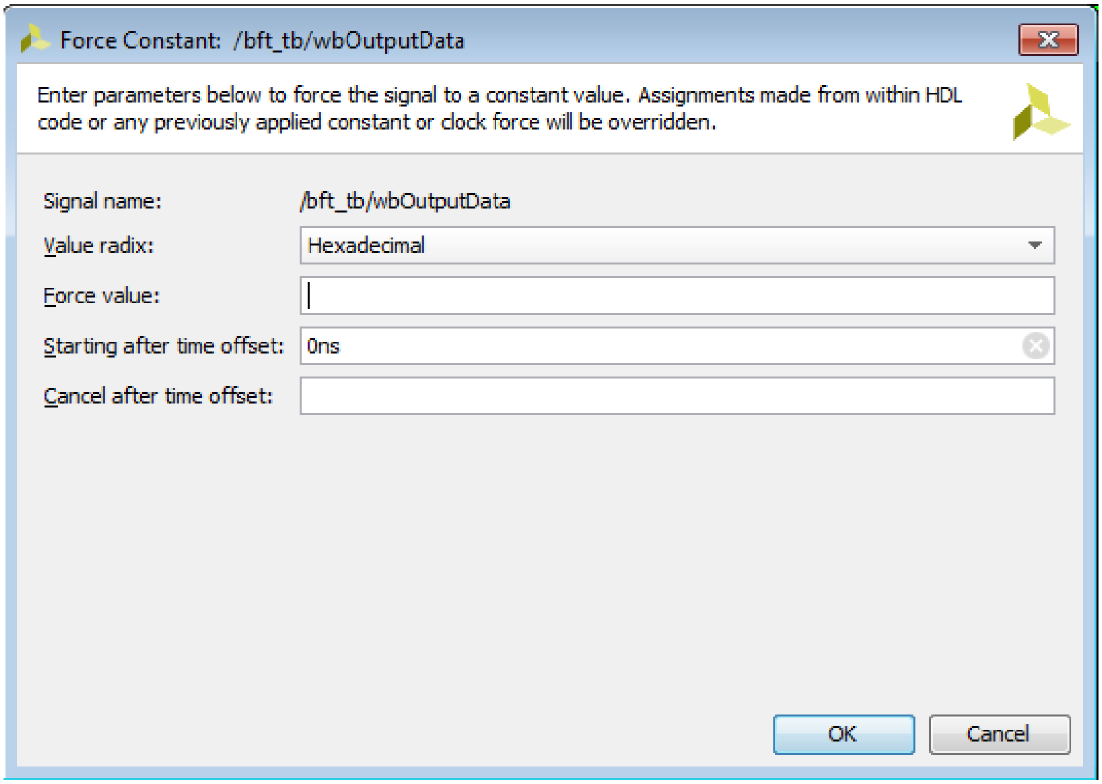


Figure 5-2: **Force Options**

When you select the Force Constant option, the Force Constant dialog box opens so you can enter the relevant values, as shown in Figure 5-3.


Force Constant: /bft_tb/wbOutputData

Enter parameters below to force the signal to a constant value. Assignments made from within HDL code or any previously applied constant or clock force will be overridden.

Signal name: /bft_tb/wbOutputData

Value radix: Hexadecimal

Force value:

Starting after time offset: 0ns

Cancel after time offset:

Figure 5-3: **Force Selected Signal Dialog Box**

The following are Force Constant option descriptions:

- **Signal name**: Displays the default signal name, that is, the full path name of the selected object.
- **Value radix**: Displays the current radix setting of the selected signal. You can choose one of the supported radix types: Binary, Hexadecimal, Unsigned Decimal, Signed Decimal, Octal, and ASCII. The GUI then disallows entry of the values based on the

Radix setting. For example: if you choose Binary, no numerical values other than 0 and 1 are allowed.

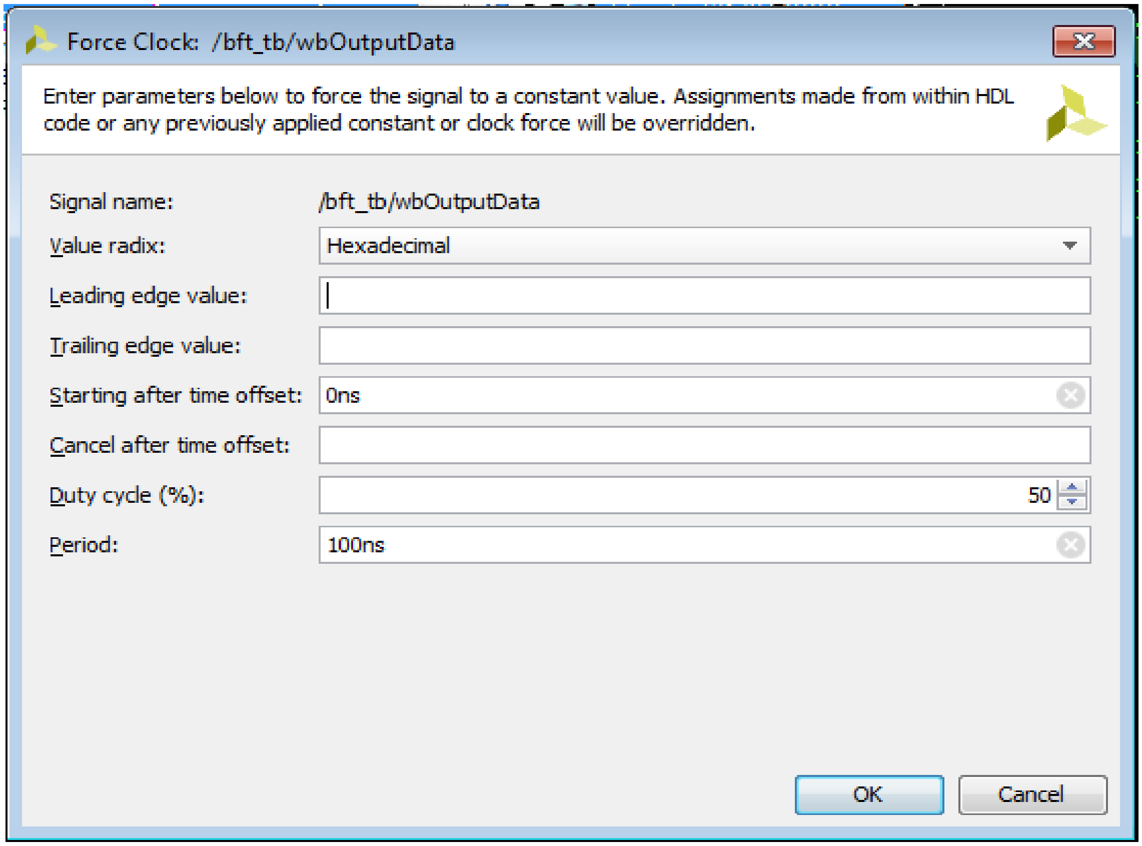
- **Force value**: Specifies a force constant value using the defined radix value. (For more information about radixes, see About Radixes, page 59 and Using Radixes and Analog Waveforms, page 61.)
- **Starting at time offset**: Starts after the specified time. The default starting time is 0. Time can be a string, such as 10 or 10 ns. When you enter a number without a unit, the Vivado simulator uses the default (ns).
- **Cancel after time offset**: Cancels after the specified time. Time can be a string such as 10 or 10 ns. If you enter a number without a unit, the default simulation time unit is used.

  Tcl command:

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

Force Clock

The Force Clock command lets you assign a signal a value that toggles at a specified rate between two states, in the manner of a clock signal, for a specified length of time. When you select the **Force Clock** option in the Objects window menu, the Force Clock dialog box opens, as shown in Figure 5-4.



Force Clock: /bft_tb/wbOutputData

Enter parameters below to force the signal to a constant value. Assignments made from within HDL code or any previously applied constant or clock force will be overridden.

Signal name: /bft_tb/wbOutputData
Value radix: Hexadecimal
Leading edge value:
Trailing edge value:
Starting after time offset: 0ns
Cancel after time offset:
Duty cycle (%): 50
Period: 100ns

Figure 5-4: **Force Clock Dialog Box**

The options in the Force Clock dialog box are shown below.

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

TIP: *The Force Clock command can be applied to any signal (not just clock signals) to define an oscillating value.*

- Value radix: Displays the current radix setting of the selected signal. Select one of the displayed radix types from the drop-down menu: `Binary`, `Hexadecimal`, `Unsigned Decimal`, `Signed Decimal`, `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 (default is 50%).
- Period: Specifies the length of the clock pulse, defined as a time value. Time can be a string, such as 10 or 10 ns.

Note: For more information about radixes, see About Radixes, page 59 and Using Radixes and Analog Waveforms, page 61.)

Example Tcl command:

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

Remove Force

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

```
remove_forces <force object>
remove_forces <HDL object>
```

Using Force in Batch Mode

The code examples below show how to force a signal to a specified value using the add_force command. A simple verilog circuit is provided. The first example shows the interactive use of the add_force command and the second example shows the scripted use.

Example 1: Adding Force

Verilog Code (tmp.v)

The following code snippet is a Verilog circuit:

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

module top;
reg in1, in2;
wire out1;
bot I1(in1, in2, out1);
initial
begin
    #10 in1 = 1'b1; in2 = 1'b0;
    #10 in1 = 1'b0; in2 = 1'b1;
end
initial
    $monitor("out1 = %b\n", out1);
endmodule
```

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
```

Tcl Commands

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

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

For more info on this and other Tcl commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 8].

Example 2: Scripted Use of add_force with remove_forces

Verilog Code (top.v)

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

```
module counter(input clk,reset,updown,output [4:0] out1);

reg [4:0] r1;

always@(posedge clk)
begin
    if(reset)
        r1 <= 0;
    else
        if(updown)
            r1 <= r1 + 1;
        else
            r1 <= r1 - 1;
end

assign out1 = r1;
endmodule

module top;
reg clk;
reg reset;
reg updown;
wire [4:0] out1;

counter I1(clk, reset, updown, out1);

initial
begin
    reset = 1;
    #20 reset = 0;
end

initial
begin
    updown = 1; clk = 0;
end

initial
    #500 $finish;

initial
    $monitor("out1 = %b\n", out1);
endmodule
```

Command Example

1. Create a file called `add_force.tcl` with following command:

   ```
   create_project add_force -force
   add_files top.v
   set_property top top [get_filesets sim_1]
   set_property -name xelab.more_options -value {-debug all} -objects
   [get_filesets sim_1]
   set_property runtime {0} [get_filesets sim_1]
   launch_simulation -simset sim_1 -mode behavioral
   add_wave /top/*
   ```

2. Invoke the Vivado Design Suite in Tcl mode, and source the `add_force.tcl` file.
3. In the Tcl Console, type:

   ```
   set force1 [add_force clk {0 1} {1 2} -repeat_every 3 -cancel_after 500]
   set force2 [add_force updown {0 10} {1 20} -repeat_every 30]
   run 100
   ```

   Observe that the value of `out1` increments as well as decrements in the Wave window. You can observe the waveforms in the Vivado IDE using the `start_gui` command.

   Observe the value of `updown` signal in the Wave window.

4. In the Tcl Console, type:

   ```
   remove_forces $force2
   run 100
   ```

   Observe that only the value of `out1` increments.

5. In the Tcl Console, type:

   ```
   remove_forces $force1
   run 100
   ```

   Observe that the value of `out1` is not changing because the `clk` signal is not toggling.

Power Analysis Using Vivado Simulator

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

Switching Activity Interchange Format (SAIF) dumping is optimized for Xilinx power tools and for use by the `report_power` Tcl command. The Vivado simulator writes the following HDL types to the SAIF file. Refer to this link in the *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907) [Ref 9] for additional information.

- Verilog:
  - Input, Output, and Inout ports
  - Internal wire declarations
- VHDL:
  - Input, Output, and Inout ports of type `std_logic`, `std_ulogic`, and `bit` (scalar, vector, and arrays).

Note: a VHDL netlist is not generated in the Vivado Design Suite for timing simulations; consequently, the VHDL sources are for RTL-level code only, and not for netlist simulation.

For RTL-level simulations, only block-level ports are generated and not the internal signals.

For information about power analysis using third-party simulation tools, see Chapter 8, Using Third-Party Simulators, Dumping SAIF for Power Analysis, page 152

Generating SAIF Dumping

Before you use the `log_saif` command, you must call `open_saif`. The `log_saif` command does not return any object or value.

1. Compile your RTL code with the `-debug typical` option to enable SAIF dumping:

   ```
   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, use the Tcl command:

   ```
   log_saif [get_objects -r *]
   ```

4. Run the simulation (use any of the run commands).
5. Import simulation data into an SAIF format using the following Tcl command:

   ```
   close_saif
   ```

Example SAIF Tcl Commands

To log SAIF for:

- All signals in the scope: `/tb: log_saif /tb/*`
- All the ports of the scope: `/tb/UUT`
- Those objects having names that start with `a` and end in `b` and have digits in between: `log_saif [get_objects -regexp {^a[0-9]+b$}]`
- The objects in the `current_scope` and `children_scope`: `log_saif [get_objects -r *]`
- The objects in the `current_scope`: `log_saif * or log_saif [get_objects]`
- Only the ports of the `scope /tb/UUT`, use the command:

  ```
  log_saif [get_objects -filter {type == in_port || type == out_port || type ==
  inout_port || type == port } /tb/UUT/* ]
  ```

- Only the internal signals of the `scope /tb/UUT`, use the command:

  ```
  log_saif [get_objects -filter { type == signal } /tb/UUT/* ]
  ```

TIP: *This filtering is applicable to all Tcl commands that require HDL objects.*

Dumping SAIF using a Tcl Simulation Batch File

```
sim.tcl:
open_saif xsim_dump.saif
log_saif /tb/dut/*
run all
close_saif
quit
```

Using the report_drivers Tcl Command

You can use the `report_drivers` Tcl command to determine what signal is *driving* a value on an HDL object. The syntax is as follows:

```
report_drivers <hdl_object>
```

The command prints drivers (HDL statements doing the assignment) to the Tcl Console along with current driving values on the right side of the assignment to a wire or signal-type HDL object.

You can also call the `report_drivers` command from the Object or Wave window context menu or text editor. To open the context menu (shown in the figure below), right-click any signal and click **Report Drivers.** The result appears in the Tcl console.

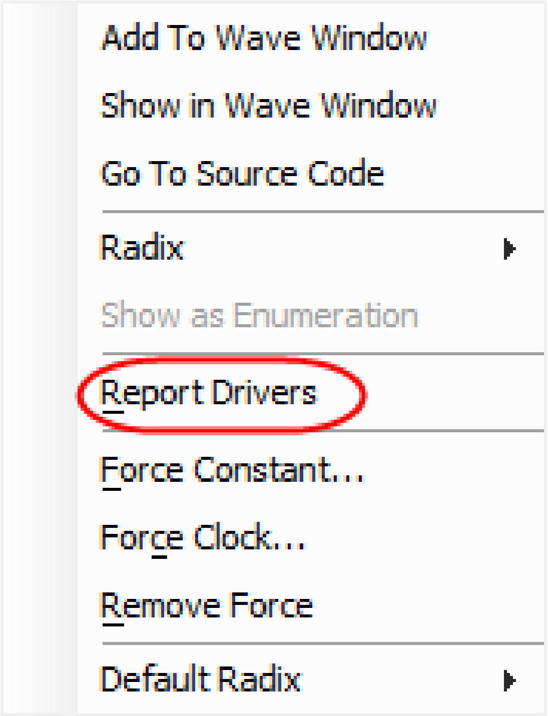


Figure 5-5: **Context Menu with Report Drivers Command Option**

Using the Value Change Dump Feature

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

For the VCD feature, the Tcl commands listed in the table below model the Verilog system tasks.

Table 5-1: **Tcl Commands for VCD**

Tcl Command	Description
`open_vcd`	Opens a VCD file for capturing simulation output. This Tcl command models the behavior of `$dumpfile` Verilog system task.
`checkpoint_vcd`	Models the behavior of the `$dumpall` Verilog system task.
`start_vcd`	Models the behavior of the `$dumpon` Verilog system task.

Table 5-1: **Tcl Commands for VCD *(Cont'd)***

Tcl Command	Description
`log_vcd`	Logs VCD for the specified HDL objects. This command models behavior of the `$dumpvars` Verilog system task.
`flush_vcd`	Models behavior of the `$dumpflush` Verilog system task.
`limit_vcd`	Models behavior of the `$dumplimit` Verilog system task.
`stop_vcd`	Models behavior of the `$dumpoff` Verilog system task.
`close_vcd`	Closes the VCD generation.

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 7], or type the following in the Tcl Console:

```
<command> -help
```

Example:

```
open_vcd xsim_dump.vcd
log_vcd /tb/dut/*
run all
close_vcd
quit
```

See Verilog Language Support Exceptions in Appendix B for more information.

You can use the VCD data to validate the output of the simulator to debug simulation failures.

Using the log_wave Tcl Command

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

TIP: *To display object values prior to the time of insertion, the simulation must be restarted. To avoid having to restart the simulation because of missing value changes: issue the log_wave -r / Tcl command at the start of a simulation run to capture value changes for all display-able HDL objects in your design.*

Syntax:

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

Example log_wave TCL Command Usage

To log the waveform output for:

- All signals in the design (excluding those of alternate top modules):

  ```
  log_wave -r /
  ```
- All signals in a scope: `/tb`:

  ```
  log_wave /tb/*
  ```
- Those objects having names that start with a and end in b and have digits in between:

  ```
  log_wave [get_objects -regexp {^a[0-9]+b$}]
  ```
- All objects in the current scope and all child scopes, recursively:

  ```
  log_wave -r *
  ```
- The objects in the current scope:

  ```
  log_wave *
  ```
- Only the ports of the scope `/tb/UUT`, use the command:

  ```
  log_wave [get_objects -filter {type == in_port || type == out_port || type ==
  inout_port || type == port} /tb/UUT/*]
  ```
- Only the internal signals of the scope `/tb/UUT`, use the command:

  ```
  log_wave [get_objects -filter {type == signal} /tb/UUT/*]
  ```

The wave configuration settings; which include the signal order, name style, radix, and color; are saved to the wave configuration (WCFG) file upon demand. See Chapter 4, Analyzing Simulation Waveforms.

Cross Probing Signals in the Object, Wave, and Text Editor Windows

In Vivado simulator, you can do cross probing on signals present in the Objects, Wave, and text editor windows.

From the Objects window, you can check to see if a signal is present in the Wave window and vice versa. Right-click the signal to open the context menu shown in the figure below.

Click Show in Wave Window or **Add to Wave Window** (if signal is not yet present in the Wave window).

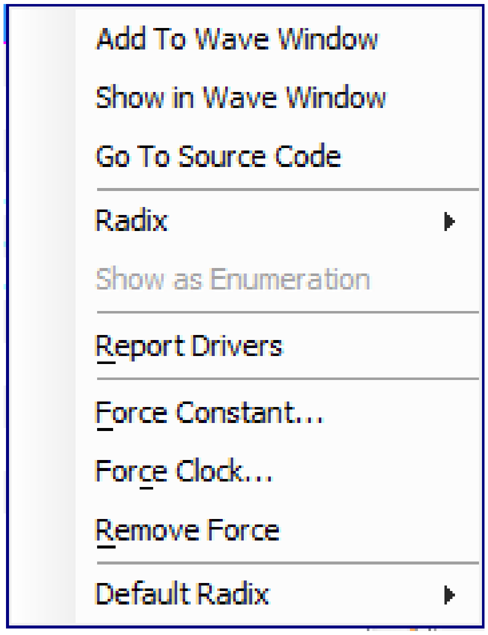


Figure 5-6: **Objects Window Context Menu Options**

You can also cross probe a signal from the text editor. Right-click a signal to open the context menu shown in the figure below. Select **Add to Wave Window**, **Show in Waveform** or **Show in Objects**. The signal then appears highlighted in the Wave or Objects window.

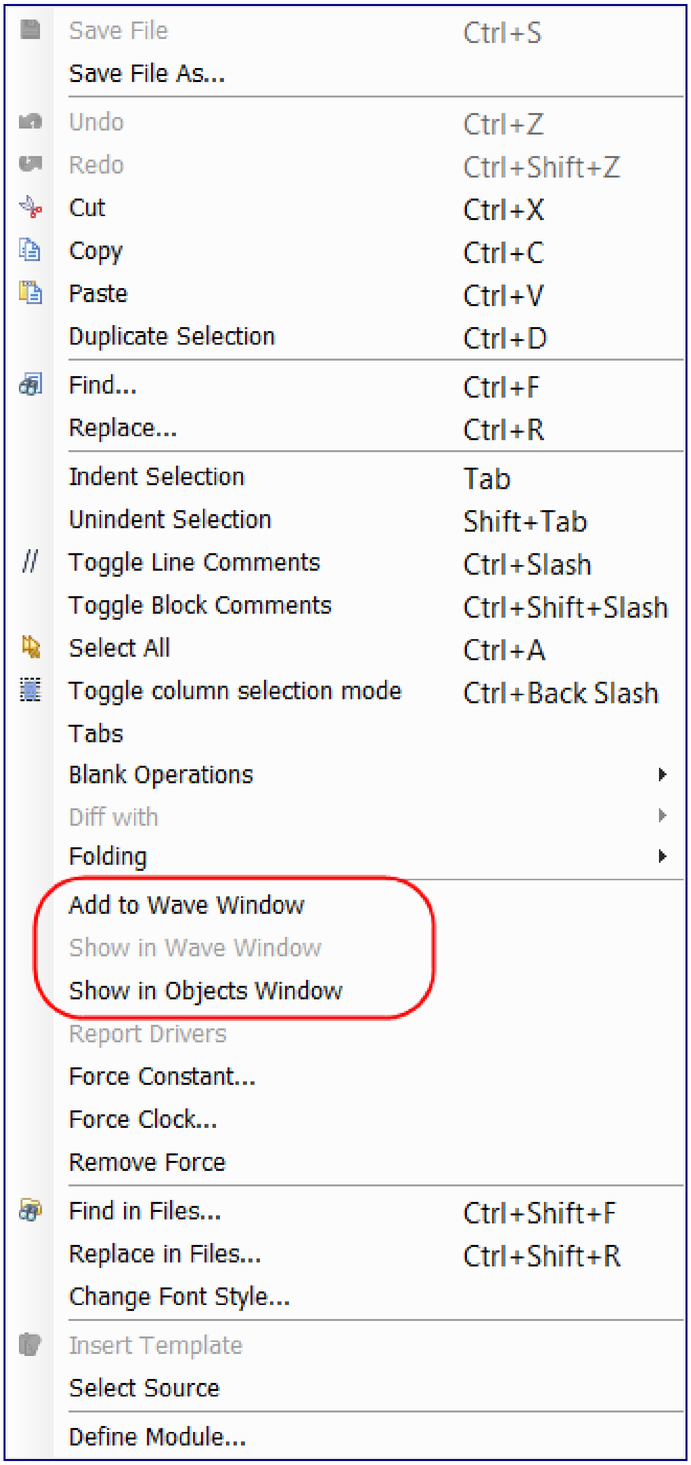


Figure 5-7: **Text Editor Right-Click (Context) Menu**

Chapter 6

Handling Special Cases

Using Global Reset and 3-State

Xilinx® devices have dedicated routing and circuitry that connect to every register in the device.

Global Set and Reset Net

During configuration, the dedicated Global Set/Reset (GSR) signal is asserted. The GSR signal is deasserted upon completion of device configuration. All the flip-flops and latches receive this reset, and are set or reset depending on how the registers are defined.

✓ **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 6-1 shows how Global 3-State (GTS) and Global Set/Reset (GSR) signals are used in an FPGA.

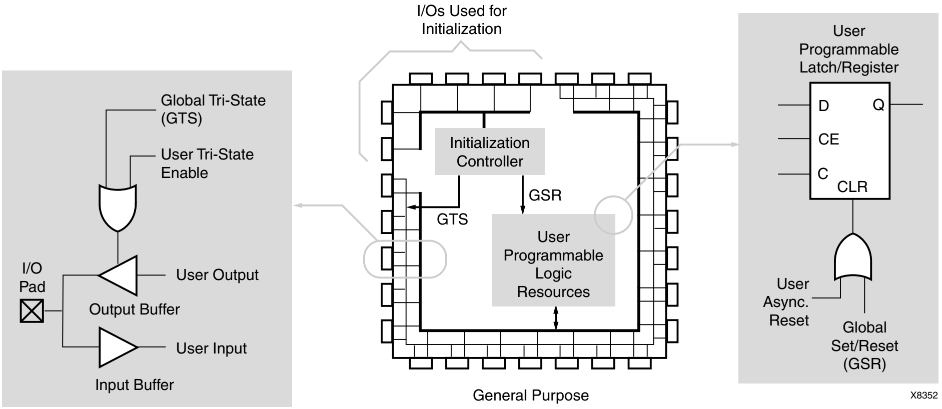


Figure 6-1: **Built-in FPGA Initialization Circuitry Diagram**

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

```
clk_b <= clk;
clk_prcs : process (clk)
begin
  if (clk'event and clk='1') then
    result <= data;
  end if;
end process;

clk_b_prcs : process (clk_b)
begin
  if (clk_b'event and clk_b='1') then
    result1 <= result;
  end if;
end process;
```

In this example, there are two synchronous processes:

- `clk_prcs`
- `clk_b_prcs`

The simulator performs the `clk_b <= clk` assignment before advancing the simulation time. As a result, events that should occur in two clock edges occur in one clock edge instead, causing a race condition.

Recommended ways to introduce causality in simulators for such cases include:

- Do not change clock and data at the same time. Insert a delay at every output.
- Use the same clock.

- Force a delta delay by using a temporary signal, as shown in the following example:

```
clk_b <= clk;
clk_prcs : process (clk)
begin
  if (clk'event and clk='1') then
    result <= data;
  end if;
end process;

result_temp <= result;
clk_b_prcs : process (clk_b)
begin
if (clk_b'event and clk_b='1') then
    result1 <= result_temp;
  end if;
end process;
```

Most event-based simulators can display delta cycles. Use this to your advantage when debugging simulation issues.

Using the ASYNC_REG Constraint

The `ASYNC_REG` constraint:

- Identifies asynchronous registers in the design
- Disables X propagation for those registers

The `ASYNC_REG` constraint can be attached to a register in the front-end design by using either:

- An attribute in the HDL code
- A constraint in the Xilinx Design Constraints (XDC)

The registers to which `ASYNC_REG` are attached retain the previous value during timing simulation, and do not output an X to simulation. Use care; a new value might have been clocked in as well.

The `ASYNC_REG` constraint is applicable to CLB and Input Output Block (IOB) registers and latches only. For more information, refer to ASYNC_REG constraint at this link in the *Vivado Design Suite properties Reference Guide* (UG912) [Ref 13].

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 10].*

Disabling X Propagation for Synchronous Elements

When a timing violation occurs during a timing simulation, the default behavior of a latch, register, RAM, or other synchronous elements is to output an X to the simulator. This occurs because the actual output value is not known. The output of the register could:

- Retain its previous value
- Update to the new value
- Go metastable, in which a definite value is not settled upon until some time after the clocking of the synchronous element

Because this value cannot be determined, and accurate simulation results cannot be guaranteed, the element outputs an X to represent an unknown value. The X output remains until the next clock cycle in which the next clocked value updates the output if another violation does not occur.

The presence of an X output can significantly affect simulation. For example, an X generated by one register can be propagated to others on subsequent clock cycles. This can cause large portions of the design under test to become unknown.

To correct X-generation:

- On a synchronous path, analyze the path and fix any timing problems associated with this or other paths to ensure a properly operating circuit.
- On an asynchronous path, if you cannot otherwise avoid timing violations, disable the X propagation on synchronous elements during timing violations by using the `ASYNC_REG` property.

When X propagation is disabled, the previous value is retained at the output of the register. In the actual silicon, the register might have changed to the 'new' value. Disabling X propagation might yield simulation results that do not match the silicon behavior.

CAUTION! *Exercise care when using this option. Use it only if you cannot otherwise avoid timing violations.*

Simulating Configuration Interfaces

This section describes the simulation of the following configuration interfaces:

- JTAG simulation
- SelectMAP simulation

JTAG Simulation

BSCAN component simulation is supported on all devices.

The simulation supports the interaction of the JTAG ports and some of the JTAG operation commands. The JTAG interface, including interface to the scan chain, is not fully supported. To simulate this interface:

1. Instantiate the BSCANE2 component and connect it to the design.
2. Instantiate the JTAG_SIME2 component into the test bench (not the design).

This becomes:

- The interface to the external JTAG signals (such as TDI, TDO, and TCK)
- The communication channel to the BSCAN component

The communication between the components takes place in the VPKG VHDL package file or the 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 models (SIM_CONFIGE2 and SIM_CONFIGE3) with an instantiation template allow supported configuration interfaces to be simulated to ultimately show the DONE pin going HIGH. This is a model of how the supported devices react to stimulus on the supported configuration interface.

Table 6-1 lists the supported interfaces and devices.

Table 6-1: **Supported Configuration Devices and Modes**

Devices	SelectMAP	Serial	SPI	BPI
7 Series and Zynq®-7000 AP SoC Devices	Yes	Yes	No	No
UltraScale™ Devices	Yes	Yes	No	No

The models handle control signal activity as well as bit file downloading. Internal register settings such as the CRC, IDCODE, and status registers are included. You can monitor the Sync Word as it enters the device and the start-up sequence as it progresses. Figure 6-2, 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.

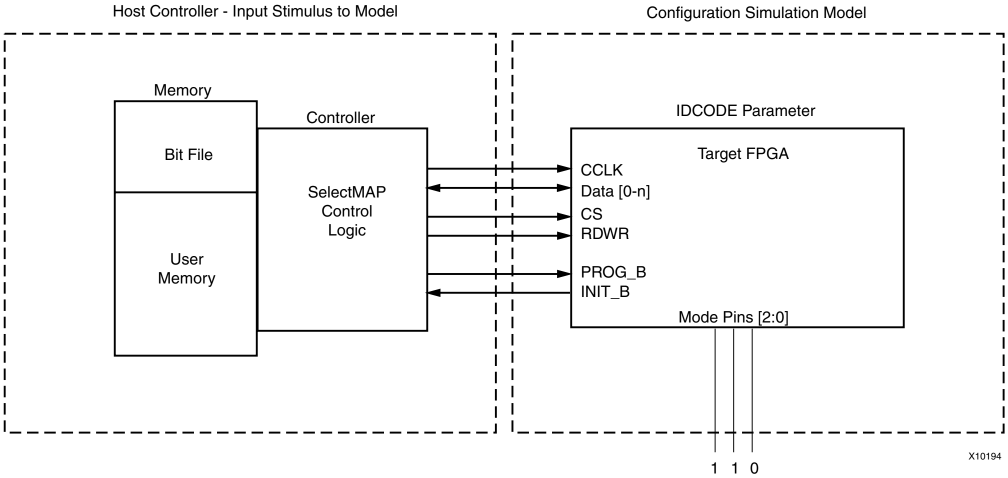


Figure 6-2: **Block Diagram of Model Interaction**

System Level Description

The configuration models allow the configuration interface control logic to be tested before the hardware is available. It simulates the entire device, and is used at a system level for:

- Applications using a processor to control the configuration logic to ensure proper wiring, control signal handling, and data input alignment.
- Applications that control the data loading process with the `CS` (SelectMAP Chip Select) or `CLK` signal to ensure proper data alignment.
- Systems that need to perform a SelectMAP `ABORT` or `Readback`.

The 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 configuration models also demonstrate what is occurring inside the device during the configuration procedure when a BIT file is loaded into the device.

During the BIT file download, the model processes each command and changes registers settings that mirror the hardware changes.

You can monitor the CRC register as it actively accumulates a CRC value. The model also shows the Status Register bits being set as the device progresses through the different states of configuration.

Debugging with the Model

Each configuration model provides an example of a correct configuration. You can leverage this example to assist in the debug procedure if you encounter device programming issues.

You can read the Status Register through JTAG using the Vivado Device Programmer tool. This register contains information relating to the current status of the device and is a useful debugging resource. If you encounter issues on the board, reading the Status Register in Vivado Device Programmer is one of the first debugging steps to take.

After the status register is read, you can map it to the simulation to pinpoint the configuration stage of the device.

For example, the `GHIGH` bit is set HIGH after the data load process completes successfully; if this bit is not set, then the data loading operation did not complete. You can also monitor the `GTW`, `GWE`, and `DONE` signals set in BitGen that are released in the start-up sequence.

The configuration models also allow for error injection. The active CRC logic detects any issue if the data load is paused and started again with any problems. It also detects bit flips manually inserted in the BIT file, and handles them just as the device would handle this error.

Feature Support

Each device-specific configuration user guide outlines the supported methods of interacting with each configuration interface.The table below shows which features discussed in the configuration user guides are supported.

The `SIM_CONFIGE2` model:

- Does not support Readback of configuration data.
- Does not store configuration data provided, although it does calculate a CRC value.
- Can perform Readback on specific registers only to ensure that a valid command sequence and signal handling is provided to the device.
- Is not intended to allow Readback data files to be produced.

Table 6-2: **Model-Supported Slave SelectMAP and Serial Features**

Slave SelectMAP and Serial Features	Supported
Master mode	No
Daisy chain - slave parallel daisy chains	No
SelectMAP data loading	Yes
Continuous SelectMAP data loading	Yes
Non-continuous SelectMAP data loading	Yes
SelectMAP `ABORT`	Yes
SelectMAP reconfiguration	No
SelectMAP data ordering	Yes
Reconfiguration and MultiBoot	No
Configuration CRC—CRC checking during configuration	Yes
Configuration CRC—post-configuration CRC	No

Disabling Block RAM Collision Checks for Simulation

Xilinx block RAM memory is a true dual-port RAM where both ports can access any memory location at any time. Be sure that the same address space is not accessed for reading and writing at the same time. This causes a block RAM address collision. These are valid collisions, because the data that is being read from the read port is not valid.

In the hardware, the value that is read might be the old data, the new data, or a combination of the old data and the new data.

In simulation, this is modeled by outputting X because the value read is unknown. For more information on block RAM collisions, see the user guide for the device.

In certain applications, this situation cannot be avoided or designed around. In these cases, the block RAM can be configured not to look for these violations. This is controlled by the generic (VHDL) or parameter (Verilog) `SIM_COLLISION_CHECK` string in block RAM primitives.

Table 6-3 shows the string options you can use with SIM_COLLISION_CHECK to control simulation behavior in the event of a collision.

Table 6-3: **SIM_COLLISION_CHECK Strings**

String	Write Collision Messages	Write Xs on the Output
ALL	Yes	Yes
WARNING_ONLY	Yes	No. Applies only at the time of collision. Subsequent reads of the same address space could produce Xs on the output.
GENERATE_X_ONLY	No	Yes
None	No	No. Applies only at the time of collision. Subsequent reads of the same address space could produce Xs on the output.

Apply the SIM_COLLISION_CHECK at an instance level so you can change the setting for each block RAM instance.

Dumping the Switching Activity Interchange Format File for Power Analysis

- Vivado simulator: Power Analysis Using Vivado Simulator, page 88
- Chapter 8, Using Third-Party Simulators, Dumping SAIF for Power Analysis, page 152

Simulating a Design with AXI Bus Functional Models

For information on this topic see Simulating a Design with AXI Bus Functional Models, page 155.

Skipping Compilation or Simulation

Skipping Compilation

You can run simulation on an existing snapshot and skip the compilation (or recompilation) of the design by setting the SKIP_COMPILATION property on the simulation fileset:

```
set_property SKIP_COMPILATION 1 [get_filesets sim_1]
```

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

Skipping Simulation

To perform a semantic check on the design HDL files, by elaborating and compiling the simulation snapshot without running simulation, you can set the SKIP_SIMULATION property on the simulation fileset:

```
set_property SKIP_SIMULATION true [get_filesets sim_1]
```

IMPORTANT: *If you elect to use one of the properties above, disable the* `clean up simulation files` *checkbox in the simulations settings or, if you are running in batch/Tcl mode, call* `launch_simulation` *with* `-noclean_dir`*.*

Chapter 7

Simulating in Batch or Scripted Mode

Introduction

This chapter describes the command line compilation and simulation process.

Vivado supports an integrated simulation flow where the tool can launch Vivado simulator, or a third party simulator, from the IDE. However, many users also want to run simulation in batch or scripted mode in their verification environment, which may include system-level simulation, or advanced verification such as UVM. The Vivado Design Suite supports batch or scripted simulation for third party verification, as well as in the Vivado simulator.

This chapter describes a process to gather the needed design files, to generate simulation scripts for your target simulator, and to run simulation in batch mode. The simulation scripts can be generated for a top-level HDL design, or for hierarchical modules, managed IP projects, or block designs from Vivado IP integrator. Batch simulation is supported in both project and non-project script-based flow.

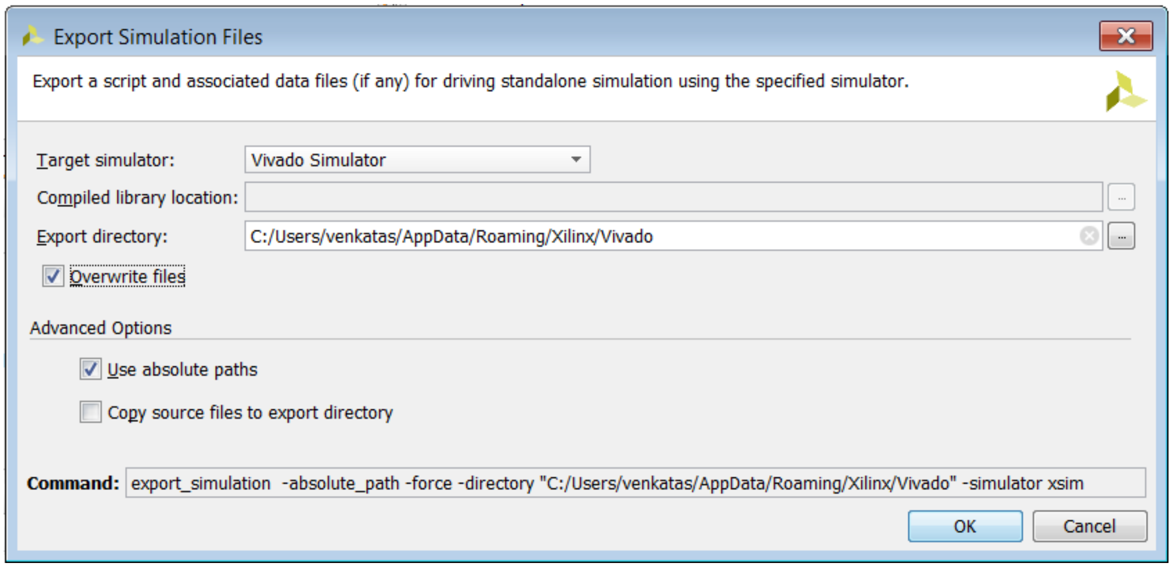
Exporting Simulation Files and Scripts

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

1. Identifying and parsing design files.
2. Elaborating and generating an executable simulation snapshot of the design.
3. Running simulation using the executable snapshot.

The Vivado Design Suite provides an Export Simulation command to let you quickly gather the design files required for simulation, and generate the simulation scripts for the top-level RTL design, or a sub-design. The `export_simulation` command will generate scripts for all of the supported third-party simulators, or for the target simulator of your choice.

From within the Vivado IDE, use the **File> Export > Export Simulation** command to open the Export Simulation dialog box as shown in Figure 7-1, page 107.



Export Simulation Files

Export a script and associated data files (if any) for driving standalone simulation using the specified simulator.

Target simulator: Vivado Simulator

Compiled library location:

Export directory: C:/Users/venkatas/AppData/Roaming/Xilinx/Vivado

☑ Overwrite files

Advanced Options

☑ Use absolute paths

□ Copy source files to export directory

Command: export_simulation -absolute_path -force -directory "C:/Users/venkatas/AppData/Roaming/Xilinx/Vivado" -simulator xsim

Figure 7-1: **Export Simulation dialog box**

The Export Simulation command writes a simulation script file for all supported simulators, or for the specified Target simulator. The generated scripts will contain simulator commands for compiling, elaborating, and simulating the design.

The features of the Export Simulation dialog box include the following:

- **Target simulator:** Specifies all simulators, or a specific simulator to generate command line scripts for. Target simulators include Vivado simulator as well as various supported third-party simulators. Refer to Chapter 8, Using Third-Party Simulators for more information.

  Note: On the Windows operating system, scripts will only be generated for those simulators that run on Windows.

- **Compiled library location:** In order to perform simulation with the script generated by Export Simulation, your simulation libraries must first be compiled with the compile_simlib Tcl command. The generated scripts will automatically include the setup files needed for the target simulator from the compiled library directory. Refer to Compiling Simulation Libraries, page 138 for more information.

TIP: *It is recommended to provide the path to the Compile library location whenever running Export SImulation. This insures that the scripts will always point to the correct simulation libraries.*

- **Export directory:** Specifies the output directory for the scripts written by Export Simulation. By default, the simulation scripts are written to the local directory of the current project.
- **Overwrite files:** Force the creation of new files even if files of the same name already exist.

- **Use absolute paths:** Source files and directory paths in the generated scripts will be relative to a reference directory that is defined in the scripts. Use this switch to make file paths in the script absolute rather than relative.
- **Copy source files to export directory:** Copy design files to the output directory. This copies the simulation source files as well as the generated scripts to make the entire simulation folder more portable.
- **Command:** This field provides the Tcl command syntax for the export_simulation command that will be run as a result of the various options and settings that you have specified in the Export Simulation dialog box.

The Export Simulation command supports both project and non-project designs. It does not read properties from the current project except to query for Verilog ‘defines and ‘include directories. Instead, the Export Simulation command gets directives from the dialog box or from `export_simulation` command options. You must specify the appropriate options to get the results you want. In addition, you must have output products generated for all IP and BD that are used in the top-level design.

IMPORTANT: *The* `export_simulation` *command will not generate output products for IP and BD if they do not exist. Instead it will return an error and exit.*

When you click OK on the Export Simulation dialog box, the command gets the simulation compile order of all design files required for simulating the specified design object: the top-level design, a hierarchical module, IP core, a block design from Vivado IP integrator, or a Managed IP project with multiple IP. The simulation compile order of the required design files is exported to a shell script with compiler commands and options for the target simulator.

The simulation scripts are written to separate folders in the Export directory as specified in the Export Simulation dialog box. A separate folder is created for each specified simulator, and `compile`, `elaborate`, and `simulate` scripts are written for the simulator.

The scripts generated by the Export Simulation command uses a 3-step process, analyze/compile, elaborate and simulate, that is common to many simulators including the Vivado simulator. However, for ModelSim the generated scripts use the two-step process of compile and simulate that the tool requires.

TIP: *To use the two-step process in the Questa or Aldec simulators, you can start with the scripts generated for ModelSim and modify them as needed.*

The Export Simulation command will also copy data files (if any) from the fileset, or from an IP, to the specified export directory. If the design contains Verilog sources, then the generated script will also copy the "glbl.v" file from the Vivado software installation path to the output directory.

Tcl Command Example for Export Simulation

```
set_property export.sim.base_dir C:/Data/project_wave1 [current_project]
export_ip_user_files -no_script -force
export_simulation -directory "C:/Data/project_wave1" -simulator all
```

When you run the Export Simulation command from the dialog box, the Vivado IDE actually runs a sequence of commands that defines the base directory (or location) for the exported scripts, exports the IP user files, and then runs the export_simulation command.

The `export_ip_user_files` command is run automatically by the Vivado IDE to ensure that all required files needed to support simulation for both core container and non-core container IP, as well as block designs, are available. See this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 3] for more information. While `export_ip_user_files` is run automatically when working with the Export Simulation dialog box, you must be sure to run it manually before running the `export_simulation` command.

TIP: *Notice the `-no_script` option is specified when `export_ip_user_files` is run automatically by the Vivado IDE. This is to prevent the generation of simulation scripts for the individual IP and BDs that are used in the top-level design since it can add significant run time to the command. However, you can generate these simulation scripts for individual IP and BD by running `export_ip_user_files` on specified objects (`-of_objects`), or without the `-no_script` option.*

The `export_ip_user_files` command sets up the user file environment for IP and block design needed for simulation and synthesis. The command creates a folder called `ip_user_files` which contains instantiation templates, stub files for use with 3rd party synthesis tools, wrapper files, memory initialization files, and simulation scripts.

The `export_ip_user_files` command also consolidates static simulation files that are shared across all IP and block designs in the project and copies them to an `ipstatic` folder. Many of the IP files that are shared across multiple IP and BDs in a project do not change for specific IP customizations. These static files are copied into the `ipstatic` directory. The scripts created for simulation reference the shared files in this directory as needed. The dynamic simulation files that are specific to an IP customization are copied to the IP folder. See this link, or "Understanding IP User Files" in *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 3] for more information.

IMPORTANT: *The scripts and files generated by the `export_simulation` command point to the files in the `ip_user_files` directory. You must run the `export_ip_user_files` command before you run `export_simulation` or simulation errors may occur.*

Exporting the Top level design

To create simulation scripts for the top-level RTL design use `export_simulation` and provide the simulation fileset object. In the following example `sim_1` is the simulation fileset, and export simulation will create simulation scripts for all the RTL entities, IP, and BD objects in the design.

```
export_ip_user_files -no_script
export_simulation -of_objects [get_filesets sim_1] -directory C:/test_sim \
-simulator questa
```

Exporting IP from the Xilinx Catalog and Block Designs

To generate scripts for an IP, or a Vivado IP integrator block design, you can simply run the command on the IP or block design object:

```
export_ip_user_files -of_objects [get_ips blk_mem_gen_0] -no_script -force
export_simulation -simulator ies -directory ./export_script \
-of_objects [get_ips blk_mem_gen_0]
```

Or, export the ip_user_files for all IP and BDs in the design:

```
export_ip_user_files -no_script -force
export_simulation -simulator ies -directory ./export_script
```

You can also generate simulation scripts for block design objects:

```
export_ip_user_files -of_objects [get_files base_microblaze_design.bd] \
-no_script -force
export_simulation -of_objects [get_files base_microblaze_design.bd] \
-directory ./sim_scripts
```

IMPORTANT: *You must have output products generated for all IP and BD that are used in the top-level design. The `export_simulation` command will not generate output products for IP and BD if they do not exist. Instead it will return an error and exit.*

Exporting a Manage IP Project

Manage IP project provides users an ability to create and manage a centralized repository of customized IPs. See this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 3] for more information on Manage IP projects. When generating the IP output products for Manage IP projects, the Vivado tool also generates simulation scripts for each IP using the `export_ip_user_files` command as previously discussed.

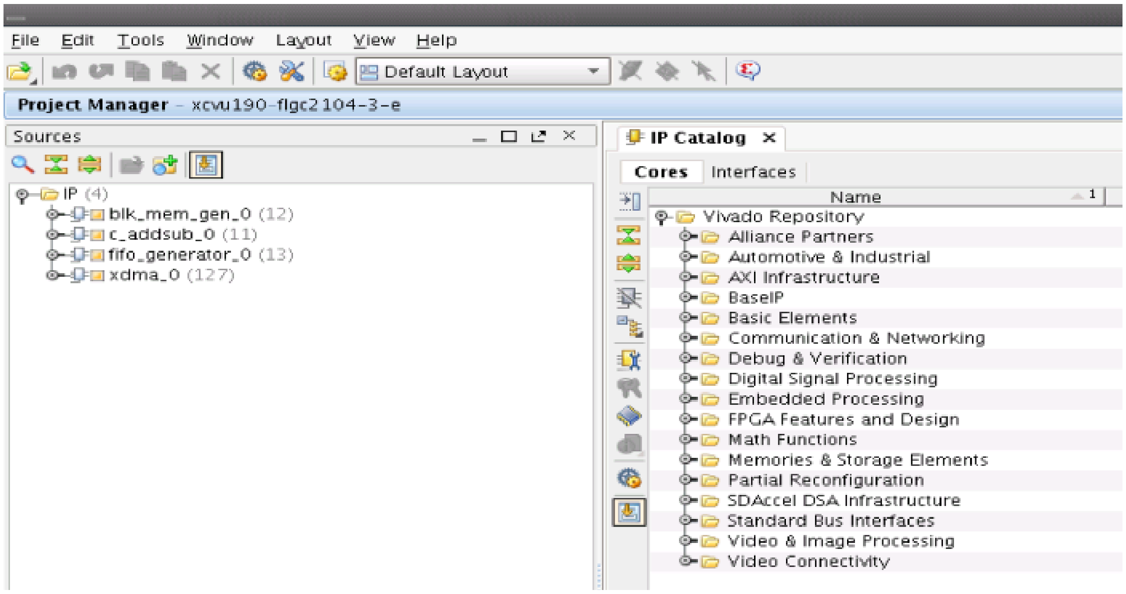


Figure 7-2: **Managed IP Project**

The Managed IP Project shown above features four different customized IP: `blk_mem_gen_0`, `c_addsub_0`, `fifo_generator_0`, `xdma_0`. For this project the Vivado Design Suite creates an `ip_user_files` folder as shown in the following figure.

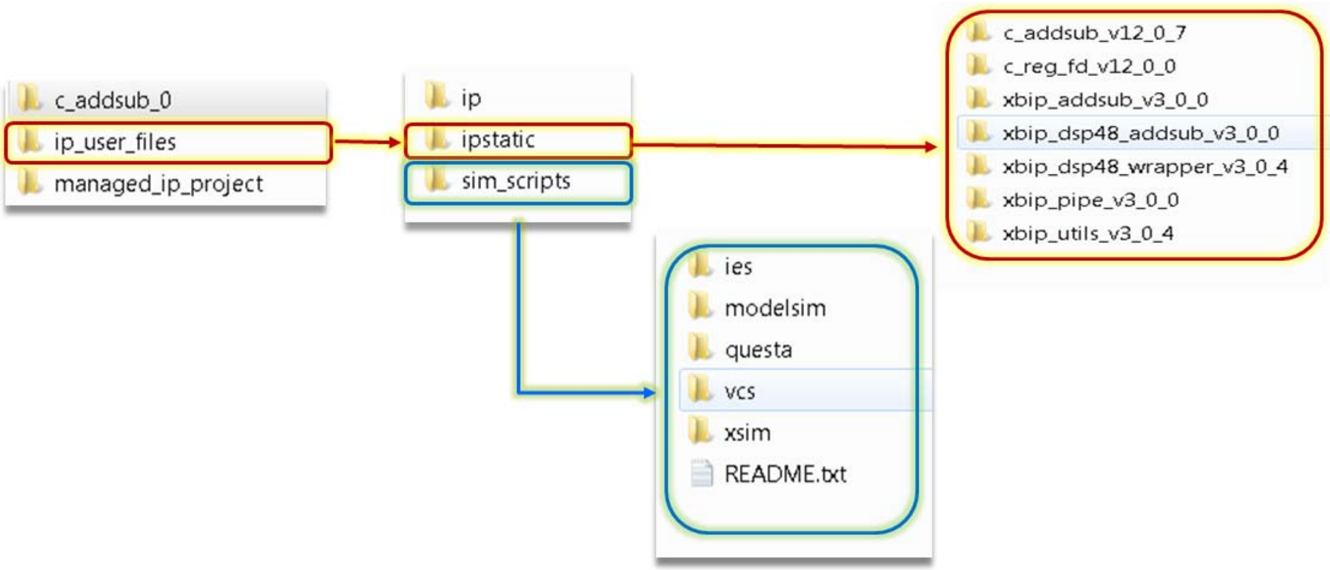


Figure 7-3: **Managed IP Directory Structure**

The `ip_user_files` folder is generated by the `export_ip_user_files` command as previously described. When this command is run on a Manage IP project, it will recursively process all the IP in the project and generate the scripts and other files needed for synthesis and simulation of the IP. The `ip_user_files` folder contains the scripts used for batch simulation, as well as the dynamic and static IP files needed to support simulation.

The simulation scripts for your target simulator, or for all supported simulators, are located in the ./`sim_scripts` folder as seen in Figure 7-3, page 111. You can go to the folder of your target simulator and incorporate the `compile`, `elaborate`, and `simulate` scripts into your simulation flow.

The Vivado tool consolidates all the shared simulation files, used by multiple IP and BD in the design, into a folder called ./`ipstatic`. The dynamic files that vary depending on the specifics of an IP customization are located in the ./`ip` folder.

TIP: *In addition to exporting all the IP in a Manage IP project, you can use the steps outlined in Exporting IP from the Xilinx Catalog and Block Designs, page 110 to generate scripts for individual IP in the project.*

Running Third-Party Simulators in Batch Mode

With the design files gathered, and the scripts generated to support your target simulator, you can inspect the scripts and incorporate them into your verification environment. Xilinx recommends that you use the `export_simulation` scripts as a starting point for your simulation flow rather than building a custom API to generate scripts.

Make sure you have the environment setup correctly for the simulator before running the scripts. See Preparing for Simulation Using Third-Party Tools, page 137. Refer to the User Guide of your specific simulator for the details of running batch or scripted mode.

Running the Vivado Simulator in Batch Mode

To run in batch or scripted mode, the Vivado simulator relies on three processes which are supported by the files generated by the `export_simulation` command.

- Parsing Design Files, xvhdl and xvlog.
- Elaborating and Generating a Design Snapshot, xelab.
- Simulating the Design Snapshot, xsim.

For timing simulation, there are additional steps and data required to complete the simulation, as described in the following:

- Generating a Timing Netlist in Chapter 2
- Running Post-Synthesis and Post-Implementation Simulations, page 131

Parsing Design Files, xvhdl and xvlog

The `xvhdl` and `xvlog` commands parse VHDL and Verilog files, respectively. Descriptions for each option are available in Table 7-2, page 120.

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>]
[-nolog]
[-noname_unamed_generate]
[-relax]
[-prj <filename>]
[-sourcelibdir <sourcelib_dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-sv]
[-v [verbose] [0|1|2]]
[-version]
[-work <library_name> [=<library_dir>]
```

xvlog Examples

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

Note: xelab, xvlog and xvhdl are not Tcl commands. The xvlog, xvhdl, xelab are Vivado-independent compiler executables. Hence, there is no Tcl command for them.

The simulation launching is Vivado dependent and hence is done through xsim Tcl command.

For usage of simulation outside Vivado, an executable by the same name as xsim is provided. The xsim executable launches Vivado in project less mode and executes xsim Tcl command to launch simulation. Hence, to get help on xvlog, xvhdl, xelab form within Vivado IDE, please precede the command with exec.

```
Example: exec xvlog -help.
```

To get help on xsim, use xsim –help.

Elaborating and Generating a Design Snapshot, xelab

Simulation with the Vivado simulator happens in two phases:

- In the first phase, the simulator compiler xelab, compiles your HDL model into a snapshot, which is a representation of the model in a form that the simulator can execute.
- In the second phase, the simulator loads and executes (using the `xsim` command) the snapshot to simulate the model. In Non-Project Mode, you can reuse the snapshot by skipping the first phase and repeating the second.

When the simulator creates a snapshot, it assigns the snapshot a name based on the names of the top modules in the model. You can, however, override the default by specifying a snapshot name as an option to the compiler. Snapshot names must be unique in a directory or *SIMSET*; reusing a snapshot name, whether default or custom, results in overwriting a previously-built snapshot with that name.

IMPORTANT: *you cannot run two simulations with the same snapshot name in the same directory or SIMSET.*

xelab

The `xelab` command, for given top-level units, does the following:

- Loads children design units using language binding rules or the `–L <library>` command line specified HDL libraries
- Performs a static elaboration of the design (sets parameters, generics, puts generate statements into effect, and so forth)
- Generates executable code
- Links the generated executable code with the simulation kernel library to create an executable simulation snapshot

You then use the produced executable simulation snapshot name as an option to the `xsim` command along with other options to effect HDL simulation.

TIP: `xelab` *can implicitly call the parsing commands,* `xvlog` *and* `xvhdl`*. You can incorporate the parsing step by using the* `xelab -prj` *option. See Project File (.prj) Syntax, page 128 for more information about project files.*

Note: xelab, xvlog and xvhdl are not Tcl commands. The xvlog, xvhdl, xelab are Vivado-independent compiler executables. Hence, there is no Tcl command for them.

xelab Command Syntax Options

Descriptions for each option are available in Table 7-2, page 120.

```
xelab
[-d [define] <name>[=<val>]
[-debug <kind>]
[-f [-file] <filename>]
[-generic_top <value>]
[-h [-help]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-log <filename>]
[-L [-lib] <library_name> [=<library_dir>]
[-maxdesigndepth arg]
[-mindelay]
[-typdelay]
[-maxarraysize arg]
[-maxdelay]
[-mt arg]
[-nolog]
[-noname_unnamed_generate]
[-notimingchecks]
[-nosdfinterconnectdelays]
[-nospecify]
[-O arg]
[-Odisable_acceleration arg]
[-Odisable_always_combine]
[-Odisable_pass_through_elimination]
[-Odisable_process_opt]
[-Odisable_unused_removal]
[-Oenable_cdfg]
[-Odisable_cdfg]
[-Oenable_always_combine]
[-Oenable_pass_through_elimination]
[-Oenable_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>]
```

```
[-stats]
[-timescale]
[-timeprecision_vhdl arg]
[-transport_int_delays]
[-v [verbose] [0|1|2]]
[-version]
[-sv_root arg]
[-sv_lib arg]
[-sv_liblist arg]
[-dpiheader arg]
[-driver_display_limit arg]
[-dpi_absolute]
```

xelab Examples

```
xelab work.top1 work.top2 -s cpusim
xelab lib1.top1 lib2.top2 -s fftsim
xelab work.top1 work.top2 -prj files.prj -s pciesim
xelab lib1.top1 lib2.top2 -prj files.prj -s ethernetsim
```

Verilog Search Order

The `xelab` command uses the following search order to search and bind instantiated Verilog design units:

1. A library specified by the `‘uselib` directive in the Verilog code. For example:

```
module
full_adder(c_in, c_out, a, b, sum)
input c_in,a,b;
output c_out,sum;
wire carry1,carry2,sum1;
`uselib lib = adder_lib
half_adder adder1(.a(a),.b(b),.c(carry1),.s(sum1));
half_adder adder1(.a(sum1),.b(c_in),.c(carry2),.s(sum));
c_out = carry1 | carry2;
endmodule
```

2. Libraries specified on the command line with `-lib|-L` switch.
3. A library of the parent design unit.
4. The `work` library.

Verilog Instantiation Unit

When a Verilog design instantiates a component, the xelab command treats the component name as a Verilog unit and searches for a Verilog module in the user-specified list of unified logical libraries in the user-specified order.

- If found, xelab binds the unit and the search stops.
- If the case-sensitive search is not successful, xelab performs a case-insensitive search for a VHDL design unit name constructed as an extended identifier in the user-specified list and order of unified logical libraries, selects the first one matching name, then stops the search.
- If xelab finds a unique binding for any one library, it selects that name and stops the search.

Note: For a mixed language design, the port names used in a named association to a VHDL entity instantiated by a Verilog module are always treated as case insensitive. Also note that you cannot use a defparam statement to modify a VHDL generic. See 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 7-1: **‘uselib Examples**

File half_adder.v compiled into logical library named adder_lib	File full_adder.v compiled into logical library named work

module half_adder(a,b,c,s);
input a,b;
output c,s;
s = a ^ b;
c = a & b;
endmodule
	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

xelab, xvhdl, and xvlog xsim Command Options

Table 7-2 lists the command options for the xelab, xvhdl, and xvlog xsim commands.

Table 7-2: **xelab, xvhd, and xvlog Command Options**

Command Option	Description	Used by Command
-d [define] <name>[=<val>]	Define Verilog macros. Use -d|--define for each Verilog macro. The format of the macro is <name>[=<val>] where <name> is name of the macro and <value> is an optional value of the macro.	xelab xvlog
-debug <kind>	Compile with specified debugging ability turned on. The <kind> options are: • typical: Most commonly used abilities, including: line and wave. • line: HDL breakpoint. • wave: Waveform generation, conditional execution, force value. • xlibs: Visibility into Xilinx® precompiled libraries. This option is only available on the command line. • off: Turn off all debugging abilities (Default). • all: Uses all the debug options.	xelab
-encryptdumps	Encrypt parsed dump of design units being compiled.	xvhdl xvlog

Table 7-2: **xelab, xvhd, and xvlog Command Options *(Cont'd)***

Command Option	Description	Used by Command
`-f [-file] <filename>`	Read additional options from the specified file.	xelab xsim xvhdl xvlog
`-generic_top <value>`	Override generic or parameter of a top-level design unit with specified value. Example: `-generic_top "P1=10"`	xelab
`-h [-help]`	Print this help message.	xelab xsim xvhdl xvlog
`-i [include] <directory_name>`	Specify directories to be searched for files included using Verilog `` `include``. Use `-i|--include` for each specified search directory.	xelab xvlog
`-initfile <init_filename>`	User-defined simulator initialization file to add to or override settings provided by the default `xsim.ini` file.	xelab xvhdl xvlog
`-L [-lib] <library_name> [=<library_dir>]`	Specify search libraries for the instantiated non-VHDL design units; for example, a Verilog design unit. Use `-L|--lib` for each search library. The format of the argument is `<name>[=<dir>]` where `<name>` is the logical name of the library and `<library_dir>` is an optional physical directory of the library.	xelab xvhdl xvlog
`-log <filename>`	Specify the log file name. Default: `<xvlog|xvhdl|xelab|xsim>.log`.	xelab xsim xvhdl xvlog
`-maxarraysize arg`	Set maximum vhdl array size to be 2**n (Default: n = 28, which is 2**28)	xelab
`-maxdelay`	Compile Verilog design units with maximum delays.	xelab
`-maxdesigndepth arg`	Override maximum design hierarchy depth allowed by the elaborator (Default: 5000).	xelab
`-maxlogsize arg (=-1)`	Set the maximum size a log file can reach in MB. The default setting is unlimited.	xsim
`-mindelay`	Compile Verilog design units with minimum delays.	xelab

Table 7-2: **xelab, xvhd, and xvlog Command Options *(Cont'd)***

Command Option	Description	Used by Command
`-mt arg`	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`). Advanced usage: to further control the `-mt` option, you can set the Tcl property as follows: `set_property XELAB.MT_LEVEL off|N [get_filesets sim_1]`	xelab
`-nolog`	Suppress log file generation.	xelab xsim xvhdl xvlog
`-noieeewarnings`	Disable warnings from VHDL IEEE functions.	xelab
`-noname_unnamed_generate`	Do not generate name for an unnamed generate block.	xelab xvlog
`-notimingchecks`	Ignore timing check constructs in Verilog specify block(s).	xelab
`-nosdfinterconnectdelays`	Ignore SDF port and interconnect delay constructs in SDF.	xelab
`-nospecify`	Ignore Verilog path delays and timing checks.	xelab
`-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
`-Odisable_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
`-Odisable_process_opt`	Turn off the process-level optimization (default on)	xelab
`-Oenable_cdfg` `-Odisable_cdfg`	Turn on (enable) or off (disable) the building of the control+data flow graph (default: on)	xelab
`-Oenable_unused_removal` `-Odisable_unused_removal`	Turn on (enable or off (disable) the optimization to remove unused signals and statements (default: on)	xelab

Table 7-2: **xelab, xvhd, and xvlog Command Options *(Cont'd)***

Command Option	Description	Used by Command
`-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
`-prj <filename>`	Specify the Vivado simulator project file containing one or more entries of `vhdl|verilog <work lib> <HDL file name>`.	xelab xvhdl xvlog
`-r [-run]`	Run the generated executable snapshot in command-line interactive mode.	xelab
`-rangecheck`	Enable run time value range check for VHDL.	xelab
`-R [-runall`	Run the generated executable snapshot until the end of simulation.	xelab xsim
`-relax`	Relax strict language rules.	xelab xvhdl xvlog
`-s [-snapshot] arg`	Specify the name of output simulation snapshot. Default is `<worklib>.<unit>`; for example: `work.top`. Additional unit names are concatenated using #; for example: `work.t1#work.t2`.	xelab
`-sdfnowarn`	Do not emit SDF warnings.	xelab
`-sdfnoerror`	Treat errors found in SDF file as warning.	xelab
`-sdfmin arg`	`<root=file>` SDF annotate `<file>` at `<root>` with minimum delay.	xelab
`-sdftyp arg`	`<root=file>` SDF annotate `<file>` at `<root>` with typical delay.	xelab

Table 7-2: **xelab, xvhd, and xvlog Command Options *(Cont'd)***

Command Option	Description	Used by Command
`-sdfmax arg`	`<root=file>` SDF annotate `<file>` at `<root>` with maximum delay.	xelab
`-sdfroot <root_path>`	Default design hierarchy at which SDF annotation is applied.	xelab
`-sourcelibdir <sourcelib_dirname>`	Directory for Verilog source files of uncompiled modules. Use `-sourcelibdir <sourcelib_dirname>` for each source directory.	xelab xvlog
`-sourcelibext <file_extension>`	File extension for Verilog source files of uncompiled modules. Use -`sourcelibext <file extension>` for source file extension	xelab xvlog
`-sourcelibfile <filename>`	File name of a Verilog source file with uncompiled modules. Use `-sourcelibfile <filename>`.	xelab xvlog
`-stat`	Print tool CPU and memory usages, and design statistics.	xelab
`-sv`	Compile input files in System Verilog mode.	xvlog
`-timescale`	Specify default timescale for Verilog modules. Default: `1ns/1ps`.	xelab
`-timeprecision_vhdl arg`	Specify time precision for vhdl designs. Default: `1ps`.	xelab
`-transport_int_delays`	Use transport model for interconnect delays.	xelab
`-typdelay`	Compile Verilog design units with typical delays (Default).	xelab
`-v [verbose] [0|1|2]`	Specify verbosity level for printing messages. Default = 0.	xelab xvhdl xvlog
`-version`	Print the compiler version to screen.	xelab xsim xvhdl xvlog
`-work <library_name> [=<library_dir>]`	Specify the work library. The format of the argument is `<name>[=<dir>]` where: • `<name>` is the logical name of the library. • `<library_dir>` is an optional physical directory of the library.	xvhdl xvlog
`-sv_root arg`	Root directory off which DPI libraries are to be found. Default: <current_directory/xsim.dir/xsc>	xelab
`-sv_lib arg`	Shared library name for DPI imported functions (.dll/.so) without the file extension.	xelab

Table 7-2: **xelab, xvhd, and xvlog Command Options *(Cont'd)***

Command Option	Description	Used by Command
`-sv_liblist arg`	Bootstrap file pointing to DPI shared libraries.	xelab
`-dpiheader arg`	Header filename for the exported and imported functions.	xelab
`-driver_display_limit arg`	Enable driver debugging for signals with maximum size (Default: n = 65536)	xelab
`-dpi_absolute`	Use absolute paths instead of `LD_LIBRARY_PATH` on Linux for DPI libraries that are formatted as lib<libname>.so	xelab

Simulating the Design Snapshot, xsim

The `xsim` command loads a simulation snapshot to effect a batch mode simulation or provides a workspace (GUI) and/or a Tcl-based interactive simulation environment.

xsim Executable Syntax

The command syntax is as follows:

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

Where:

- `xsim` is the command.
- `<options>` are the options specified in Table 7-3.
- `<snapshot>` is the simulation snapshot.

xsim Executable Options

Table 7-3: **xsim Executable Command Options**

xsim Option	Description
`-f [-file] <filename>`	Load the command line options from a file.
`-g [-gui]`	Run with interactive workspace.
`-h [-help]`	Print help message to screen.
`-log <filename>`	Specify the log file name.
`-maxdeltaid arg (=-1)`	Specify the maximum delta number. Report an error if it exceeds maximum simulation loops at the same time.
`-maxlogsize arg (=-1)`	Set the maximum size a log file can reach in MB. The default setting is unlimited.
`-ieeewarnings`	Enable warnings from VHDL IEEE functions.

Table 7-3: **xsim Executable Command Options *(Cont'd)***

xsim Option	Description
`-nolog`	Suppresses log file generation.
`-nosignalhandlers`	Disables the installation of OS-level signal handlers in the simulation. For performance reasons, the simulator does not check explicitly for certain conditions, such as an integer division by zero, that could generate an OS-level fatal run time error. Instead, the simulator installs signal handlers to catch those errors and generates a report. With the signal handlers disabled, the simulator can run in the presence of such security software, but OS-level fatal errors could crash the simulation abruptly with little indication of the nature of the failure. ⚠ **CAUTION!** Use this option only if your security software prevents the simulator from running successfully.
`-onfinish <quit|stop>`	Specify the behavior at end of simulation.
`-onerror <quit|stop>`	Specify the behavior upon simulation run time error.
`-R [-runall]`	Runs simulation till end (such as `do 'run all;quit'`).
`-stats`	Display memory and CPU stats upon exiting.
`-testplusarg <arg>`	Specify `plusargs` to be used by `$test$plusargs` and `$value$plusargs` system functions.
`-t [-tclbatch] <filename>`	Specify the Tcl file for batch mode execution.
`-tp`	Enable printing to screen of hierarchical names of process being executed.
`-tl`	Enable printing to screen of file name and line number of statements being executed.
`-wdb <filename.wdb>`	Specify the waveform database output file.
`-version`	Print the compiler version to screen.
`-view <wavefile.wcfg>`	Open a wave configuration file. Use this switch together with `-gui` switch.

TIP: *When running the xelab, xsc, xsim, xvhdl, 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.)*

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 7-4: **File Types and Associated Commands for Design File Analysis**

File Type	Command
Verilog	`xvlog  <VerilogFileName(s)>`
SystemVerilog	`xvlog -sv <SystemVerlilogFileName(s)>`
VHDL	`xvhdl <VhdlFileName(s)>`

Step2: Elaborating and Creating a Snapshot

After analysis, elaborate the design and create a snapshot for simulation using the `xelab` command:

```
xelab <topDesignUnitName> -debug typical
```

IMPORTANT: *You can provide multiple top-level design unit names with* `xelab`*. To use the Vivado simulator workspace for purposes similar to those used during* `launch_simulation`*, you must set debug level to* `typical`*.*

Step 3: Running Simulation

After successful completion of the `xelab` phase, the Vivado simulator creates a snapshot used for running simulation.

To invoke the Vivado simulator workspace, use the following command:

```
xsim  <SnapShotName> -gui
```

To open the wave config file:

```
xsim <SnapShotName> -view <wcfg FileName> -gui
```

You can provide multiple wcfg files using multiple –view flags. For example:

```
xsim  <SnapShotName> -view <wcfg FileName> -view <wcfg FileName>
```

Project File (.prj) Syntax

Note: The project file discussed here is a Vivado simulator text-based project file. It is not the same as the project file (.xpr) created by the Vivado Design Suite.

To parse design files using a project file, create a text file called <proj_name>.prj, and use the syntax shown below inside the project file.

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

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:

```
`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:

```
<logical_library1> = <physical_dir_path1>
<logical_library2> = <physical_dir_path2>
```

The following is an example `xsim.ini` file:

```
std=<Vivado_Install_Area>/xsim/vhdl/std
ieee=<Vivado_Install_Area>/xsim/vhdl/ieee
vl=<Vivado_Install_Area>/xsim/vhdl/vl
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 7-4 illustrates the behavioral simulation process:

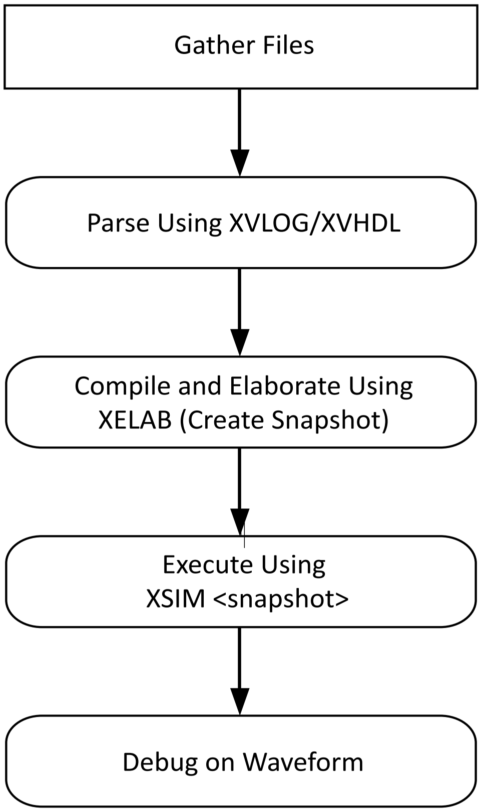

Gather Files
Parse Using XVLOG/XVHDL
Compile and Elaborate Using XELAB (Create Snapshot)
Execute Using XSIM <snapshot>
Debug on Waveform

Figure 7-4: **Behavioral Simulation Process**

To run behavioral simulation from within the Vivado Design Suite, use the Tcl command: `launch_simulation -mode behavioral`.

Running Post-Synthesis and Post-Implementation Simulations

At post-synthesis and post-implementation, you can run a functional or a Verilog timing simulation. Figure 7-5 illustrates the post-synthesis and post-implementation simulation process:

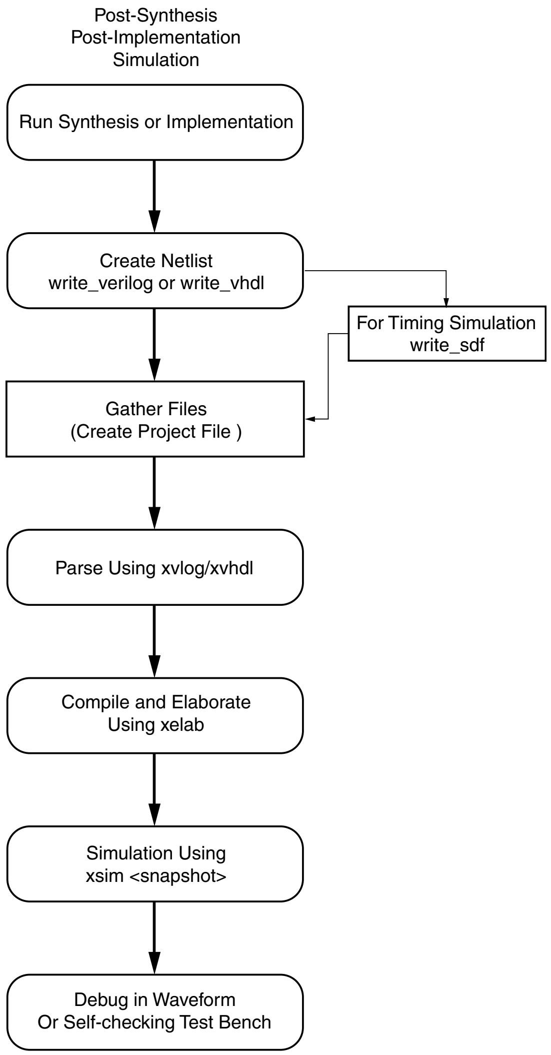


Figure 7-5: **Post-Synthesis and Post-Implementation Simulation**

The following is an example of running a post-synthesis functional simulation from the command line:

```
synth_design -top top -part xc7k70tfbg676-2
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
launch_simulation
```

TIP: *When you run a post-synthesis or post-implementation timing simulation, you must run the* `write_sdf` *command after the* `write_verilog` *command, and the appropriate annotate command is needed for elaboration and simulation.*

Using Tcl Commands and Scripts

You can run Tcl commands on the Tcl Console individually, or batch the commands into a Tcl script to run simulation.

Using a -tclbatch File

You can type simulation commands into a Tcl file, and reference the Tcl file with the following command: `-tclbatch <filename>`

Use the `-tclbatch` option to contain commands within a file and execute those command as simulation starts. For example, you can have a file named `run.tcl` that contains the following:

```
run 20ns
current_time
quit
```

Then launch simulation as follows:

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

You can set a variable to represent a simulation command to quickly run frequently used simulation commands.

Launching Vivado Simulator from the Tcl Console

The following is an example of Tcl commands that create a project, read in source files launch the Vivado simulator, do placing and routing, write out an SDF file, and re-launch simulation.

```
Vivado -mode Tcl
Vivado% create_project prj1
Vivado% read_verilog dut.v
Vivado% synth_design -top dut
Vivado% launch_simulation -simset sim_1 -mode post-synthesis -type functional
Vivado% place_design
Vivado% route_design
Vivado% write_verilog -mode timesim -sdf_anno true -sdf_file postRoute.sdf
postRoute_netlist.v
Vivado% write_sdf postRoute.sdf
Vivado% launch_simulation -simset sim_1 -mode post-implementation -type timing
Vivado% close_project
```

export_simulation

```
export_simulation  [-simulator <arg>] [-language <arg>] [-of_objects <arg>]
[-lib_map_path <arg>] [-script_name <arg>] [-directory <arg>] [-runtime <arg>]
[-absolute_path] [-single_step] [-ip_netlist] [-export_source_files] [-32bit]
[-force] [-quiet] [-verbose]

Usage:
```

Table 7-5: **Export Simulation**

Name	Description
[-simulator]	Simulator for which the simulation script will be created value=all|xsim|modelsim|questa|ies|vcs)(Default: all)
[-language]	simulator language(value=mixed|verilog|vhdl)(Default: mixed)
[-of_objects]	Export simulation script for the specified object (Default: None)
[-lib_map_path]	Precompiled simulation library directory path. If not specified, then please follow the instructions in the generated script header to manually provide the simulation library mapping information.(Default: Empty)
[-script_name]	Output shell script filename. If not specified, then file with a default name will be created.(Default: top_module.sh/.bat)
[-directory]	Directory where the simulation script will be exported (Default: export_sim)
[-runtime]	Run simulation for this time (default: full simulation run or until a logical break or finish condition) Default: Empty
[-absolute_path]	Make all file paths absolute wrt the reference directory
[-single_step]	Generate script to launch all steps in one step
[-ip_netlist]	Select the netlist files for IP(s) in the project or the selected object for the specified simulator language (default: verilog)
[-export_source_files]	Copy design files to output directory
[-32bit]	Perform 32bit compilation
[-force]	Overwrite previous files
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

```
simulation, xilinxtclstore, user-written
```

Description:

Export a simulation script file for the target simulator (please see the list of supported simulators below). The generated script will contain simulator commands for compiling, elaborating and simulating the design.

The command will retrieve the simulation compile order of specified objects, and export this information in a shell script with the compiler commands and default options for the target simulator. The specified object can be either a simulation fileset or an IP.

If the object is not specified, then this command will generate the script for the active simulation `top`. Any verilog include directories or file paths for the files containing verilog define statements will be added to the compiler command line.

By default, the design source file and include directory paths in the compiler command line will be set relative to the `reference_dir` variable that is set in the generated script. To make these paths absolute, specify the `-absolute_path` switch.

The command will also copy data files (if any) from the fileset, or from an IP, to the output directory. If the design contains Verilog sources, then the generated script will also copy the `glbl.v` file from the software installation path to the output directory.

A default`.do` file that is used in the compiler commands in the simulation script for the target simulator, will be written to the output directory.

Note: In order to perform simulation with the generated script, the simulation libraries must be compiled first using the `compile_simlib` Tcl command with the compiled library directory path specified, when generating this script. The generated script will automatically include the setup files for the target simulator from the compiled library directory.

The option provided in Simulation Settings will not have any impact on `export_simulation` scripts.

Supported simulators

```
Vivado Simulator (xsim)
ModelSim Simulator (modelsim)
Questa Advanced Simulator (questa)
Incisive Enterprise Simulator (ies)
Verilog Compiler Simulator (vcs)
```

Arguments

```
-of_objects - (Optional) Specify the target object for which the simulation script
file needs to be generated. The target object can be either a simulation fileset
(simset) or an IP. If this option is not specified then this command will generate
file for the current simulation fileset.

-lib_map_path - (Optional) Specify the pre-compiled simulation library directory
path where the Xilinx simulation libraries are compiled. Please see the header
section in the generated script for more information.
```

-script_name - (Optional) Specify the name of the shell script. If this option is not specified then the filename with the following syntax will be generated based on the object type selected with -of_objects switch:

```
   <simulation_top_name>_sim_<simulator>.sh
   <ip_name>_sim_<simulator>.sh
```

-absolute_path - (Optional) Specify this option to make source and include directory paths used in the script absolute. By default, all paths are written as relative to the directory path that is specified with the -directory switch. A reference_dir variable will be set in the script to the directory path that is specified with the -directory switch.

-32bit - (Optional) Specify this option to perform 32bit simulation. If this option is not specified then by default 64bit option will be added to the simulation command line.

-force - (Optional) Overwrite an existing script file of the same name. If the script file already exists, the tool returns an error unless the -force argument is specified.

-directory - (Required) Specify the directory path where the script file will be exported.

-simulator - (Required) Specify the target simulator name for the simulation script. The valid simulators names are xsim, modelsim, questa, ies, and vcs (or vcs_mx).

-language - (Optional) Select the simulator language for which the IP netlist will be referred in the generated script. By default, the language is 'mixed'. Other values are 'vhdl' and 'verilog'.

-ip_netlist - (Optional) Use the netlist file for the IP(s) in the generated script. By default, if this switch is specified, the verilog netlist will be referenced in the script. To override this netlist language type, use the -language switch in conjunction with -ip_netlist switch. By default, the simulator language value from the project settings will be selected if -ip_netlist is specified.

export_ip_user_files

Generate and export IP/IPI user files from a project. This can be scoped to work on one or more IPs.

```
Syntax:
export_ip_user_files  [-of_objects <arg>] [-ip_user_files_dir <arg>]
                      [-ipstatic_source_dir <arg>] [-no_script] [-sync]
                      [-force] [-quiet] [-verbose]
```

Returns: list of files that were exported

Usage

Table 7-6: **export_ip_user_files**

Name	Description
`[-of_objects]`	`IP,IPI or a fileset (Default: None)`
`[-ip_user_files_dir]`	`Directory path to simulation base directory (for dynamic and other IP non static files)(Default: None)`
`[-ipstatic_source_dir]`	`Directory path to the static IP files (Default: None)`
`[-no_script]`	`Do not export simulation scripts (Default: 1)`
`[-sync]`	`Delete IP/IPI dynamic and simulation script files`
`[-force]`	`Overwrite files`
`[-quiet]`	`Ignore command errors`
`[-verbose]`	`Suspend message limits during command execution`

Description

Export IP user files repository with static, dynamic, netlist, verilog/vhdl stubs and memory initialization files.

Arguments

```
-of_objects - (Optional) Specify the target object for which the IP static and
dynamic files needs to be exported.

-ip_user_files_dir - (Optional) Directory path to IP user files base directory (for
dynamic and other IP non static files). By default, if this switch is not specified
then this command will use the path specified with the IP.USER_FILES_DIR project
property value.

-ipstatic_source_dir - (Optional) Directory path to the static IP files. By default,
if this switch is not specified then this command will use the path specified with
the SIM.IPSTATIC_SOURCE_DIR project property value.
```

Note: If the -ip_user_files_dir switch is specified, by default the IP static files will be exported under the sub-directory with the name `ipstatic`. If this switch is specified in conjunction with `-ipstatic_source_dir`, then the IP static files will be exported in the path specified with the `-ipstatic_source_dir` switch.

```
-clean_dir - (Optional) Delete all files from central directory (including static,
dynamic and other files)

Examples:
The following command will export, ‘char_fifo’ IP dynamic files to
'<project>/<project>.ip_user_files/ip/char_fifo' directory 'char_fifo' IP static
files to '<project>/<project>.ip_user_files/ipstatic' directory

% export_ip_user_files -of_objects [get_ips char_fifo]
```

Chapter 8

Using Third-Party Simulators

Introduction

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

The following third-party tools are supported:

- QuestaSim
- ModelSim (PE and DE)
- IES
- VCS
- Riviera PRO simulator (Aldec)

Note: See Aldec simulator documentation for using Active-HDL/Riviera PRO Simulators[Ref 14].

IMPORTANT: Use only supported versions of third-party simulators. For more information on supported Simulators and Operating Systems, see the Compatible Third-Party Tools table in the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) [Ref 1].

The *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 4] describes the use of the Vivado IDE.

For links to more information on your third party simulator see [Ref 14].

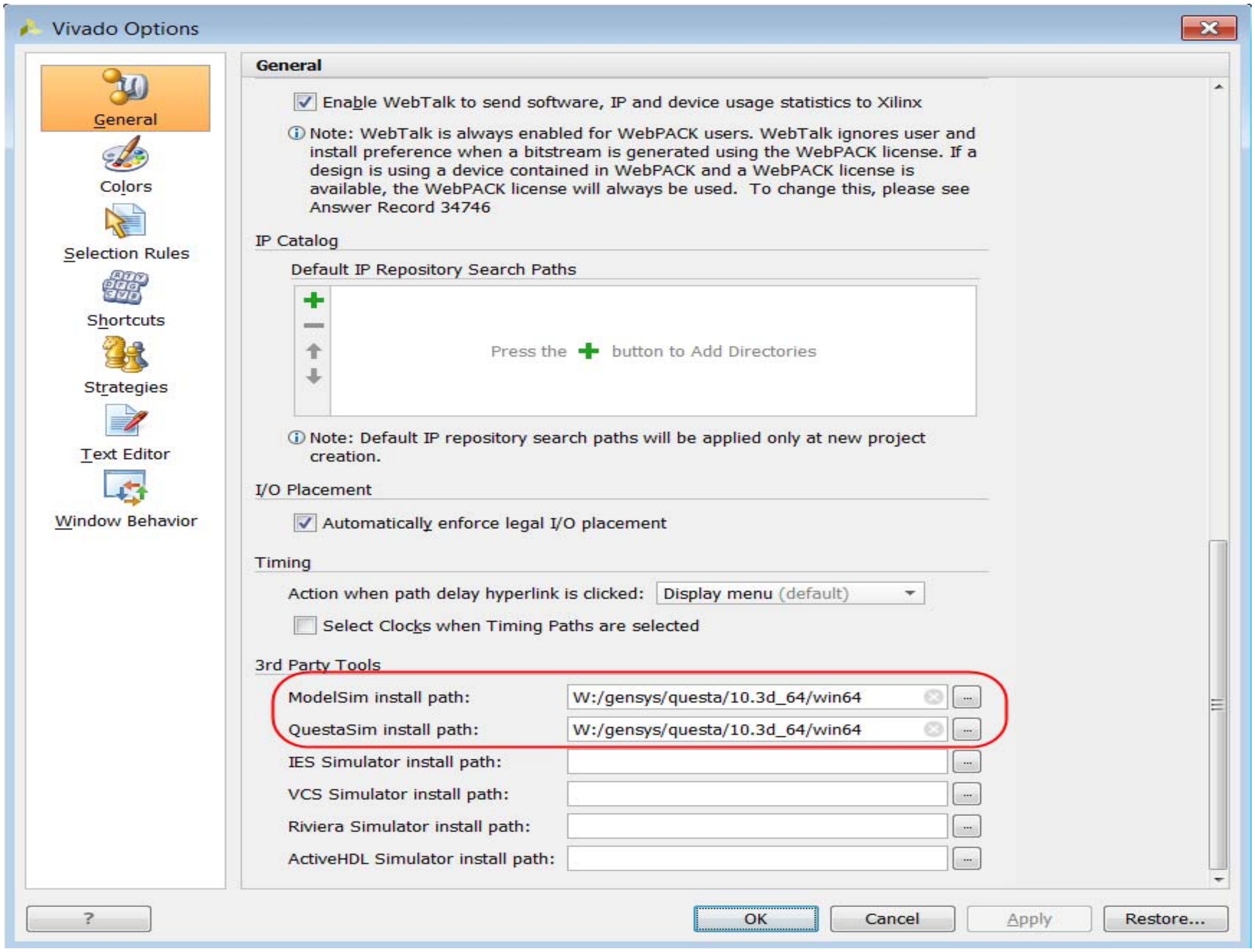
Preparing for Simulation Using Third-Party Tools

Pointing to the Simulator Install Location

To define the installation path:

1. Select **Tools > Options > General.**

2. In the Vivado Options, General dialog box, *scroll down* to the appropriate **install path** field, shown in the figure below, and browse to the installation path.



General

☑ Enable WebTalk to send software, IP and device usage statistics to Xilinx

ⓘ Note: WebTalk is always enabled for WebPACK users. WebTalk ignores user and install preference when a bitstream is generated using the WebPACK license. If a design is using a device contained in WebPACK and a WebPACK license is available, the WebPACK license will always be used. To change this, please see Answer Record 34746

IP Catalog

Default IP Repository Search Paths

Press the ✚ button to Add Directories

ⓘ Note: Default IP repository search paths will be applied only at new project creation.

I/O Placement

☑ Automatically enforce legal I/O placement

Timing

Action when path delay hyperlink is clicked: Display menu (default)

☐ Select Clocks when Timing Paths are selected

3rd Party Tools

ModelSim install path: W:/gensys/questa/10.3d_64/win64

QuestaSim install path: W:/gensys/questa/10.3d_64/win64

IES Simulator install path:

VCS Simulator install path:

Riviera Simulator install path:

ActiveHDL Simulator install path:

Figure 8-1: **Vivado Design Suite General Options, Install Path**

Compiling Simulation Libraries

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

Compilation of the libraries is typically a one-time operation, as long as you are using the same version of the tools.

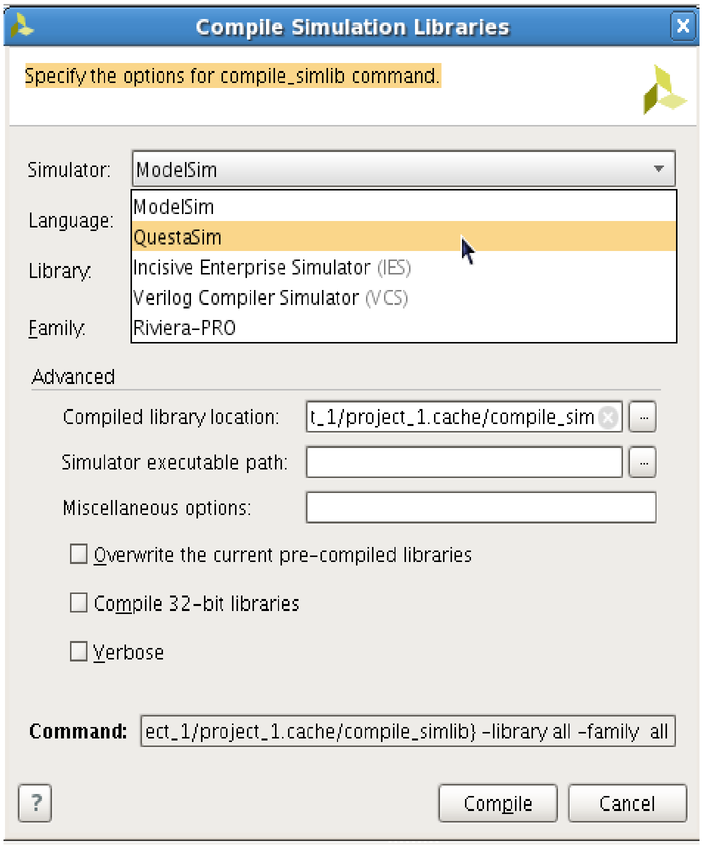
IMPORTANT: *Any change to the Vivado tools or the simulator versions requires that libraries be recompiled.*

Before you begin simulation, run the `compile_simlib` Tcl command to compile the Xilinx® simulation libraries for the target simulator.

Compiling Simulation Libraries Using Vivado IDE

Whenever you change the third party tool, you must recompile the simulation libraries.

1. In the **Tools**, click **Compile Simulation Libraries** to open the dialog box shown in Figure 8-2.



Compile Simulation Libraries

Specify the options for compile_simlib command.

Simulator: ModelSim

Compiled library location: t_1/project_1.cache/compile_sim

Simulator executable path:

Miscellaneous options:

□ Overwrite the current pre-compiled libraries

□ Compile 32-bit libraries

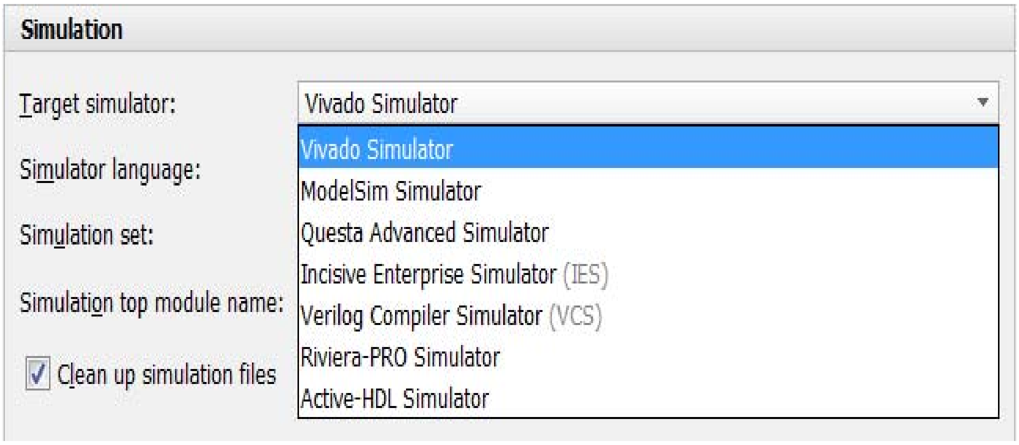
□ Verbose

Command: ect_1/project_1.cache/compile_simlib} -library all -family all

Figure 8-2: **Compile Simulation Libraries Dialog Box**

Dialog Box Options

Simulator: From the Simulator drop-down menu, select a simulator, as shown in Figure 8-3.



Simulation

Target simulator: Vivado Simulator

Simulator language:

Simulation set:

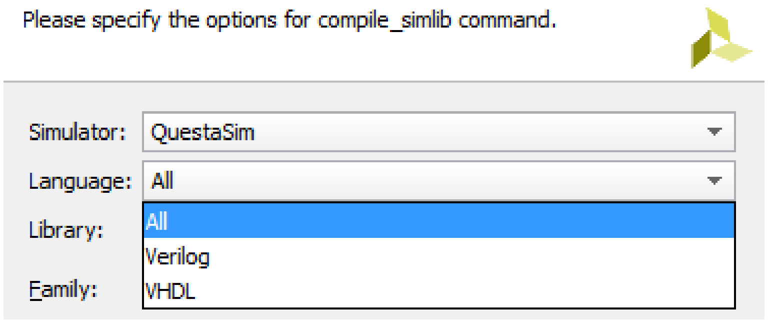
Simulation top module name:

☑ Clean up simulation files

Vivado Simulator
ModelSim Simulator
Questa Advanced Simulator
Incisive Enterprise Simulator (IES)
Verilog Compiler Simulator (VCS)
Riviera-PRO Simulator
Active-HDL Simulator

Figure 8-3: **Simulator Drop-Down Selections**

- **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 8-4.



Please specify the options for compile_simlib command.

Simulator: QuestaSim

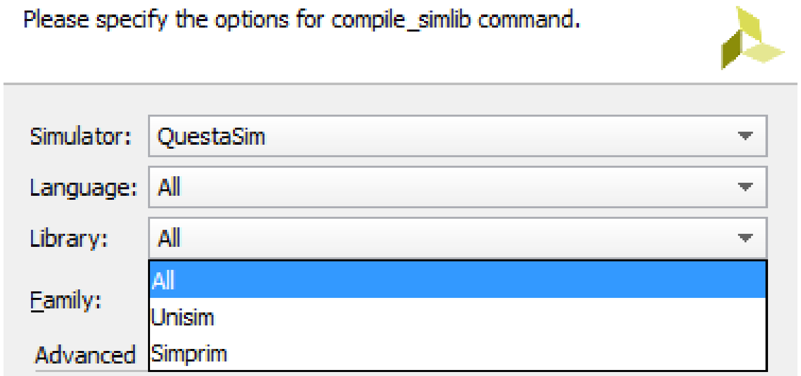
Language: All

Library:

Family:

Figure 8-4: **Language Selection**

- **Library:** Specifies the simulation library to compile. By default, the `compile_simlib` command compiles all simulation libraries. See Figure 8-5.



Please specify the options for compile_simlib command.

Simulator: QuestaSim

Language: All

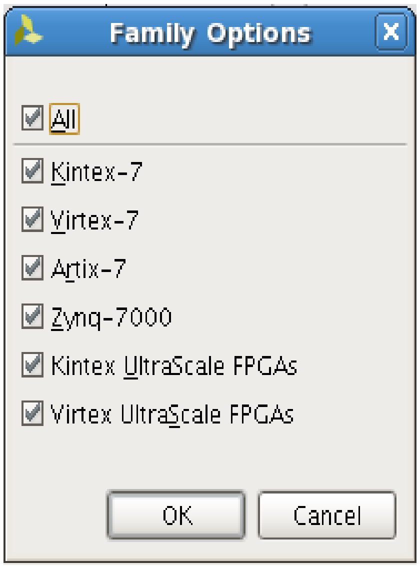
Library: All

Family:

Advanced

Figure 8-5: **Simulation Libraries**

- **Family:** Compiles selected libraries to the specified device family. All device families are generated by default. See Figure 8-6.



☑ All

☑ Kintex-7

☑ Virtex-7

☑ Artix-7

☑ Zynq-7000

☑ Kintex UltraScale FPGAs

☑ Virtex UltraScale FPGAs

Figure 8-6: **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 12] 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 8-2, 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:

```
compile_simlib [-directory <arg>] [-family <arg>] [-force] [-language <arg>]
               [-library <arg>] [-print_library_info <arg>] -simulator <arg>
               [-simulator_exec_path <arg>] [-source_library_path <arg>]
               [-32bit] [-quiet] [-verbose]
```

Table 8-1: **Tcl Mode Options**

Name	Description
`[-directory]`	Directory path for saving the compiled results.
`[-family]`	Selects device architecture. Default: all
`[-force]`	Overwrites the pre-compiled libraries
`[-language]`	Compiles libraries for this language. Default: all
`[-library]`	Selects library to compile. Default: all
`[-print_library_info]`	Prints pre-compiled library information
`-simulator`	Compiles libraries for this simulator
`[-simulator_exec_path]`	Uses simulator executables from this directory
`[-source_library_path]`	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.
`[-32bit]`	Performs the 32-bit compilation
`[-quiet]`	Ignores command errors
`[-verbose]`	Suspends message limits during command execution

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.

  ```
  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`.

  ```
   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`.

  ```
  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>.

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

About the Compiled Libraries

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

The name of the initialization file varies depending on the simulator you are using, as shown in the table below.

Table 8-2: **Files the Simulator Uses to Reference the Compiled Libraries**

QuestaSim/ModelSim	IES	VCS
modelsim.ini	cds.lib	synopsys_sim.setup

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

Running Simulation with Third-Party Tools

The **Flow Navigator > Simulation Settings** section lets you configure the simulation settings in Vivado IDE. The Flow Navigator Simulation menu is shown in the figure below.

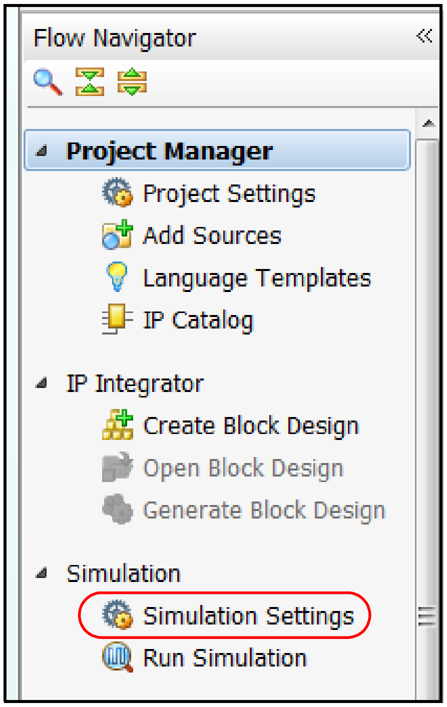


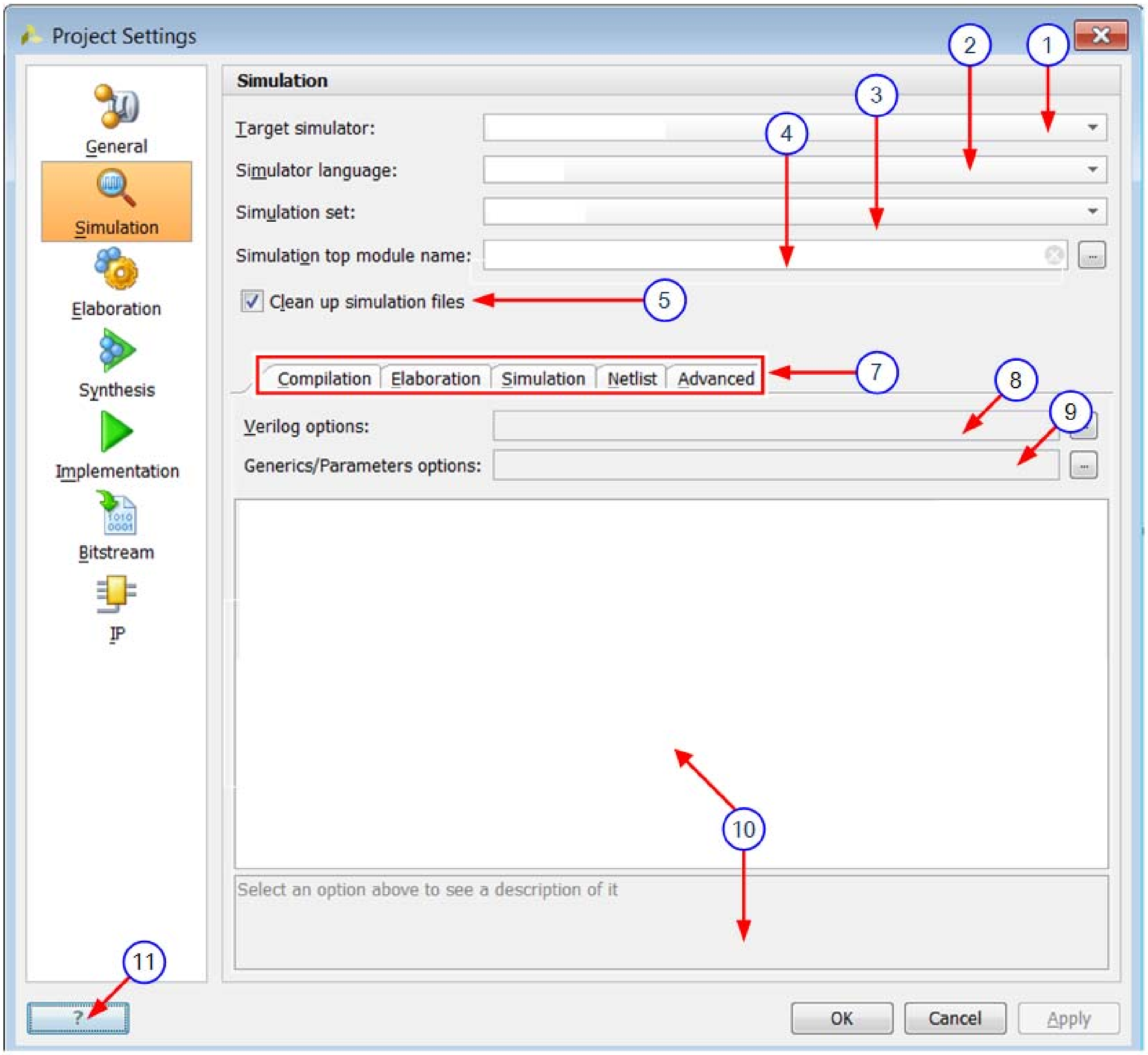
Figure 8-7: **Simulation Settings**

TIP: *This section describes how to run simulation interactively from within the Vivado IDE. For information on creating batch scripts to run simulation from the command line, refer to Chapter 7, Simulating in Batch or Scripted Mode.*

Simulation Settings opens the Simulation Settings dialog box where you can select and configure the simulator.

Selecting Simulation Options Using Third-Party Tools

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



☑ Clean up simulation files

Select an option above to see a description of it

X16599-040816

1 Selects the target simulator.
2 Selects the simulator language.
3 Selects the simulation set.
4 Browses to the simulation top-level design name.
5 Cleans simulation files before re-run. Keeping the option enabled is recommended.
6 Generates scripts without running simulation.
7 Select tabs to set options in the respective categories.
8 Specifies the location of the compiled simulation libraries.
9 Compilation tab only. Browse to set include path or to define macros.
10 Compilation tab only. Browse to select generics/parameters location.
11 For each tab, an option list appears in the window, and when selected, an option description displays.

Figure 8-8: **Project Settings Dialog Box Options: Third-Party Tools**

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

Adding or Creating Simulation Source Files

Simulation sources are contained in Simulation Sets and might contain a mix of design and simulation-only files. When adding files to a project, you can specify the simulation set and designate the association for simulation and implementation.

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).

     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.

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

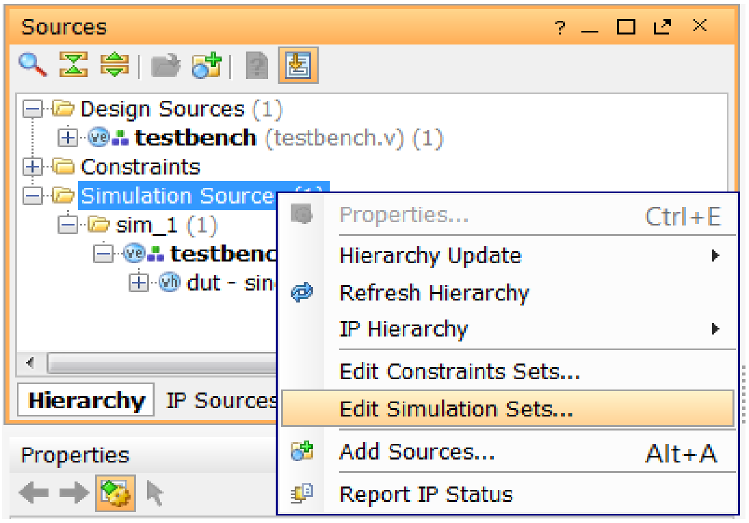


Figure 8-9: **Edit Simulation Sets Option**

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.*

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

Running Simulation Using the Vivado IDE and Third-Party Tools

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

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

From the Flow Navigator, click **Run Simulation**, as shown in Figure 8-10.

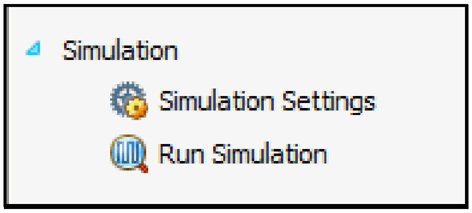


Figure 8-10: **Flow Navigator Simulation Options**

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

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

Running Simulation Using Third-Party Tools

To run simulation: In the Flow Navigator, select **Run Simulation** and choose the appropriate option from the popup menu shown in the figure below.

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

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

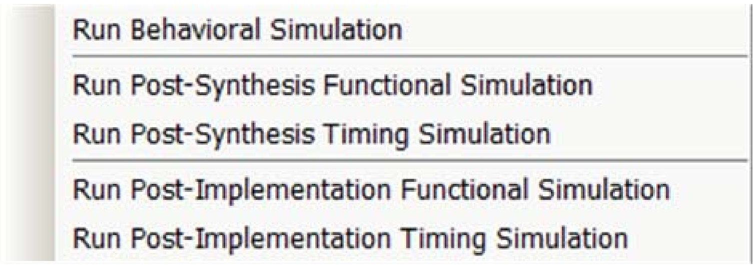

Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation

Figure 8-11: **Simulation Run Options**

Running RTL/Behavioral Simulation Using Third-Party Tools

When the `compile_simlib` command runs successfully and you have created a project without any errors, you can launch behavioral/RTL level simulation. This ensures that the RTL reflects the intended functionality.

To run behavioral simulation: in the Flow Navigator, select the **Run Simulation > Run Behavioral Simulation** option (shown in Figure 8-11).

Running Functional Simulation Using Third-Party Tools

Post-Synthesis Functional Simulation

When synthesis runs successfully, the **Run Simulation > Post-Synthesis Functional Simulation** option (shown in Figure 8-11) becomes available.

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.

Post-Implementation Functional Simulations

When implementation is successful, the **Run Simulation > Post-Implementation Functional Simulation** option (shown in Figure 8-11) becomes available.

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 Timing Simulation Using Third Party Tools

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

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

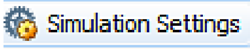
- Gate-level netlist containing SIMPRIMS library components
- SECUREIP
- Standard Delay Format (SDF) files

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

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

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

You control these options using Simulation Settings as described in Using Simulation Settings, page 25.



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 8]

.Post-Synthesis Timing Simulation

When synthesis runs successfully, the **Run Simulation > Post-Synthesis Timing Simulation** option (shown in Figure 8-11) becomes available.

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

Post-Implementation Timing Simulations

When post-implementation is successful, the **Run Simulation > Post-Implementation Timing Simulation** option (shown in Figure 8-11) becomes available.

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

Annotating the SDF File for Timing Simulation

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

TIP: *To find the SDF file options settings, in the Vivado IDE Flow Navigator, select* ***Simulation Settings****. In the* ***Project Settings*** *dialog box, select the* ***Netlist*** *tab. (See also, Vivado Simulator Project Settings in Chapter 2).*

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`

Running Standalone Timing Simulation

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

For IUS:

```
-PULSE_R/0  -PULSE_E/0
```

During elaboration (with ncelab)

For VCS:

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

During elaboration (with VCS)

For modelsim/Questasim:

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

During elaboration (With vsim)

IMPORTANT: *The Vivado simulator models use interconnect delays; consequently, additional switches are required for proper timing simulation, as follows:* `-transport_int_delays  -pulse_r 0 -pulse_int_r 0`. *Table 7-2, page 120 provides descriptions for the these commands.*

After Running Simulation with Third-Party Tools

Dumping SAIF for Power Analysis

- Dumping SAIF in QuestaSim/ModelSim
- Dumping SAIF in IES
- Dumping SAIF in VCS

See Dumping the Switching Activity Interchange Format File for Power Analysis, page 104 for more information about Switching Activity Interchange Format (SAIF).

Dumping SAIF in QuestaSim/ModelSim

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 documentation [Ref 14].

Example DO File

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

Dumping SAIF 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 14].

Dumping SAIF in VCS

VCS provides power commands to generate SAIF with specific requirements.

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

```
power <hdl_objects>
```

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

```
power -enable
```

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

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

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

Dumping VCD for Power Analysis or Debugging

- Dumping VCD in QuestaSim/ModelSim
- Dumping VCD in IES
- Dumping VCD in In VCS

Dumping VCD in QuestaSim/ModelSim

QuestaSim/ModelSim uses explicit VCD commands to dump a VCS file, as follows:

1. Open the VCD file:

   ```
   vcd file my_vcdfile.vcd
   ```

2. Specify the scope or signals to dump:

   ```
   vcd add <hdl_objects>
   ```

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

For more detailed usage or information about each commands, see the ModelSim documentation [Ref 14].

Example DO File

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

Dumping VCD in IES

1. The following command opens a VCD database named `vcddb`. The filename is `verilog.dump`. The `-timescale` option sets the `$timescale` value in the VCD file to 1 ns. Value changes in the VCD file are scaled to 1 ns.

   ```
   database -open -vcd vcddb -into verilog.dump -default -timescale ns
   ```

2. The following probe command creates a probe on all ports in the scope top.counter. Data is sent to the default VCD database.

   ```
   probe -create -vcd top.counter -ports
   ```

3. Run the simulation.

Dumping VCD in In VCS

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

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

Simulating IP

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

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

Using the Verilog UNIFAST Library

There are two methods for simulating with the UNIFAST models.

Method 1

Recommended method for simulating with all the UNIFAST models.

Select the **Simulation Settings > Enable fast simulation models** check box to enable UNIFAST support in a Vivado project environment for ModelSim. See UNIFAST Library, page 22 for more information.

Method 2

Recommended for more advanced users who want to specify which modules to simulate with the `UNIFAST` models. See Method 2: Using specific UNIFAST modules, page 24 for the description of this method.

Simulating a Design with AXI Bus Functional Models

The AXI Bus Functional Model (BFM) performs a license check to verify the AXI BFM license is present prior to use. If an AXI BFM exists in your design, perform the additional steps below to reference the license path.

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"
   ```

Using a Custom DO File During an Integrated Simulation Run

The Vivado IDE deletes the simulation directory after every re-launch and creates a new directory. So, if you are using a custom DO or UDO file, move it out of the simulation directory and point the simulator to the file. Specify the location of the custom DO or UDO file using the appropriate command, as shown below:

In QuestaSim/ModelSim

```
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]
```

In IES

```
set_property IES.CUSTOM_DO <Customized do file name> [get_filesets sim_1]

set_property IES.CUSTOM_UDO <Customized udo file name> [get_filesets sim_1]
```

In VCS

```
set_property VCS.CUSTOM_DO <Customized do file name> [get_filesets sim_1]

set_property VCS.CUSTOM_UDO <Customized udo file name> [get_filesets sim_1]
```

Generating a Simulator Specific Run Directory

The Vivado IDE generates run files in the same directory by default, even if you change the simulator.

You may create a separate directory based on simulator. Use the following Tcl command to generate a directory:

```
set_param project.addSimulatorDirForUnifiedSim 1
```

Set this command immediately after invoking Vivado IDE. The best practice would be to include this command in `init.tcl` file.

Simulation Step Control Constructs for ModelSim and Questa

the following tables outline the constructs used for controlling the step execution based on the do file format:

Native do file: In native do file format, the compile and elaborate shell scripts calls

"`source<tb>_compile/elaborate.do`". The simulate script calls "`vsim -c -do<tb>_simulate.do`".

The following is the default format:
`(project.writeNativeScriptForUnifiedSimulation is 1)`

Classic do file: In classic do file format, the compile and elaborate shell script calls "`vsim -c -do<tb>_compile/elaborate.do`". The simulate shell script calls "`vsim -c -do<tb>_simulate.do`".

The following is the default format:
`(project.writeNativeScriptForUnifiedSimulation is 0)`

Table 8-3: **Simulation Step Control Construct Parameters**

Parameter	Description	Default
project.writeNativeScriptForUnifiedSimulation	write a pure .do file with simulator command only (no Tcl or Shell constructs)	1 (true)
simulator.quitOnSimulationComplete	Quit simulator on simulator completion for ModelSim/Questa Advanced Simulator simulation. To disable quit, set this parameter to false.	1 (true)
simulator.modelsimNoQuitOnError	Do not quit on error or break by default for ModelSim/Questa Advanced Simulator simulation. To quit simulation on error or break, set this parameter to false.	1 (true)
project.enable2StepFlowForModelSim	Execute 2-step simulation flow for ModelSim-PE/DE/SE editions for Unified Simulation.	1 (true)

Appendix A

Value Rules in Vivado Simulator Tcl Commands

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:

- VHDL, see *IEEE Standard VHDL Language Reference Manual* (IEEE-STD-1076-1993) [Ref 15]
- Verilog, see *IEEE Standard Verilog Hardware Description Language (*IEEE-STD-1364-2001) [Ref 16]
- System Verilog Synthesizable subset. See IEEE Standard Verilog Hardware Description Language (IEEE-STD-1800-2009) [Ref 17]
- IEEE P1735 encryption, see *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) (*IEEE-STD-P1735) [Ref 19]

This appendix lists the application of Mixed Language in the Vivado simulator, and the exceptions to Verilog, 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/System Verilog (SV) 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. 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 163.

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:

```
-- 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
```

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 Verilog Search Order, page 117 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 163.

Instantiating a Verilog Module in a VHDL Design Unit

1. Declare a VHDL component with the same name and in the same case as the Verilog module that you want to instantiate. For example:

```
COMPONENT MY_VHDL_UNIT PORT (
   Q : out  STD_ULOGIC;
   D : in   STD_ULOGIC;
   C : in   STD_ULOGIC );
END COMPONENT;
```

2. Use named or positional association to instantiate the Verilog module. For example:

```
UUT : MY_VHDL_UNIT PORT MAP(
   Q => O,
   D => I,
        C => CLK);
```

Instantiating a VHDL Component in a Verilog/SV Design Unit

To instantiate a VHDL component in a Verilog/SV design unit, instantiate the VHDL component as if it were a Verilog/SV module.

For example:

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

Port Mapping and Supported Port Types

Table B-1 lists the supported port types.

Table B-1: **Supported Port Types**

VHDL [1]	Verilog/SV [2]
IN	INPUT
OUT	OUTPUT
INOUT	INOUT

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**

VHDL Port	Verilog Port
`bit`	net
`std_logic`	net
`bit_vector`	vector net
`signed`	vector net
`unsigned`	vector net
`std_ulogic_vector`	vector net
`std_logic_vector`	vector net

Note: Verilog output port of type `reg` is supported on the mixed language boundary. On the boundary, an output `reg` port is treated as if it were an output net (wire) port. Any other type found on mixed language boundary is considered an error.

Note: The Vivado simulator supports the record element as an actual in the port map of a Verilog module that is instantiated in the mixed domain. All those types that are supported as VHDL port (listed in Table B-2) are also supported as a record element.

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

SV Data type	VHDL Data type
Int	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
byte	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
shortint	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
longint	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
integer	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`

Table B-3: **Supported SV and VHDL Data Types *(Cont'd)***

SV Data type	VHDL Data type
vector of bit(1D)	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
vector of logic(1D)	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
vector of reg(1D)	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
logic/bit	
	`bit`
	`std_logic`
	`std_ulogic`
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`

Generics (Parameters) Mapping

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

- integer
- real
- string
- boolean

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

VHDL and Verilog Values Mapping

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

Table B-4: **Verilog States mapped to std_logic and bit**

Verilog	std_logic	bit
Z	Z	0
0	0	0
1	1	1
X	X	0

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

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

Table B-5: **VHDL bit Mapping to Verilog States**

bit	Verilog
0	0
1	1

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

Table B-6: **VHDL std_logic mapping to Verilog States**

std_logic	Verilog
U	X
X	X
0	0
1	1
Z	Z
W	X
L	0

Table B-6: **VHDL std_logic mapping to Verilog States *(Cont'd)***

std_logic	Verilog
H	1
-	X

Because Verilog is case sensitive, named associations and the local port names that you use in the component declaration must match the case of the corresponding Verilog port names.

VHDL Language Support Exceptions

Certain language constructs are not supported by the Vivado simulator. Table B-7 lists the VHDL language support exceptions.

Table B-7: **VHDL Language Support Exceptions**

Supported VHDL Construct	Exceptions
`abstract_literal`	Floating point expressed as based literals are not supported.
`alias_declaration`	Alias to non-objects are in general not supported; particularly the following: Alias of an alias Alias declaration without subtype_indication Signature on alias declarations Operator symbol as alias_designator Alias of an operator symbol Character literals as alias designators
`alias_designator`	Operator_symbol as alias_designator Character_literal as alias_designator
`association_element`	Globally, locally static range is acceptable for taking slice of an actual in an association element.
`attribute_name`	Signature after prefix is not supported.
`binding_indication`	Binding_indication without use of entity_aspect is not supported.
`bit_string_literal.`	Empty bit_string_literal (" ") is not supported
`block_statement`	Guard_expression is not supported; for example, guarded blocks, guarded signals, guarded targets, and guarded assignments are not supported.
`choice`	Aggregate used as choice in case statement is not supported.
`concurrent_assertion_statement`	Postponed is not supported.
`concurrent_signal_assignment_statement`	Postponed is not supported.

Table B-7: **VHDL Language Support Exceptions *(Cont'd)***

Supported VHDL Construct	Exceptions
`concurrent_statement`	Concurrent procedure call containing wait statement is not supported.
`conditional_signal_assignment`	Keyword guarded as part of options is not supported as there is no supported for guarded signal assignment.
`configuration_declaration`	Non locally static for generate index used in configuration is not supported.
`entity_class`	Literals, unit, file, and group as entity class are not supported.
`entity_class_entry`	Optional <> intended for use with group templates is not supported.
`file_logical_name`	Although `file_logical_name` is allowed to be any wild expression evaluating to a string value, only string literal and identifier is acceptable as file name.
`function_call`	Slicing, indexing, and selection of formals is not supported in a named parameter association within a `function_call`.
`instantiated_unit`	Direct configuration instantiation is not supported.
`mode`	Linkage and Buffer ports are not supported completely.
`options`	Guarded is not supported.
`primary`	At places where primary is used, allocator is expanded there.
`procedure_call`	Slicing, indexing, and selection of formals is not supported in a named parameter association within a `procedure_call`.
`process_statement`	Postponed processes are not supported.
`selected_signal_assignment`	The `guarded` keyword as part of options is not supported as there is no support for guarded signal assignment.
`signal_declaration`	The `signal_kind` is not supported. The `signal_kind` is used for declaring guarded signals, which are not supported.
`subtype_indication`	Resolved subtype of composites (arrays and records) is not supported
`waveform`	Unaffected is not supported.
`waveform_element`	Null waveform element is not supported as it only has relevance in the context of guarded signals.

Verilog Language Support Exceptions

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

Table B-8: **Verilog Language Support Exceptions**

Verilog Construct	Exception
Compiler Directive Constructs	
`unconnected_drive	not supported
`nounconnected_drive	not supported
Attributes	
attribute_instance	not supported
attr_spec	not supported
attr_name	not supported
Primitive Gate and Switch Types	
cmos_switchtype	not supported
mos_switchtype	not supported
pass_en_switchtype	not supported
Generated Instantiation	
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 16]: generate_item_or_null ::= generate_conditonal_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_blockgenerate_condition

Table B-8: **Verilog Language Support Exceptions** ***(Cont'd)***

Verilog Construct	Exception

`genvar_assignment`	Partially supported.
	All generate blocks must be named.
	Production from standard (see *IEEE Standard Verilog Hardware Description Language* (IEEE 1364-2001) section 13.2 [Ref 16]:
	`generate_block ::=`
	`begin`
	`[ : generate_block_identifier ]`
	`{ generate_item }`
	`end`
	`Production supported by Simulator:`
	`generate_block ::=`
	`begin:`
	`generate_block_identifier {`
	`generate_item }`
	`end`

Source Text Constructs	
Library Source Text	
`library_text`	not supported
`library_descriptions`	not supported
`library_declaration`	not supported
`include_statement`	This refers to include statements within library map files (See *IEEE Standard Verilog Hardware Description Language* (IEEE 1364-2001) section 13.2 [Ref 16]. 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

Table B-8: **Verilog Language Support Exceptions** ***(Cont'd)***

Verilog Construct	Exception
`$sync$or$array`	not supported
`$async$and$plane`	not supported
`$async$nand$plane`	not supported
`$async$nor$plane`	not supported
`$async$or$plane`	not supported
`$sync$and$plane`	not supported
`$sync$nand$plane`	not supported
`$sync$nor$plane`	not supported
`$sync$or$plane`	not supported
Value Change Dump (VCD) Files	

`$dumpportson`
`$dumpports`
`$dumpportsoff`
`$dumpportsflush`
`$dumpportslimit`
`$vcdplus`
	not supported

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**

Parsing HDL Files	
Vivado Simulator supports three HDL file types: Verilog, SystemVerilog and VHDL. You can parse the supported files using XVHDL and XVLOG commands.	

Parsing VHDL files	`xvhdl file1.vhd file2.vhd`
	`xvhdl -work worklib file1.vhd file2.vhd`
	`xvhdl -prj files.prj`

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

Parsing SystemVerilog files	`xvlog -sv file1.v file2.v`
	`xvlog -work worklib -sv file1.v file2.v`
	`xvlog -prj files.prj`
	Note: For information about the PRJ file format, see Project File (.prj) Syntax in Chapter 7.

Table C-1: **Standalone Mode: Parsing, Elaborating, and Running Simulation from a Command Line** ***(Cont'd)***

Additional xvlog and xvhdl Options		
xvlog and xvhdl Key Options	See Table 7-2, page 120 for a complete list of command options. The following are key options for xvlog and xvhdl:	
	Key Option	Applies to:
	-d [define] <name>[=<val>]	xvlog
	-h [-help]	xvlog, xvhdl
	-i [include] <directory_name>	xvlog
	-initfile <init_filename>	xvlog, xvhdl
	-L [-lib] <library_name> [=<library_dir>]	xvlog, xvhdl
	-log <filename>	xvlog, xvhdl
	-prj <filename>	xvlog, xvhdl
	-relax	xvhdl, vlog
	-work <library_name> [=<library_dir>]	xvlog, xvhdl
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 have 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.

Table C-1: **Standalone Mode: Parsing, Elaborating, and Running Simulation from a Command Line *(Cont'd)***

Command Line Help and xelab Options	`xelab -help` xelab, xvhd, and xvlog Command Options, page 120	
Running Simulation		
After parsing, elaboration and compilation stages are successful; xsim generates an executable snapshot to run simulation.		
Usage	`xsim top -R`	Simulates the design to through completion.
	`xsim top -gui`	Opens the Vivado simulator workspace (GUI).

	`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.		

	Three Stage	`xvlog bot.v`
		`xvhdl top.vhd`
		`xelab work.top -s top`
		`xsim top -R`

	Two Stage	`xelab -prj my_prj.prj work.top -s top`
		`xsim top -R`
		where `my_prj.prj` file contains:
		`verilog  work bot.v`
		`vhdl  work top.vhd`

	Single Stage	`xelab -prj my_prj.prj work.top -s top -R`
		where `my_prj.prj` file contains:
		`verilog  work bot.v`
		`vhdl  work top.vhd`

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 C-1: **Standalone Mode: Parsing, Elaborating, and Running Simulation from a Command Line** ***(Cont'd)***

Common Vivado Simulator Tcl Commands:	`add_bp`	Add break point at a line of HDL source. A Tcl command example is provided on page 78.
	`add_force`	Force the value of a signal, wire, or register to a specified value. Tcl command exampled are provided on page 82.
	`current_time` `now`	Report current simulation time. See current_time, page 132 for an example of this command within a Tcl script.
	`current_scope`	Report or set the current, working HDL scope. See Additional Scopes and Sources Options, page 39 for more information.
	`get_objects`	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 || type == out_port || type == inout_port || type == port } /tb/UUT/*], page 89 .
	`get_scopes`	Get a list of child HDL scopes. See Additional Scopes and Sources Options, page 39 for more information.
	`get_value`	Get the current value of the selected HDL object (variable, signal, wire, register). Type `get_value -help` in Tcl Console for more information.
	`launch_simulation`	Launch simulation using the Vivado simulator.
	`remove_bps`	Remove breakpoints from a simulation. A Tcl command example is provided on page 78.
	`report_drivers`	Print drivers along with current driving values for an HDL wire or signal object. Reference for more information: Using the report_drivers Tcl Command, page 89.
	`report_values`	Print current simulated value of given HDL objects (variables, signals, wires, or registers). For example Tcl command usage, see page 39.
	`restart`	Rewind simulation to post loading state (as though the design was reloaded); time is set to 0. For additional information, see page 33.
	`set_value`	Set the HDL object (variable, signal, wire, or register) to a specified value. Reference for more information: Appendix A, Value Rules in Vivado Simulator Tcl Commands.
	`step`	Step simulation to the next statement. See Stepping Through a Simulation, page 76.

Appendix D

System Verilog Constructs Supported by the Vivado Simulator

Introduction

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

Targeting System Verilog for a Specific File

By default, the Vivado simulator tool compiles `.v` files with the Verilog 2001 syntax and `.sv` files with the System Verilog syntax.

To target System Verilog for a specific `.v` file in the Vivado IDE:

1. Right-click the file and select **Set file type** as shown in the figure below.

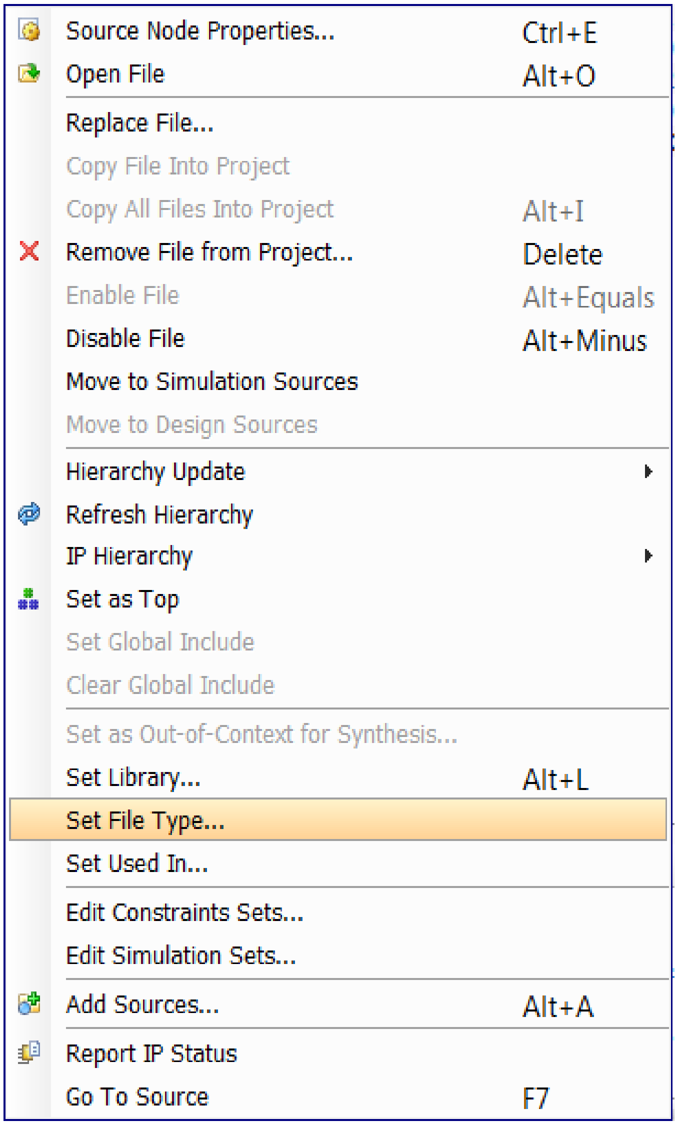


Figure D-1: **Context Menu with Set File Type Option**

2. In the **Set Type** menu, shown in the figure below, change the file type from Verilog to **System Verilog** and click **OK**.

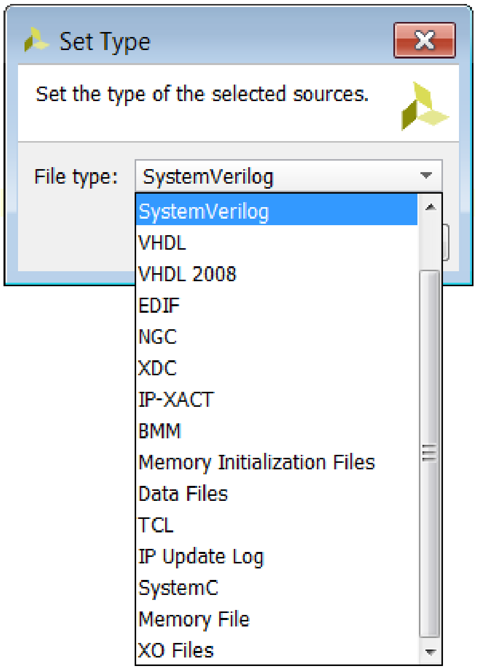


Figure D-2: **Source Node Properties > Set File Type**

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

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

Running System Verilog in Standalone or prj Mode

Standalone Mode

A new `-sv` flag has been introduced to `xvlog`, so if you want to read any System Verilog file, you can use following command:

```
xvlog  -sv <Design file list>
xvlog -sv  -work <LibraryName> <Design File List>
xvlog -sv -f  <FileName> [Where FileName contain path of test cases]
```

prj Mode

If you want to run the Vivado simulator in the `prj`-based flow, use `sv` as the file type, as you would `verilog` or `vhdl`.

```
xvlog -prj <prj File>
xelab -prj <prj File>  <topModuleName> <other options>
```

. . .where the entry in `prj` file appears as follows:

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

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

Primary construct	Secondary construct	LRM section	Status
Data type		6	
	Singular and aggregate types	6.4	Supported
	Nets and variables	6.5	Supported
	Variable declarations	6.8	Supported
	Vector declarations	6.9	Supported
	2-state (two-value) and 4-state (four-value) data types	6.11.2	Supported
	Signed and unsigned integer types	6.11.3	Supported
	Real, shortreal and realtime data types	6.12	Supported
	User-defined types	6.18	Supported
	Enumerations	6.19	Supported
	Defining new data types as enumerated types	6.19.1	Supported
	Enumerated type ranges	6.19.2	Supported
	Type checking	6.19.3	Supported
	Enumerated types in numerical expressions	6.19.4	Supported
	Enumerated type methods	6.19.5	Supported
	Type parameters	6.20.3	Supported
	Const constants	6.20.6	Supported
	Type operator	6.23	Supported
	Cast operator	6.24.1	Supported
	`$cast` dynamic casting	6.24.2	Not Supported
	Bitstream casting	6.24.3	Supported
Aggregate data types		7	
	Structures	7.2	Supported
	Packed/Unpacked structures	7.2.1	Supported
	Assigning to structures	7.2.2	Supported
	Unions	7.3	Supported

Table D-1: **Synthesizable Set of System Verilog 1800-2009 *(Cont'd)***

Primary construct	Secondary construct	LRM section	Status
	Packed/Unpacked unions	7.3.1	Supported
	Tagged unions	7.3.2	Not Supported
	Packed arrays	7.4.1	Supported
	Unpacked arrays	7.4.2	Supported
	Operations on arrays	7.4.3	Supported
	Multidimensional arrays	7.4.5	Supported
	Indexing and slicing of arrays	7.4.6	Supported
	Array assignments	7.6	Supported
	Arrays as arguments to subroutines	7.7	Supported
	Array querying functions	7.11	Supported
	Array manipulation methods (those that do not return queue type)	7.12	Supported
Processes		9	
	Combinational logic `always_comb` procedure	9.2.2	Supported
	Implicit `always_comb` sensitivities	9.2.2.1	Supported
	Latched logic `always_latch` procedure	9.2.2.3	Supported
	Sequential logic `always_ff` procedure	9.2.2.4	Supported
	Sequential blocks	9.3.1	Supported
	Parallel blocks	9.3.2	Not Supported
	Procedural timing controls	9.4	Supported
	Conditional event controls	9.4.2.3	Supported
	Sequence events	9.4.2.4	Not Supported
Assignment statement		10	
	The continuous assignment statement	10.3.2	Supported
	Variable declaration assignment (variable initialization)	10.5	Supported
	Assignment-like contexts	10.8	Supported
	Array assignment patterns	10.9.1	Supported
	Structure assignment patterns	10.9.2	Supported
	Unpacked array concatenation	10.10	Supported

Table D-1: **Synthesizable Set of System Verilog 1800-2009 *(Cont'd)***

Primary construct	Secondary construct	LRM section	Status
	Net aliasing	10.11	Not Supported
Operators and expressions		11	
	Constant expressions	11.2.1	Supported
	Aggregate expressions	11.2.2	Supported
	Operators with real operands	11.3.1	Supported
	Operations on logic (4-state) and bit (2-state) types	11.3.4	Supported
	Assignment within an expression	11.3.6	Supported
	Assignment operators	11.4.1	Supported
	Increment and decrement operators	11.4.2	Supported
	Arithmetic expressions with unsigned and signed types	11.4.3.1	Supported
	Wildcard equality operators	11.4.6	Supported
	Concatenation operators	11.4.12	Supported
	Set membership operator	11.4.13	Supported
	Concatenation of `stream_expressions`	11.4.14.1	Supported
	Re-ordering of the generic stream	11.4.14.2	Supported
	Streaming concatenation as an assignment target (unpack)	11.4.14.3	Not Supported
	Streaming dynamically sized data	11.4.14.4	Not Supported
Procedural programming statement		12	
	`Unique-if`, `unique0-if` and `priority-if`	12.4.2	Supported
	Violation reports generated by `B-if`, `unique0-if`, and priority-if constructs	12.4.2.1	Supported
	If statement violation reports and multiple processes	12.4.2.2	Supported

Table D-1: **Synthesizable Set of System Verilog 1800-2009 *(Cont'd)***

Primary construct	Secondary construct	LRM section	Status
	`unique-case`, `unique0-case`, and `priority-case`	12.5.3	Supported
	Violation reports generated by `unique-case`, `unique0-case`, and `priority-case` construct	12.5.3.1	Supported
	Case statement violation reports and multiple processes	12.5.3.2	Supported
	Set membership case statement	12.5.4	Supported
	Pattern matching conditional statements	12.6	Not Supported
	Loop statements	12.7	Supported
	Jump statement	12.8	Supported
Tasks		13.3	
	Static and Automatic task	13.3.1	Supported
	Tasks memory usage and concurrent activation	13.3.2	Supported
Function		13.4	
	Return values and void functions	13.4.1	Supported
	Static and Automatic function	13.4.2	Supported
	Constant function	13.4.3	Supported
	Background process spawned by function call	13.4.4	Not Supported
Subroutine calls and argument passing		13.5	
	Pass by value	13.5.1	Supported
	Pass by reference	13.5.2	Supported
	Default argument value	13.5.3	Supported
	Argument binding by name	13.5.4	Supported
	Optional argument list	13.5.5	Supported
	Import and Export function	13.6	Not Supported
	Task and function name	13.7	Supported

Table D-1: **Synthesizable Set of System Verilog 1800-2009 *(Cont'd)***

Primary construct	Secondary construct	LRM section	Status
Utility system tasks and system functions (only synthesizable set)		20	Supported
I/O system tasks and system functions (only synthesizable set)		21	Supported
Compiler directives		22	Supported
Modules and hierarchy		23	
	Default port values	23.2.2.4	Supported
	Top-level modules and `$root`	23.3.1	Supported
	Module instantiation syntax	23.3.2	Supported
	Nested modules	23.4	Supported
	Extern modules	23.5	Supported
	Hierarchical names	23.6	Supported
	Member selects and hierarchical names	23.7	Supported
	Upwards name referencing	23.8	Supported
	Overriding module parameters	23.10	Supported
	Binding auxiliary code to scopes or instances	23.11	Not Supported
Interfaces		25	
	Interface syntax	25.3	Supported
	Nested interface	25.3	Supported
	Ports in interfaces	25.4	Supported
	Example of named port bundle	25.5.1	Supported
	Example of connecting port bundle	25.5.2	Supported
	Example of connecting port bundle to generic interface	25.5.3	Supported
	Modport expressions	25.5.4	Supported
	Clocking blocks and modports	25.5.5	Not Supported
	Interfaces and specify blocks	25.6	Not Supported
	Example of using tasks in interface	25.7.1	Supported

Table D-1: **Synthesizable Set of System Verilog 1800-2009 *(Cont'd)***

Primary construct	Secondary construct	LRM section	Status
	Example of using tasks in modports	25.7.2	Supported
	Example of exporting tasks and functions	25.7.3	Supported
	Example of multiple task exports	25.7.4	Not Supported
	Parameterized interfaces	25.8	Supported
	Virtual interfaces	25.9	Not Supported
Packages		26	
	Package declarations	26.2	Supported
	Referencing data in packages	26.3	Supported
	Using packages in module headers	26.4	Supported
	Exporting imported names from packages	26.6	Supported
	The std built-in package	26.7	Not Supported
Generate constructs		27	Supported

Dynamic Types

In Vivado simulator, support for some of the commonly used dynamic types has been added, as shown in the table below.

Table D-2: **Supported Dynamic Type Constructs**

Primary Construct	Secondary Construct	LRM Section	Status
String data type		6.16	Supported
	String operators (table 6-9)	6.16	Supported
	Len()	6.16.1	Supported
	Putc()	6.16.2	Supported
	Getc()	6.16.3	Supported
	Toupper()	6.16.4	Supported
	Tolower()	6.16.5	Supported
	Compare	6.16.6	Supported
	Icompare()	6.16.7	Supported
	Substr()	6.16.8	Supported
	Atoi(), atohex(), atooct(), atobin()	6.16.9	Supported
	Atoreal()	6.16.10	Supported
	Itoa()	6.16.11	Supported
	Hextoa()	6.16.12	Supported
	Octtoa()	6.16.13	Supported
	Bintoa()	6.16.14	Supported
	Realtoa()	6.16.15	Supported
Dynamic Array		7.5	Supported
	Dynamic array new	7.5.1	Supported
	Size	7.5.2	Supported
	Delete	7.5.3	Supported
Associative Array		7.8	Supported
	Wildcard index	7.8.1	Supported
	String index	7.8.2	Supported
	Class index	7.8.3	Supported
	Integral index	7.8.4	Supported
	Other user-defined types	7.8.5	Supported
	Accessing invalid index	7.8.6	Supported
	Associative array methods	7.9	Supported

Table D-2: **Supported Dynamic Type Constructs *(Cont'd)***

Primary Construct	Secondary Construct	LRM Section	Status
	Num() and Size()	7.9.1	Supported
	Delete()	7.9.2	Supported
	Exists()	7.9.3	Supported
	First()	7.9.4	Supported
	Last()	7.9.5	Supported
	Next()	7.9.6	Supported
	Prev()	7.9.7	Supported
	Arguments to traversal Method	7.9.8	Supported
	Associative array assignment	7.9.9	Supported
	Associative array arguments	7.9.10	Supported
	Associative Array literals	7.9.11	Supported
Queue		7.10	Supported
	Queue operators	7.10.1	Supported
	Queue methods	7.10.2	Supported
	Size()	7.10.2.1	Supported
	Insert()	7.10.2.2	Supported
	Delete()	7.10.2.3	Supported
	Pop_front()	7.10.2.4	Supported
	Pop_back()	7.10.2.5	Supported
	Push_front()	7.10.2.6	Supported
	Push_back()	7.10.2.7	Supported
	Persistence of references to elements of a queue	7.10.3	Supported
	Updating a queue using assignment and unpacked array concatenation	7.10.4	Supported
	Bounded queues	7.10.5	Supported
Class		8	Supported
	Class General	8.1	Supported
	Overviews	8.2	Supported
	Syntax	8.3	Supported
	Objects(Class instance)	8.4	Supported
	Object properties and object parameter data	8.5	Supported
	Object methods	8.6	Supported

Table D-2: **Supported Dynamic Type Constructs *(Cont'd)***

Primary Construct	Secondary Construct	LRM Section	Status
	Constructors	8.7	Supported
	Static class properties	8.8	Supported
	Static methods	8.9	Supported
	This	8.10	Supported
	Assignment, renaming, and copying	8.11	Supported
	Inheritance and subclasses	8.12	Supported
	Overridden members	8.13	Supported
	Super	8.14	Supported
	Casting	8.15	Supported
	Chaining constructors	8.16	Supported
	Data hiding and encapsulation	8.17	Supported
	Constant class properties	8.18	Supported
	Virtual methods	8.19	Supported
	Abstract classes and pure virtual methods	8.20	Not Supported
	Polymorphism: dynamic method lookup	8.21	Supported
	Class scope resolution operator ::	8.22	Supported
	Out-of-block declarations	8.23	Supported
	Parameterized classes	8.24	Supported
	Class resolution operator for parameterized classes	8.24.1	Supported
	Typedef class	8.25	Supported
	Classes and structures	8.26	Supported
	Memory management	8.27	Supported

VHDL 2008 Support for Vivado Simulator

Introduction

The Vivado® simulator supports the subset of VHDL 2008(IEEE 1076-2008). The complete list is given in Table E-1.

Compiling and Simulating

The Vivado Simulator executable `xvhdl` is used to convert a VHDL design unit into parser dump (.vdb). The default mode for `xvhdl` is to compile in VHDL 93 mode. To compile a file with VHDL 2008 mode, you need to pass `–2008` switch to `xvhdl`.

For example, to compile a design called top.vhdl in VHDL-2008, following command line can be used:

```
xvhdl -2008 -work mywork top.vhdl
```

The Vivado Simulator executable `xelab` is used to elaborate a design and produce an executable image for simulation.

xelab can do either of the following:

- Elaborate on parser dumps produced by `xvhdl`
- Directly use vhdl source files.

No switch is needed to elaborate on parser dumps produced by xvhdl. You can pass `–vhdl2008` to xelab to directly use vhdl source files.

Example 1:

```
xelab top -s mysim; xsim mysim -R
```

Example 2:

```
xelab -vhdl2008 top.vhdl top -s mysim; xsim mysim -R
```

Instead of specifying VHDL files in the command line for `xvhdl` and `xelab`, a .prj file can be used. If you have two files for a design called `top.vhdl` (2008 mode) and `bot.vhdl` (93 mode), you can create a project file named example.prj as follows:

```
vhdl xil_defaultlib bot.vhdl
vhdl2008 xil_defaultlib top.vhdl
```

In the project file, each line starts with the language type of the file, followed by the library name such as `xil_defaultlib` or `IEEE`, and one or more file names with a space separator. For VHDL 93, one should use `vhdl` as the language type. For VHDL 2008, use `vhdl2008` instead.

A .prj file can be used as shown in the example below:

```
xelab -prj example.prj top -s mysim; xsim mysim -R
```

Alternatively, to mix VHDL 93 and VHDL 2008 design units, compile the files separately with a proper language mode specified to `xvhdl`. Then, elaborate on top(s) of the design. For example, if we have a VHDL 93 module called bot in file `bot.vhdl`, and a VHDL-2008 module called top in file `top.vhdl`, you can compile them as shown in the example below:

```
xvhdl bot.vhdl
xvhdl -2008 top.vhdl
xelab -debug typical top -s mysim
```

Once the executable is produced by xelab, you can run the simulation as usual.

Example 1:

```
xsim mysim -gui
```

Example 2:

```
xsim mysim -R
```

Supported Features

Table E-1: **Supported features of VHDL 2008(IEEE1076-2008): Early Access**

Features	Example/Comment
VHDL-2008 STD and IEEE packages precompiled, including new fixed and float packages, unsigned bit etc.	Limited by other language features such as generic package which XSIM does not yet support. All newly added std functions are not yet supported. Notably, `stop` and `finish` are supported.
Simplified sensitivity list	process(all)

Table E-1: **Supported features of VHDL 2008(IEEE1076-2008): Early Access *(Cont'd)***

Features	Example/Comment

Conditional and selected sequential statements	`process(clk)`
	`...`
	`  with x select`
	`      y := "111" when "110",`
	`            "000" when others;`
	
	`  a := '1' when b = '1' else`
	`    '0' when b = '0';`
	`...`

Protected types	`type areaOfSquare is protected`
	`    procedure setx(newx : real);`
	`    impure function area return real;`
	`end protected;`
	
	`type areaOfSquare is protected body`
	` variable x : real = 0.0;`
	`...`
	Note: Protected type shared variable is supported in HDL simulation. However, TCL and GUI does not yet allow examining value of protected type shared variables.

Keyword `parameter` in procedure declaration	`procedure proc parameter (a : in`
	`std_logic)`

Array element resolution function in subtype definition	`type bit_word is array  (natural range <>)`
	`of bit;`
	`function resolve_array (s : bit_word)`
	`return bit;`
	`subtype resolved_array is (resolve_array)`
	`bit_word;`

Block comments	`/*`
	`X <= 1;`
	`Process(all)`
	`...`
	`*/`

Predefined array types	boolean_vector, integer_vector etc.

Type passed as Generic	`entity test is`
	`  generic (type data_type);`
	`  port (`
	`         x : in data_type;`
	`         s : out data_type);`
	`end entity test;`

Appendix F

Direct Programming Interface (DPI) in Vivado Simulator

Introduction

You can use the SystemVerilog Direct Programming Interface (DPI) to bind C code to SystemVerilog code. Using DPI, SystemVerilog code can call a C function, which in turn can call back a SystemVerilog task or function. Vivado® simulator supports all the constructs except the open array as DPI task/function, as described below.

Compiling C Code

A new compiler executable, `xsc`, is provided to convert C code into an object code file and to link multiple object code files into a shared library (`.a` on Windows and `.so` on Linux). The xsc compiler is available in the `<Vivado installation>/bin` directory. You can use `-sv_lib` to pass the shared library containing your C code to the Vivado simulator/elaborator executable. The xsc compiler works in the same way as a C compiler, such as gcc. The xsc compiler:

- Calls the LLVM clang compiler to convert C code into object code
- Calls the GNU linker to create a shared library (`.a` on Windows and `.so` on Linux) from one or more object files corresponding to the C files

The shared library generated by the xsc compiler is linked with the Vivado simulator kernel using one or more newly added switches in xelab, as described below. The simulation snapshot created by xelab thus has ability to connect the compiled C code with compiled SystemVerilog code and effect communication between C and SystemVerilog.

xsc Compiler

The xsc compiler helps you to create a shared library (`.a` on Windows or `.so` on Linux) from one or more C files. Use xelab to bind the shared library generated by xsc into the rest of your design. You can create a shared library using the following processes:

One-step process:

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

Table F-1: **XSC Compiler 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`)

Table F-1: **XSC Compiler Switches**

`-additional_option arg`	Provide an additional option to the compiler. You can use multiple -additional_option switches.
`-C [ --cc ] arg`	Specify Compiler to compile generated C code. Choices are: `gcc`: Use GCC `clang`: Use CLANG C Compiler (Default: gcc)
`-dpi_absolute`	Use absolute paths instead of `LD_LIBRARY_PATH` on Linux for DPI libraries that are formatted as lib<libname>.so

Examples

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

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

Note: By default, Linux uses the `LD_LIBRARY_PATH` for searching the DPI libraries . Hence, provide `-dpi_absolute` flag to xelab on Linux if library name start with `lib*`.

Binding Compiled C Code to SystemVerilog Using xelab

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

`-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.

For more information on `r-sv_liblist arg`, refer to the *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language* [Ref 17], Appendix J.4.1, page 1228.

Data Types Allowed on the Boundary of C and SystemVerilog

The *IEEE Standard for SystemVerilog* [Ref 17] 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 F-2: **Data Types Allowed on the C-SystemVerilog Boundary**

SystemVerilog	C	Supported	Comments
byte	char	Yes	None
shortint	short int	Yes	None
int	int	Yes	None
longint	long long	Yes	None
real	double	Yes	None
shortreal	float	Yes	None
chandle	void *	Yes	None
string	const char*	Yes	None
bit	unsigned char	Yes	sv_0, sv_1
Available on C side using svdpi.h			
logic, reg	unsigned char	Yes	sv_0, sv_1, sv_z, sv_x:
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	Yes	None
Packed structs, unions	Passed as array	Yes	None
Unpacked arrays of bit, logic	Passed as array	Yes	C can call SystemVerilog
Unpacked structs,	Passed as struct	Yes	None
Unpacked 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 17].

Mapping for User-Defined Types

Enum

You can define an enumerated type (`enum`) for conversion to the equivalent SystemVerilog types, `svLogicVecVal` or `svBitVecVal`, depending on the base type of `enum`. For enumerated arrays, equivalent SystemVerilog arrays are created.

Examples:

SystemVerilog types:

```
typedef enum reg [3:0] { a = 0, b = 1, c} eType;
eType e;
eType e1[4:3];

typedef enum bit { a = 0, b = 1} eTypeBit;
eTypeBit    e3;
eTypeBit    e4[3:1] ;
```

C types:

```
svLogicVecVal e[SV_PACKED_DATA_NELEMS(4)];
svLogicVecVal e1[2][SV_PACKED_DATA_NELEMS(4)];
svBit  e3;
svBit  e4[3];
```

TIP: *The C argument types depend on the base type of the* `enum` *and the direction.*

Packed Struct/Union

When using a packed struct or union type, an equivalent SystemVerilog type, `svLogicVecVal` or `svBitVecVal`, is created on the DPI C side.

Examples

SystemVerilog type:

```
typedef struct packed {
        int i;
        bit b;
        reg [3:0]r;
        logic [2:0] [4:8][9:1] l;
    } sType;
    sType c_obj;
   sType [3:2] c_obj1[5];
```

C type:

```
svLogicVecVal  c_obj[SV_PACKED_DATA_NELEMS(172)];
svLogicVecVal  c_obj1[5][SV_PACKED_DATA_NELEMS(344)];
```

Arrays, both packed and unpacked, are represented as arrays of `svLogicVecVal` or `svBitVecVal`.

Unpacked Struct

An equivalent unpacked type is created on the C side, in which all the members are converted to the equivalent C representation.

Examples:

SystemVerilog type:

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

C type:

```
typedef struct {
    int i;
    svBit b;
    svLogic r[4];
    svLogicVecVal l[5][9][SV_PACKED_DATA_NELEMS(3)];
}  sType;
```

Support for svdpi.h functions

The `svdpi.h` header file is provided in this directory:
`<vivado installation>/data/xsim/include`.

The following `svdpi.h` functions are supported:

```
svBit svGetBitselBit(const svBitVecVal* s, int i);

svLogic svGetBitselLogic(const svLogicVecVal* s, int i);

void svPutBitselBit(svBitVecVal* d, int i, svBit s);

void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);

void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);

void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int w);

void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);

void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int w);

const char* svDpiVersion();
    svScope svGetScope();
    svScope svSetScope(const svScope scope);
    const char* svGetNameFromScope(const svScope);
    int svPutUserData(const svScope scope, void*userKey, void* userData);
    void* svGetUserData(const svScope scope, void* userKey);
    int svIsDisabledState();
    void svAckDisabledState();
```

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

function1.c

```
#include "svdpi.h"

DPI_DLLESPEC
int myFunction1()
{
    return 5;
}
```

function2.c

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

file.sv

```
module m();

import "DPI-C" pure function int myFunction1 ();
import "DPI-C" pure function int myFunction2 ();

integer i, j;

initial
begin
#1;
  i = myFunction1();
  j = myFunction2();
  $display(i, j);
  if( i == 5 && j == 10)
    $display("PASSED");
  else
    $display("FAILED");
end

endmodule
```

Usage

Methods for compiling and linking the C files into the Vivado simulator are described below.

Single-step flow (simplest flow)

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

Flow description:

The xsc compiler compiles and links the C code to create the shared library `xsim.dir/xsc/dpi.so`, and xelab references the shared library through the switch `-sv_lib`.

Two-step flow

```
xsc -compile function1.c function2.c -work abc
xsc -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`.*

TIP: *On Linux, if `sv_root` is not specified and the DPI library is named with the prefix `lib` and the suffix `.so`, then use the LD_LIBRARY_PATH environment variable for the location of shared library.*

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

```
xsc -compile function1.c function2.c -work "abc" -v 1
xsc -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.

Function with Output

Code

file.sv

```
/*- - - -*/
package pack1;
import "DPI-C"  function int myFunction1(input int v, output int o);
import "DPI-C"  function void myFunction2 (input int v1, input int v2, output int o);
endpackage

/*-- ---*/
module m();
int i, j;
int o1 ,o2, o3;

initial
begin
#1;
   j = 10;
   o3 =pack1:: myFunction1(j, o1);//should be 10/2 = 5
   pack1::myFunction2(j, 2+3, o2); // 5 += 10 + 2+3
   $display(o1, o2);
   if( o1 == 5 && o2 == 15)
      $display("PASSED");
   else
      $display("FAILED");
end


endmodule
```

function.c

```
#include "svdpi.h"

DPI_DLLESPEC
int myFunction1(int j, int* o)
{
   *o = j /2;
   return 0;
}

DPI_DLLESPEC
void myFunction2(int i, int j, int* o)
{
   *o = i+j;
   return;
}
```

run.ksh

```
xsc function.c
xelab -vlog file.sv -sv -sv_lib dpi -R
```

Simple Import-Export Flow (illustrates xelab -dpiheader flow)

In this flow:

- 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.

file.c

```
#include "file.h"
/* NOTE: This file is generated by xelab -dpiheader <filename> flow */

int cfunc (int a, int b) {
    //Call the function exported from SV.
    return c_exported_func (a,b);
}
```

test.sv

```
module m();
export "DPI-C" c_exported_func = function func;
import "DPI-C" pure function int cfunc (input int a ,b);

/*This function can be called from both SV or C side. */
function int func(input int x, y);
begin
    func = x + y;
end
endfunction

int z;

initial
begin
    #5;
    z  = cfunc(2, 3);
    if(z == 5)
        $display("PASSED");
    else
        $display("FAILED");

end
endmodule
```

run.ksh

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

```
   file.h
/**********************************************************************/
/*   ____  ____                                                       */
/*  /   /\/   /                                                       */
/* /___/  \  /                                                        */
/* \   \   \/                                                         */
/*  \   \        Copyright (c) 2003-2013 Xilinx, Inc.                 */
/*  /   /        All Right Reserved.                                  */
/* /---/   /\                                                         */
/* \   \  /  \                                                        */
/*  \___\/\___\                                                       */
/**********************************************************************/

   /* NOTE: DO NOT EDIT. AUTOMATICALLY GENERATED FILE. CHANGES WILL BE LOST. */

   #ifndef DPI_H
   #define DPI_H
   #ifdef __cplusplus
   #define DPI_LINKER_DECL  extern "C"
   #else
   #define DPI_LINKER_DECL
   #endif

   #include "svdpi.h"


   /* Exported (from SV) function */
   DPI_LINKER_DECL DPI_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."*

Appendix G

Using Xilinx Simulator Interface

Introduction

The Xilinx® Simulator Interface (XSI) is a C/C++ application programming interface (API) to the Xilinx Vivado Simulator (xsim) that enables a C/C++ program to serve as the test bench for a HDL design. Using XSI, the C/C++ program controls the activity of the Vivado Simulator which hosts the HDL design.

The C/C++ program controls the simulation in the following methods:

- Setting the values of the top-level input ports of the HDL design
- Instructing the Vivado Simulator to run the simulation for a certain amount of simulation time

Additionally, the C/C++ program can read the values of the top-level output ports of the HDL design.

Perform the following steps to use XSI in your C/C++ program:

1. Prepare the XSI API functions to be called through dynamic linking
2. Write your C/C++ test bench code using the API functions
3. Compile and link your C/C++ program
4. Package the Vivado Simulator and the HDL design together into a shared library

Preparing the XSI Functions for Dynamic Linking

Xilinx recommends the usage of dynamic linking for indirectly calling the XSI functions. While this technique involves more steps than simply calling XSI functions directly, dynamic linking allows you to keep the compilation of your HDL design independent of the compilation of your C/C++ program. You can compile and load your HDL design at any time, even while your C/C++ program continues to run.

To call a function through dynamic linking requires your program to perform the following steps:

1. Open the shared library containing the function
2. Look up the function by name to get a pointer to the function
3. Call the function using the function pointer
4. Close the shared library (optional)

Steps 1, 2, and 4 require the use of OS-specific library calls, as shown in Table G-1. Refer to your operating system documentation for details about these functions.

Table G-1: **Operating System Specific Library Calls**

Function	Linux	Windows
Open shared library	void ***dlopen**(const char *filename, int flag);	HMODULE WINAPI **LoadLibrary**(_In_ LPCTSTR lpFileName);
Look up function by name	void ***dlsym**(void *handle, const char *symbol);	FARPROC WINAPI **GetProcAddress**(_In_ HMODULE hModule,_In_ LPCSTR lpProcName);
Close shared library	int **dlclose**(void *handle);	BOOL WINAPI **FreeLibrary**(_In_ HMODULE hModule);

XSI requires you to call functions from two shared libraries: the kernel shared library and your design shared library. The kernel shared library ships with the Vivado Simulator and is called `librdi_simulator_kernel.so` (Linux) or `librdi_simulator_kernel.dll` (Windows). It resides in the following directory:

```
<Vivado Installation Root>/lib/<platform>
```

where `<platform>` is `lnx64.o` or `win64.o`. Make sure to include this directory in your library path while running your program. On Linux, include the directory in the environment variable `LD_LIBRARY_PATH`, and on Windows, in the environment variable `PATH`.

Your design shared library, which the Vivado Simulator creates in the course of compiling your HDL design, as described in Preparing the Design Shared Library, is called `xsimk.so` (Linux) or `xsimk.dll` (Windows) and typically resides at the following location:

```
<HDL design directory>/xsim.dir/<snapshot name>
```

where `<HDL design directory>` is the directory from which your design shared library was created, and `<snapshot name>` is the name of the snapshot that you specify during the creation of the library. Your C/C++ program will call the XSI function `xsi_open()`

residing in your design shared library and all other XSI functions from the kernel shared library.

The XSI code examples that ship with the Vivado Simulator consolidate the XSI functions into a C++ class called `Xsi::Loader`. The class accepts the names of the two shared libraries, internally executes the necessary dynamic linking steps, and exposes all the XSI functions as member functions of the class. Wrapping the XSI functions in this manner eliminates the necessity of calling the dynamic linking OS functions directly. You can find the source code for the class that can be copied into your own program at the following location under your Vivado installation:

```
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.h
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.cpp
```

To use `Xsi::Loader`, simply instantiate it by passing the names of the two shared libraries as shown in the following example:

```
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so", "librdi_simulator_kernel.so");
```

Writing the Test Bench Code

A C/C++ test bench using XSI typically uses the following steps:

1. Open the design
2. Fetch the IDs of each top-level port
3. Repeat the following until the simulation is finished:
   a. Set values on top-level input ports
   b. Run the simulation for a specific amount of time
   c. Fetch the values of top-level output ports
4. Close the design

The following table lists the XSI functions and their `Xsi::Loader` member function equivalents to use for each step. You can find the usage details for each XSI function in the XSI Function Reference section.

Table G-2: **Xsi::Loader member functions**

Activity	XSI Function	`Xsi::Loader` Member Function
Open the design	`xsi_open`	`open`
Fetch a port ID	`xsi_get_port_number`	`get_port_number`

Table G-2: **Xsi::Loader member functions *(Cont'd)***

Activity	XSI Function	`Xsi::Loader` Member Function
Set an input port value	`xsi_put_value`	`put_value`
Run the simulation	`xsi_run`	`run`
Fetch an output port value	`xsi_get_value`	`get_value`
Close the design	`xsi_close`	`close`

You can find the example C++ programs that use XSI in your Vivado Simulator installation at the following location:

```
<Vivado Installation Root>/examples/xsim/<HDL language>/xsi
```

Compiling Your C/C++ Program

You can use the XSI example programs as a guideline. Each example supplies one or two scripts for compiling and running the example. Refer to your compiler's documentation for details on compiling a program. On Linux, compiling and running is a two-step process.

1. In a C shell, source set_env.csh
2. Invoke run.csh

On Windows, simply run the batch file `run.bat`.

Note the following from the scripts:

1. The compilation lines specify (via `-I`) the inclusion of the directory containing the `xsi.h` include file.
2. There is no mention of the design shared library or kernel shared library during the compilation of a C++ program.

The XSI include file resides at the following location:

```
<Vivado Installation Root>/data/xsim/include/xsi.h
```

Preparing the Design Shared Library

The last step for producing a working XSI-based C/C++ program involves the compilation of a HDL design and packaging it together with the Vivado Simulator to become your design shared library. You may repeat this step whenever there is a change in HDL designs source code.

CAUTION! *If you intend to rebuild the design shared library for your C/C++ program while your program continues to run, be sure to close the design in your program before executing this step.*

Create your design shared library by invoking xelab on the HDL design and including the -dll switch to instruct xelab to produce a shared library instead of the usual snapshot for use with the Vivado Simulator's user interface.

For example:

Type the following in the Linux command line to create a design shared library at `./xsim.dir/design/xsimk.so`:

```
xelab work.top1 work.top2 -dll -s design
```

where `work.top1` and `work.top2` are the top module names and `design` is the snapshot name.

See xelab, xvhdl, and xvlog xsim Command Options for more details on compiling an HDL design.

XSI Function Reference

This section presents each of the XSI API functions in plain (direct C call) and `Xsi::Loader` member function forms. The plain form functions take an `xsiHandle` argument, whereas the member functions do not take this argument. The `xsiHandle` contains state information about the opened HDL design. The plain form `xsi_open` produces the `xsiHandle`. `Xsi::Loader` contains an `xsiHandle` internally.

xsi_close

```
void xsi_close(xsiHandle design_handle);
void Xsi::Loader::close();
```

This function closes an HDL design, freeing the memory associated with the design. Call this function to end the simulation.

xsi_get_error_info

```
const char* xsi_get_error_info(xsiHandle design_handle);
const char* Xsi::Loader::get_error_info();
```

This function returns a string description of the last error encountered.

xsi_get_port_number

```
XSI_INT32 xsi_get_port_number(xsiHandle design_handle, const char* port_name);
int Xsi::Loader::get_port_number(const char* port_name);
```

This function returns an integer ID for the requested top-level port of the HDL design. You may subsequently use the ID to specify the port in `xsi_get_value` and `xsi_put_value` calls. `port_name` is the name of the port and is case sensitive for Verilog and case insensitive for VHDL. The function returns `-1` if no port of the specified name exists.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so","librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
```

xsi_get_status

```
XSI_INT32 xsi_get_status(xsiHandle design_handle);
int Xsi::Loader::get_status();
```

This function returns the status of the simulation. The status may be equal to one of the following identifiers:

Table G-3: **Xsi Simulation Status Identifiers**

Status code Identifiers	Description
`xsiNormal`	No error
`xsiError`	The simulation has encountered an HDL run-time error
`xsiFatalError`	The simulation has encountered an error condition for which the Vivado Simulator cannot continue.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so","librdi_simulator_kernel.so");
...
if (loader.get_status() == xsiError)
    printf("HDL run-time error encountered.\n");
```

xsi_get_value

```
void xsi_get_value(xsiHandle design_handle, XSI_INT32 port_number, void* value);
int Xsi::Loader::get_value(int port_number, void* value);
```

This function fetches the value of the port indicated by port ID `port_number`. The value is placed in the memory buffer to which value points. See xsi_get_port_number for information on obtaining an ID for a port.

IMPORTANT: *Your program must allocate sufficient memory for the buffer before calling the function.See Vivado Simulator VHDL Data Format and Vivado Simulator Verilog Data Format to determine the necessary size of the buffer.*

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Buffer for value of port "count"
s_xsi_vlog_logicval count_val = {0X00000000, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so","librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
loader.get_value(count, &count_val);
```

xsi_open

```
typedef struct t_xsi_setup_info {
    char* logFileName;
    char* wdbFileName;
} s_xsi_setup_info, *p_xsi_setup_info;

xsiHandle xsi_open(p_xsi_setup_info setup_info);
void Xsi::Loader::open(p_xsi_setup_info setup_info);
bool Xsi::Loader::isopen() const;
```

This function opens an HDL design for simulation. To use this function, you must first initialize an `s_xsi_setup_info` struct to pass to the function. Use `logFileName` for the name of the simulation log file, or `NULL` to disable logging. If waveform tracing is on (see xsi_trace_all), `wdbFileName` is the name of the output WDB (waveform database) file. Use `NULL` for the default name of `xsim.wdb`. If the waveform tracing is off, the Vivado Simulator ignores the `wdbFileName` field.

TIP: *To protect your program from future changes to the XSI API, Xilinx recommends that you zero out the `s_xsi_setup_info` struct before filling in the fields, as shown in the Example.*

The plain (non-loader) form of the function returns an `xsiHandle`, a C object containing process state information about the design, to be used with all other plain-form XSI functions. The loader form of the function has no return value. However, you may check whether the loader has opened a design by querying the `isopen` member function, which returns true if the `open` member function had been invoked.

Example

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so","librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
info.logFileName = NULL;
char wdbName[] = "test.wdb";  // make a buffer for holding the string "test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
```

xsi_put_value

```
void xsi_put_value(xsiHandle design_handle, XSI_INT32 port_number,  void* value);
void Xsi::Loader::put_value(int port_number, const void* value);
```

This function deposits the value stored in `value` onto the port specified by port ID `port_number`. See xsi_get_port_number for information on obtaining an ID for a port. `value` is a pointer to a memory buffer that your program must allocate and fill. See the Vivado Simulator VHDL Data Format and Vivado Simulator Verilog Data Format for information on the proper format of value.

CAUTION! *For maximum performance, the Vivado Simulator performs no checking on the size or type of the value you pass to `xsi_put_value`. Passing a value to `xsi_put_value` which does not match the size and type of the port may result in unpredictable behavior of your program and the Vivado Simulator.*

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Hard-coded Buffer for a 1-bit "1" Verilog 4-state value
const s_xsi_vlog_logicval one_val  = {0X00000001, 0X00000000};

Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so","librdi_simulator_kernel.so");
...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val);  // set clk to 1
```

xsi_restart

```
void xsi_restart(xsiHandle design_handle);
void Xsi::Loader:: restart();
```

This function resets the simulation to simulation time 0.

xsi_run

```
void xsi_run(xsiHandle design_handle, XSI_UINT64 time_ticks);
void Xsi::Loader::run(XSI_INT64 step);
```

This function runs the simulation for the given amount of time specified in kernel precision units. A kernel precision unit is the smallest unit of time precision specified among all HDL source files of the design. For example, if a design has two source files, one of which that specifies a precision of 1 ns and the other specifies a precision of 1 ps, the kernel precision unit becomes 1 ps, as that time unit is the smaller of the two.

A Verilog source file may specify the time precision using the `` `timescale `` directive.

Example:

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

In this example, the time unit after the / (1 ps) is the time precision. VHDL has no equivalent of `` `timescale ``.

You may additionally adjust the kernel precision unit through the use of the `xelab` command-line options `--timescale`, `--override_timeprecision`, and `--timeprecision_vhdl`. See xelab, xvhdl, and xvlog xsim Command Options for information on the use of these command-line options.

Note: `xsi_run` blocks until the specified simulation run time has elapsed. Your program and the Vivado Simulator share a single thread of execution.

xsi_trace_all

```
void xsi_trace_all(xsiHandle design_handle);
void Xsi::Loader:: trace_all();
```

Call this function after `xsi_open` to turn on waveform tracing for all signals of the HDL design. Running the simulation with waveform tracing on causes the Vivado Simulator to produce a waveform database (WDB) file containing all events for every signal in the design. The default name of the WDB file is `xsim.wdb`. To specify a different WDB file name, set the `wdbFileName` field of the `s_xsi_setup_info` struct when calling `xsi_open`, as shown in the example code.

Example code:

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so","librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
char wdbName[] = "test.wdb";  // make a buffer for holding the string "test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
loader.trace_all();
```

After the simulation completes, you can open the WDB file in Vivado to examine the waveforms of the signals. See Opening a Previously Saved Simulation Run for more information on how to view WDB files in Vivado.

IMPORTANT: *When compiling the HDL design, you must specify* `-debug all` *or* `-debug typical` *on the xelab command line. The Vivado Simulator will not record the waveform data without the* `-debug` *command line option.*

Vivado Simulator VHDL Data Format

This section describes how to convert between VHDL values and the format of the memory buffers to use with the XSI functions xsi_get_value and xsi_put_value.

IEEE std_logic Type

A single bit of VHDL `std_logic` and `std_ulogic` is represented in C/C++ as a single byte (char or unsigned char). Table G-4 shows the values of `std_logic/std_ulogic` and their C/C++ equivalents.

Table G-4: **std_logic/std_ulogic values and their C/C++ Equivalents**

`std_logic` **Value**	**C/C++ Byte Value (Decimal)**
‘U’	0
‘X’	1
‘0’	2
‘1’	3
‘Z’	4
‘W’	5
‘L’	6
‘H’	7
‘_’	8

Example code:

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : std_logic;
const char one_val = 3; // C encoding for std_logic '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val);  // set clk to 1
```

VHDL bit Type

A single bit of VHDL `bit` type is represented in C/C++ as a single byte. Table G-5 shows the values of `bit` and their C/C++ equivalents.

Table G-5: **Values of bit and their C/C++ equivalents**

`bit` **Value**	**C/C++ Byte Value (Decimal)**
`‘0’`	0
`‘1’`	1

Example code:

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : bit;
const char one_val = 1; // C encoding for bit '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val);  // set clk to 1
```

VHDL character Type

A single VHDL `character` value is represented in C/C++ as a single byte. VHDL `character` values are exactly identical to C/C++ `char` literals and are also equal to their ASCII numeric values. For example, the VHDL character value ‘`m`’ is equivalent to the C/C++ `char` literal ‘`m`’ or decimal value 109.

Example code:

```
// Put a 'T' on signal "myChar," where "myChar" is defined as
// signal myChar : character;
const char tVal = 'T';
int myChar = loader.get_port_number("myChar");
loader.put_value(myChar, &tVal);
```

VHDL integer Type

A single VHDL `integer` value is represented in C/C++ as an `int`.

Example code:

```
// Put 1234 (decimal) on signal "myInt," where "myInt" is defined as
// signal myInt : integer;
const int intVal = 1234;
int myInt = loader.get_port_number("myInt");
loader.put_value(myInt, &intVal);
```

VHDL real Type

A single VHDL `real` value is represented in C/C++ as a `double`.

Example code:

```
// Put 3.14 on signal "myReal," where "myReal" is defined as
// signal myReal : real;
const double doubleVal = 3.14;
int myReal = loader.get_port_number("myReal");
loader.put_value(myReal, &doubleVal);
```

VHDL Array Types

A VHDL array is represented in C/C++ as an array of whatever C/C++ type represents the element type of the VHDL array. Table G-6 shows the examples of VHDL arrays and their C/C++ equivalent types.

Table G-6: **VHDL Arrays and their C/C++ Equivalent Types**

VHDL Array Type	C/C++ Array Type
`std_logic_vector` (array of `std_logic`)	char [ ]
`bit_vector` (array of `bit`)	char [ ]
`string` (array of `character`)	char [ ]
array of `integer`	int [ ]
array of `real`	double [ ]

VHDL arrays are organized in C/C++ with the left index of the VHDL array mapped to C/C++ array element 0 and the right index mapped to C/C++ element <array size> - 1.

Table G-7: **VHDL Array mapping to C/C++**

C/C++ Array Index	0	1	2	...	*<array size>* - 1
VHDL array(*left* TO *right*) Index	*left*	*left* + 1	*left* + 2	...	*right*
VHDL array(*left* DOWNTO *right*) Index	*left*	*left* – 1	*left* – 2	...	*right*

Example code:

```
// For the following VHDL definitions

// signal slv : std_logic_vector(7 downto 0);
// signal bv : bit_vector(3 downto 0);
// signal s : string(1 to 11);
// type IntArray is array(natural range <>) of integer;
// signal iv : IntArray(0 to 3);

// do the following assignments
//
// slv <= "11001010";
// bv <= B"1000";
// s <= "Hello world";
// iv <= (33, 44, 55, 66);
```

```
const unsigned char slvVal[] = {3, 3, 2, 2, 3, 2, 3, 2}; // 3 = '1', 2 = '0'
loader.put_value(slv, slvVal);
const unsigned char bvVal[] = {1, 0, 0, 0};
loader.put_value(bv, bvVal);
const char sVal[] = "Hello world"; // ends with extra '\0' that XSI ignores
loader.put_value(s, sVal);
const int ivVal[] = {33, 44, 55, 66};
loader.put_value(iv, ivVal);
```

Vivado Simulator Verilog Data Format

Verilog logic data is encoded in C/C++ using the following struct, defined in xsi.h:

```
typedef struct t_xsi_vlog_logicval {
    XSI_UINT32 aVal;
    XSI_UINT32 bVal;
} s_xsi_vlog_logicval, *p_xsi_vlog_logicval;
```

Each four-state bit of Verilog value occupies one bit position in aVal and the corresponding bit position in bVal.

Table G-8: **Verilog Value Mapping**

Verilog Value	aVal Bit Value	bVal Bit Value
0	0	0
1	1	0
X	1	1
Z	0	1

For two-state SystemVerilog bit values, an aVal bit holds the bit value, and the corresponding bVal bit is unused. Xilinx recommends that you zero out bVal when composing two-state values for xsi_put_value.

Verilog vectors are organized in C/C++ with the right index of the Verilog vector mapped to `aVal/bVal` bit position 0 and the left index mapped to `aVal/bVal` bit position <vector size> - 1

Table G-9: **Verilog Vectors**

aVal/bVal Bit Position	*<vector size>* to 31	*<vector size>* - 1	*<vector size>* - 2	...	1	0
Index of `wire [left:right] vec` **(where left > right)**	*unused*	*left*	*left* - 1	...	*right* + 1	*right*
Index of `wire [left:right] vec` **(where left < right)**	*unused*	*left*	*left* + 1	...	*right* - 1	*right*

For example, Table G-10 shows the Verilog and C/C++ equivalents of the following Verilog vector.

```
wire [7:4] w = 4'bXX01;
```

Table G-10: **Verilog and C/C++ Equivalents of the Verilog Vector**

Verilog Bit Index				**7**	**6**	**5**	**4**
Verilog Bit Value				X	X	0	1
C/C++ Bit Position	**31**	**...**	**4**	**3**	**2**	**1**	**0**
`aVal` Bit Value	unused	...	unused	1	1	0	1
`bVal` Bit Value	unused	...	unused	1	1	0	0

The C/C++ representation of a Verilog vector with more than 32 elements is an array of `s_xsi_vlog_logicval`, for which the right-most 32 bits of the Verilog vector maps to element 0 of the C/C++ array. The next 32 bits of the Verilog vector maps to element 1 of the C/C++ array, and so forth. For example, Table G-11 shows the mapping of Verilog vector

```
wire [2:69] vec;
```

to the C/C++ array

```
s_xsi_vlog_logicval val[3];
```

Table G-11: **Verilog Index Range**

Verilog Index Range	**C/C++ Array Element**
vec[38:69]	val[0]
vec[6:37]	val[1]
vec[2:5]	val[3]

Hence, `vec[2]` maps to `val[3]` bit position 3, and `vec[69]` maps to `val[0]` bit position 0.

A multi-dimensional Verilog array maps to the bits of a `s_xsi_vlog_logicval` or `s_xsi_vlog_logicval` array as if the Verilog array were flattened in row-major order before mapping to C/C++.

For example, the two-dimensional array

```
reg [7:0] mem[0:1];
```

is treated as if copied to a vector before mapping to C/C++:

```
reg [15:0] vec;
vec[7:0] = mem[1];
vec[8:15] = mem[0];
```

Appendix H

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

1. Vivado® Design Suite Documentation: *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
2. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
3. *Vivado Design Suite User Guide: Designing with IP* (UG896)
4. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
5. *Vivado Design Suite User Guide: Using the Tcl Scripting Capabilities* (UG894)
6. *Writing Efficient Testbenches* (XAPP199)
7. *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* (UG953)
8. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
9. *Vivado Design Suite User Guide: Power Analysis and Optimization* (UG907)
10. *Vivado Design Suite User Guide: Using Constraints* (UG903)
11. Vivado Design Suite Tutorial: Simulation *(*UG937)

12. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
13. *Vivado Design Suite Properties Reference Guide* (UG912)

Links to Additional Information on Third-Party Simulators

14. For more information on:
    - Questa Advanced Simulator/ModelSim simulators:
        - www.mentor.com/products/fv/questa/
        - www.mentor.com/products/fv/modelsim/
    - Cadence IES simulators:
        - www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx
    - Synopsys VCS simulators:
        - www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx
    - Active-HDL Simulators:
        - https://www.aldec.com/support/resources/documentation/articles/1579
    - Riviera PRO Simulators:
        - https://www.aldec.com/support/resources/documentation/articles/1525

Links to Language and Encryption Support Standards

15. *IEEE Standard VHDL Language Reference Manual* (IEEE-STD-1076-1993)
16. *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1364-2001)
17. *IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language* (IEEE-STD-1800-2009)
18. *Standard Delay Format Specification (SDF)* (Version 2.1)
19. *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)* (IEEE-STD-P1735).

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
2. Vivado Design Suite Hands-on Introductory Workshop Training Course
3. Vivado Design Suite Quick Take Video: Logic Simulation
4. Vivado Design Suite QuickTake Video Tutorials

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at 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.

PCI, PCIe and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.