

Vivado Design Suite User Guide

Logic Simulation

UG900 (v2015.2) June 24, 2015

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/24/2015	2015.2	<p>Chapter 2, Preparing for Simulation</p> <ul style="list-style-type: none"> Updated Using Verilog UNIFAST Library section. <p>Chapter 7, Using Vivado Simulator in Batch or Scripted Mode</p> <ul style="list-style-type: none"> Added new xelab Command Syntax Options and updated xelab, xvhd, and xvlog Command Options table. <p>Chapter 8, Using Third-Party Simulators</p> <ul style="list-style-type: none"> Updated Compiling Simulation Libraries section. Added Generating a Simulator Specific Run Directory section. <p>Global Changes</p> <ul style="list-style-type: none"> Updated figures to match feature changes in the 2015.2 release; some figures enhanced for improved viewing.
04/01/2015	2015.1	<p>Chapter 1, Logic Simulation Overview</p> <ul style="list-style-type: none"> Added Aldec Active-HDL and Riviera-PRO to the list of supported simulators. <p>Chapter 2, Preparing for Simulation</p> <ul style="list-style-type: none"> In section UNIFAST Library GTXE2_CHANNEL/GTXE2_COMMON, added note to bypass the DRP production reset sequence when using the UNIFAST model. <p>Chapter 3, Understanding Vivado Simulator</p> <ul style="list-style-type: none"> Added description of new right-click options for items in the Objects Window. <p>Chapter 5, Debugging a Design with Vivado Simulator</p> <ul style="list-style-type: none"> Added new section on Cross Probing Signals in the Object, Wave, and Text Editor Windows. <p>Chapter 7, Using Vivado Simulator in Batch or Scripted Mode</p> <ul style="list-style-type: none"> Added new xelab Command Syntax Options: <ul style="list-style-type: none"> -Oenable_pass_through_elimination, -Odisable_pass_through_elimination, -Oenable_always_combine, -Odisable_always_combine <p>Chapter 8, Using Third-Party Simulators</p> <ul style="list-style-type: none"> Added Riviera PRO simulator (Aldec) to list of supported third-party simulators. <p>Appendix D, System Verilog Constructs Supported by the Vivado Simulator</p> <ul style="list-style-type: none"> Noted newly supported constructs in Table D-1, Synthesizable Set of System Verilog 1800-2009. Added Table D-2, Supported Dynamic Types Constructs: Early Access. <p>Appendix E, Direct Programming Interface (DPI) in Vivado Simulator</p> <ul style="list-style-type: none"> Updated Table E-2, Data Types Allowed on the C-SystemVerilog Boundary <p>Global Changes</p> <ul style="list-style-type: none"> Book reorganized to reflect design flow structure. Extensive enhancements to content. Updated figures to match feature changes in the 2015.1 release; some figures enhanced for improved viewing.

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	13
Adding or Creating Simulation Source Files	14
Using Xilinx Simulation Libraries.	16
Using Simulation Settings	25
Understanding the Simulator Language Option	28
Recommended Simulation Resolution	30
Generating a Netlist.	30

Chapter 3: Understanding Vivado Simulator

Introduction	32
Vivado Simulator Features	32
Running the Vivado Simulator	33
Running Functional and Timing Simulation	45
Saving Simulation Results	48
Distinguishing Between Multiple Simulation Runs	48
Closing a Simulation.	49
Adding a Simulation Start-up Script File.	49
Viewing Simulation Messages.	51
Using the launch_simulation Command	52

Chapter 4: Analyzing Simulation Waveforms

Introduction	54
Using Wave Configurations and Windows	54
Opening a Previously Saved Simulation Run	55

Understanding HDL Objects in Waveform Configurations	57
Customizing the Waveform.....	60
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.....	87
Using the report_drivers Tcl Command	88
Using the Value Change Dump Feature	89
Using the log_wave Tcl Command	90
Cross Probing Signals in the Object, Wave, and Text Editor Windows	91
Chapter 6: Handling Special Cases	
Using Global Reset and 3-State.....	93
Delta Cycles and Race Conditions.....	95
Using the ASYNC_REG Constraint.....	96
Simulating Configuration Interfaces.....	97
Disabling Block RAM Collision Checks for Simulation	101
Dumping the Switching Activity Interchange Format File for Power Analysis	102
Simulating a Design with AXI Bus Functional Models	102
Skipping Compilation or Simulation	102
Chapter 7: Using Vivado Simulator in Batch or Scripted Mode	
Introduction	104
Vivado Simulator Command Line Steps	104
Elaborating and Generating a Design Snapshot, -xelab	106
Simulating the Design Snapshot, xsim	116
Example of Running Vivado Simulator in Standalone Mode	118
Project File (.prj) Syntax	119
Predefined Macros.....	120
Library Mapping File (xsim.ini)	120
Running Simulation Modes	122
Using Tcl Commands and Scripts	124
Chapter 8: Using Third-Party Simulators	
Introduction	125

Preparing for Simulation Using Third-Party Tools	125
Tcl Mode	130
Running Simulation with Third-Party Tools	132
After Running Simulation with Third-Party Tools	140
Simulating IP.	142
Using the Verilog UNIFAST Library	142
Simulating a Design with AXI Bus Functional Models	143
Using a Custom DO File During an Integrated Simulation Run	143
Generating a Simulator Specific Run Directory	144
Appendix A: Value Rules in Vivado Simulator Tcl Commands	
Introduction	145
String Value Interpretation	145
Vivado Design Suite Simulation Logic.	146
Appendix B: Vivado Simulator Mixed Language Support and Language Exceptions	
Introduction	147
Using Mixed Language Simulation	147
VHDL Language Support Exceptions.	154
Verilog Language Support Exceptions	155
Appendix C: Vivado Simulator Quick Reference Guide	
Introduction	159
Appendix D: System Verilog Constructs Supported by the Vivado Simulator	
Introduction	163
Dynamic Types: Early Access.	171
Appendix E: Direct Programming Interface (DPI) in Vivado Simulator	
Introduction	172
Compiling C Code	172
Description of the xsc Compiler	172
Binding Compiled C Code to SystemVerilog Using xelab.	174
Data Types Allowed on the Boundary of C and SystemVerilog	174
Mapping for User-Defined Types	176
Support for svdpi.h functions	178
Examples.	178
DPI Examples Shipped with the Vivado Design Suite	184

Appendix F: Additional Resources and Legal Notices

Xilinx Resources	185
Solution Centers	185
Documentation References	185
Links to Additional Information on Third-Party Simulators	186
Links to Language and Encryption Support Standards	186
Training Resources	186
Please Read: Important Legal Notices	187

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 3](#)]
 - *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [[Ref 11](#)]
-

Simulation Flow

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

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

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

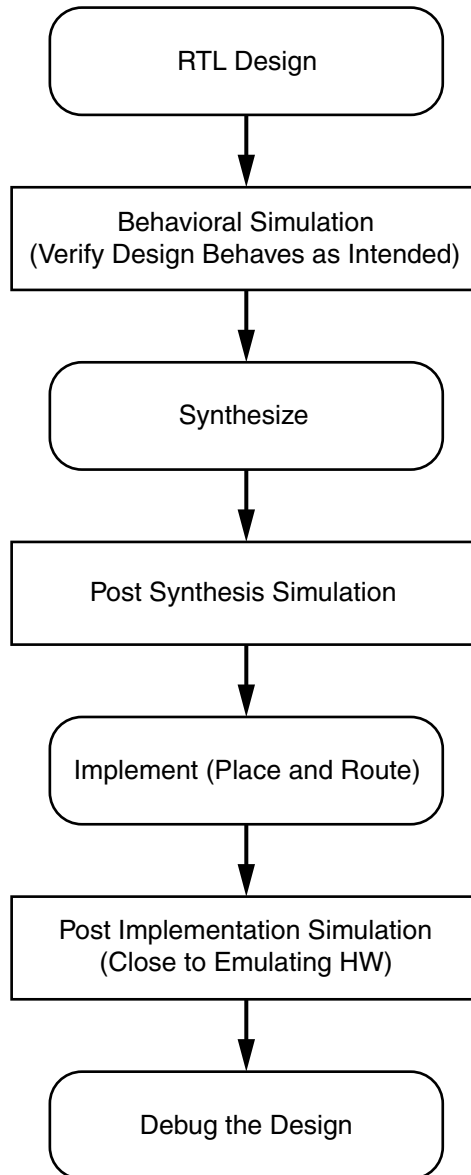


Figure 1-1: **Simulation Flow**

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 at system level.

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

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

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

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 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 QuestaSim/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, Verilog, SystemVerilog, and Standard Delay Format (SDF) [Ref 17]. See also [Appendix D, System Verilog Constructs Supported by the Vivado Simulator](#).
- IEEE standards for language and encryption. See *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)*, (P1735) [Ref 18].

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.

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 93](#).

For more information about test benches, see *Writing Efficient TestBenches (XAPP199)* [Ref 5].



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](#).

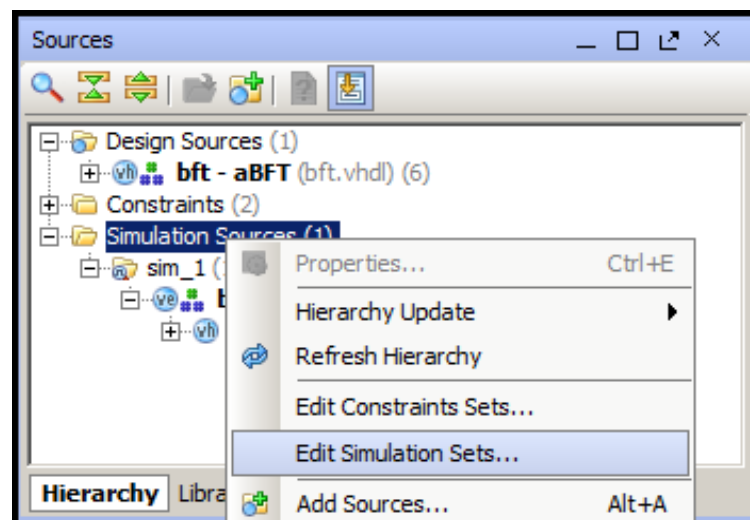


Figure 2-1: Edit Simulation Sets Option

The Add or Create Simulation Sources wizard opens.

- 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.

- 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

Table 2-1: Simulation Libraries (Cont'd)

Library Name	Description	VHDL Library Name	Verilog Library Name
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. See [Method 1: Using the complete UNIFAST library \(Recommended\)](#), page 24 for more information.

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

VHDL UNISIM Library

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

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

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 6].

VHDL UNIMACRO Library

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

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

Verilog UNIMACRO Library

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

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



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

SIMPRIM Library

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



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



TIP: If you are a VHDL user, you can run post synthesis and post implementation functional simulation (in which case no standard default format (SDF) annotation is required and the simulation netlist uses the `UNISIM` library). You can create the netlist using the [write_vhdl](#) Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 7].

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 18]. 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		
QuestaSim		
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
```

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

UNIFAST Library

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



IMPORTANT:

This model cannot be used for timing-driven simulations.

UNIFAST libraries cannot be used for sign-off simulations because the library components do not have all the checks/features that are available in a full model



RECOMMENDED: Use the UNIFAST library for initial verification of the design and then 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.01 use unifast_ver.MMCME2;`)

Note: For ModelSim (vsim) only `-genblk` is added to hierarchy name. (For example: `instance testbench.genblk1.inst.genblk1.01 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.01 use unifast_ver.DSP48E1;
//If using ModelSim or Questa, add in the genblk to the name
(instance testbench.genblk1.inst.genblk1.01 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;
    
```

Using Simulation Settings

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

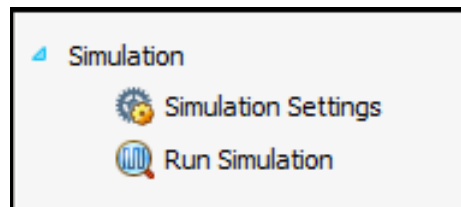


Figure 2-2: Flow Navigator Simulation Options

- Simulation Settings: Opens the Project Settings dialog box where you can select and configure the Vivado simulator. See [Vivado Simulator Project Settings, page 27](#).
- 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.

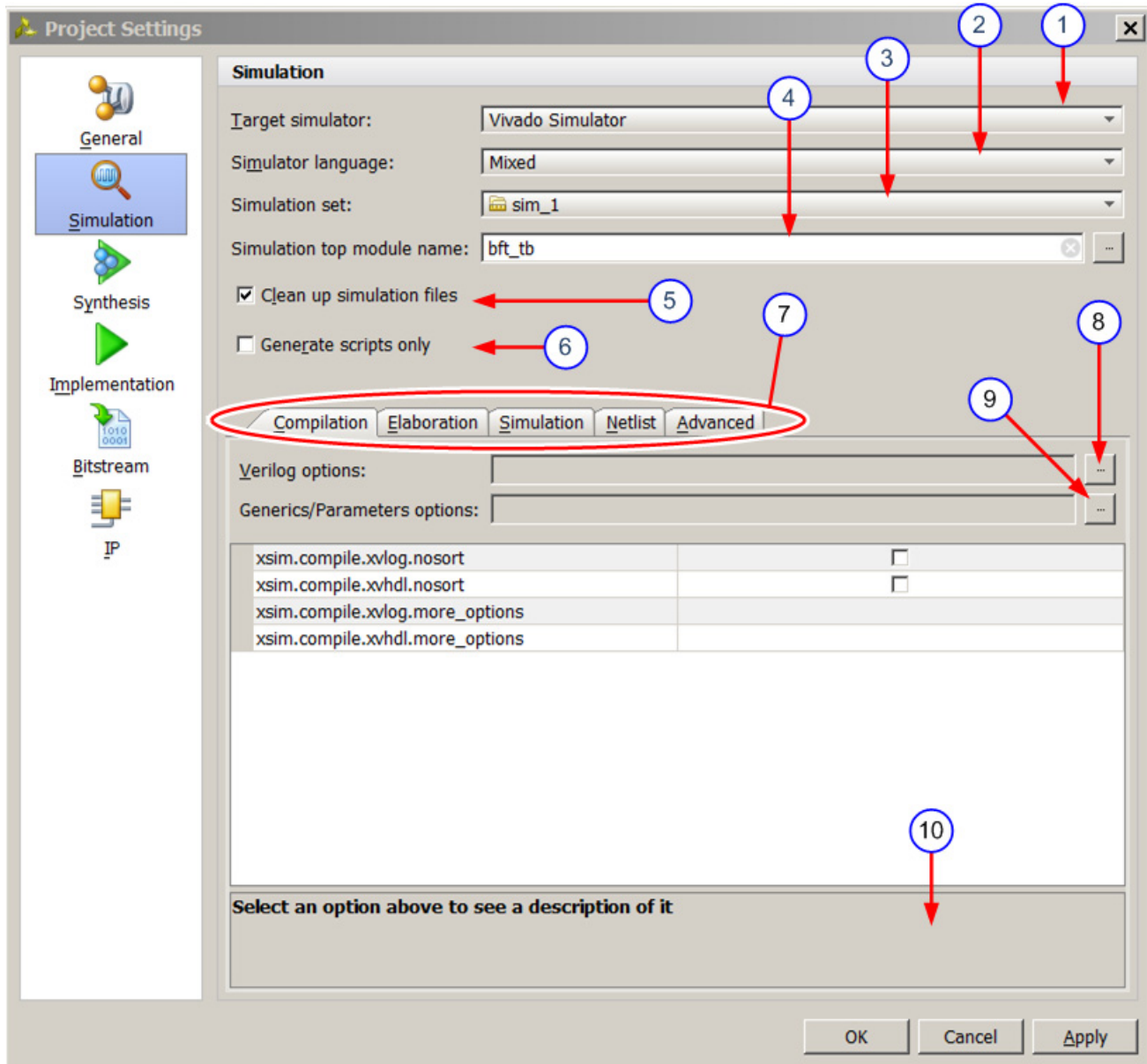
To use the corresponding Tcl command, type: `launch_simulation`.



TIP: *This command has a `-scripts_only` option that writes a script to run the Vivado simulator.*

Vivado Simulator Project Settings

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



- | | |
|---|--|
| <ul style="list-style-type: none"> 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. | <ul style="list-style-type: none"> 7 Select tabs to set options in the respective categories. 8 Compilation tab only. Browse to set include path or to define macros. 9 Compilation tab only. Browse to select generics/parameters location. 10 For each tab, an option list appears in the window, and when selected, an option description displays. |
|---|--|

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.

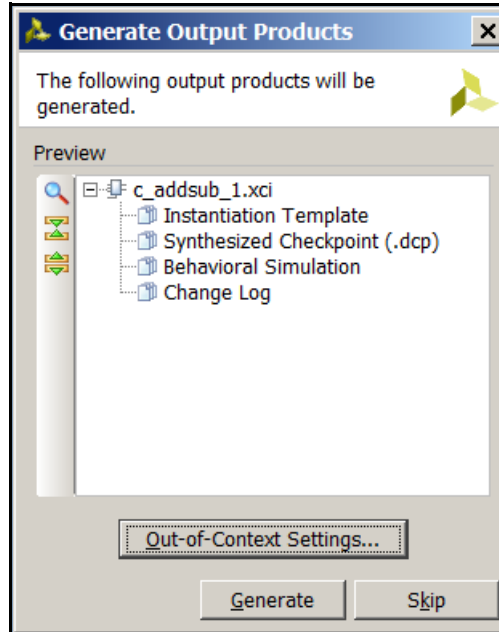


Figure 2-4: Dialog Box Showing Generate Synthesized Checkpoint (.dcp) Option

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

Table 2-6: Function of `simulator_language` Property

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

Notes:

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

```
Tcl: set_property target_language VHDL [current_project]
```

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.*

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

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

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 10] for a step-by-step demonstration of how to run Vivado simulation.

Vivado Simulator Features

The Vivado simulator supports the following features:

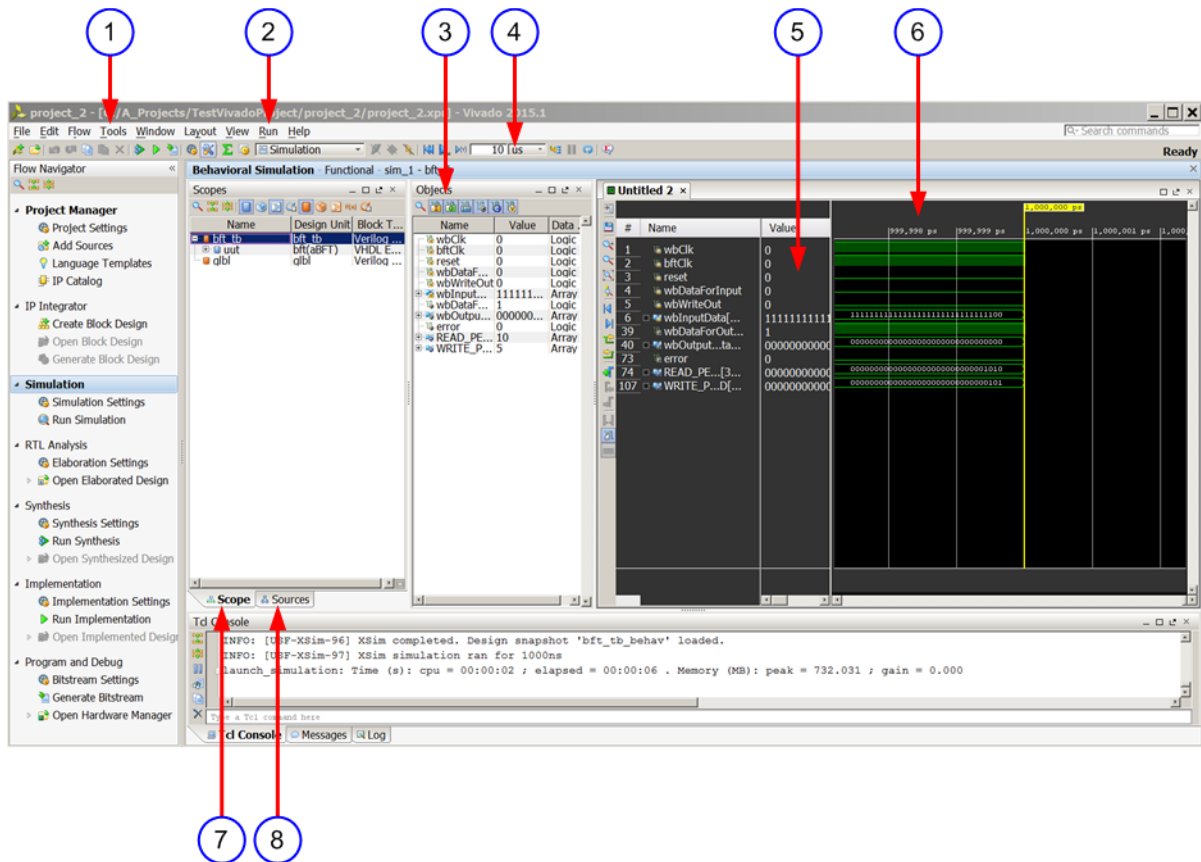
- Source code debugging (step, breakpoint, current value display)
- SDF annotation for timing simulation
- VCD dumping
- SAIF dumping for power analysis and optimization
- Native support for HardIP blocks (such as serial transceivers and PCIe®)
- Multi-threaded compilation
- Mixed language (VHDL, Verilog, 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.



- | | |
|----------------------|------------------|
| 1 Main Toolbar | 5 Wave Objects |
| 2 Run Menu | 6 Wave window |
| 3 Objects Window | 7 Scopes Window |
| 4 Simulation Toolbar | 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](#).

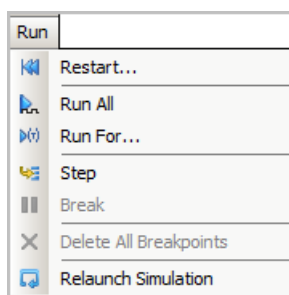


Figure 3-2: Simulation Run Menu Options

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.

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.

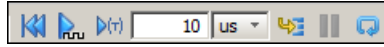


Figure 3-3: Simulation Toolbar

These are the same buttons labeled in Figure 3-2, page 34, 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).

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.

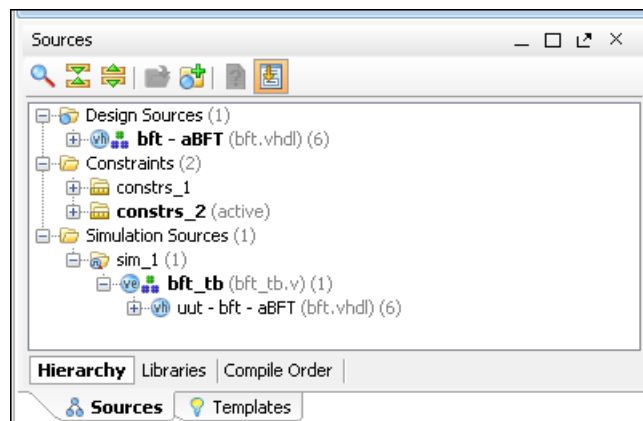


Figure 3-4: Sources Window

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.

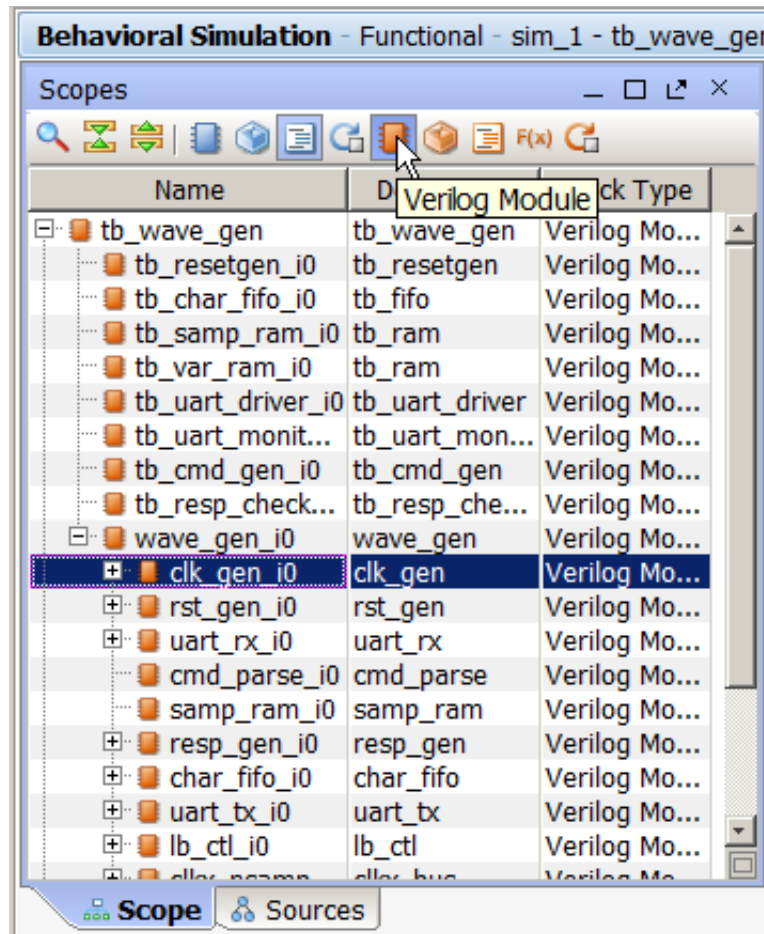



Figure 3-5: Scopes Window

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 `tb_wave_gen` scope but also `uart` (even though `uart` 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.



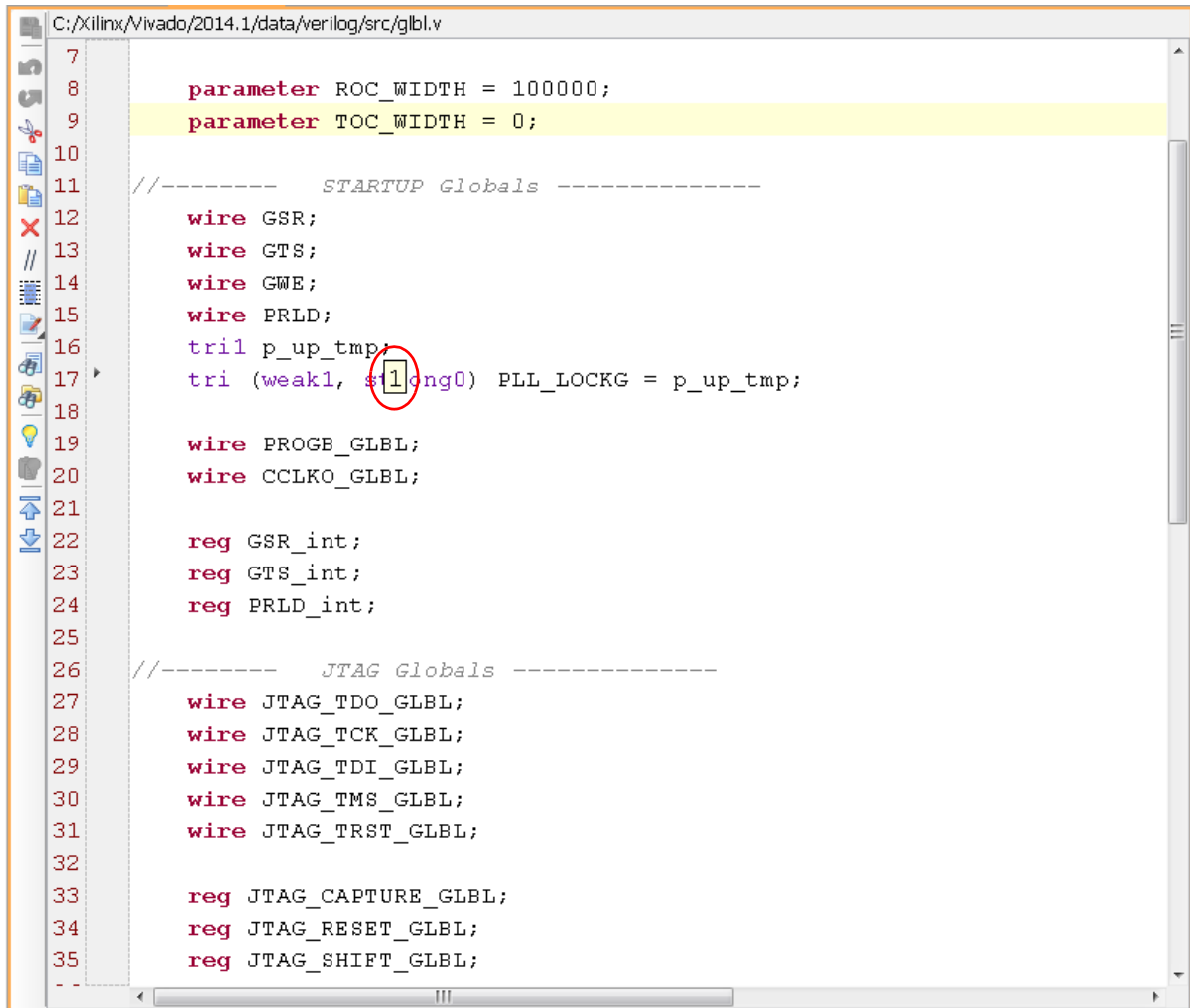
IMPORTANT: The Wave window displays the value changes of an object over time, starting from the simulation time at which the object was added.



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 90](#).

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.




```

C:/Xilinx/Vivado/2014.1/data/verilog/src/glbl.v
7
8     parameter ROC_WIDTH = 100000;
9     parameter TOC_WIDTH = 0;
10
11 //----- STARTUP Globals -----
12     wire GSR;
13     wire GTS;
14     wire GWE;
15     wire PRLD;
16     tri1 p_up_tmp;
17     tri (weak1, s1ong0) PLL_LOCKG = p_up_tmp;
18
19     wire PROGB_GLBL;
20     wire CCLKO_GLBL;
21
22     reg GSR_int;
23     reg GTS_int;
24     reg PRLD_int;
25
26 //----- JTAG Globals -----
27     wire JTAG_TDO_GLBL;
28     wire JTAG_TCK_GLBL;
29     wire JTAG_TDI_GLBL;
30     wire JTAG_TMS_GLBL;
31     wire JTAG_TRST_GLBL;
32
33     reg JTAG_CAPTURE_GLBL;
34     reg JTAG_RESET_GLBL;
35     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](#).

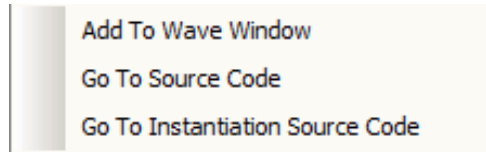
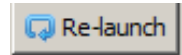


Figure 3-8: Context Menu in Scopes Window



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.

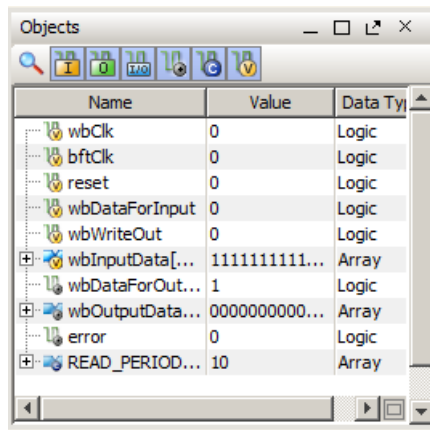


Figure 3-9: HDL Objects Window








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.
	Input signals
	Output signals
	Input/Output signals
	Internal signals
	Constant signals
	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 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.

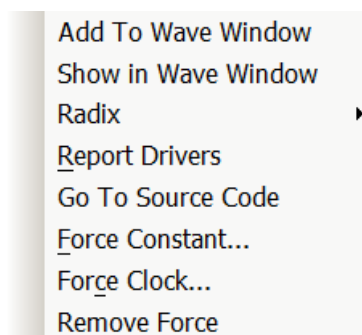


Figure 3-10: Context Menu in Objects Window

- **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.
- **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:
 - Binary (default)
 - 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.*

- **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.*

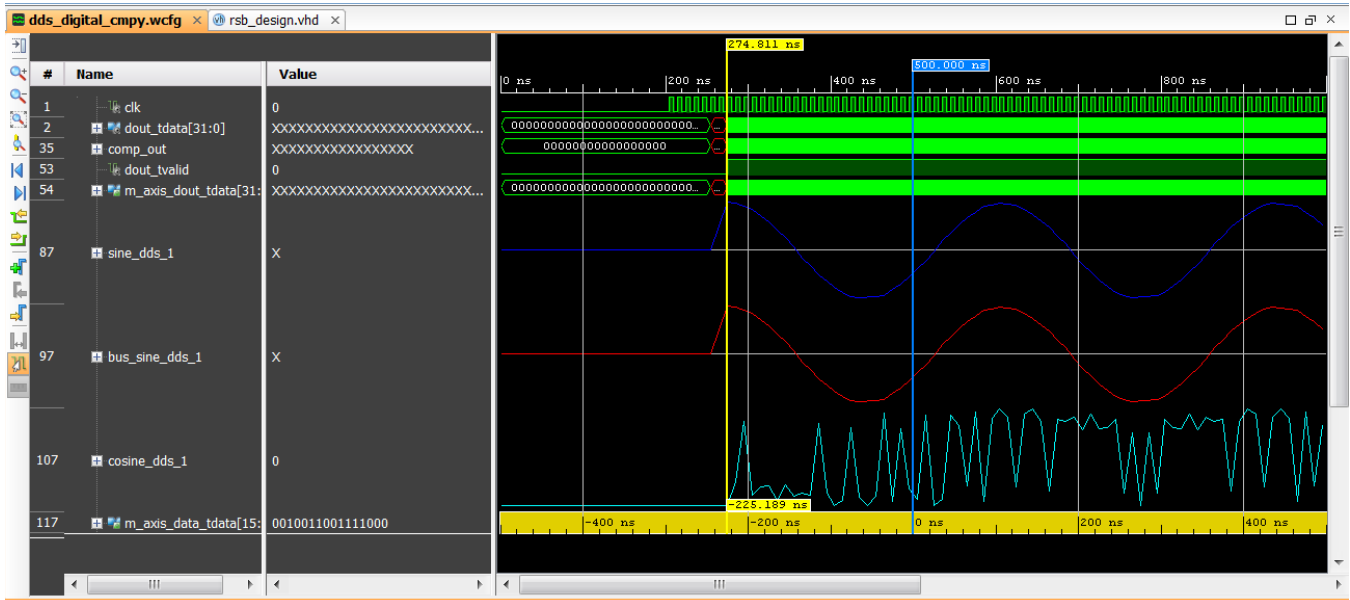


Figure 3-11: Wave Window

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.

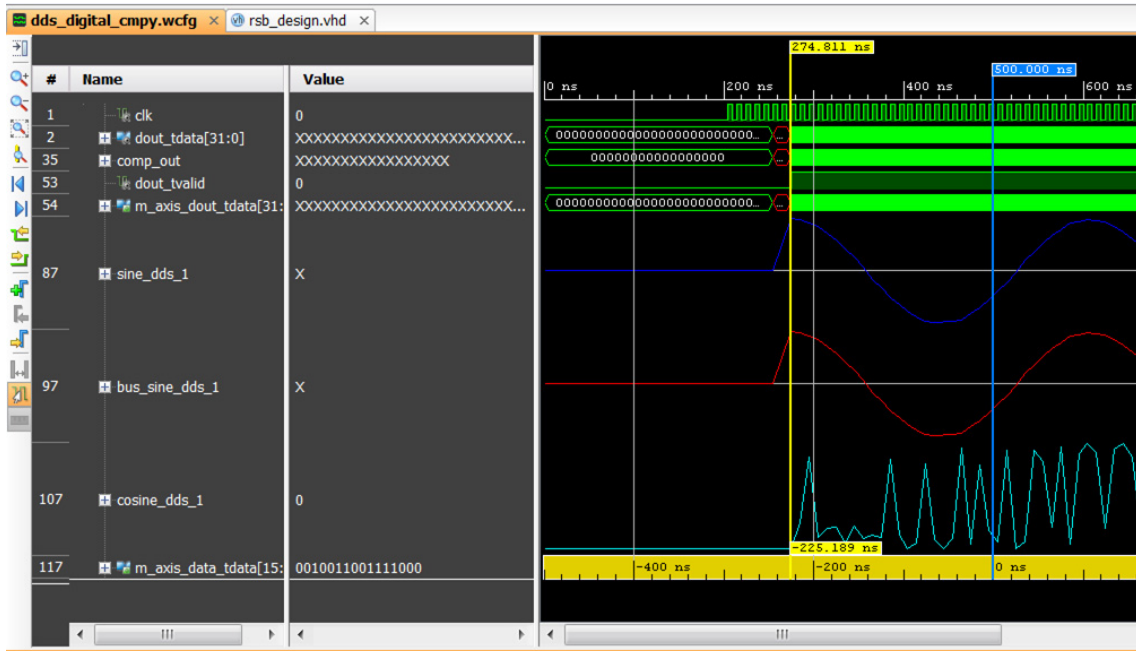


Figure 3-12: HDL Objects in Waveform

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

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 48](#) and [Creating a New Wave Configuration, page 54](#) 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.*

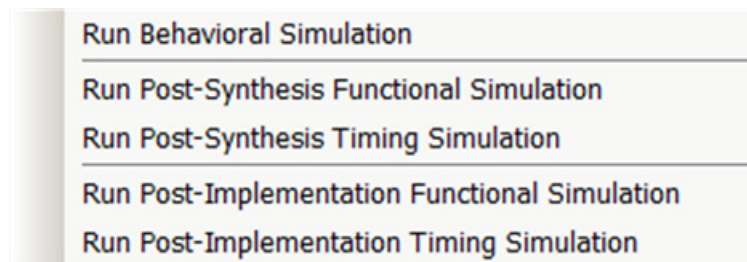


Figure 3-13: Simulation Run Options

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.

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.

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 7].



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.

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.

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 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

Saving Simulation Results

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

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

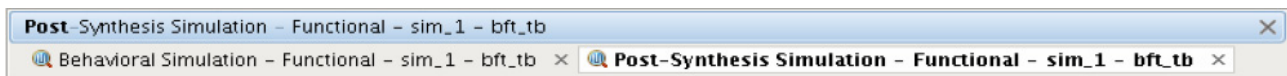


Figure 3-14: Active Simulation Type

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.

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](#).

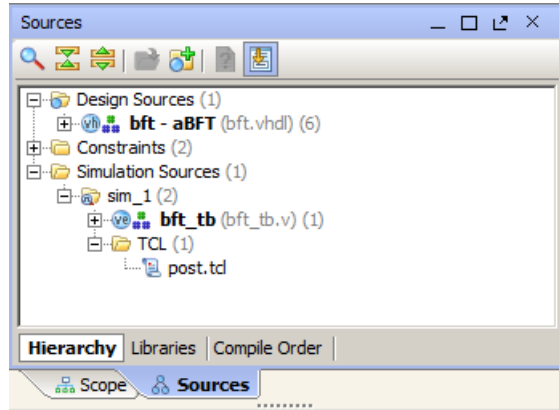



Figure 3-15: Using the post.tcl File in a Design

- 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.

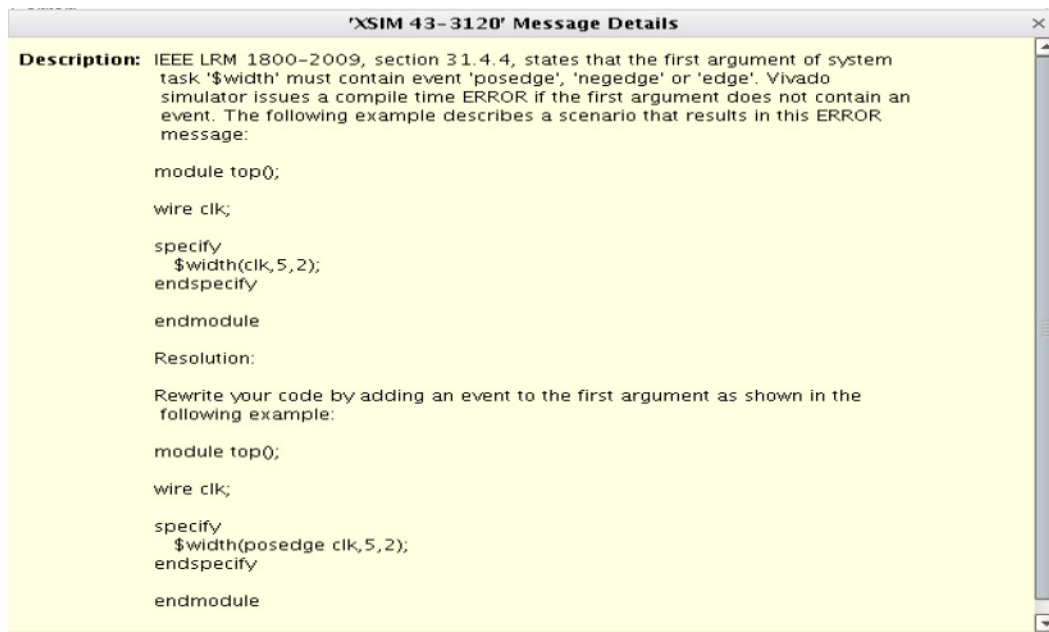


Figure 3-16: Simulator Message Description and Resolution Information

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 <arg>]
                 [-install_path <arg>] [-noclean_dir] [-quiet] [-verbose]
```

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

Table 3-2: launch_simulation Options

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

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 QuestaSim.

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

- Launching post-synthesis functional simulation using Synopsys VCS

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

- Running post-implementation timing simulation using Cadence IUS

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

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 5].

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: `add_wave <HDL_Object>`.

2. Add HDL objects to the waveform configuration using the steps listed in [Understanding HDL Objects in Waveform Configurations](#), page 57.

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

Interactive Method

- If a Vivado Design Suite project is loaded, click **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 earlier versions 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.

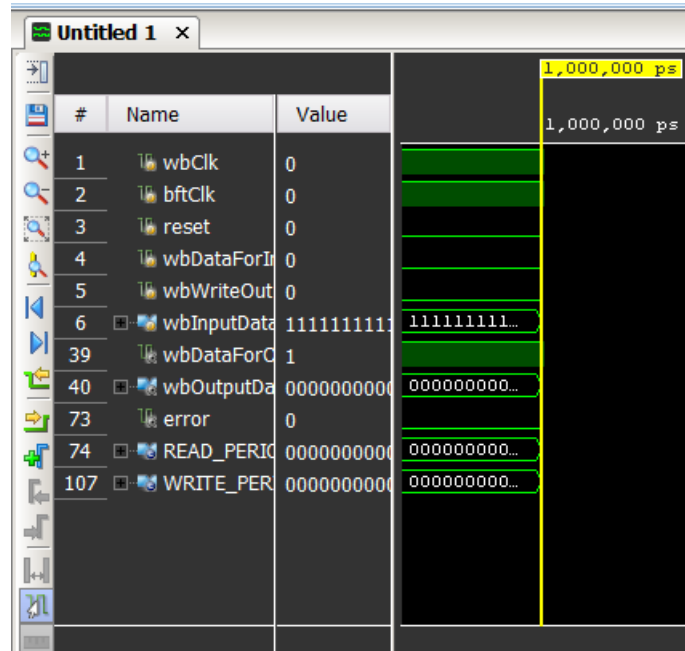


Figure 4-1: Waveform HDL Objects

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 36](#).

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.




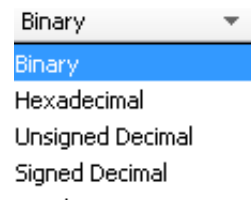
IMPORTANT: *Make a change to the radix setting in the window in which you wish to see the change. A change to the radix of an item in the Objects window does not apply to values in the Wave window or the Tcl Console. For example, the item `wbOutputData[31:0]` can be set to Signed Decimal in the objects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

To change the default waveform radix:

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



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:
 - Binary (default)
 - Hexadecimal
 - Unsigned Decimal
 - Signed Decimal
 - Octal
 - ASCII
 - Real
 - Real Settings

Note: For a description of the usage for Real and Real Settings see. . .

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.
- Real radixes cause bus values to be interpreted as fixed point or floating point real numbers, based on settings of the Real Settings dialog box.

To set a wave object to the Real radix:

1. Right-click an HDL object in the Name column of the waveform configuration window and select **Radix > Real Settings** from the drop-down menu to open the Real Settings dialog box (shown in [Figure 4-2.](#))

Use the dialog box to set parameters specifying how bus values convert to real numbers. The figure below shows the defaults. You must set these parameters before you can use the Real radix.

2. Right-click the HDL object again and select **Radix > Real** to display the values of the HDL object as real numbers.

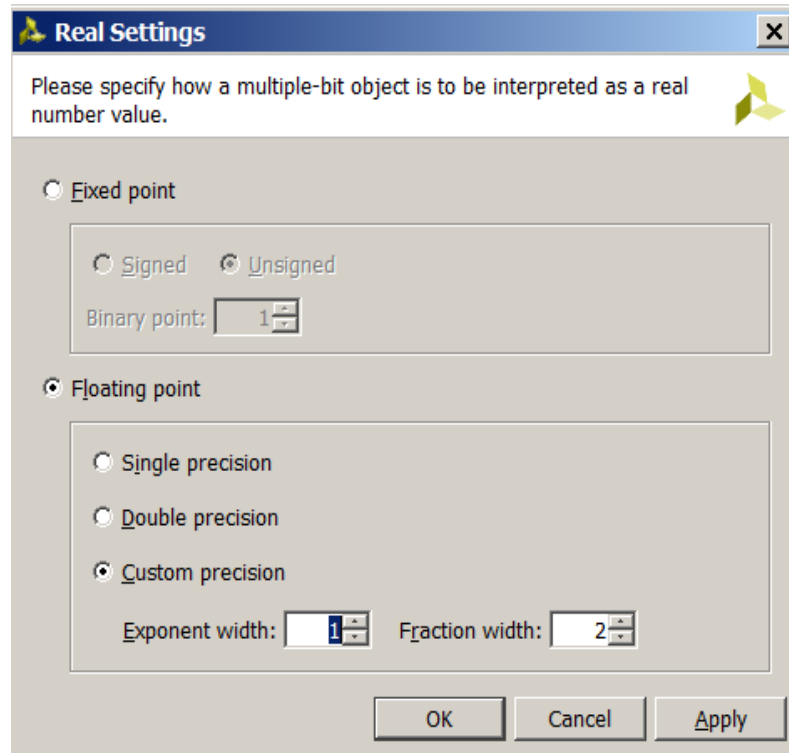


Figure 4-2: Real Settings Dialog Box

Real Settings Dialog Box Option Descriptions

- Fixed Point: Specifies that the bits of the selected bus wave object(s) is interpreted as a fixed point, signed, or unsigned real number.
 - Binary Point: Specifies how many bits to use for the fractional part of the fixed point number.



CAUTION! If Binary Point is larger than the bit width of the wave object, wave object values cannot be interpreted as fixed point, and when the wave object is shown in Digital waveform style, all values show as <Bad Radix>. When shown as analog, all values are interpreted as 0.

- Floating Point: Specifies that the bits of the selected bus wave object(s) should be interpreted as an IEEE floating point real number.

Note: Only single precision and double precision (and custom precision with values set to those of single and double precision) are supported. Other values result in <Bad Radix> values as in Fixed Point. Exponent Width and Fraction Width must add up to the bit width of the wave object, or else <Bad Radix> values result.

Be aware of the following limitations:

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

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-complement 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 4-3](#)), 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.*



Figure 4-3: 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. 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. Select **Name >**:
 - **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 64](#).



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.*

Using the Waveform Options Dialog Box

Select the **Waveforms Options** button  to open the Waveform Options dialog box, shown in [Figure 4-4](#).

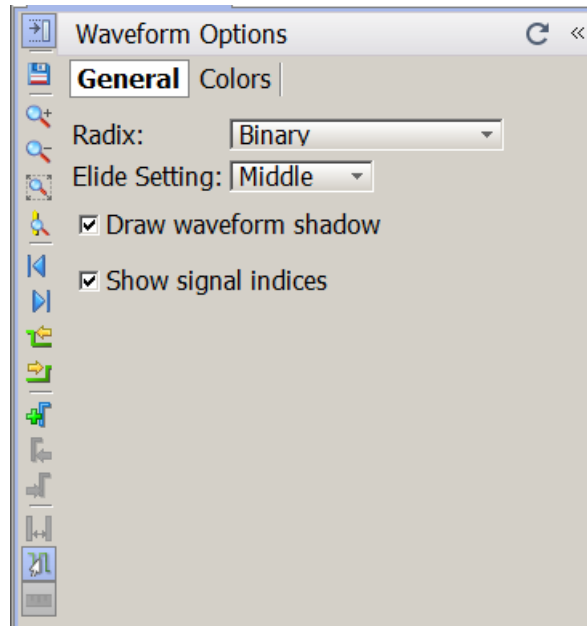


Figure 4-4: Waveform Options Dialog Box

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 3] 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 4-5](#).

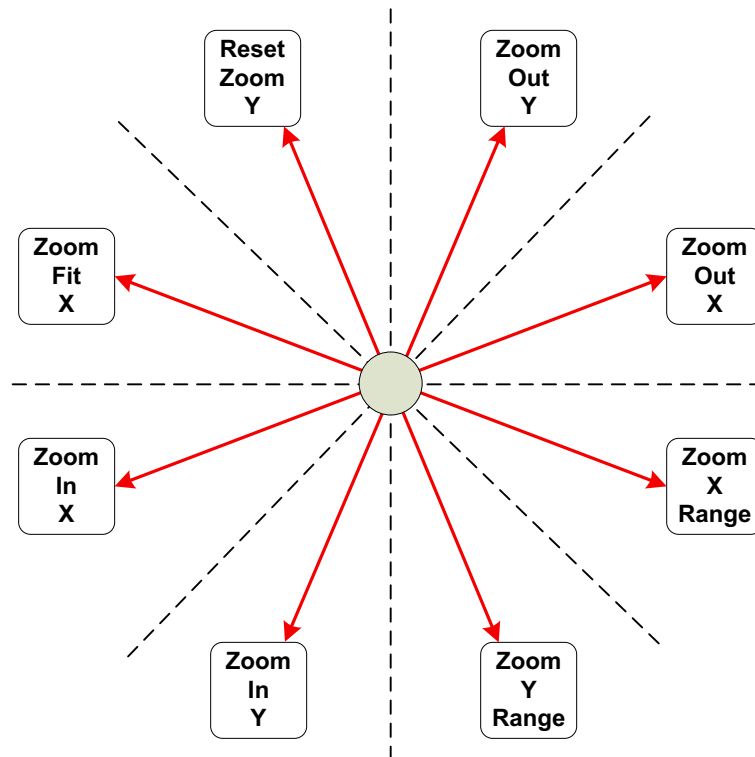


Figure 4-5: Analog Zoom Options

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 64](#).



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 64](#).

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 64](#).

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 7] 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.

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, see the section, [Using Breakpoints](#), below.

1. To step through a simulation:

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

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

- From the start (0 ns), restart the simulation. Use the **Restart** command to reset time to the beginning of the test bench. See [Chapter 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 6] 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
```

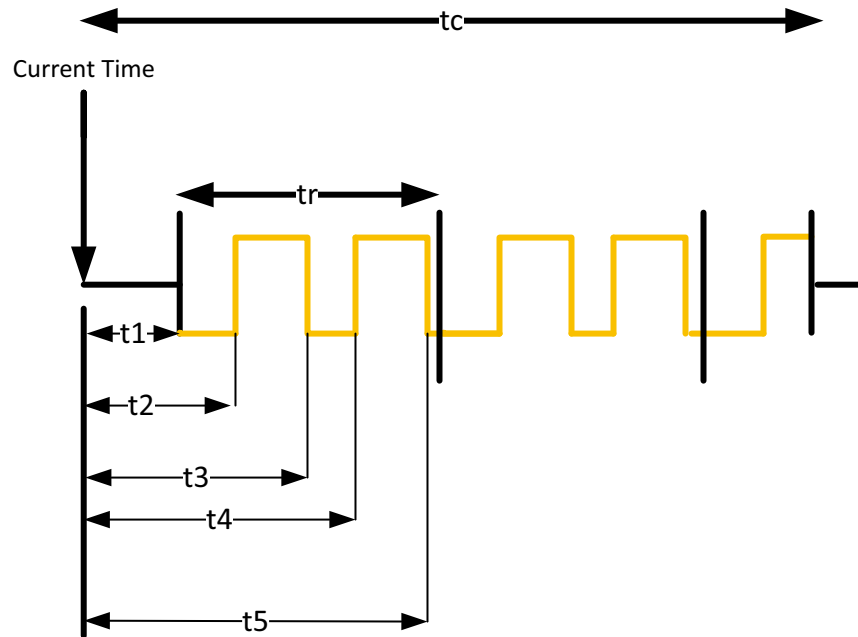


Figure 5-1: Illustration of `-add_force` Functionality

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 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 3].

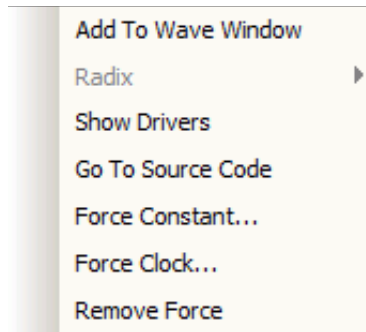


Figure 5-2: Force Options

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.

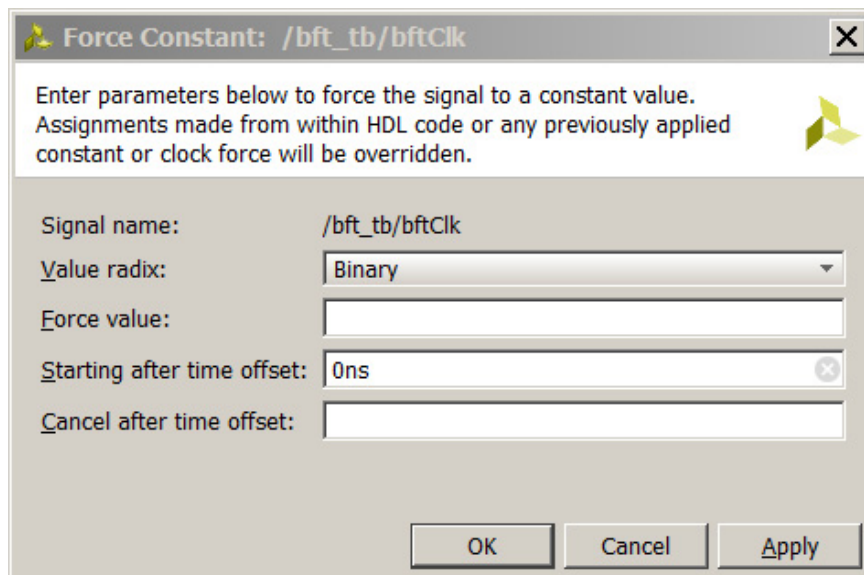


Figure 5-3: Force Selected Signal Dialog Box

The Force Constant options 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 radices, see [About Radices, page 58](#) and [Using Radices and Analog Waveforms, page 60](#).)

- 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](#).

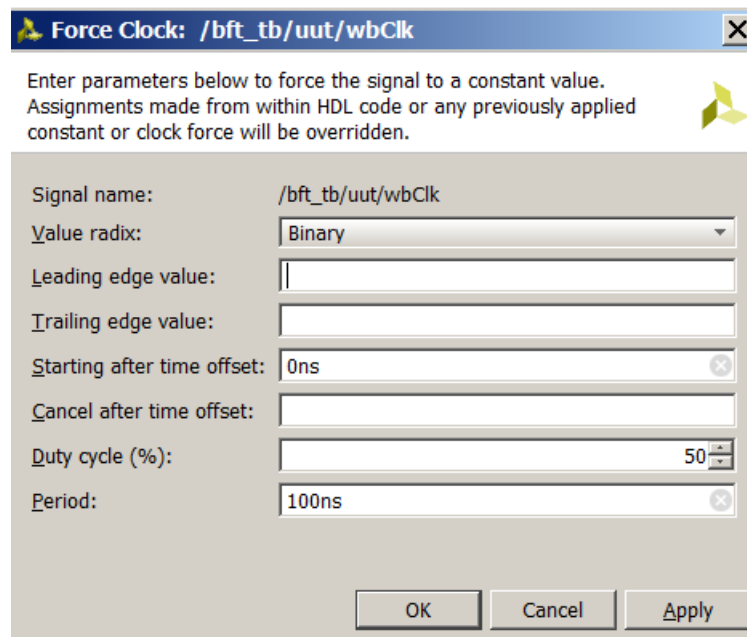


Figure 5-4: Force Clock Dialog Box

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 radices, see [About Radices, page 58](#) and [Using Radices and Analog Waveforms, page 60.](#)

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 7].

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 8] 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 140](#)

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 <code>\$dumpfile</code> Verilog system task.
checkpoint_vcd	Models the behavior of the <code>\$dumpall</code> Verilog system task.
start_vcd	Models the behavior of the <code>\$dumpon</code> Verilog system task.
log_vcd	Logs VCD for the specified HDL objects. This command models behavior of the <code>\$dumpvars</code> Verilog system task.
flush_vcd	Models behavior of the <code>\$dumpflush</code> Verilog system task.
limit_vcd	Models behavior of the <code>\$dumplimit</code> Verilog system task.
stop_vcd	Models behavior of the <code>\$dumpoff</code> Verilog system task.
close_vcd	Closes the VCD generation.

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

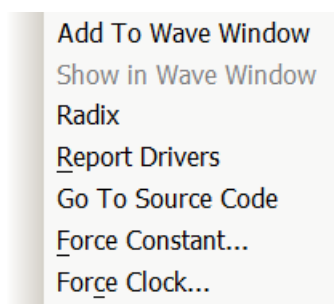


Figure 5-6: Objects Window Context Menu Options

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.

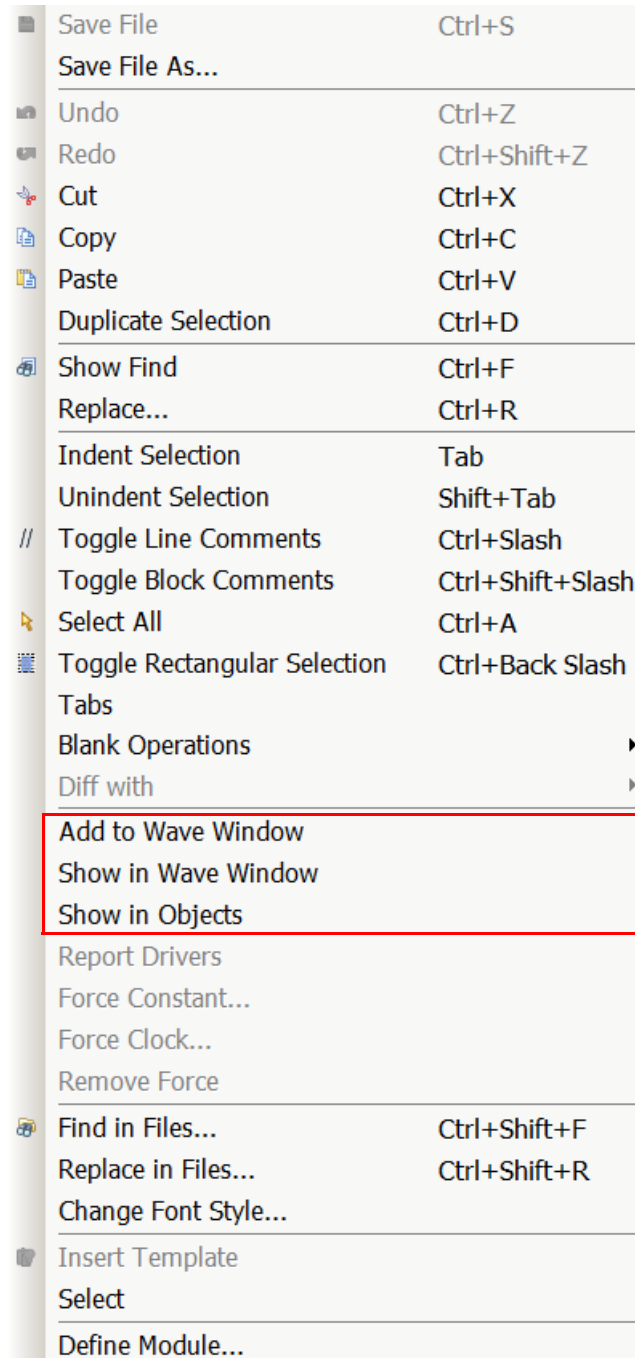


Figure 5-7: Text Editor Right-Click (Context) Menu

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.

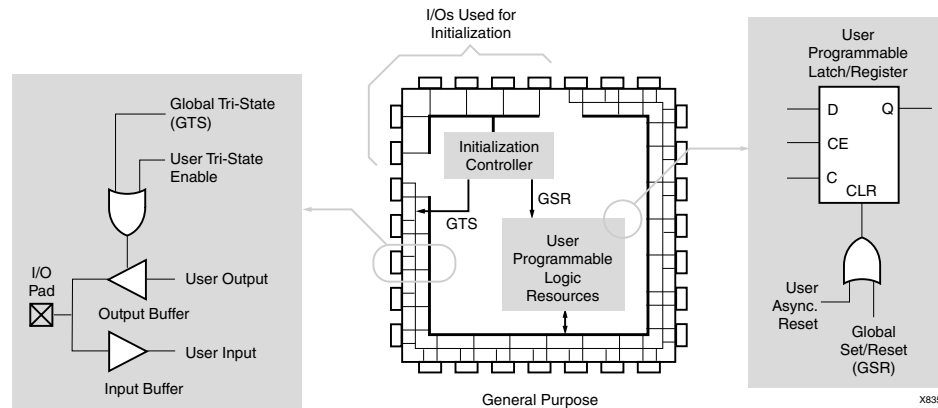


Figure 6-1: Built-in FPGA Initialization Circuitry Diagram

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

result_temp <= result;
clk_b_prccs : 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 12].



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 9].*

Disabling X Propagation for Synchronous Elements

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

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

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

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

To correct X-generation:

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

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



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

Simulating Configuration Interfaces

This section describes the simulation of the following configuration interfaces:

- JTAG simulation
- SelectMAP simulation

JTAG Simulation

BSCAN component simulation is supported on all devices.

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

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

This becomes:

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

The communication between the components takes place in the `VPKG` VHDL package file or the `g1b1` Verilog global module. Accordingly, no implicit connections are necessary between the specific `JTAG_SIME2` component and the design, or the specific `BSCANE2` symbol.

Stimulus can be driven and viewed from the specific `JTAG_SIME2` component within the test bench to understand the operation of the JTAG/BSCAN function. Instantiation templates for both of these components are available in both the Vivado® Design Suite templates and the specific-device libraries guides.

SelectMAP Simulation

The configuration simulation models (`SIM_CONFIGE2` and `SIM_CONFIGE3`) with an instantiation template allow supported configuration interfaces to be simulated to ultimately show the `DONE` pin going HIGH. This is a model of how the supported devices react to stimulus on the supported configuration interface.

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.

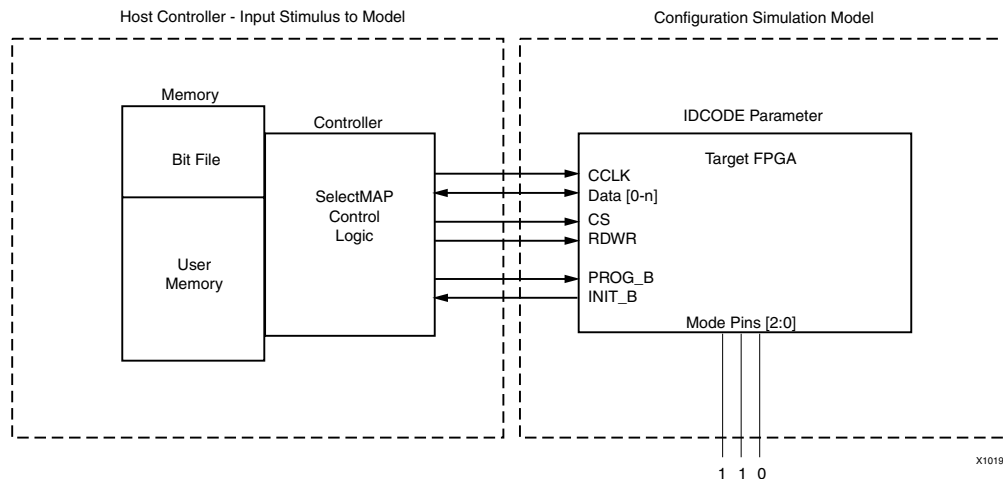


Figure 6-2: Block Diagram of Model Interaction

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 87
- Chapter 8, [Using Third-Party Simulators, Dumping SAIF for Power Analysis](#), page 140

Simulating a Design with AXI Bus Functional Models

For information on this topic see [Simulating a Design with AXI Bus Functional Models](#), page 143.

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`.*

Using Vivado Simulator in Batch or Scripted Mode

Introduction

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

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

```
help -category sim
```

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 7] for Tcl command usage.

When you issue the simulation command, the Vivado simulator

- Runs `xvlog` and `xvhdl` to analyze the design
- Runs `xelab` in the background to elaborate and compile the design into a simulation snapshot, which the Vivado simulator can run

When the latter process is complete, the Vivado tool launches `xsim` to run the simulation.

Vivado Simulator Command Line Steps

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

1. [Parsing Design Files, xvhdl and xvlog](#)
2. [Elaborating and Generating a Design Snapshot, -xelab](#)
3. [Simulating the Design Snapshot, xsim](#)

The following subsections describe these steps.

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

- [Generating a Timing Netlist in Chapter 2](#)
- [Running Post-Synthesis and Post-Implementation Simulations, page 123](#)

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 112](#).

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

xvlog Examples

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

Elaborating and Generating a Design Snapshot, -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 119](#) for more information about project files.*

xelab Command Syntax Options

Descriptions for each option are available in [Table 7-2, page 112](#).

```
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]

```

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 ethernetstim

```

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 <code>half_adder.v</code> compiled into logical library named <code>adder_lib</code>	File <code>full_adder.v</code> compiled into logical library named <code>work</code>
<pre> module half_adder(a,b,c,s); input a,b; output c,s; s = a ^ b; c = a & b; endmodule </pre>	<pre> module full_adder(c_in, c_out, a, b, sum) input c_in,a,b; output c_out,sum; wire carry1,carry2,sum1; `uselib lib = adder_lib half_adder adder1(.a(a),.b(b),. c(carry1),.s(sum1)); half_adder adder1(.a(sum1),.b(c_in),.c (carry2),.s(sum)); c_out = carry1 carry2; endmodule </pre>

xelab, xvhdl, and xvlog Command Options

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

Table 7-2: `xelab`, `xvhdl`, and `xvlog` Command Options

Command Option	Description	Used by Command
<code>-d [define] <name>[=<val>]</code>	Define Verilog macros. Use <code>-d</code> <code>--define</code> for each Verilog macro. The format of the macro is <code><name>[=<val>]</code> where <code><name></code> is name of the macro and <code><value></code> is an optional value of the macro.	xelab xvlog
<code>-debug <kind></code>	Compile with specified debugging ability turned on. The <code><kind></code> options are: <ul style="list-style-type: none"> • <code>typical</code>: Most commonly used abilities, including: <code>line</code> and <code>wave</code>. • <code>line</code>: HDL breakpoint. • <code>wave</code>: Waveform generation, conditional execution, force value. • <code>xlibs</code>: Visibility into Xilinx® precompiled libraries. This option is only available on the command line. • <code>off</code>: Turn off all debugging abilities (Default). • <code>all</code>: Uses all the debug options. 	xelab
<code>-encryptdumps</code>	Encrypt parsed dump of design units being compiled.	xvhdl xvlog
<code>-f [-file] <filename></code>	Read additional options from the specified file.	xelab xsim xvhdl xvlog
<code>-generic_top <value></code>	Override generic or parameter of a top-level design unit with specified value. Example: <code>-generic_top "P1=10"</code>	xelab
<code>-h [-help]</code>	Print this help message.	xelab xsim xvhdl xvlog
<code>-i [include] <directory_name></code>	Specify directories to be searched for files included using Verilog <code>`include</code> . Use <code>-i</code> <code>--include</code> for each specified search directory.	xelab xvlog
<code>-initfile <init_filename></code>	User-defined simulator initialization file to add to or override settings provided by the default <code>xsim.ini</code> file.	xelab xvhdl xvlog

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

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

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

Command Option	Description	Used by Command
-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
-override_timeunit	Override timeunit for all Verilog modules, with the specified time unit in -timescale option.	xelab
-override_timeprecision	Override time precision for all Verilog modules, with the specified time precision in -timescale option.	xelab
-pulse_e arg	Path pulse error limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_r arg	Path pulse reject limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_int_e arg	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_int_r arg	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).	xelab
-pulse_e_style arg	Specify when error about pulse being shorter than module path delay should be handled. Choices are: ondetect: report error right when violation is detected onevent: report error after the module path delay. Default: onevent	xelab

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

Command Option	Description	Used by Command
-prj <filename>	Specify the Vivado simulator project file containing one or more entries of vhdl verilog <work lib> <HDL file name>.	xelab xvhd 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 xvhd 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
-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

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

Command Option	Description	Used by Command
-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: <ul style="list-style-type: none"> <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
-sv_liblist arg	Bootstrap file pointing to DPI shared libraries.	xelab
-dpiheader arg	Header filename for the exported and imported functions.	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
<code>-f [-file] <filename></code>	Load the command line options from a file.
<code>-g [-gui]</code>	Run with interactive workspace.
<code>-h [-help]</code>	Print help message to screen.
<code>-log <filename></code>	Specify the log file name.
<code>-maxdeltaid arg (=-1)</code>	Specify the maximum delta number. Report an error if it exceeds maximum simulation loops at the same time.
<code>-maxlogsize arg (=-1)</code>	Set the maximum size a log file can reach in MB. The default setting is unlimited.
<code>-ieewarnings</code>	Enable warnings from VHDL IEEE functions.
<code>-nolog</code>	Suppresses log file generation.
<code>-nosignalhandlers</code>	<p>Disables the installation of OS-level signal handlers in the simulation. For performance reasons, the simulator does not check explicitly for certain conditions, such as an integer division by zero, that could generate an OS-level fatal run time error. Instead, the simulator installs signal handlers to catch those errors and generates a report. With the signal handlers disabled, the simulator can run in the presence of such security software, but OS-level fatal errors could crash the simulation abruptly with little indication of the nature of the failure.</p>  <p>CAUTION! Use this option only if your security software prevents the simulator from running successfully.</p>
<code>-onfinish <quit stop></code>	Specify the behavior at end of simulation.
<code>-onerror <quit stop></code>	Specify the behavior upon simulation run time error.
<code>-R [-runall]</code>	Runs simulation till end (such as <code>do 'run all;quit'</code>).
<code>-stats</code>	Display memory and CPU stats upon exiting.
<code>-testplusarg <arg></code>	Specify <code>plusargs</code> to be used by <code>\$test\$plusargs</code> and <code>\$value\$plusargs</code> system functions.
<code>-t [-tclbatch] <filename></code>	Specify the Tcl file for batch mode execution.
<code>-tp</code>	Enable printing to screen of hierarchical names of process being executed.
<code>-tl</code>	Enable printing to screen of file name and line number of statements being executed.

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

xsim Option	Description
-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	<code>xvlog <VerilogFileName(s)></code>
SystemVerilog	<code>xvlog -sv <SystemVerilogFileName(s)></code>
VHDL	<code>xvhdl <VhdlFileName(s)></code>

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_simulator`, you must set debug level to `typical`.

Step 3: Running Simulation

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

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

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

To open the wave config file:

```
xsim <SnapShotName> -view <wcfg 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 7-1 illustrates the behavioral simulation process:

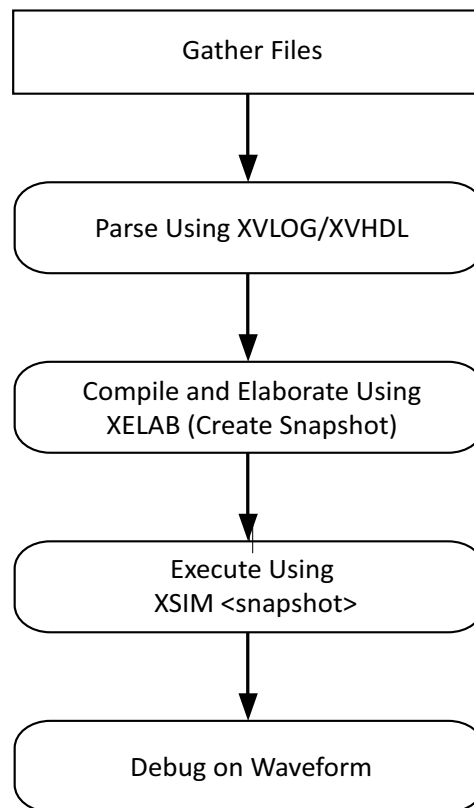


Figure 7-1: Behavioral Simulation Process

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

Running Post-Synthesis and Post-Implementation Simulations

At post-synthesis and post-implementation, you can run a functional or a Verilog timing simulation. Figure 7-2 illustrates the post-synthesis and post-implementation simulation process:

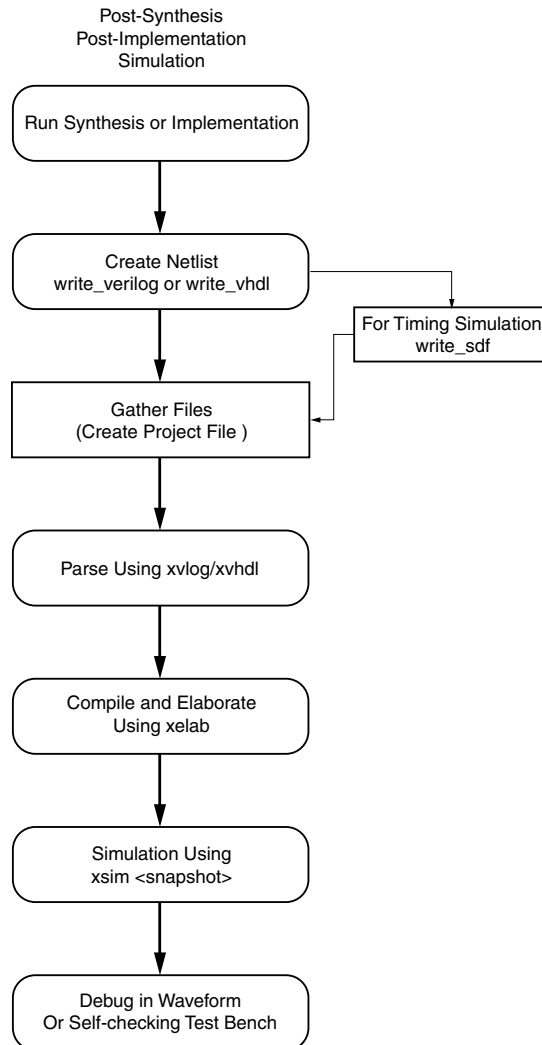


Figure 7-2: Post-Synthesis and Post-Implementation Simulation

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_simulator -simset sim_1 -mode post-synthesis -type functional
Vivado% place_design
Vivado% route_design
Vivado% write_verilog -mode timesim -sdf_anno true -sdf_file postRoute.sdf
postRoute_netlist.v
Vivado% write_sdf postRoute.sdf
Vivado% launch_simulator -simset sim_1 -mode post-implementation -type timing
Vivado% close_project
```

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)



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 3] describes the use of the Vivado IDE.

For links to more information on your third party simulator see [Ref 13].

Preparing for Simulation Using Third-Party Tools

Pointing to the Simulator Install Location

To define the installation path:

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

- 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.

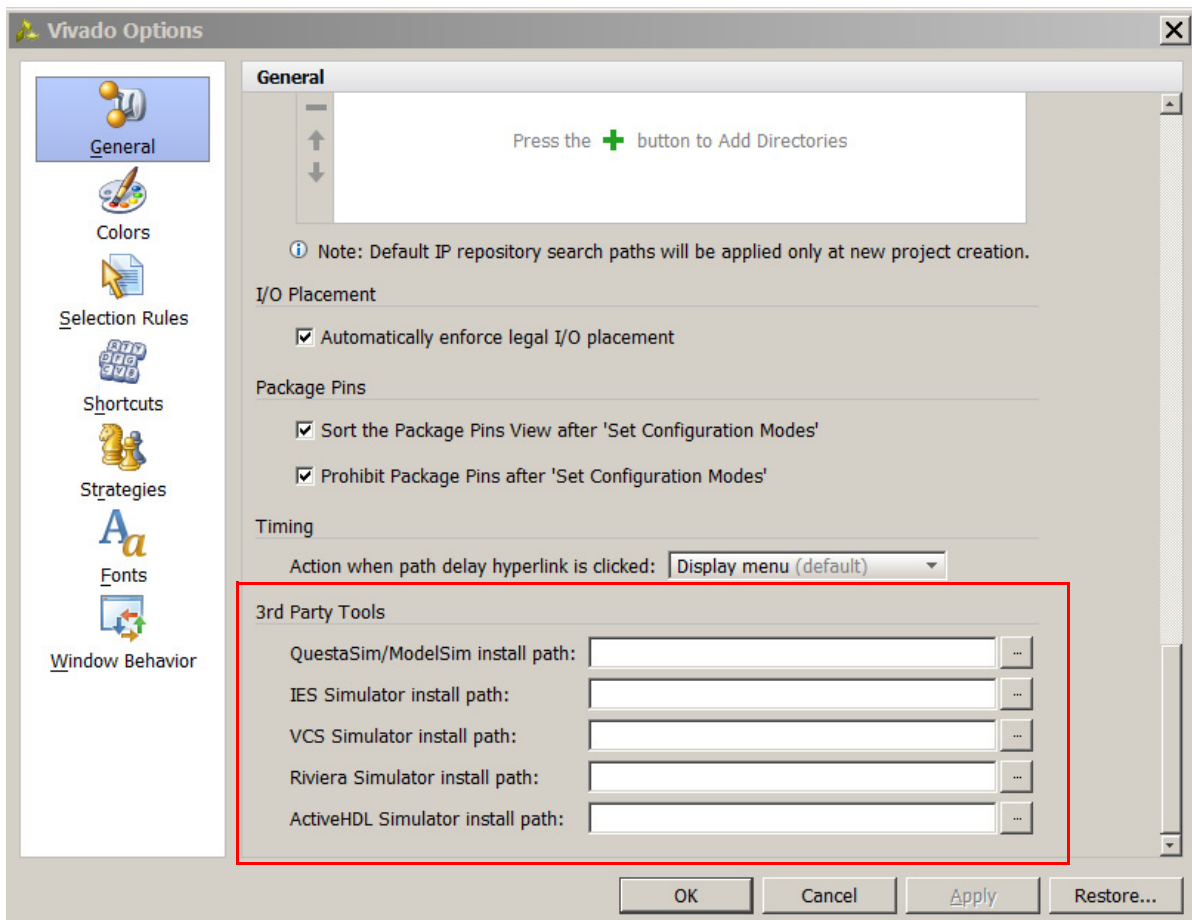


Figure 8-1: Vivado Design Suite General Options, Install Path

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.

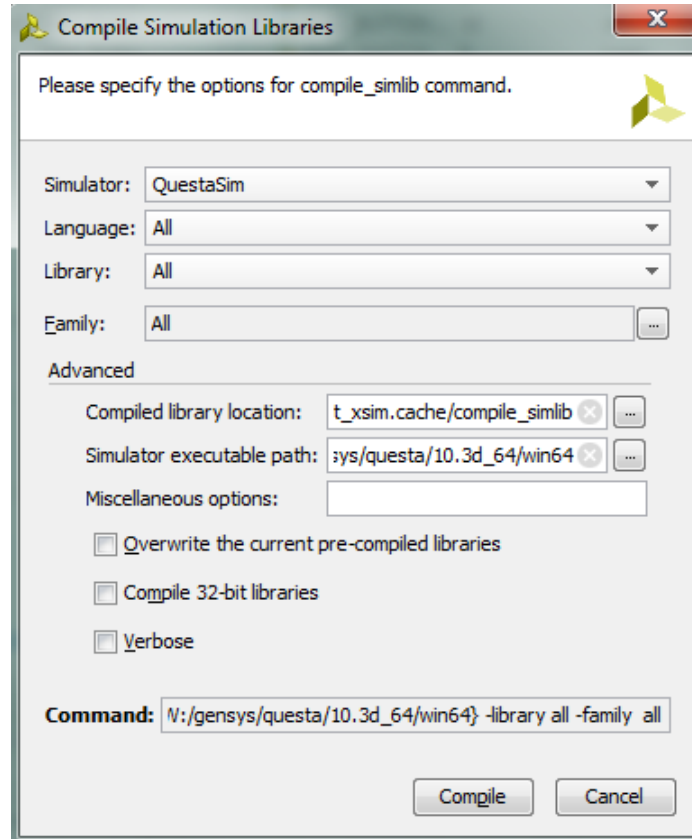


Figure 8-2: Compile Simulation Libraries Dialog Box

Dialog Box Options

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

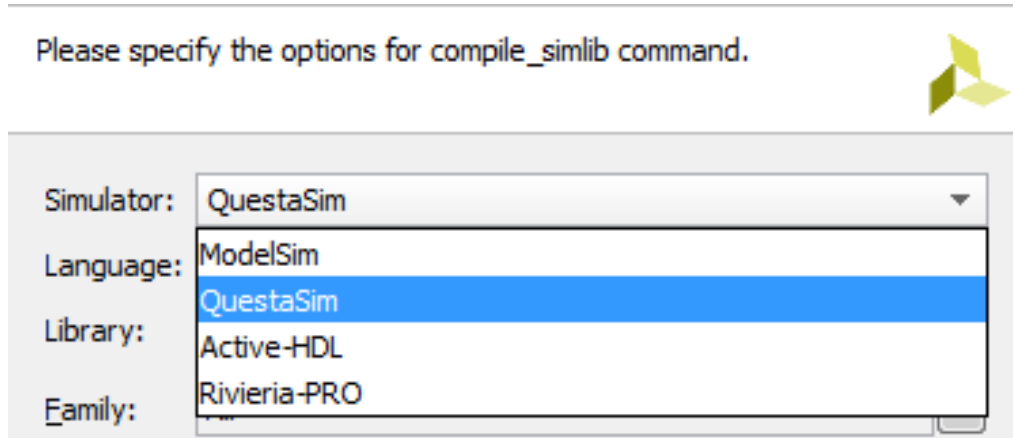


Figure 8-3: Simulator Drop-Down Selections

- **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.

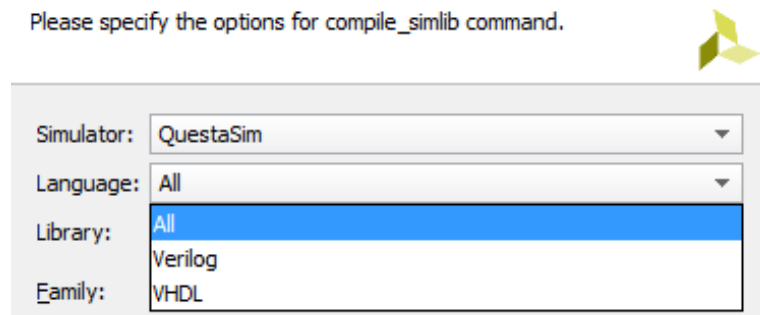


Figure 8-4: Language Selection

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

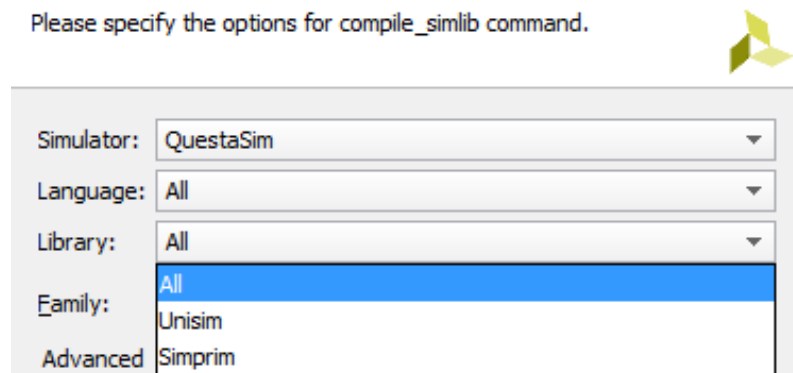


Figure 8-5: Simulation Libraries

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

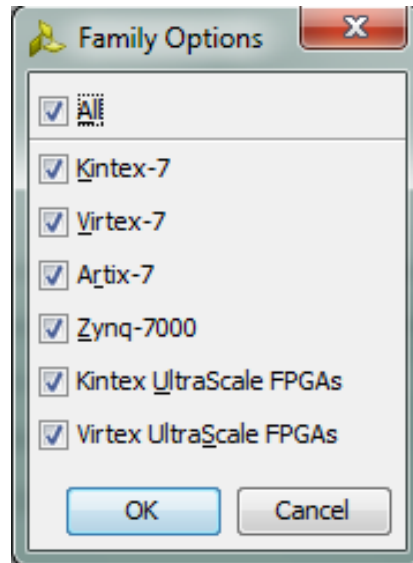


Figure 8-6: Family Options

- **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 11] 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.

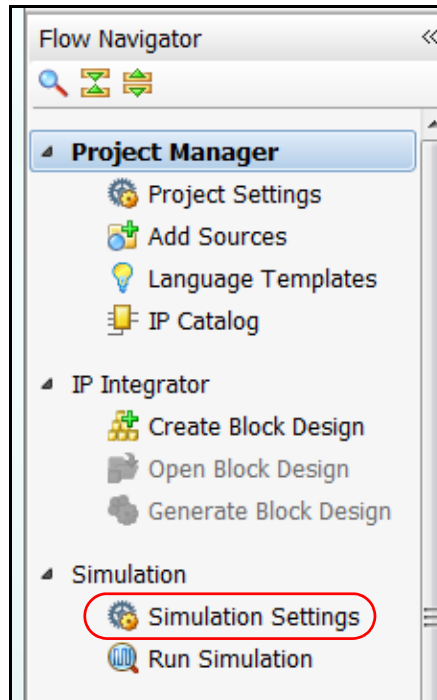
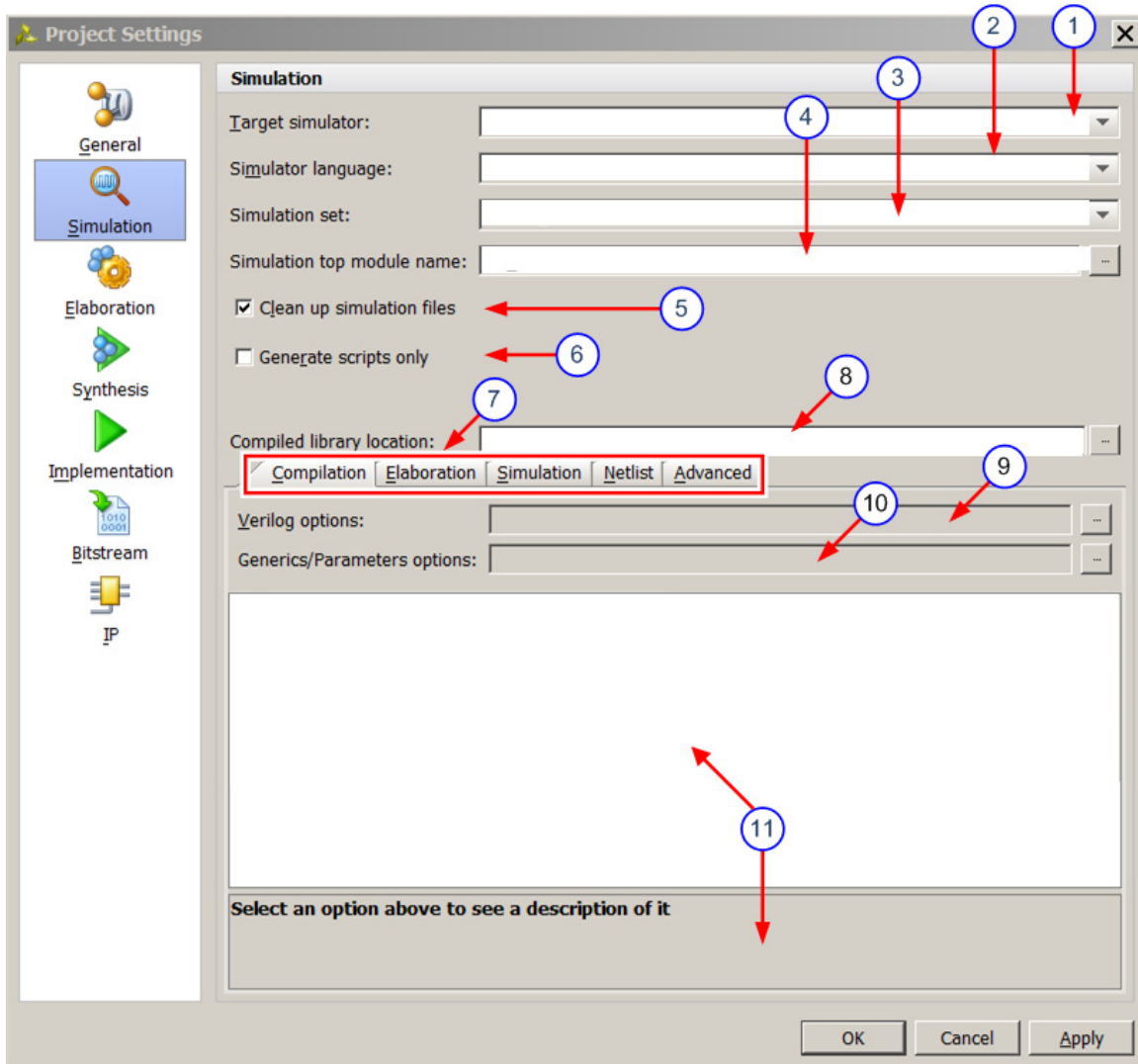


Figure 8-7: Simulation Settings

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.



- | | |
|---|---|
| <ol style="list-style-type: none"> 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. | <ol style="list-style-type: none"> 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](#).

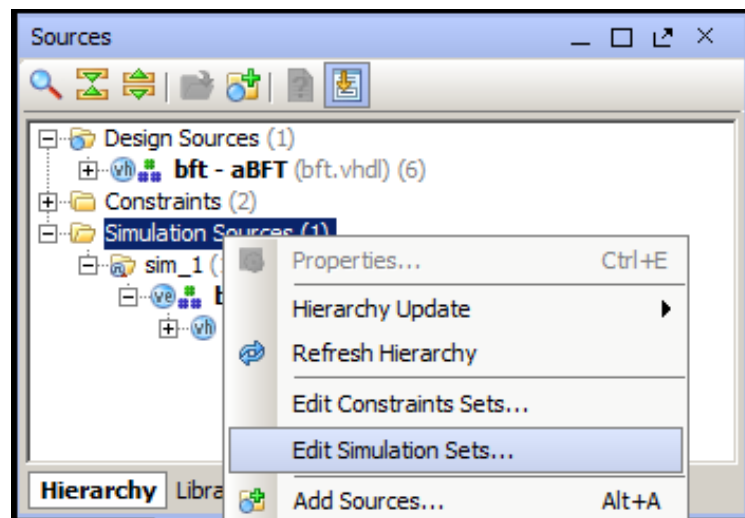


Figure 8-9: Edit Simulation Sets Option

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](#).

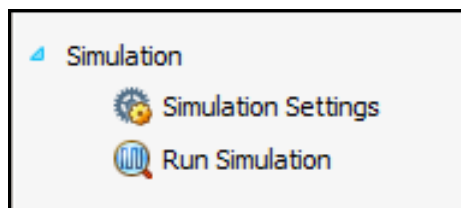


Figure 8-10: Flow Navigator Simulation Options

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.



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

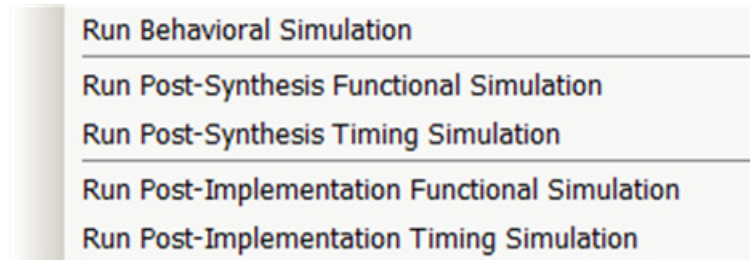


Figure 8-11: Simulation Run Options

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



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 7].*



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 112](#) provides descriptions for these commands.*

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

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 102](#) 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 [\[Ref 13\]](#).

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 13].

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 13].

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. Now you can run them using the following commands:

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

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.

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 object's type is `bit`, `std_logic`, or any enumeration type whose enumerators are a subset of those of `std_logic` and include at least 0 and 1, or type of the object is a one-dimensional array of such a type.
- For HDL objects, which are of VHDL enumeration type, the value can be one of the enumerator literals, without single quotes if the enumerator is a character literal. Radix is ignored.
- For VHDL objects, of integral type, the value can be a signed decimal integer in the range of the type. Radix is ignored.
- For VHDL and Verilog floating point types the value can be a floating point value. Radix is ignored.
- For all other types of HDL objects, the Tcl command set does not support setting values.

Vivado Simulator Mixed Language Support and Language Exceptions

Introduction

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

- VHDL, see *IEEE Standard VHDL Language Reference Manual* (IEEE-STD-1076-1993) [Ref 14]
- Verilog, see *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1364-2001) [Ref 15]
- System Verilog Synthesizable subset. See *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1800-2009) [Ref 16]
- IEEE P1735 encryption, see *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)* (IEEE-STD-P1735) [Ref 18]

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 150](#).



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 109](#) 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 150](#).

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
vector of bit(1D)	bit_vector

Table B-3: Supported SV and VHDL Data Types (Cont'd)

SV Data type	VHDL Data type
	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	<code>std_logic</code>	<code>bit</code>
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

<code>bit</code>	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

<code>std_logic</code>	Verilog
U	X
X	X
0	0
1	1
Z	Z
W	X
L	0
H	1
-	X

Because Verilog is case sensitive, named associations and the local port names that you use in the component declaration must match the case of the corresponding Verilog port names.

VHDL Language Support Exceptions

Certain language constructs are not supported by the Vivado simulator. [Table B-7](#) lists the VHDL language support exceptions.

Table B-7: VHDL Language Support Exceptions

Supported VHDL Construct	Exceptions
<code>abstract_literal</code>	Floating point expressed as based literals are not supported.
<code>alias_declaration</code>	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
<code>alias_designator</code>	Operator_symbol as alias_designator Character_literal as alias_designator
<code>association_element</code>	Globally, locally static range is acceptable for taking slice of an actual in an association element.
<code>attribute_name</code>	Signature after prefix is not supported.
<code>binding_indication</code>	Binding_indication without use of entity_aspect is not supported.
<code>bit_string_literal.</code>	Empty bit_string_literal (" ") is not supported
<code>block_statement</code>	Guard_expression is not supported; for example, guarded blocks, guarded signals, guarded targets, and guarded assignments are not supported.
<code>choice</code>	Aggregate used as choice in case statement is not supported.
<code>concurrent_assertion_statement</code>	Postponed is not supported.
<code>concurrent_signal_assignment_statement</code>	Postponed is not supported.
<code>concurrent_statement</code>	Concurrent procedure call containing wait statement is not supported.
<code>conditional_signal_assignment</code>	Keyword guarded as part of options is not supported as there is no supported for guarded signal assignment.
<code>configuration_declaration</code>	Non locally static for generate index used in configuration is not supported.
<code>entity_class</code>	Literals, unit, file, and group as entity class are not supported.

Table B-7: VHDL Language Support Exceptions (Cont'd)

Supported VHDL Construct	Exceptions
entity_class_entry	Optional <> intended for use with group templates is not supported.
file_logical_name	Although file_logical_name is allowed to be any wild expression evaluating to a string value, only string literal and identifier is acceptable as file name.
function_call	Slicing, indexing, and selection of formals is not supported in a named parameter association within a function_call.
instantiated_unit	Direct configuration instantiation is not supported.
mode	Linkage and Buffer ports are not supported completely.
options	Guarded is not supported.
primary	At places where primary is used, allocator is expanded there.
procedure_call	Slicing, indexing, and selection of formals is not supported in a named parameter association within a procedure_call.
process_statement	Postponed processes are not supported.
selected_signal_assignment	The guarded keyword as part of options is not supported as there is no support for guarded signal assignment.
signal_declaration	The signal_kind is not supported. The signal_kind is used for declaring guarded signals, which are not supported.
subtype_indication	Resolved subtype of composites (arrays and records) is not supported
waveform	Unaffected is not supported.
waveform_element	Null waveform element is not supported as it only has relevance in the context of guarded signals.

Verilog Language Support Exceptions

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

Table B-8: Verilog Language Support Exceptions (Cont'd)

Verilog Construct	Exception
<code>`nounconnected_drive</code>	not supported
Attributes	
<code>attribute_instance</code>	not supported
<code>attr_spec</code>	not supported
<code>attr_name</code>	not supported
Primitive Gate and Switch Types	
<code>cmos_switchtype</code>	not supported
<code>mos_switchtype</code>	not supported
<code>pass_en_switchtype</code>	not supported
Generated Instantiation	
<code>generated_instantiation</code>	<p>The <code>module_or_generate_item</code> alternative is not supported.</p> <p>Production from standard (see <i>IEEE Standard Verilog Hardware Description Language</i> (IEEE 1364-2001) section 13.2 [Ref 15]:</p> <pre>generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_block module_or_generate_item</pre> <p>Production supported by Simulator:</p> <pre>generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_blockgenerate_condition</pre>
<code>genvar_assignment</code>	<p>Partially supported.</p> <p>All generate blocks must be named.</p> <p>Production from standard (see <i>IEEE Standard Verilog Hardware Description Language</i> (IEEE 1364-2001) section 13.2 [Ref 15]:</p> <pre>generate_block ::= begin [: generate_block_identifier] { generate_item } end</pre> <p>Production supported by Simulator:</p> <pre>generate_block ::= begin: generate_block_identifier { generate_item } end</pre>
Source Text Constructs	

Table B-8: Verilog Language Support Exceptions (Cont'd)

Verilog Construct	Exception
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 <i>IEEE Standard Verilog Hardware Description Language</i> (IEEE 1364-2001) section 13.2 [Ref 15]. This does not refer to the <code>`include</code> compiler directive.
System Timing Check Commands	
\$skew_timing_check	not supported
\$timeskew_timing_check	not supported
\$fullskew_timing_check	not supported
\$nochange_timing_check	not supported
System Timing Check Command Argument	
checktime_condition	not supported
PLA Modeling Tasks	
\$async\$nand\$array	not supported
\$async\$nor\$array	not supported
\$async\$or\$array	not supported
\$sync\$and\$array	not supported
\$sync\$nand\$array	not supported
\$sync\$nor\$array	not supported
\$sync\$or\$array	not supported
\$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

Table B-8: Verilog Language Support Exceptions (Cont'd)

Verilog Construct	Exception
Value Change Dump (VCD) Files	
\$dumpportson \$dumpports \$dumpportsoff \$dumpportsflush \$dumpportslimit \$vcdplus	not supported

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	<pre>xvhdl file1.vhd file2.vhd xvhdl -work worklib file1.vhd file2.vhd xvhdl -prj files.prj</pre>
Parsing Verilog files	<pre>xvlog file1.v file2.v xvlog -work worklib file1.v file2.v xvlog -prj files.prj</pre>
Parsing SystemVerilog files	<pre>xvlog -sv file1.v file2.v xvlog -work worklib -sv file1.v file2.v xvlog -prj files.prj</pre> <p>Note: For information about the PRJ file format, see Project File (.prj) Syntax in Chapter 7.</p>

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 112 for a complete list of command options. The following are key options for xvlog and xvhdl:	
	Key Option	Applies to:
	<code>-d [define] <name> [=<val>]</code>	xvlog
	<code>-h [-help]</code>	xvlog, xvhdl
	<code>-i [include] <directory_name></code>	xvlog
	<code>-initfile <init_filename></code>	xvlog, xvhdl
	<code>-L [-lib] <library_name> [=<library_dir>]</code>	xvlog, xvhdl
	<code>-log <filename></code>	xvlog, xvhdl
	<code>-prj <filename></code>	xvlog, xvhdl
	<code>-relax</code>	xvhdl, vlog
	<code>-work <library_name> [=<library_dir>]</code>	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	<code>xelab top1 top2</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units are compiled in the work library.
	<code>xelab lib1.top1 lib2.top2</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units have are compiled in <code>lib1</code> and <code>lib2</code> , respectively
	<code>xelab top1 top2 -prj files.prj</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units are compiled in the work library. The file <code>files.prj</code> contains entries such as: <pre> verilog <libraryName> <VerilogDesignFileName> vhdl <libraryName> <VHDLDesignFileName> sv <libraryName> <SystemVerilogDesignFileName> </pre>
	<code>xelab top1 top2 -s top</code>	Elaborates a design that has two top design units: <code>top1</code> and <code>top2</code> . In this example, the design units are compiled in the work library. After compilation, <code>xelab</code> generates an executable snapshot with the name <code>top</code> . Without the <code>-s top</code> switch, <code>xelab</code> creates the snapshot by concatenating the unit names.

Table C-1: Standalone Mode: Parsing, Elaborating, and Running Simulation from a Command Line (Cont'd)

Command Line Help and xelab Options	<code>xelab -help</code> xelab, xvhd, and xvlog Command Options, page 112	
Running Simulation		
After parsing, elaboration and compilation stages are successful; xsim generates an executable snapshot to run simulation.		
Usage	<code>xsim top -R</code>	Simulates the design to through completion.
	<code>xsim top -gui</code>	Opens the Vivado simulator workspace (GUI).
	<code>xsim top</code>	Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as: <pre>run -all run 100 ns</pre>
Important Shortcuts		
You can invoke the parsing, elaboration, and executable generation and simulation in one, two, or three stages.		
	Three Stage	<pre>xvlog bot.v xvhd1 top.vhd xelab work.top -s top xsim top -R</pre>
	Two Stage	<pre>xelab -prj my_prj.prj work.top -s top xsim top -R</pre> <p>where <code>my_prj.prj</code> file contains:</p> <pre>verilog work bot.v vhd1 work top.vhd</pre>
	Single Stage	<pre>xelab -prj my_prj.prj work.top -s top -R</pre> <p>where <code>my_prj.prj</code> file contains:</p> <pre>verilog work bot.v vhd1 work top.vhd</pre>
Vivado Simulation Tcl Commands		
The following are commonly used Tcl commands. For a complete list, invoke following commands in the Tcl Console:		
<ul style="list-style-type: none"> <code>load_features simulator</code> <code>help -category simulation</code> 		
For information on any Tcl Command, type: <code>-help <Tcl_command></code>		

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 77 .
	add_force	Force the value of a signal, wire, or register to a specified value. Tcl command examples are provided on page 82 .
	current_time now	Report current simulation time. See current_time , page 124 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 88 .
	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 <code>get_value -help</code> 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 88 .
	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 34 .
	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 .	

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.

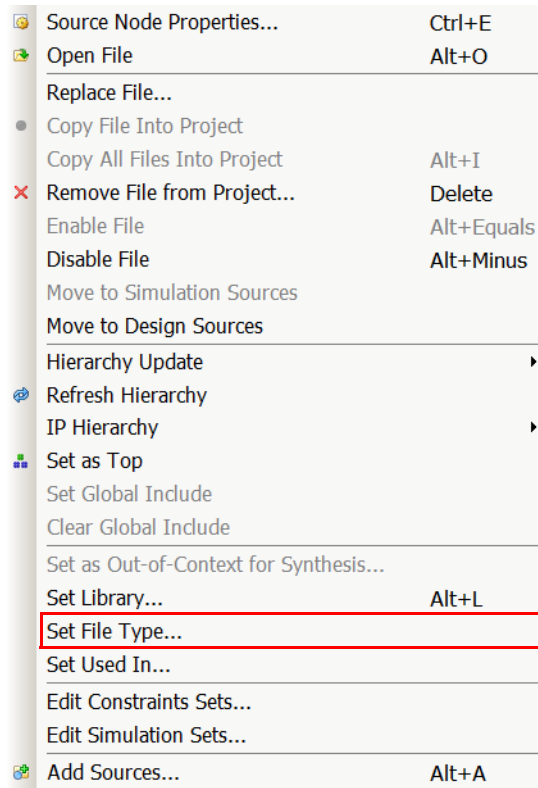


Figure D-1: Context Menu with Set File Type Option

- In the **Set Type** menu, shown in the figure below, change the file type from Verilog to **System Verilog** and click **OK**.

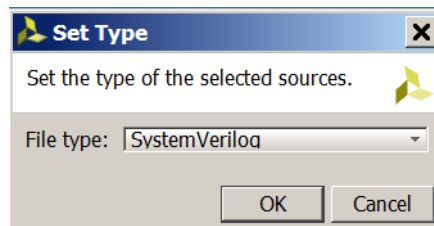


Figure D-2: Source Node Properties > Set File Type

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

Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont'd)

Primary construct	Secondary construct	LRM section	Status
	Enumerated type ranges	6.19.2	Supported
	Type checking	6.19.3	Supported
	Enumerated types in numerical expressions	6.19.4	Supported
	Enumerated type methods	6.19.5	Supported
	Type parameters	6.20.3	Supported
	Const constants	6.20.6	Supported
	Type operator	6.23	Supported
	Cast operator	6.24.1	Supported
	<code>\$cast</code> dynamic casting	6.24.2	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
	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 <code>always_comb</code> procedure	9.2.2	Supported
	Implicit <code>always_comb</code> sensitivities	9.2.2.1	Supported
	Latched logic <code>always_latch</code> procedure	9.2.2.3	Supported

Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont'd)

Primary construct	Secondary construct	LRM section	Status
	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
	Net aliasing	10.11	Not Supported
Operators and expressions		11	
	Constant expressions	11.2.1	Supported
	Aggregate expressions	11.2.2	Supported
	Operators with real operands	11.3.1	Supported
	Operations on logic (4-state) and bit (2-state) types	11.3.4	Supported
	Assignment within an expression	11.3.6	Supported
	Assignment operators	11.4.1	Supported
	Increment and decrement operators	11.4.2	Supported
	Arithmetic expressions with unsigned and signed types	11.4.3.1	Supported
	Wildcard equality operators	11.4.6	Supported
	Concatenation operators	11.4.12	Supported
	Set membership operator	11.4.13	Supported

Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont'd)

Primary construct	Secondary construct	LRM section	Status
	Concatenation of <code>stream_expressions</code>	11.4.14.1	Supported
	Re-ordering of the generic stream	11.4.14.2	Supported
	Streaming concatenation as an assignment target (unpack)	11.4.14.3	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
	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	

Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont'd)

Primary construct	Secondary construct	LRM section	Status
	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
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

Table D-1: Synthesizable Set of System Verilog 1800-2009 (Cont'd)

Primary construct	Secondary construct	LRM section	Status
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
	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: Early Access

In Vivado simulator, basic support for some of the commonly used dynamic types has been added, as shown in the table below.



IMPORTANT: *This support is early access at the current time.*

Table D-2: Supported Dynamic Types Constructs: Early Access

Construct Name	Details of Supported Features
String	<ul style="list-style-type: none"> • String assignment • String as struct member • All string operators • String methods <ul style="list-style-type: none"> ◦ <code>len()</code>, <code>putc()</code>, <code>getc()</code> ◦ <code>toupper()</code>, <code>tolower()</code> ◦ <code>compare()</code>, <code>icompare()</code> ◦ <code>substr()</code>, <code>atoi()</code> ◦ <code>atobin()</code>, <code>atohex()</code>
Queue	<ul style="list-style-type: none"> • Queue insertion in constant order; this includes growing the queue dynamically until the system memory is full. <ul style="list-style-type: none"> ◦ <code>push_front()</code>, <code>push_back()</code> • Queue deletion in constant order. • Queue function <ul style="list-style-type: none"> ◦ <code>size()</code>, <code>delete()</code> • Random access (both read and write in the form <code>my_queue[i]</code>) in constant order • Insertion via indexing at last location supported
Associative Array	<p>Keys of the following types are supported:</p> <ul style="list-style-type: none"> • Vector, multi-D array, packed struct, class <ul style="list-style-type: none"> ◦ Value of any supported type, including dynamic types • Associative array methods <ul style="list-style-type: none"> ◦ <code>size()</code>, <code>number()</code>, <code>delete()</code> ◦ <code>first()</code>, <code>next()</code>, <code>last()</code>, <code>prev()</code>
Class	<ul style="list-style-type: none"> • Constructor, called with <code>new</code> • Member with initial value • Member function • <code>this</code> as keyword
Non-synthesizable constructs	<ul style="list-style-type: none"> • <code>fork-join_none</code> • <code>\$urandom</code> and <code>\$urandom_range</code>

Direct Programming Interface (DPI) in Vivado Simulator

Introduction

You can use the SystemVerilog Direct Programming Interface (DPI) to bind C code to SystemVerilog code. Using DPI, SystemVerilog code can call a C function, which in turn can call back a SystemVerilog task or function. Vivado® simulator currently supports a subset of DPI features, as described below.

Compiling C Code

A new compiler executable, `xsc`, is provided to convert C code into 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.

Description of the xsc Compiler

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

One-step process:

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

Two-step process:

```
xsc -compile <C files>
```

```
xsc -link <object files>
```

Usage:

```
xsc [options] <files...>
```

Switches

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

Table E-1: XSC Compiler Switches

<code>-compile</code>	Generate the object files only from the source C files. The link stage is not run.
<code>-f [-file] arg</code>	Read additional options from the specified file.
<code>-h [-help]</code>	Print this help message.
<code>-i [-input_file] arg</code>	List of input files (one file per switch) for compiling or linking.
<code>-link</code>	Run only the linking stage to generate the shared library (.a or .so) from the object files.
<code>-mt arg (=auto)</code>	Specifies the number of sub-compilation jobs that can be run in parallel. Choices are: <code>auto</code> : automatic <code>n</code> : where <code>n</code> is an integer greater than 1 <code>off</code> : turn off multi-threading (Default: <code>auto</code>)
<code>-o [-output] arg</code>	Specify the name of output shared library. Works with the <code>-link</code> option only.
<code>-work arg</code>	Specify the work directory in which to place the outputs. (Default: <code><current_directory>/xsim.dir/xsc</code>)
<code>-v [-verbose] arg</code>	Specify verbosity level for printing messages. Allowed values are: 0, 1 (Default: 0)

Examples

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

Example

```
xsc -compile function1.c function2.c -work abc
```

```
xsc -link abc/function1.lnx64.o abc/function2.lnx64.o -work abc
```

Binding Compiled C Code to SystemVerilog Using **xelab**

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

<code>-sv_root arg</code>	Root directory relative to which a DPI shared library should be searched. (Default: <code><current_directory>/xsim.dir/xsc</code>)
<code>-sv_lib arg</code>	Name of the DPI shared library without the file extension defining C function imported in SystemVerilog.
<code>-sv_liblist arg</code>	Bootstrap file pointing to DPI shared libraries.
<code>-dpiheader arg</code>	Generate a DPI C header file containing C declaration of imported and exported functions.

For more information on `r-sv_liblist arg`, refer to the *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language* [Ref 16], Appendix J.4.1, page 1228.

Data Types Allowed on the Boundary of C and SystemVerilog

The *IEEE Standard for SystemVerilog* [Ref 16] allows only subsets of C and SystemVerilog data types on the C and SystemVerilog boundary. Provided below are (1) details on data types supported in Vivado simulator and (2) descriptions of mapping between the C and SystemVerilog data types.

Supported Data Types

The following table describes data types allowed on the boundary of C and SystemVerilog, along with mapping of data types from SystemVerilog to C and vice versa.

Table E-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, sv_z, sv_x
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 arrays of bit, logic	Passed as array	Yes	None
Packed structs, unions	Passed as array	Yes	None
Unpacked arrays of bit, logic	Passed as array	Export Only	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 16].

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/Union

An equivalent unpacked type is created on the C side, in which all the members are converted to the equivalent C representation.

Examples:

SystemVerilog type:

```
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);
```

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

cat function1.c

```
#include "svdpi.h"

DPI_DLLESPEC
int myFunction1()
{
    return 5;
}
```

cat function2.c

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

cat file.sv

```
module m();

import "DPI-C" pure function int myFunction1 ();
import "DPI-C" pure function int myFunction2 ();

integer i, j;

initial
begin
#1;
    i = myFunction1();
    j = myFunction2();
    $display(i, j);
    if( i == 5 && j == 10)
        $display("PASSED");
    else
        $display("FAILED");
end

endmodule
```

Usage

Methods for compiling and linking the C files into the Vivado simulator are described below.

Single-step flow (simplest flow)

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

Flow description:

The xsc compiler compiles and links the C code to create the shared library `xsim.dir/xsc/dpi.so`, and xelab references the shared library through the switch `-sv_lib`.

Two-step flow

```
xsc -compile function1.c function2.c -work abc
xsc -link abc/function1.lnx64.o abc/function2.lnx64.o -work abc
xelab -svlog file.sv -sv_root abc -sv_lib dpi -R
```

Flow description:

- Compile the two C files into corresponding object code in the work directory `abc`.
- Link these two files together to create the shared library `dpi.so`.
- Make sure that this library is picked up from the work library `abc` via the `sv_root` switch.



TIP: `sv_root` specifies where to look for the shared library specified through the switch `sv_lib`.

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

```
xsc -compile function1.c function2.c -work "abc" -v 1
xsc -link "abc/function1.lnx64.o" "abc/function2.lnx64.o" -work "abc" -o final -v 1
xelab -svlog file.sv -sv_root "abc" -sv_lib final -R
```

Flow description:

If you want to do your own compilation and linking, you can use the verbose switch to see the path and the options with which the compiler was invoked. You can then tailor those to suit your needs. In the example above, a distinct shared library `final` is created. This example also demonstrates how spaces in file path work.

Function with Output

Code

cat file.sv

```

/*- - - */
package pack1;
import "DPI-C" pure function int myFunction1(input int v, output int o);
import "DPI-C" pure function void myFunction2 (input int v1, input int v2, output int o);
endpackage

/*-- ---*/
module m();
int i, j;
int o1 ,o2, o3;

initial
begin
#1;
    j = 10;
    o3 =pack1:: myFunction1(j, o1);//should be 10/2 = 5
    pack1::myFunction2(j, 2+3, o2); // 5 += 10 + 2+3
    $display(o1, o2);
    if( o1 == 5 && o2 == 15)
        $display("PASSED");
    else
        $display("FAILED");
end

endmodule

```

cat function.c

```

#include "svdpi.h"

DPI_DLLESPEC
int myFunction1(int j, int* o)
{
    *o = j /2;
    return 0;
}

DPI_DLLESPEC
void myFunction2(int i, int j, int* o)
{
    *o = i+j;
    return;
}

```

cat run.ksh

```
xsc function.c
xelab -vlog file.sv -sv -sv_lib dpi -R
```

Simple Import-Export Flow (illustrates xelab -dpiheader flow)

In this flow:

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

cat file.c

```
#include "file.h"
/* NOTE: This file is generated by xelab -dpiheader <filename> flow */

int cfunc (int a, int b) {
    //Call the function exported from SV.
    return c_exported_func (a,b);
}
```

cat test.sv

```
module m();
export "DPI-C" c_exported_func = function func;
import "DPI-C" pure function int cfunc (input int a ,b);

/*This function can be called from both SV or C side. */
function int func(input int x, y);
begin
    func = x + y;
end
endfunction

int z;

initial
begin
    #5;
    z = cfunc(2, 3);
    if(z == 5)
        $display("PASSED");
    else
        $display("FAILED");
end

endmodule
```

cat run.ksh

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

cat 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 F

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: Using the Vivado IDE* ([UG893](#))
4. *Vivado Design Suite User Guide: Using the Tcl Scripting Capabilities* ([UG894](#))
5. *Writing Efficient Testbenches* ([XAPP199](#))
6. *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* ([UG953](#))
7. *Vivado Design Suite Tcl Command Reference Guide* ([UG835](#))
8. *Vivado Design Suite User Guide: Power Analysis and Optimization* ([UG907](#))
9. *Vivado Design Suite User Guide: Using Constraints* ([UG903](#))
10. Vivado Design Suite Tutorial: Simulation ([UG937](#))
11. *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))
12. *Vivado Design Suite Properties Reference Guide* ([UG912](#))

Links to Additional Information on Third-Party Simulators

13. For more information on:

- QuestaSim/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

Links to Language and Encryption Support Standards

14. *IEEE Standard VHDL Language Reference Manual* ([IEEE-STD-1076-1993](#))
15. *IEEE Standard Verilog Hardware Description Language* ([IEEE-STD-1364-2001](#))
16. *IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language* ([IEEE-STD-1800-2009](#))
17. *Standard Delay Format Specification (SDF)* ([Version 2.1](#))
18. *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.

© Copyright 2012-2015 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.