

Vivado Design Suite User Guide:

Logic Simulation

UG900 (v2012.2) July 25, 2012



Notice of Disclaimer

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 the Limited Warranties which can be viewed at <http://www.xilinx.com/warranty.htm>; 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 Critical Applications: <http://www.xilinx.com/warranty.htm#critapps>.

© Copyright 2012 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, 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.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/25/12	2012.2	Initial Xilinx release of the Vivado Design Suite User Guide: Logic Simulation.

Table of Contents

Revision History	2
Chapter 1: Logic Simulation Overview	
Introduction	5
Simulation Flow	5
Simulation Modes of Operation	9
Supported Simulators	11
Vivado Simulator Features	12
Language Support	12
OS Support and Release Changes	12
Simulation Libraries	13
Chapter 2: Understanding Vivado Simulation Components	
Introduction	14
About Test Benches or Stimulus Files	14
About Simulation Libraries	16
Netlist Generation Process in Non-Project Mode	24
About Global Reset and Tristate for Simulation	26
Chapter 3: Running Simulation in Vivado IDE	
Introduction	28
Using Simulation Settings	28
Managing Simulation Sources	32
Using the Vivado Simulator	36
Pausing a Simulation	40
Saving Simulation Results	41
Closing Simulation	41
Chapter 4: Compiling and Simulating the Design	
Introduction	42
Parsing Design Files	43
Elaborating and Generating a Snapshot Using xelab	44
Using xsim to Simulate the Design Snapshot	49

Project File Syntax	50
Predefined XILINX_SIMULATOR Macro for Verilog Simulation	51
Simulating the Design in Non-Project Mode	51
Library Mapping File (xsim.ini)	54
xelab, xvhd, and xvlog Command Options	55
Using Mixed Language Simulation	58

Chapter 5: Analyzing and Debugging With Waveforms

Introduction	62
Launching the Vivado Simulator	63
Adding HDL Objects to the Wave Configuration	63
Using Non-Project Mode	68
Understanding HDL Objects in Waveform Configurations	69
About Wave Configurations and Waveform Windows	70
Controlling the Display of Waveforms	71
Customizing the Wave Configuration	80
Viewing Simulation Data from Prior Simulation Settings (Static Simulation)	86
Debugging at the Source Level	88

Appendix A: Running Simulation with Third Party Simulators Outside Vivado IDE

Introduction	91
Running RTL/Behavioral Simulation	91
Running Netlist Simulation	92
Running Timing Simulation	93

Appendix B: Verilog and VHDL Exceptions

Introduction	94
VHDL Language Support Exceptions	94
Verilog Language Support Exceptions	96

Appendix C: Additional Resources

Xilinx Resources	99
Solution Centers	99
Documentation References	99

Logic Simulation Overview

Introduction

Simulation is a process of emulating the real design behavior in a software tool. A simulation helps verify the functionality of a design by injecting stimulus and observing the design outputs. Simulators interpret VHDL or Verilog code into circuit functionality, and display logical results.

This document describes the simulation options available using the Xilinx® Vivado™ Integrated Design Environment (IDE).

This chapter provides an overview of the simulation process, and the simulation tool options in the Vivado IDE. The Vivado IDE is designed to be used with several HDL simulation tools that provide a solution for programmable logic designs from beginning to end.

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; it might need to be repeated until both the design functionality and the timing are met.

[Figure 1-1](#) illustrates the simulation flow for a typical design:

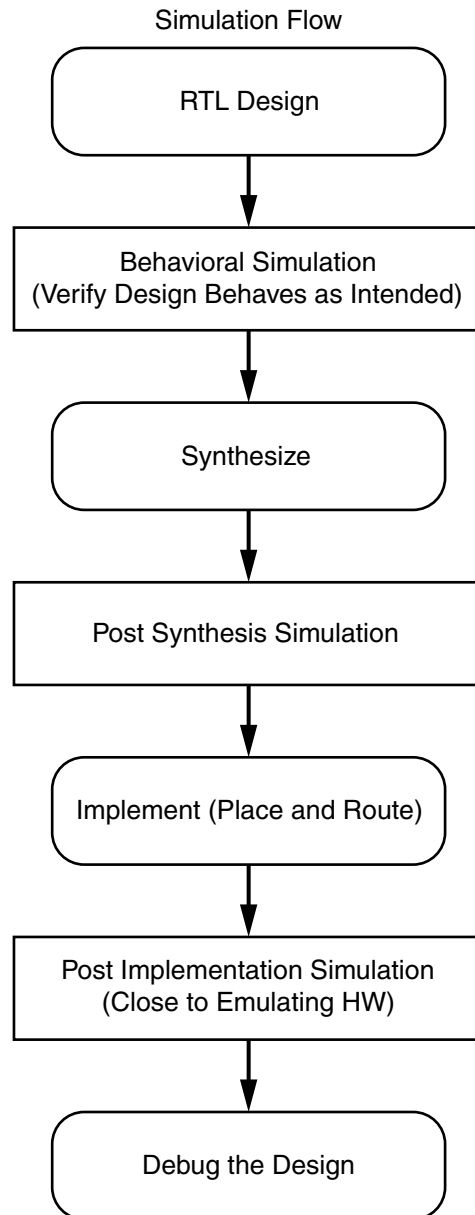


Figure 1-1: Simulation Flow

Behavioral Simulation at the Register Transfer Level

Register Transfer Level (RTL), behavioral simulation can include:

- RTL Code
- Instantiated UNISIM library components
- Instantiated UniMacro components
- XilinxCoreLib and UNISIM gate-level models (for the Vivado logic analyzer)
- SecureIP Library

The RTL-level simulation lets you verify or simulate a description at the system or chip level.

This first pass simulation is typically performed to verify code syntax, and to confirm that the code is functioning as intended. At this step, no timing information is provided, and simulation is performed in unit-delay mode to avoid the possibility of a race condition.

RTL simulation is not architecture-specific unless the design contains instantiated UNISIM or Lab Analyzer components. To support these instantiations, Xilinx® provides the UNISIM and XilinxCoreLib libraries. You can use the Vivado Lab Analyzer components if you do not want to rely on the module generation capabilities of the synthesis tool, or if the design requires larger structures.

When you verify your design decisions before the design is being implemented at the behavioral RTL you can make any necessary changes earlier and save design cycles.



TIP: *Keep the code behavioral for the initial design creation. Do not instantiate specific components unless necessary. You might find it necessary to instantiate components if the component is not inferable.*

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 do have the capability to 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 17](#) 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.

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

See [About Post-Synthesis or Post-Implementation Timing Simulation, page 54](#) for more information.

Simulation Modes of Operation

The Vivado simulator has two modes of operation: Project Mode and Non-Project Mode.

Project Mode: Using the Graphical User Interface (GUI)

When you run simulation using a project in the Vivado IDE, the Vivado simulator GUI opens and provides a graphical view of simulation data. You can use menu commands and toolbar buttons to run simulation, examine the design, and debug data.

In Project Mode, you can:

- Open or create a project.
- Create or add a test bench.

See [About Test Benches or Stimulus Files](#).

- From the Simulation section of the Flow Navigator in the Vivado IDE, select the **Simulation Settings** to select and setup the simulation options.

See [Using Simulation Settings](#) and [Managing Simulation Sources in Chapter 3](#).

- Click the **Run Simulation** button in the Flow Navigator to launch behavioral RTL simulation.
- Refer to Vivado IDE execution, error, and log messages.

See [Chapter 3, Running Simulation in Vivado IDE](#).

- Analyze and debug your design with the Vivado IDE common waveform viewer.

See [Chapter 5, Analyzing and Debugging With Waveforms](#).

[Figure 1-2](#) illustrates the simulation flow in Project Mode.

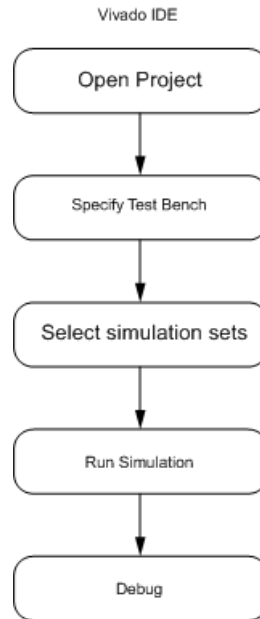


Figure 1-2: Vivado Simulation Project Mode Flow

Using Non-Project Mode: Command and Batch Script Options

In Non-Project Mode, you can:

- Create or add a test bench for functional RTL, post-synthesis, or post-implementation simulation.

Note: Post-synthesis and post-implementation simulation is supported in Non-Project Mode only.

- Specify source files, libraries, and file compilation order.
- Parse and elaborate a design using the Vivado simulator commands.

See [Chapter 4, Compiling and Simulating the Design](#).

- Run the Vivado simulator.
- Debug the design using the Vivado simulator waveform viewer.

See [Chapter 4, Compiling and Simulating the Design](#) for more information.

See [Appendix A, Running Simulation with Third Party Simulators Outside Vivado IDE](#).



IMPORTANT: *In Non-Project Mode, you must manage your source files.*

Figure 1-3 illustrates the simulation flow in Non-Project Mode.

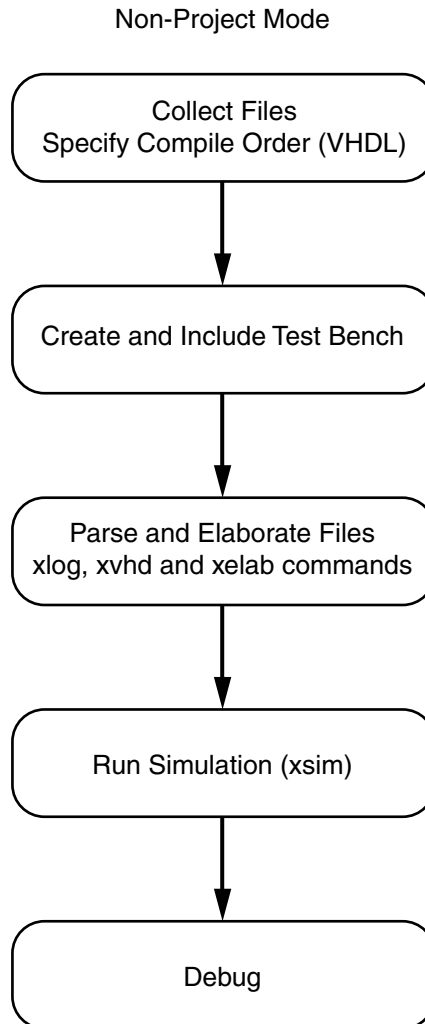


Figure 1-3: Simulation Flow in Non-Project Mode

Supported Simulators

The supported simulators are:

- Vivado simulator
- Mentor Graphics QuestaSim/ModelSim (integrated in the Vivado IDE)
- Cadence Incisive Enterprise Simulator (IES)
- Synopsys VCS and VCS MX
- Aldec Active-HDL*
- Aldec Rivera-PRO*

Note: (* Aldec simulators are compatible but not supported by Xilinx Technical Support. Contact your Aldec representative for any support issues.)

Vivado Simulator Features

The Vivado simulator supports the following features:

- Source code debugging
- SDF annotation
- VCD dumping
- SAIF dumping for power analysis and optimization
- Native support for HardIP blocks (such as MGT, PPC, and PCIe®)
- Multi-threaded compilation
- Mixed language (VHDL and Verilog) use
- Single-click simulation re-compile and re-launch
- One-click compilation and simulation
- Built-in Xilinx® simulation libraries

The *Xilinx Design Tools: Release Notes Guide (UG631)* [\[Ref 1\]](#) lists the currently supported simulator versions.

Note: Third party simulators (with the exception of QuestaSim/ModelSim, which can be selected and invoked from the Vivado IDE) must be run outside of the Vivado IDE.

Language Support

The following languages are supported:

- VHDL IEEE-STD-1076-1993
- Verilog IEEE-STD-1364-2001
- Standard Delay Format (SDF) version 2.1
- VITAL-2000

OS Support and Release Changes

The *Xilinx Design Tools: Release Notes Guide (UG631)* [\[Ref 1\]](#) provides information about the most recent release changes.

See the *Xilinx Design Tools: Installation and Licensing Guide (UG798)* [\[Ref 2\]](#) for operating systems support.

Simulation Libraries

The libraries required to support simulation flows are:

- SIMPRIM_VER - Verilog UNISIM with timing
- XILINXCoreLib- VHDL IP
- XILINXCORELIB_VER - Verilog IP
- UNISIM - VHDL UNISIM
- UNISIMS_VER - Verilog UNISIM
- UNIMACRO - VHDL UNIMACRO
- UNIMACRO_VER - Verilog UNIMACRO
- SECUREIP - Verilog Hard IP
- UNIFAST - Fast simulation VHDL library
- UNIFAST_VER - Fast simulation Verilog library

[About Simulation Libraries in Chapter 2](#) describes the simulation libraries.

Also, see the *7 Series FPGA Libraries Guide for HDL Designs (UG768)* [\[Ref 6\]](#).

Note: The Vivado simulator uses precompiled simulation device libraries and updates those libraries automatically when updates are installed.

[Table 1-1](#) lists the points of simulation and which simulation library is used at that simulation point.

Table 1-1: Simulation Points and Relevant Libraries

	UNISIM	Unifast	UniMacro	XilinxCoreLib Models	SecureIP	SIMPRIM (Verilog Only)	Standard Delay Format (SDF)
1. Register Transfer Level (RTL)	Yes	Yes	Yes	Yes	Yes	N/A	Yes
2. Post-Synthesis Simulation	Yes	Yes	N/A	N/A	Yes	Yes	Yes
3. Post-Implementation Simulation	Yes	Yes	N/A	N/A	Yes	Yes	Yes

See [Chapter 2, Understanding Vivado Simulation Components](#), for more information about simulation libraries.

Understanding Vivado Simulation Components

Introduction

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

Note: Simulation libraries are precompiled in the Vivado Design Suite for use with the Vivado simulator. You must compile libraries when using third party simulators.

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 using a third-party simulator)
- Writing a netlist (if performing Post-Synthesis or Post-Implementation simulation)
- Understanding the use of global reset and tristate in Xilinx devices

The following sections describe these required components.

About Test Benches or Stimulus Files

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

- Instantiates the design
- Initializes the design
- Generates and applies stimulus to the design
- Optionally, monitors the design output result and checks for functional correctness

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 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

A test bench has the following advantages over interactive simulation:

- It allows repeatable simulation throughout the design process.
- It provides documentation of the test conditions.



TIP: Choose Vivado simulator as the target simulator in GUI

(see [Using Simulation Settings in Chapter 3](#)) and click **Run Simulation**.

This creates a `<testbench>.prj` file in the `project/project.sim/sim_1` directory, which has the collection of all the sources needed to simulate the design. Use that file as starting point for gathering files.

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

- Specify the name as `testbench` to the main module or entity name in the test bench file.
- Always specify the ``timescale` in Verilog test bench files.
- Initialize all inputs to the design within the test bench at simulation time zero to properly begin simulation with known values.
- Apply stimulus data after 100 ns to account for the default Global Set/Reset (GSR) pulse used in UNISIM and SIMPRIM-based simulation.
- Begin the clock source before the Global Set/Reset (GSR) is released.

For more information, see [About Global Reset and Tristate for Simulation, page 26](#).

For more information about test benches, refer to the *Writing Efficient TestBenches (XAPP199)* application note [\[Ref 5\]](#).

About Simulation Libraries

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.

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

Table 2-1: Simulation Libraries

Library Name	Description	VHDL	Verilog
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
XilinxCoreLib	Functional simulation of Xilinx cores.	xilinxcorelib	xilinxcorelib_ver
SIMPRIM	Timing simulation of Xilinx primitives.	N/A	SIMPRIM_VER
SecureIP	Simulation library for both functional and timing simulation of Xilinx device features, such as the: <ul style="list-style-type: none"> • PowerPC® processors • PCIe® technology • Gigabit Transceiver • Ethernet MAC 	secureip	-lib secureip

It is important to note that:

- 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: Libraries Required in Simulation Points

Simulation Point	Required Library
Register Transfer Level (RTL)	UNISIM Unifast UniMacro XilinxCoreLib SecureIP

Table 2-2: Libraries Required in Simulation Points (Cont'd)

Simulation Point	Required Library
Post-Synthesis Simulation	UNISIM (Functional Netlist) SIMPRIM (Timing Netlist) SecureIP
Post-Implementation Simulation	UNISIM (Functional Netlist) SIMPRIM (Timing Netlist) SecureIP

Note: Verilog SIMPRIM uses the same source as UNISIM with the addition of specify blocks for timing annotation. This is enabled by ``ifdef XIL_TIMING` in UNISIM source code.

Table 2-3 lists the library locations.

Table 2-3: Simulation Library Locations

Library	HDL Type	Location
UNISIM	Verilog	<Xilinx Install>/PlanAhead/data/verilog/unisims
	VHDL	<Xilinx Install>/PlanAhead/data/vhdl/unisims
UNIFAST	Verilog	<Xilinx Install>/PlanAhead/data/verilog/unifast
	VHDL	<Xilinx Install>/PlanAhead/data/vhdl/unifast
UNIMACRO	Verilog	<Xilinx Install>/PlanAhead/data/verilog/unimacro
	VHDL	<Xilinx Install>/PlanAhead/data/vhdl/unimacro
SecureIP	Verilog	<Xilinx Install> /PlanAhead/data/secureip/<simulator>/<simulator>_cell.list

The following subsections describe the libraries in more detail.

UNISIM Library

The UNISIM library is used during functional simulation and contains descriptions for all the device primitives, or lowest-level building blocks.

VHDL UNISIM Library

The VHDL UNISIM library is divided into the following files:

- The component declarations (`unisimVCOMP.vhdl`)
- Package files (`unisim_VPKG.vhd`)
- Entity and architecture declarations (`unisim_VITAL.vhdl`)

Primitives for Xilinx device families are specified in these files.

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

```
library UNISIM;  
use UNISIM.Vcomponents.all;
```

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 built-in.

Verilog UNISIM Library

For Verilog, specify each library component is in a separate file. This allows the Verilog `-y` library specification switch to perform automatic library expansion.

Specify Verilog module names and file names in upper case.

For example, module `BUFG` is `BUFG.v`, and module `IBUF` is `IBUF.v`.

See [Appendix A, Running Simulation with Third Party Simulators Outside Vivado IDE](#) for examples that use the `-y` switch.

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

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

```
-L unisims_ver
```

UniMacro Library

The UniMacro library is used during functional simulation and contains macro descriptions for selective device primitives. You must specify the UniMacro library anytime you include a device macro listed *7 Series FPGA Libraries Guide for HDL Designs (UG768)* [\[Ref 6\]](#).

VHDL UniMacro Library

Add the following library declaration to the top of your HDL file:

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

Verilog UniMacro Library

For Verilog, specify each library component in a separate file. This allows automatic library expansion using the `-y` library specification switch.



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.

You must also compile and map the library: the method you use depends on the simulator.

The following is an compilation and mapping example for the Vivado simulator:

```
-L unimacro_ver
```

XilinxCoreLib Library

Use the `XilinxCoreLib` library during RTL Behavioral simulation for designs that contain certain cores created by the Xilinx IP catalog.

SIMPRIM Library

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



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

Specify this library as follows:

```
-L SIMPRIM_VER
```

Where:

- `-L` is the library specification command.
- `SIMPRIM_VER` is the logical library name to which the Verilog `UniMacro` has been mapped.

SecureIP Simulation Library

Use the `SecureIP` library for functional and timing simulation of complex FPGA components, such as: `PPC` and `GT`.

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

Xilinx leverages the encryption methodology as specified in Verilog LRM - IEEE Std 1364.2001. The library compilation process automatically handles encryption.

When running a simulation using Verilog code, you must reference the `SecureIP` library. For most simulators, this can be done by using the `-L` switch as an argument to the simulator, such as:

```
-L SECUREIP_VER.
```

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

Table 2-4 lists special considerations that must be arranged with your simulator vendor for using these libraries.

Table 2-4: 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		
IUS	Cadence	An export control regulation license is required.
VCS	Synopsys	Use of the <code>-lca</code> switch with the VCS commands is required.

VHDL SecureIP Library

To use SecureIP in VHDL, place the following two lines at the beginning of each file:

```
Library UNISIM;
use UNISIM.vcomponents.all;
```

Verilog SecureIP Library

You can use the Verilog SecureIP library at compile time by leveraging the `-f` switch. The following is a command-line example for VCS:

```
vcs -f <Xilinx Install>/PlanAhead/data/secureip/VCS/vcs_cell.list.f \
<other simulation commands>
```

If you are 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_VER
```

UNIFAST Library

The UNIFAST library is an optional library that can be used during RTL behavioral simulation to speed up simulation runtime. The simulation runtime speed up is achieved by supporting a subset of the primitive features in the simulation model.

Fast MMCME2 UNISIM Simulation Model

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

1. The fast simulation model just has basic clock generation function. All 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 sample stops after `LOCKED 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, then VCO to output clocks.

4. The `LOCKED` assert times are different between the standard and the fast `MMCME2` simulation model.
 - Standard Model depends on the `M` the `D` setting, for large `M` and `D` values, the lock time is relatively long for standard `MMCME2` simulation model.
 - In fast simulation model, this time is shortened.

RAMB18E1/RAMB36E1

To reduce the simulation runtimes, a fast Block RAM simulation model has the following features removed from the full model:

- Error Correction Code (ECC)
- Collision checks
- Cascade mode

FIFO18E1/FIFO36E1

To reduce the simulation runtimes, a fast FIFO simulation model has the following features removed from the full model:

- ECC
- Design Rules Check (DRC) for `RESET`
- DRC check for `almostempty` and `almostfull` offset
- Output padding – `X` for data out, `1` for counters
- First word fall-through
- `almostempty` and `almostfull` flags

DSP48E1

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

- ALU
- Pattern Detect

OverFlow/UnderFlow

To reduce the simulation runtimes, a fast OverFlow/UnderFlow simulation model has the following features removed from the full model:

- PLLE2_ADV
- DRP interface support

Using Verilog UNIFAST Library

Method 1: Configurations in Verilog

In method 1, the following are specified in a `config.v` file.

- Specify the name of the top-level module or configuration (for example: `config cfg_xilinx;`)
- Specify the name to which the design configuration applies (for example: `design testbench;`)
- Define the library search order for cells or instances that are not explicitly called out (for example: `default liblist unisims_ver unifast_ver;`)
- Map a particular CELL or INSTANCE to a particular library (for example: `instance testbench.inst.01 use unifast_ver.OR2;`)

Note: For ModelSIM (vsim) only - `genblk` gets added to hierarchy name (for example: `instance testbench.genblk1.inst.genblk1.01 use unifast_ver.OR2; - VSIM`)

Example config.v

```
config cfg_xilinx;
design testbench;
default liblist unisims_ver unifast_ver;
cell AND2 use unifast_ver.AND2;
instance testbench.inst.01 use unifast_ver.OR2;
(instance testbench.genblk1.inst.genblk1.01 use unifast_ver.OR2; - VSIM)
endconfig
```

Method 2: Using Library or File Compile Order

This second method lists the compile order by simulator and code type:

- **Vivado Simulator - Verilog -L or -sourcelibdir order selects all fast / no fast.**

```
xelab -L unifast_ver -L unisims_ver testbench glbl -s mysim
```

```
xvlog -sourcelibdir ../verilog/src/unifast -sourcelibext .v -sourcelibdir
../verilog/src/unisims; xelab testbench glbl -s mysim
```

- **Modelsim - Verilog -L or -y order selects all fast / no fast.**

```
vsim -c -L unifast_ver -L unisims_ver testbench glbl
vlog -y ../verilog/src/unifast +libext+.v -y ../verilog/src/unisims; vsim -c
testbench glbl
```

- **NCSIM - Verilog - same results with different command lines.**

```
ncelab -libname unifast_ver -libname unisims_ver
ncverilog -y ../verilog/src/unifast +libext+.v -y ../verilog/src/unisims
+libext+.v \
```

- **VCS - Verilog -lib does not imply order - change synopsys_sim.setup to order libraries**

```
vcs -work work -y ../Verilog/src/unifast +libext+.v -y ../verilog/src/unisims
+libext+.v ../netlist.v testbench glbl
```

Using VHDL UniFast Library

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

```
library work;
library unisim;
library unifast; -- (different libs use case)
configuration cfg_xilinx of testbench is for xilinx for inst:netlist use entity
work.netlist(inst); for inst
for all:AND2 use entity unifast.AND2; -- could be unisim.AND2(fast) (architecture
use case) end for;
for O1:OR2 use entity unifast.OR2; -- could be unisim.OR2(fast) (architecture
use case)
end for;
end for; end for; end for;
end cfg_xilinx;
```

Compiling Simulation Libraries

Before you can simulate your design, you must compile the applicable libraries and map them to the simulator.

Use the `compile_simlib` Tcl command in the Vivado Design Suite Tcl Console for compiling Xilinx HDL-based simulation libraries for third party simulation vendors. Libraries are typically compiled (or recompiled) anytime a new simulator version is installed.

See the *Vivado Tcl Command Reference Guide (UG835)* [Ref 4] for more information.



TIP: When you use `testbench` as the test bench name, compilation changes are necessary to perform simulation from the Vivado IDE.

Netlist Generation Process in Non-Project Mode

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

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

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

These commands can generate functional and timing simulation netlists at any point in the design process.



TIP: The SDF values 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 for Simulation

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 information) to check that the behavior of the structural netlist implemented by the Vivado IDE 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:

- `UNISIM_VER` simulation library for Verilog simulation
- `UNISIM` 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 is the Verilog and VHDL syntax for generating a functional simulation netlist:

```
write_verilog -mode funcsim <verilogNetlistName>
write_vhdl -mode funcsim <vhdlNetlistName>
```

Generating a Timing Netlist for Simulation

You can use a Verilog timing simulation to verify circuit operation after you 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.

Note: The Vivado IDE supports Verilog timing simulation only.

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

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

Annotating the SDF File

The following is the syntax for annotating an SDF file:

```
write_sdf -process_corner fast test.sdf
```

Running Setup and Hold Checks

Based on the specified process corner, the SDF file has different `min` and `max` numbers. Xilinx recommends running *two separate simulations* to check for setup and hold violations.

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

For example, in ModelSim, specify:

```
-sdfmax
```

To run a hold check, create an SDF with `-process` corner `fast`, and use the `min` column from the SDF. For example, in ModelSim, specify:

```
-sdfmin
```

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

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



TIP: *Ensure that Vivado IDE timing simulation is run with the switches specified in the simulator settings dialog box to prevent pulse swallowing through the Interconnect.*

About Global Reset and Tristate for Simulation

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

Global Set/Reset Net

When you assert the dedicated Global Set/Reset (GSR) net, that net is released during configuration immediately after the device is configured. All the flip-flops and latches receive this reset, and are either set or reset, depending on how the registers are defined.

Although you can access the GSR net after configuration, Xilinx does not recommend using 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 pulsed for the first 100 ns to simulate the reset that occurs after configuration. A GSR pulse can optionally be supplied in Early (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 Tristate Net

In addition to the dedicated global GSR, output buffers are set to a high impedance state during configuration mode with the dedicated Global Tristate (GTS) net. All general-purpose outputs are affected whether they are regular, tristate, or bidirectional outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the FPGA device 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 Early (Pre-Synthesis) functional simulation, but the GTS pulse width is set to 0 by default.

Using Global Tristate (GTS) and Global Set/Reset (GSR) Signals

Figure 2-1 shows how Global Tristate (GTS) and Global Set/Reset (GSR) signals are used in an FPGA device.

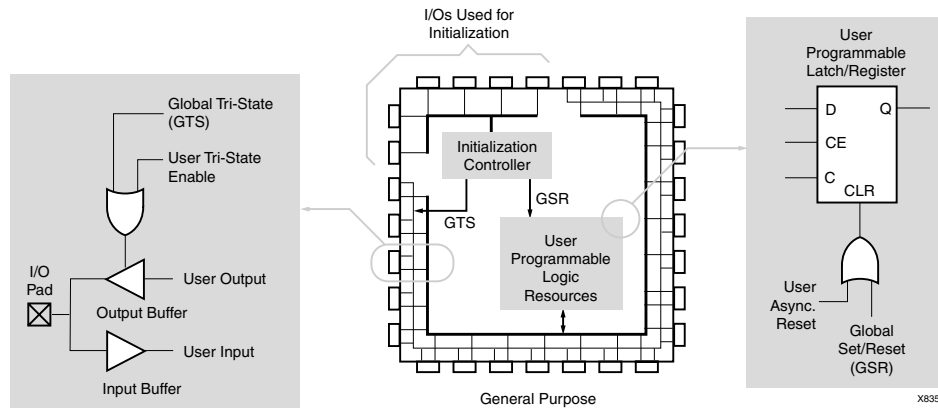


Figure 2-1: Built-in FPGA Initialization Circuitry Diagram

Global Set/Reset (GSR) and Global Tristate (GTS) in Verilog

The Global Set/Reset (GSR) and Global Tristate (GTS) signals are defined in the `$XILINX/PLANAHEAD/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.

Running Simulation in Vivado IDE

Introduction

This chapter describes the Vivado™ Integrated Design Environment (IDE) simulator Graphical User Interface (GUI), which is a selection option when you set up a native mixed language simulator in the Vivado IDE.

The Vivado IDE provides an integrated simulation environment when using the Vivado simulator for behavioral simulation.

Using Simulation Settings

The **Flow Navigator** > **Simulation Settings** section lets you configure the simulation settings in Vivado IDE. The Flow Navigator Simulation section is shown in [Figure 3-1](#).

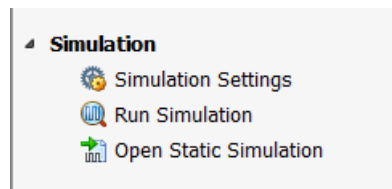


Figure 3-1: Simulation Snippet in the Flow Navigator

- **Simulation Settings:** Opens the Simulation Settings dialog box where you can select and configure the Vivado simulator.
- **Run Simulation:** Sets up the command options to compile, elaborate, and simulate the design based on the simulation settings, then launches the Vivado simulator. The Vivado simulator opens a waveform window that shows the HDL objects with the signal and bus values in either digital or analog form. See [Figure 3-2](#).
- **Open Static Simulation:** Lets you open a previously-created waveform configuration (WDB) file. See [Viewing Simulation Data from Prior Simulation Settings \(Static Simulation\)](#) in [Chapter 5](#).

When you select **Simulation Settings**, the Project Settings dialog box opens, as shown in Figure 3-2.

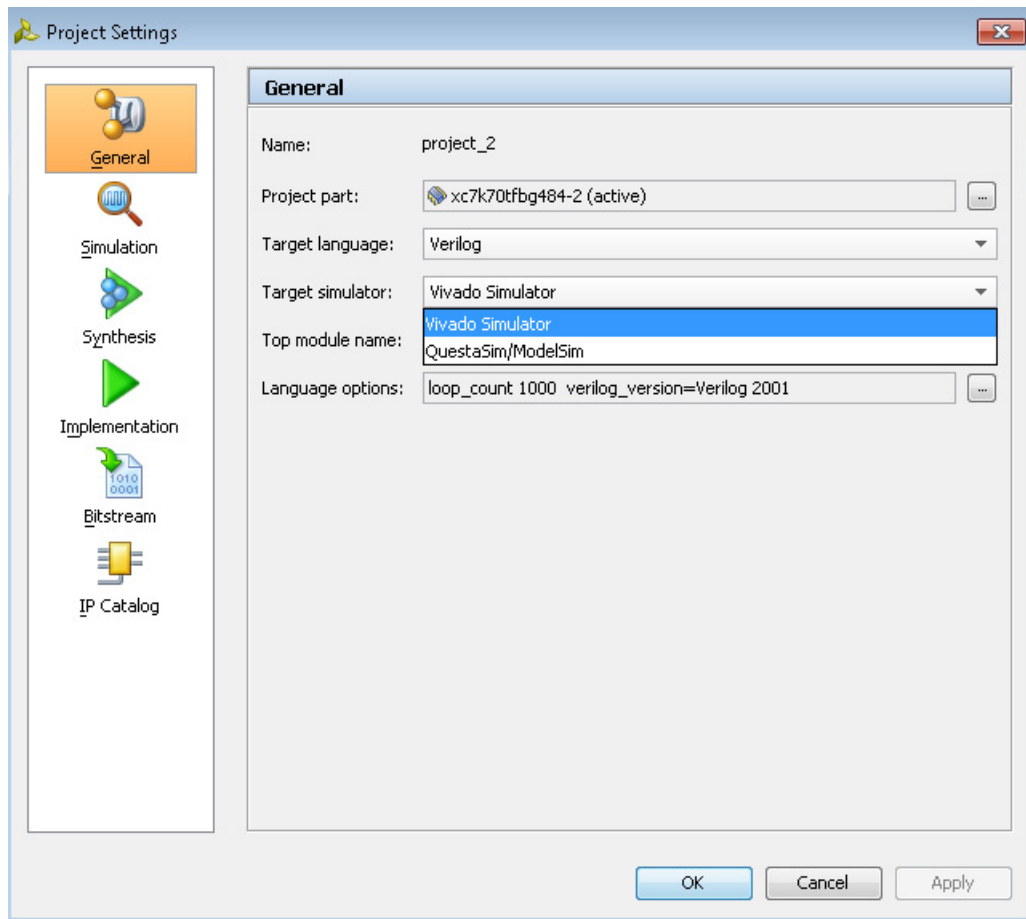


Figure 3-2: Project Setting: Target Simulator Options

The Project Setting dialog box contains the **Target Simulator** that lets you select the **Vivado simulator** or **QuestaSim/ModelSim**.

After you select the simulator, and apply the options from Project Settings, the Simulation dialog box opens, as shown in Figure 3-3, page 30.



IMPORTANT: Because the Vivado simulator has precompiled libraries, it is not necessary for you to identify the library locations. The Simulation dialog box has an additional field for library location when you select **QuestaSim/ModelSim**.

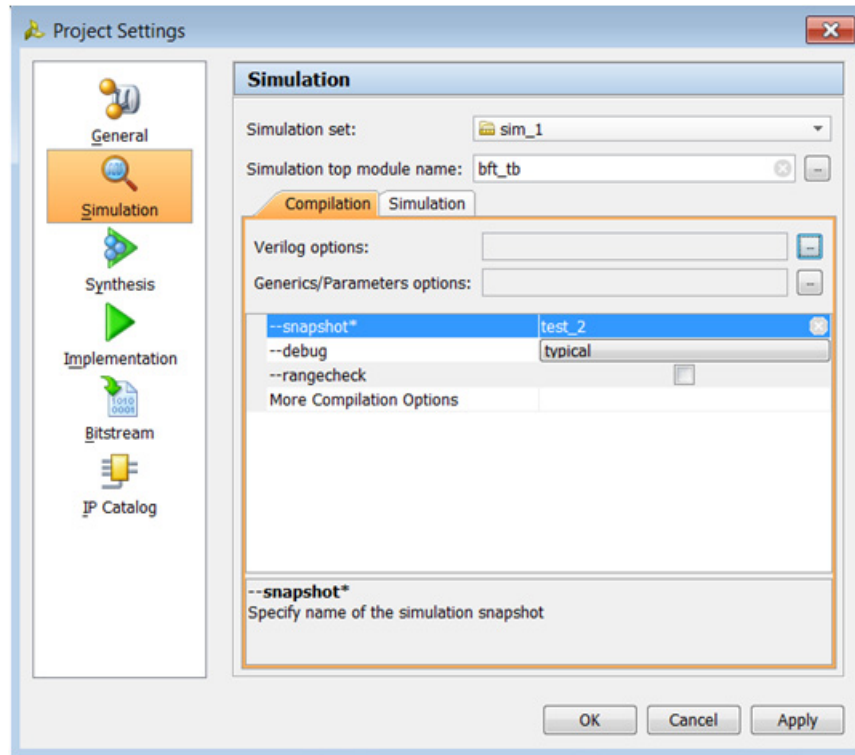


Figure 3-3: Project Settings: Simulation Dialog Box: Compilation Settings

The Simulation selection options are:

- **Simulation set:** Select an existing simulation set or use the **Create simulation set** option, described in [Editing Simulation Sets, page 34](#).
- **Simulation top module name:** Set the simulation top module.
- **Compiled library location:** When you select **QuestaSim/ModelSim** as the **Target Simulator**, this field displays the precompiled library.
 - See the [About Simulation Libraries, page 16](#) for information about how to specify simulation libraries.
 - See [Appendix A, Running Simulation with Third Party Simulators Outside Vivado IDE](#), for more information regarding QuestaSim/ModelSim.
- **Compilation View:** Provides browse and select options for frequently-used compilation options.
 - **Verilog options:** Select the version of Verilog code to use.
 - **Generics/Parameters options:** Select the required VHDL generics or Verilog parameters.

- **Command options:**
 - `-snapshot`: Specify a name for the precompiled simulation *snapshot* to use in the simulation. A snapshot is the executable output of the simulator compilation process.
 - `-debug`: Is set to `typical` by default for a faster simulation run.
 - `-rangecheck`: Is set to `off` by default for a faster simulation run.
- **Simulation View:** Provides available simulation options. You can select an option to see a description. Figure 3-4 shows the Simulation view:

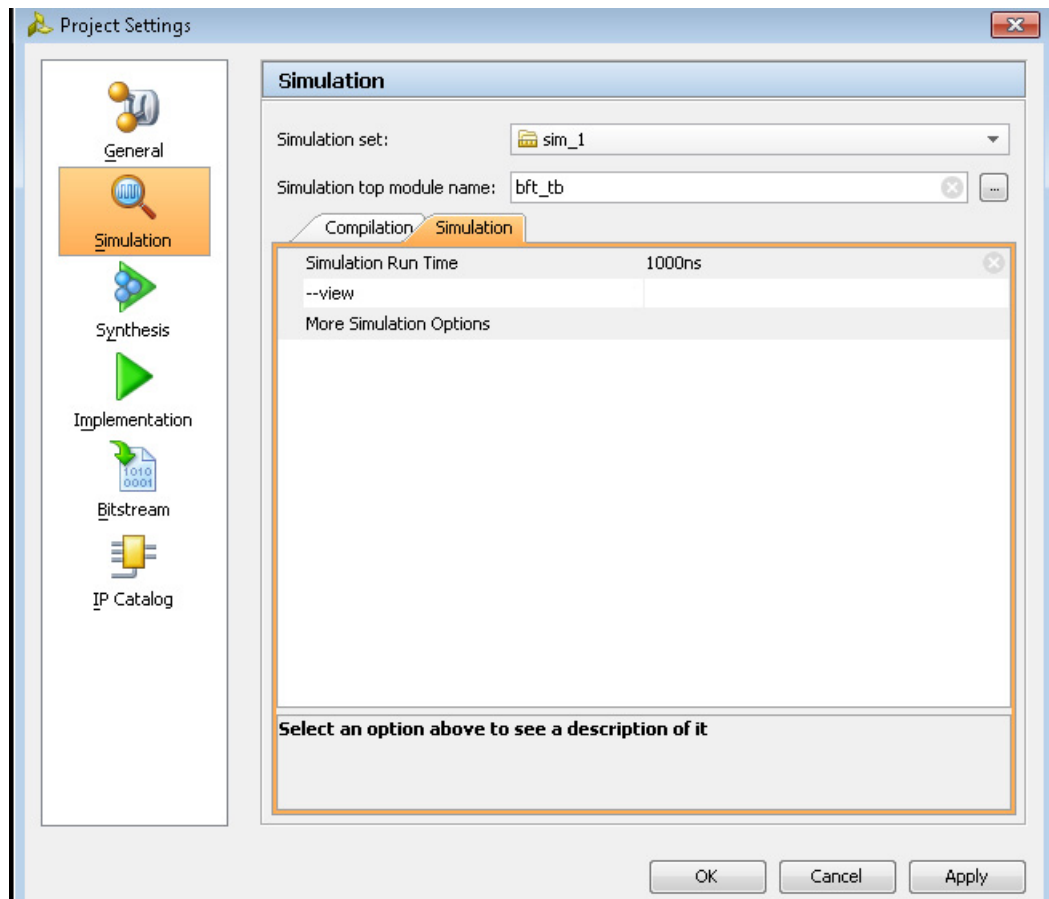


Figure 3-4: Project Settings: Simulation: Simulation View

Select from the following options in the Simulation view:

- **Simulation Run Time:** Specifies the amount of simulation time to run automatically when the simulation is launched. The defaults is 1000 ns.
- `--view`: Lets you open a previously-saved wave configuration (WCFG) file. A wave configuration is a list of HDL objects to display in a waveform window.
- **More Simulation Options:** More simulation options are available.

When you select an option a tooltip provides an option description.



TIP: You can save a WCFG file from an initial run with signals of interest in an analog or digital waveform with dividers and then have the GUI open the WCFG file using the `-view` option in later runs, which can save time on setting up simulation waveforms.

Managing Simulation Sources

The Vivado Design Suite lets you add simulation sources to the project for RTL behavioral simulation. Simulation source files include Hardware Definition Language (HDL)-based test bench files to use as a stimulus for simulation.

The Vivado Design Suite stores simulation source files in simulation sets, called *simsets*, that display in folders in the Sources view, and are either remotely referenced or stored in the local project directory.

The simset 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 simulation sources to the project, you can specify which simulation source set into which to add files.

Adding or Creating Simulation Source Files

To add simulation sources to a project:

1. Select **File > 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 as shown in [Figure 3-5](#).

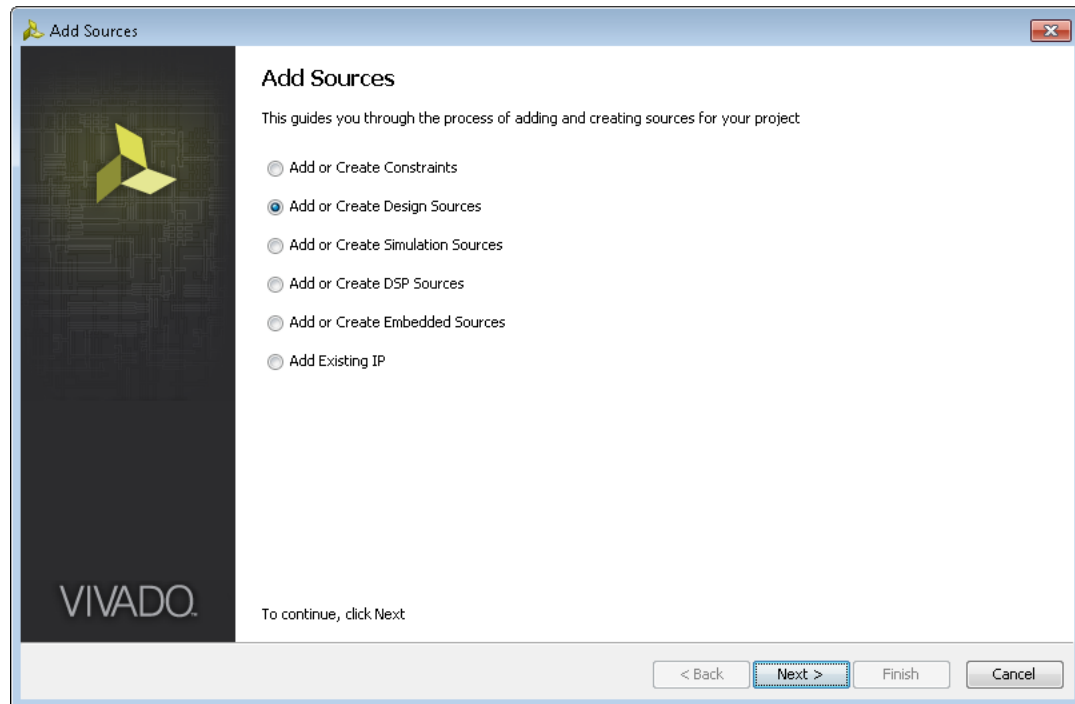





Figure 3-5: Add or Create Simulation Sources Dialog Box

The Add or Create Simulation Sources dialog box options are:

- **Specify Simulation Set:** Enter the name of the simulation set to test bench files and directories. If there is one or more simulation sets already defined, select the **Create Simulation Set** command from the drop-down menu to define a new simulation 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. See "Adding and Creating Source Files" in the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] 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. 

Checkboxes in the wizard provide the following options:

- **Scan and Add RTL Include Files into Project:** Scans the added RTL file and adds any referenced include files.
- **Copy Sources into Project:** Copies the original source files into the project and uses the local copied version of the file in the project.

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

- **Add Sources from Subdirectories:** Adds source files from the subdirectories of directories specified in the **Add Directories** option.

Editing Simulation Sets

To edit a simulation set:

1. In the Sources window popup menu, select **Simulation Sources > Edit Simulation Sets**, as shown in [Figure 3-6](#).

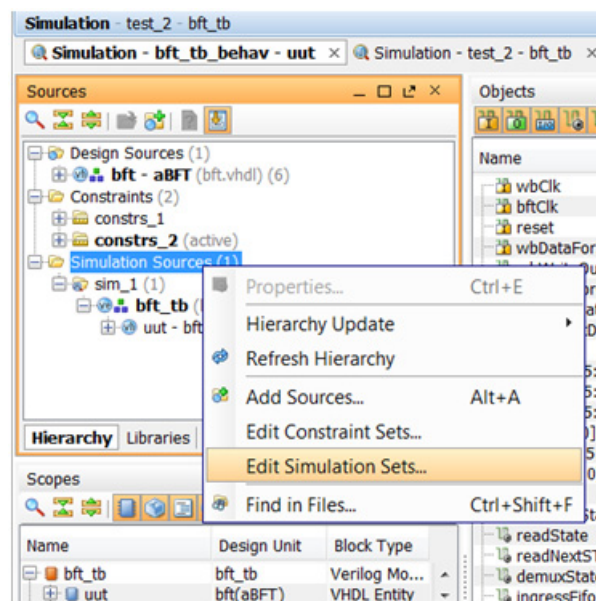


Figure 3-6: Edit Simulation Sets Option

The Add Sources wizard opens.

2. From the **Specify simulation set** drop-down, select **Create Simulation Set**, as shown in [Figure 3-7](#).

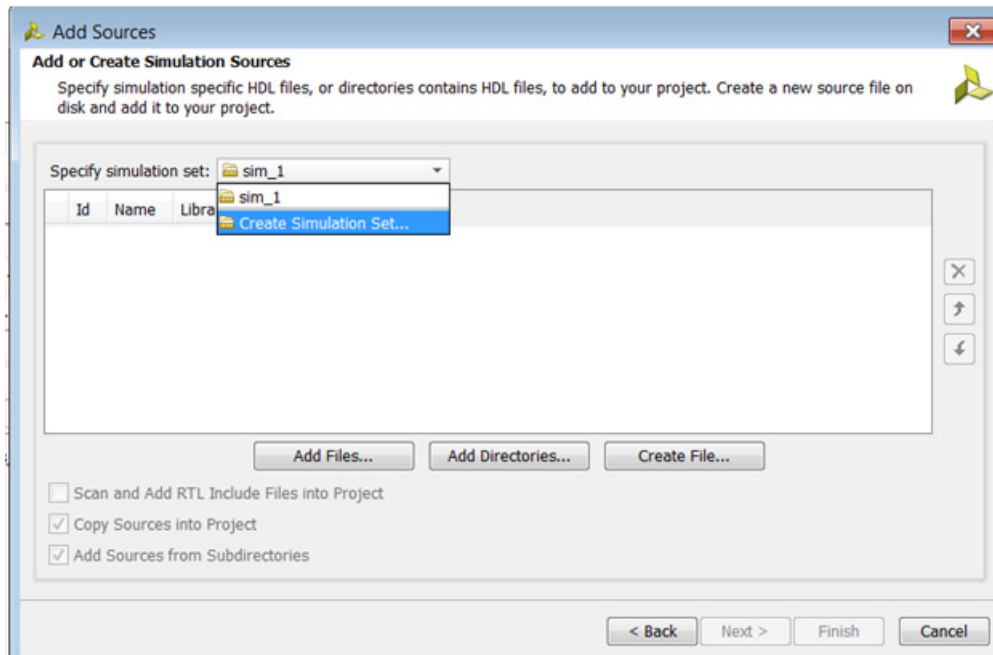


Figure 3-7: Add or Create Simulation Sources: Create Simulation Set Drop-Down

The sources associated with the project are added to the newly-created simulation set.

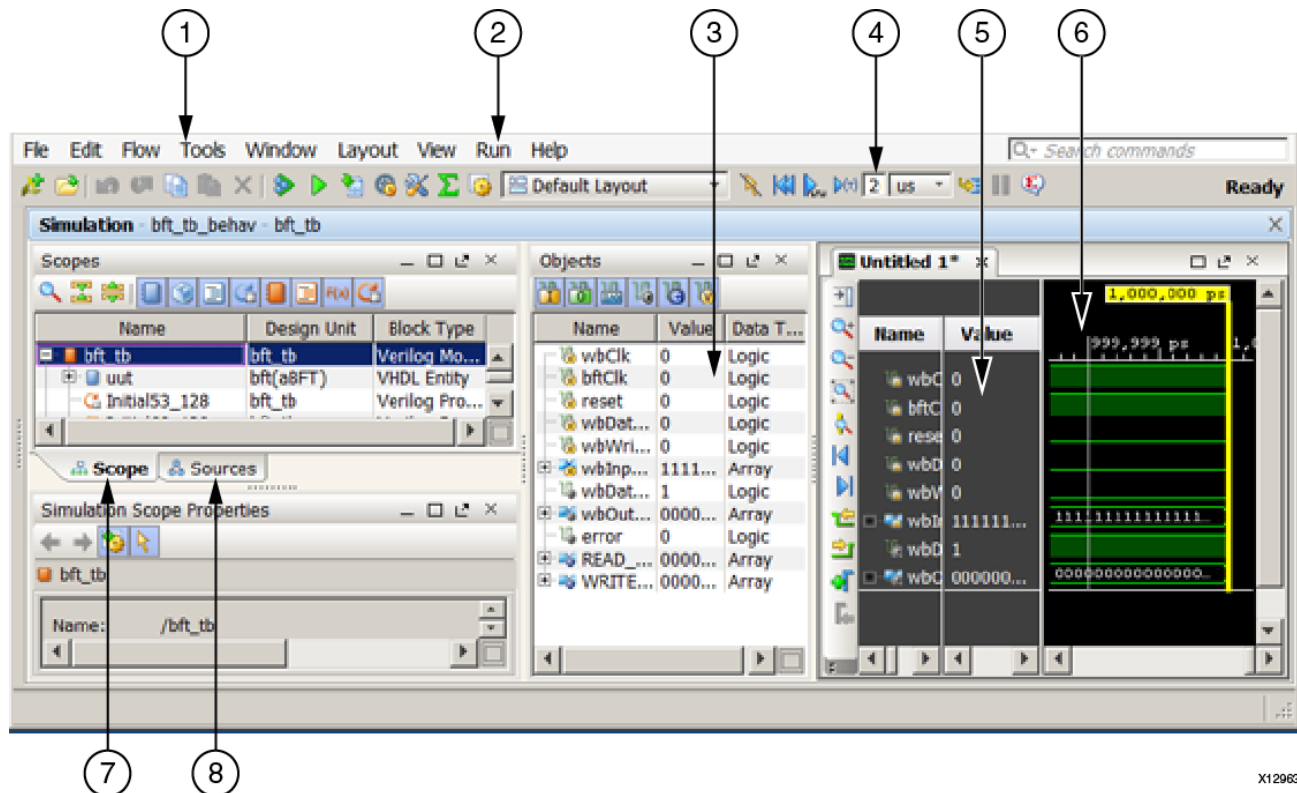
3. Add additional files as needed.



IMPORTANT: *Compilation and simulation setting set for one defined simulation set are not applied to a newly-defined simulation set: sim_1 compilation and simulation settings do not apply to sim_2, and so forth.*

Using the Vivado Simulator

When you have selected the Vivado simulator, from the Flow Navigator select **Run Simulation**. The Vivado simulator GUI, shown in [Figure 3-8](#), opens and displays the simulation.



X12963

Figure 3-8: Vivado Simulator GUI

Understanding the Vivado IDE Simulator Window

The following subsections correspond to the numbered areas of the Vivado simulator.

1. Flow Navigator Simulation Section

The Flow Navigator Simulation section, described in [Using Simulation Settings, page 28](#), is available.

2. 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.

See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] for a description of the Vivado IDE menus. The **Run** menu for simulation is shown in [Figure 3-9](#).

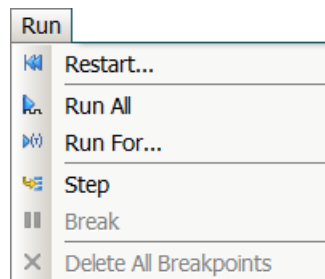


Figure 3-9: Simulation Menu Options

The Simulation menu options are as follows:

- **Restart:** Lets you restart an existing simulation time to 0.
- **Run All:** Lets you run an open simulation to completion.
- **Run For:** Lets you specify a time for the simulation to run.
- **Step:** Runs the simulation up to the next HDL source line.
- **Break:** Lets you interrupt a running simulation.
- **Delete All Breakpoints:** Deletes all breakpoints.

3. Objects Windows

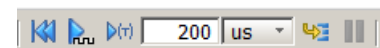
The HDL Objects window displays the HDL objects .

See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] for a description of the Sources window.

Buttons beside the HDL objects show the language or process type. This view lists the **Name**, **Design Unit**, and **Block Type** of the simulation objects. You can hover over the **Object** buttons for descriptions.

4. Simulation Toolbar

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



When you hover your mouse over the toolbar buttons, a tooltip describes the button function. The buttons are also labeled in [Figure 3-9](#).

5. and 6. HDL Objects and Waveform Window

The Vivado IDE waveform window is common across a number of Vivado Design Suite tools.

See [Chapter 5, Analyzing and Debugging With Waveforms](#) for more information about using the waveform.

An example of the waveform window is shown in [Figure 3-10](#).

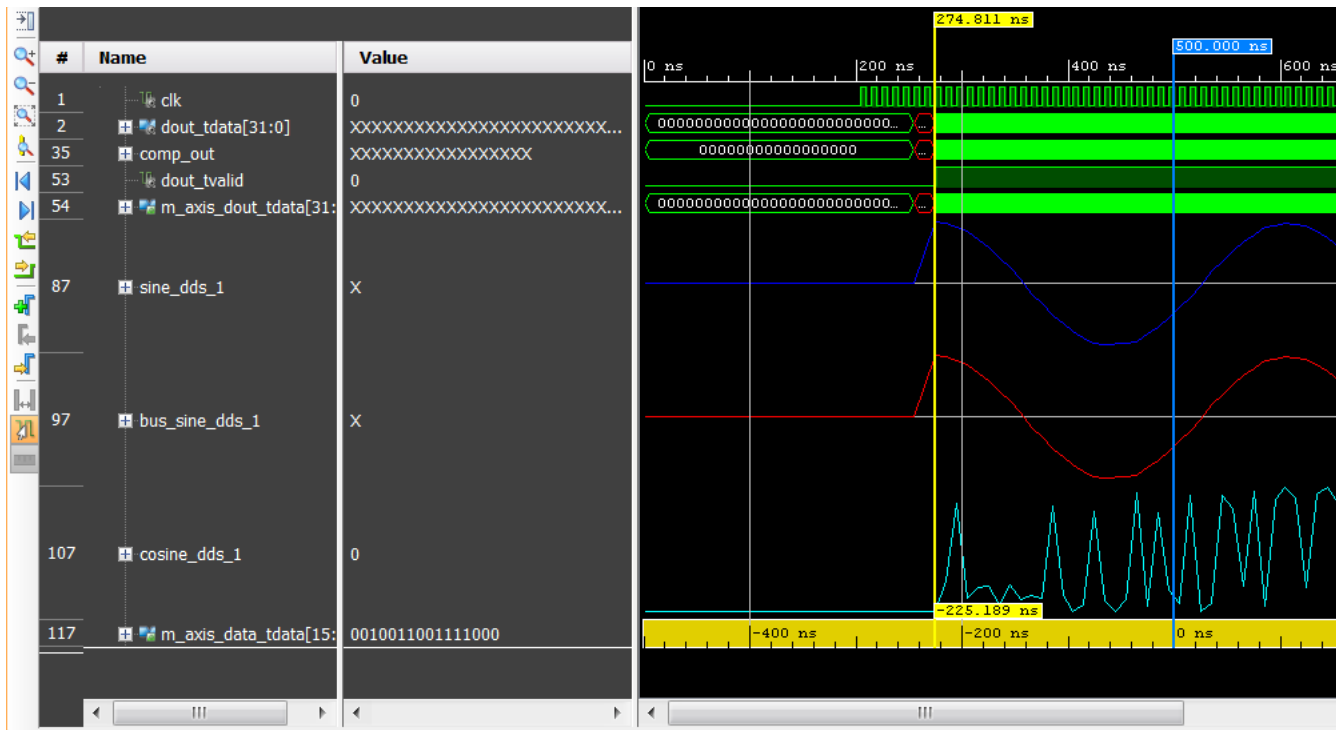


Figure 3-10: Waveform Window

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

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

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

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

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

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

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

7. 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 3-11](#).

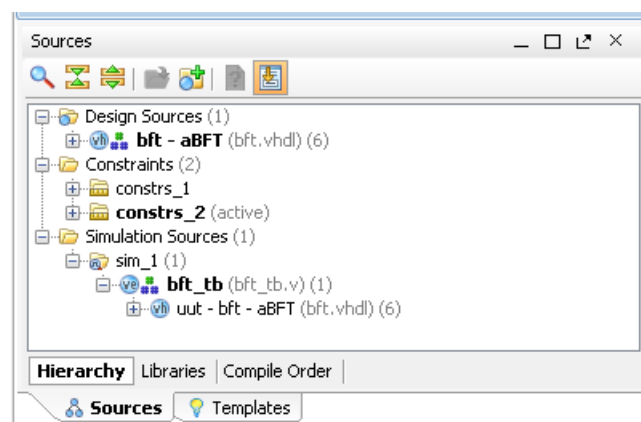


Figure 3-11: Sources Window

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

8. Scopes Window

Figure 3-12 shows the Scopes window, where you can view and filter HDL objects by type using the filter icons at the top of the window. Hover over an icon for a tooltip description of what object type it represents.

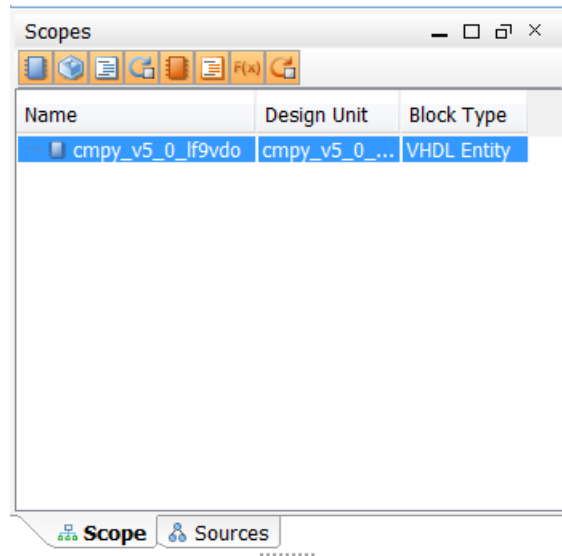


Figure 3-12: Scopes Window

See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] for a description of the Vivado IDE menus.

See [Understanding HDL Objects in Waveform Configurations](#), page 69 for more detail on how to use the Scopes window in the Vivado simulator.

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.

The simulation can be resumed at any time by using the **Run All**, **Run**, or **Step** commands. See [Stepping Through a Simulation in Chapter 5](#) for more information.

Saving Simulation Results

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

If you add objects to the Wave window and run the simulation, the design hierarchy for the complete design and the transitions for the added objects are automatically saved to the WDB file.

The wave configuration settings; which include the signal order, name style, radix, and color, and so forth; are saved to the wave configuration (WCFG) file upon demand. See [Chapter 5, Analyzing and Debugging With Waveforms](#).

Closing Simulation

To terminate a simulation, select **File > Exit** or click the **X** at the top-right corner of the project window.

Compiling and Simulating the Design

Introduction

Running a simulation from the command line for either a Behavioral or a Timing simulation requires the following steps:

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

The following subsections describe these steps.

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

- [Generating a Timing Netlist for Simulation in Chapter 2](#)
- [Running Post-Synthesis and Post-Implementation Simulations, page 53.](#)

Parsing Design Files

The `xvhdl` and `xvlog` commands parse VHDL and Verilog files, respectively. Descriptions for each option are available in [Table 4-2, page 55](#). To go to the command description click the option link.

Note: In your PDF reader, turn on the **View > Toolbars > More Tools > Previous View** and **Next View** buttons to navigate back and forth.

xvhdl

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

xvhdl Syntax

```
xvhdl
[-f [-file] <filename>]
[-encryptdumps]
[-h [-help]]
[-initfile <init_filename>]
[-L [-lib] <library_name>[=<library_dir>]]
[-log <filename>]
[-nolog]
[-prj <filename>]
[-relax]
[-v [verbose] [0|1|2]]
[-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]
[-relax]
[-prj <filename>]
[-sourcelibdir <sourcelib_dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-v [verbose] [0|1|2]]
[-version]]
```

xvlog Examples

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

Elaborating and Generating a Snapshot Using 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 (settles 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 Syntax, page 50](#) for more information about project files.

xelab Command Syntax Options

Descriptions for each option are available in [Table 4-2, page 55](#). To go to the command description click the option link.

Note: In your PDF reader, turn on the **View > Toolbars > More Tools > Previous View** and **Next View** buttons to navigate back and forth.

```
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]
[-maxdelay]
[-mt arg]
[-nolog]
[-notimingchecks]
[-nosdfinterconnectdelays]
[-nospecify]
[-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>]
[-timescale]
[-timeprecision_vhdl arg]
[-transport_int_delays]
[-v [verbose] [0|1|2] ]
[-version]

```

xelab Examples

```

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

```

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 adder2(.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 `xelab` cannot find a Verilog unit, it treats the name of the instantiated module as a VHDL entity name and continues a case-insensitive search.

The `xelab` command searches for an entity with the same name as the instantiated module name in the user-specified list and order of unified logical libraries, searches for and selects the first one matching name, then stops the search.

- 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.
- If `xelab` finds a unique binding for any one library, it selects that name and stops the search.

Note: For a mixed language design, the port names used in a named association to a VHDL entity instantiated by a Verilog module are always treated as case insensitive. Also note that you cannot use a `defparam` statement to modify a VHDL generic. See [Using Mixed Language Simulation, page 58](#) for more information.

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 determine 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 library `mylib`, which is the library of C's parent B.
- 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

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

Using xsim to Simulate the Design Snapshot

The `xsim` command loads a simulation snapshot to effect either a batch mode simulation or provides a GUI and/or 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 4-1](#).
- `<snapshot>` is the simulation snapshot.

xsim Executable Options

Table 4-1: 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 GUI.
<code>-h [-help]</code>	Print help message to screen.
<code>-log <filename></code>	Specify the log file name.

Table 4-1: xsim Executable Command Options (Cont'd)

XSIM Option	Description
-maxdeltaid arg (= -1)	Specify the maximum delta number. Report an error if it exceeds maximum simulation loops at the same time.
-nolog	Suppress log file generation.
-onfinish <quit stop>	Specifies the behavior at end of simulation.
-onerror <quit stop>	Specifies the behavior upon simulation runtime error.
-t [-tclbatch] <filename>	Specifies the Tcl file for batch mode execution.
-R [-runall]	Runs simulation till end (such as do 'run all;quit').
-testplusarg <arg>	Specifies plusargs to be used by \$test\$plusargs and \$value\$plusargs system functions.
-vcdfile <vcd_filename>	Specify the VCD output file.
-vcdunit <vcd_unit>	Specify the VCD output unit. The default is the same as the engine precision unit.
-view <wavefile.wcfg>	Open a wave configuration file. This switch should be used together with -gui switch.
-wdb <filename.wdb>	Specify the waveform database output file.
-tp	Enable printing to screen of hierarchical names of process being executed.
-t1	Enable printing to screen of file name and line number of statements being executed.

Project File Syntax

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

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

Where:

- <work_library> is the library into which the HDL files on the given line should 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, [-d <macro>] optionally lets you define one or more macros.
- For Verilog, [-i <include_path>] optionally lets you define one or more <include_path> directories.

Predefined XILINX_SIMULATOR Macro for Verilog Simulation

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
```

Simulating the Design in Non-Project Mode

You can type simulation commands into a Tcl file, and reference the Tcl file with the `-tclbatch` option of the simulation executable.

You can 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.

For information on using Tcl, see the following:

- Documentation at <http://www.tcl.tk/>
- *Vivado Tcl Command Reference Guide (UG835)* [Ref 4]
- *Vivado Design Suite User Guide: Using the Tcl Scripting Capabilities (UG894)* [Ref 7]
- *Tcl and SDC Command Tutorial (UG760)* [Ref 8]

After you parse the design files and create a simulation executable using `xelab`, you can run a functional or a timing simulation.

Running a Behavioral Simulation

Figure 4-1 illustrates the behavioral simulation process:

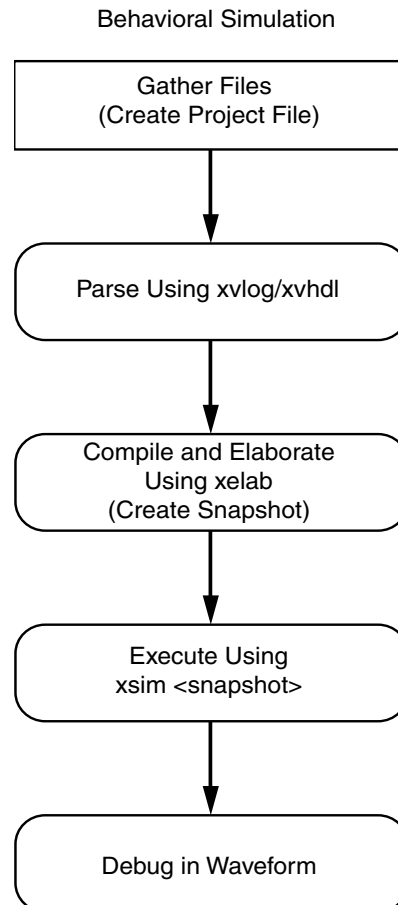


Figure 4-1: Behavioral Simulation Process

To run behavioral simulation on the Tcl Console, type:

```
launch_xsim -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 4-2 illustrates the post-synthesis and post-implementation simulation process:

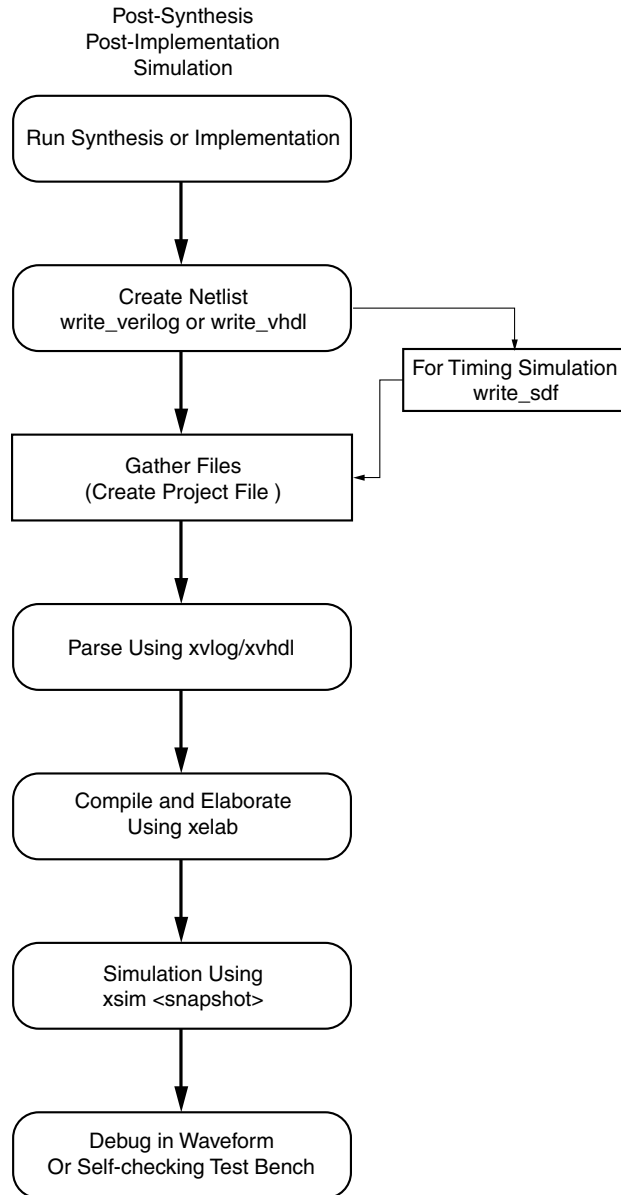


Figure 4-2: Post-Synthesis and Post-Implementation Simulation

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

```
synth_design -top top -part xc7k70tfbg676-2 -flatten_hierarchy none
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
xvlog -work work test_synth.v
xvlog -work work testbench.v
xelab -L unisms_ver testbench glbl -s test
xsim test -gui
```

About Post-Synthesis or Post-Implementation Timing Simulation

When you are run a Post-Synthesis or Post-Implementation timing simulation, the `write_sdf` command must be run after `write_verilog` and the appropriate `annotate` command is needed for elaboration and simulation.

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
```

Library Mapping File (xsim.ini)

The HDL compile programs, `xvh1`, `xvlog`, and `xelab`, use the `xisim.ini` configuration file to find the definitions and physical locations of VHDL and Verilog logical libraries.

The compilers attempt to read `xisim.ini` from these locations in the following order.

1. `$XILINX/PLANAHEAD/data/xsim/vhdl/<platform>`.
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>
<logical_libraryn> = <physical_dir_pathn>
```

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

```
VHDL
std=C:/libs/vhdl/hdl/
stdieee=C:/libs/vhdl/hdl/ieee
work=C:/workVerilog
unisims_ver=$XILINX/PLANAHEAD/data/verilog/hdl/nt/unisims_ver
xilinxcorelib_ver=C:/libs/verilog/hdl/nt/xilinxcorelib_ver
mylib=./mylib
work=C:/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 it.
- 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/<logical_library_name>`.

File comments must start with `--`

xelab, xvhd, and xvlog Command Options

Table 4-2 lists the command options for the `xelab`, `xvhdl`, and `xvlog` commands.

Table 4-2: **xelab, xvhd, and xvlog Command Options**

Command Option	Description
<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.
<code>-debug <kind></code>	Compile with specified debugging ability turned on. The <code><kind></code> options are: <ul style="list-style-type: none"> • <code>line</code>: HDL breakpoint • <code>wave</code>: Waveform generation, conditional execution, force value • <code>xlibs</code>: Visibility into libraries precompiled by xilinx • <code>typical</code>: Most commonly used abilities: <code>line</code> and <code>wave</code> • <code>off</code>: Turn off all debugging abilities (Default)
<code>-encryptdumps</code>	Encrypt parsed dump of design units being compiled.
<code>-f [-file] <filename></code>	Read additional options from the specified file.
<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>
<code>-h [-help]</code>	Print this help message.

Table 4-2: xelab, xvhd, and xvlog Command Options (Cont'd)

Command Option	Description
-i [include] <directory_name>	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.
-initfile <init_filename>	User-defined simulator initialization file to add to or override settings provided by the default <code>xsim.ini</code> file.
-incremental	Compile files only if they have changed since the last compile.
-L [-lib] <library_name> [=<library_dir>]	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.
-log <filename>	Specify the log file name. Default: <code>xvlog</code> <code>xvhd1</code> <code>xelab.log</code> .
-maxdelay	Compile Verilog design units with minimum delays.
-mindelay	Compile Verilog design units with maximum delays.
-mt arg	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>)
-maxdesigndepth arg	Override maximum design hierarchy depth allowed by the elaborator (Default: 5000)
-nolog	Suppress log file generation
-notimingchecks	Ignore timing check constructs in Verilog specify block(s).
-nosdfinterconnectdelays	Ignore SDF port and interconnect delay constructs in SDF.
-nospecify	Ignore Verilog path delays and timing checks.
-override_timeunit	Override timeunit for all Verilog modules, with the specified time unit in <code>-timescale</code> option.
-override_timeprecision	Override time precision for all Verilog modules, with the specified time precision in <code>-timescale</code> option.
-pulse_e arg	Path pulse error limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).
-pulse_r arg	Path pulse reject limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).
-pulse_int_e arg	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).
-pulse_int_r arg	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).
-pulse_e_style arg	Specify when error about pulse being shorter than module path delay should be handled. Choices are: <code>ondetect</code> : report error right when violation is detected <code>onevent</code> : report error after the module path delay (Default: <code>onevent</code>)

Table 4-2: xelab, xvhd, and xvlog Command Options (Cont'd)

Command Option	Description
-prj <filename>	Specify XSim project file containing one or more entries of vhdl verilog <work lib> <HDL file name>.
-rangecheck	Enable runtime value range check for VHDL.
-r [-run]	Run the generated executable in command line interactive mode.
-R [-runall]	Run the generated executable till end of simulation.
-relax	Relax strict language rules.
-s [-snapshot] arg	Specify the name of output executable simulation snapshot. Default is <worklib>.<unit>; for example: work.top. Additional unit names are concatenated using #; for example: work.t1#work.t2.
-sdfnowarn	Do not emit SDF warnings.
-sdfnoerror	Treat errors found in SDF file as warning.
-sdfmin arg	<root=file> SDF annotate <file> at <root> with maximum delay.
-sdftyp arg	<root=file> SDF annotate <file> at <root> with typical delay.
-sdfmax arg	<root=file> SDF annotate <file> at <root> with maximum delay.
-sdfroot <root_path>	Default design hierarchy at which SDF annotation is applied.
-sourcelibdir <sourcelib_dirname>	Directory for Verilog source files of uncompiled modules. Use -sourcelibdir <sourcelib_dirname> for each source directory.
-sourcelibext <file_extension>	File extension for Verilog source files of uncompiled modules. Use -sourcelibext <file extension> for source file extension
-sourcelibfile <filename>	Filename of a Verilog source file with uncompiled modules. Use -sourcelibfile <filename>.
-timescale	Specify default timescale for Verilog modules. Default: 1ns/1ps.
-timeprecision_vhdl arg	Specify time precision for vhdl designs. Default:1ps.
-transport_int_delays	Use transport model for interconnect delays.
-typdelay	Compile Verilog design units with typical delays (Default).
-v [verbose] [0 1 2]	Specify verbosity level for printing messages. Default = 0.
-version]	Print the compiler version to screen.
-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.

Using Mixed Language Simulation

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

Restrictions on Mixed Language in Simulation

- Mixing VHDL and Verilog is restricted to the module instance or component only.
- A VHDL design can instantiate Verilog modules and a Verilog design can instantiate VHDL components. Any other mix use of VHDL and Verilog is not supported.
- A Verilog hierarchical reference cannot refer to a VHDL unit nor can a VHDL expanded/selected name refer to a Verilog unit.
- Only a small subset of VHDL types, generics and ports are allowed on the boundary to a Verilog module. Similarly, a small subset of Verilog types, parameters and ports are allowed on the boundary to VHDL design unit.
- Component instantiation-based default binding is used for binding a Verilog module to a VHDL design unit. Specifically, configuration specification, direct instantiation and component configurations are not supported for a Verilog module instantiated inside a VHDL design unit.

Key Steps in a Mixed Language Simulation

1. Optionally, specify the search order for VHDL entity or Verilog modules in the design libraries of a mixed language project.
2. Use `xelab -L` to specify the binding order of a VHDL entity or a Verilog 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 or a Verilog module, 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 `-lib` option.

The search order is the same as the order of appearance of libraries on the `xelab` command line. See 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 module in a VHDL design unit or a VHDL module in a Verilog design unit as described in the following subsections.

Mixed Language Boundary and Mapping Rules

The following restrictions apply to the boundaries between VHDL and Verilog design units/modules:

- The boundary between VHDL and Verilog is enforced at design unit level.
- A VHDL design is allowed to instantiate one or more Verilog modules.
- Instantiation of a Verilog UDP inside a VHDL design is not supported.
- A Verilog design can instantiate a VHDL component corresponding to a VHDL entity only.
- Instantiation of a VHDL configuration in a Verilog design is not supported.

Instantiating a Verilog Module in a VHDL Design Unit

1. Declare a VHDL component with the same name as the Verilog module (respecting case sensitivity) 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 association to instantiate the Verilog module. For example:

```
UUT : MY_VHDL_UNIT PORT MAP (
  Q => O,
  D => I,
  C => CLK);
```

To ensure that you are correctly matching port types, review the [Port Mapping and Supported Port Types](#).

Port Mapping and Supported Port Types

[Table 4-3](#) lists the supported port types.

Table 4-3: Supported Port Types

VHDL ¹	Verilog ²
IN	INPUT
OUT	OUTPUT

Table 4-3: Supported Port Types (Cont'd)

VHDL ¹	Verilog ²
INOUT	INOUT

1. Buffer and linkage ports of VHDL are not supported.
2. Connection to bidirectional pass switches in Verilog are not supported. Unnamed Verilog ports are not allowed on mixed design boundary.

Table 4-4 shows the supported VHDL and Verilog data types for ports on the mixed language design boundary.

Table 4-4: Supported VHDL and Verilog data types

VHDL Port	Verilog Port
bit	net
std_ulogic	net
std_logic	net
bit_vector	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.

Generics (Parameters) Mapping

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

- integer
- real
- string
- boolean

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

VHDL and Verilog Values Mapping

Table 4-5 lists the Verilog states mappings to `std_logic` and `bit`.

Table 4-5: Verilog States mapped to `std_logic` and `bit`

Verilog	std_logic	bit
Z	Z	0
0	0	0

Table 4-5: Verilog States mapped to std_logic and bit (Cont'd)

Verilog	std_logic	bit
1	1	1
X	X	0

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

Table 4-6 lists the VHDL type `bit` mapping to Verilog states.

Table 4-6: VHDL bit Mapping to Verilog States

bit	Verilog
0	0
1	1

Table 4-7 lists the VHDL type `std_logic` mappings to Verilog states.

Table 4-7: VHDL std_logic mapping to Verilog States

std_logic	Verilog
U	X
X	X
0	0
1	1
Z	Z
W	X
L	0
H	1
-	X

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

Instantiating a VHDL Module in a Verilog Design Unit

To instantiate a VHDL module in a Verilog design unit, instantiate the VHDL entity as if it were a Verilog module. For example:

```

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

```

Analyzing and Debugging With Waveforms

Introduction

With the Vivado™ Integrated Design Environment (IDE) simulator GUI open, you can begin working with the waveform to analyze your design and debug your code. The simulator populates design data in other areas of the GUI, such as the Objects and the Scopes windows.

Typically, simulation is setup 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].

Simulating from the Vivado IDE

When you launch the Vivado simulator, a wave configuration with top-level HDL objects displays. The Vivado simulator populates design data in other areas of the GUI, such as the Scopes and Objects windows. You can then add additional HDL objects, or run the simulation using the **Run** command. See [Launching the Vivado Simulator](#).

Using the GUI in Non-Project Mode

When you launch Vivado simulator using the `xsim -gui <snapshot_name>` command option, an empty wave configuration displays.

Launching the Vivado Simulator

The options to open the Vivado simulator waveform are as follows:

- From the Flow Navigator simulation section, click:
 - **Run Simulation**, as described in [Using Simulation Settings in Chapter 3](#).
 - **Open Static Simulation** as described in [Viewing Simulation Data from Prior Simulation Settings \(Static Simulation\)](#), page 86.
- Open the simulation from an `xsim` command on the command line, as described in [Using xsim to Simulate the Design Snapshot in Chapter 4](#).

Design data populates in other areas of the Vivado IDE, such as the Scopes window, and the Objects window. You must add HDL objects to the wave configuration before you run the simulation in Vivado Simulator.

Adding HDL Objects to the Wave Configuration

To populate the waveform window with the HDL objects from your design, you can use the Vivado IDE menu commands or drag and drop capabilities, or using Tools Command Language (Tcl) commands in the Tcl Console.

Note: Changes to the wave configuration, including creating the wave configuration or adding HDL objects, do not become permanent until you save the WCFG file. For more information, see [Saving Simulation Results in Chapter 3](#).

Adding Simulator HDL Objects in Vivado IDE

To add simulation HDL objects to the waveform in the Vivado IDE, do the following:

1. In the Scopes window, expand the design hierarchy.

The objects that correspond to the selected instance or process display in the Objects window.

2. Select one or more objects.
3. Add objects to the wave configuration from the popup menu by selecting **Add to Wave Window**.

Alternately, you can drag and drop the objects from the Objects window to the **Name** column of the waveform window.



TIP: Some HDL objects cannot be viewed as a waveform, such as: Verilog-named events, Verilog parameters, VHDL constants, objects with more elements than the max traceable size (see the `trace_limit` property in the Vivado Tcl Command Reference Guide (UG835) [Ref 4]). Alternatively, type `trace_limit -help` in the Tcl Console.

When you launch the Vivado simulator, by default a waveform configuration with top-level HDL objects displays, as shown in Figure 5-1.

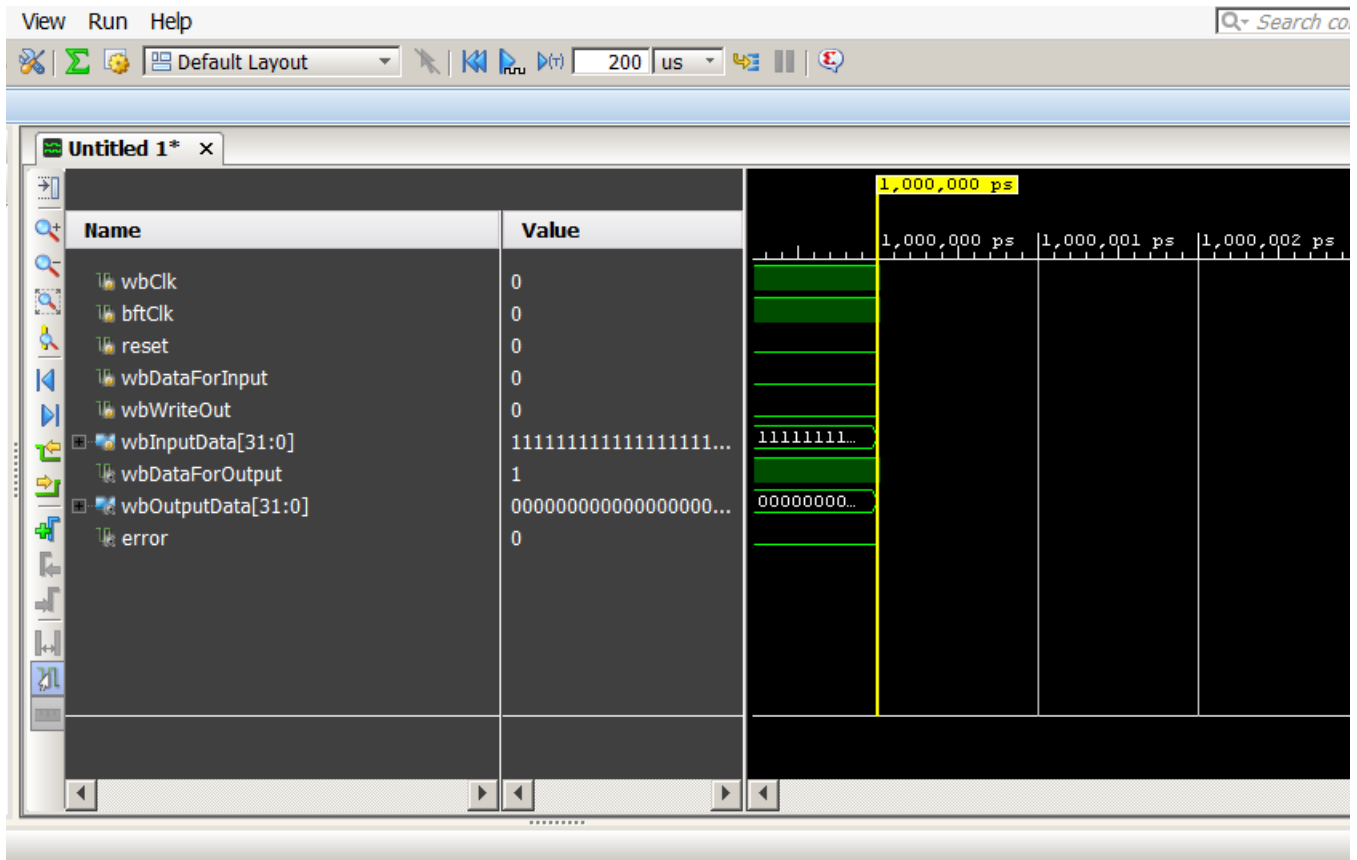


Figure 5-1: Example Waveform Window

Design data is populated in other areas of the Vivado simulator GUI, such as the Scopes window, and the Objects window. You can then proceed to add additional HDL objects, or run the simulation using the **Run Simulation** command.

Figure 5-2 shows the Vivado simulator Scopes window.

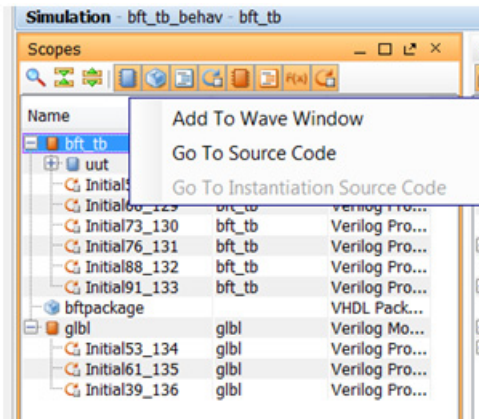




Figure 5-2: Scopes Window



IMPORTANT: Waveforms for an object show only from the simulation time when the object was added to the window. Changes to the waveform configuration, including creating the waveform configuration or adding HDL objects, do not become permanent until you save the WCFG file.

You can filter scopes within the Scopes window using one of the following methods:

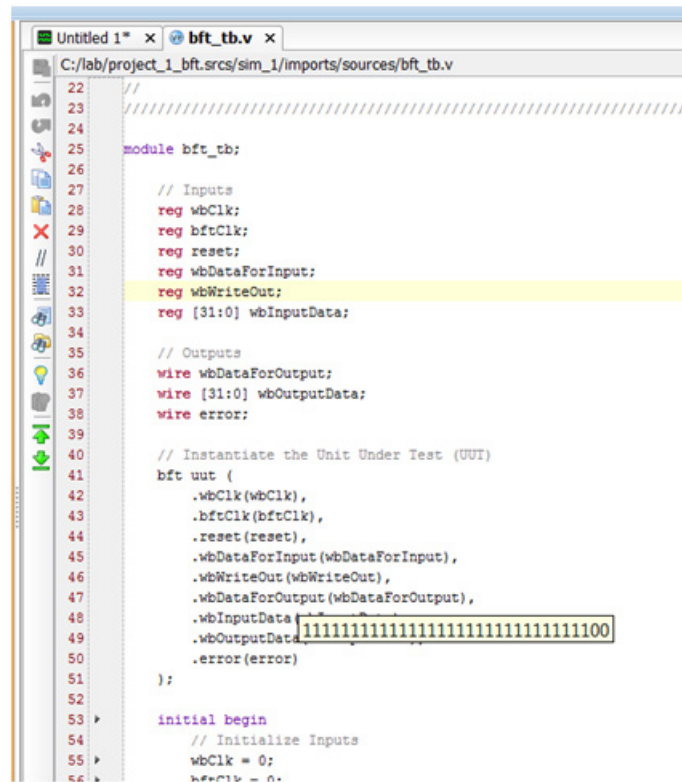
- To hide certain types of scope from display, click one or more scope-filtering buttons. 
- To limit the display to scopes containing a specified string, click the **Zoom** button.  and type the string in the text box that appears

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

- Add to Wave Window:** Adds all viewable HDL objects of the selected scope to the waveform configuration.
- Go To Source Code:** Opens the source code at the definition of the selected scope.
- Go To Instantiation Source code:** For Verilog module and VHDL entity instances, opens the source code at the point of instantiation for the selected instance.

In the source code text editor, you can hover over an identifier in a file get the value, as shown in Figure 5-3.

IMPORTANT: You need to have the correct scope in the Scopes window selected to use this feature.



```
22 //
23 ///////////////////////////////////////////////////////////////////
24
25 module bft_tb;
26
27 // Inputs
28 reg wbClk;
29 reg bftClk;
30 reg reset;
31 reg wbDataForInput;
32 reg wbWriteOut;
33 reg [31:0] wbInputData;
34
35 // Outputs
36 wire wbDataForOutput;
37 wire [31:0] wbOutputData;
38 wire error;
39
40 // Instantiate the Unit Under Test (UUT)
41 bft uut (
42     .wbClk(wbClk),
43     .bftClk(bftClk),
44     .reset(reset),
45     .wbDataForInput(wbDataForInput),
46     .wbWriteOut(wbWriteOut),
47     .wbDataForOutput(wbDataForOutput),
48     .wbInputData,
49     .wbOutputData 11111111111111111111111111111100
50     .error(error)
51 );
52
53 initial begin
54     // Initialize Inputs
55     wbClk = 0;
```

Figure 5-3: Source Code with Identifier Value Displayed

TIP: To change the numerical format of the displayed values, in the Tcl Console, type: `set_property radix <radix> [current_sim]` where <radix> is one the following: `bin`, `unsigned`, `hex`, `dec`, `ascii`, or `oct`.

Using the Objects Window

Figure 5-4 shows the Vivado simulator Objects window.

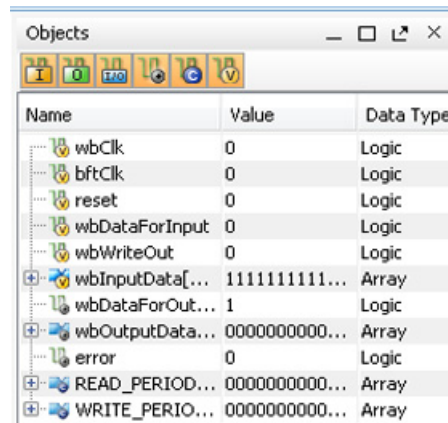
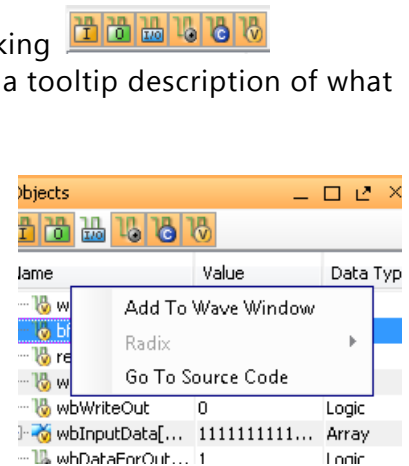


Figure 5-4: Objects Window

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

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

- **Add to Wave Window:** Add the selected object to the waveform configuration
- **Radix:** Select the numerical format to use when displaying the value of the selected object in the Objects window and in the source code window
- **Go To Source Code:** Open the source code at the definition of the selected object.



Using Non-Project Mode

The following are some of the commands that you can use on the Tcl Console or within a batch script. See the *Vivado Tcl Command Reference Guide (UG835)* [Ref 4] for more information about these commands.

Exploring Design Hierarchy

As an equivalent to using the Scopes and Objects windows, you can navigate the HDL design by typing `current_scope` in the TCL Console to set or show the current scope.

Use the `report_scopes` and `report_values` commands, respectively, to list the scopes and objects, respectively, under the current scope.

Adding HDL Objects to the Waveform Window

To add an individual HDL object or set of objects to the waveform window in the Tcl Console, type:

```
add_wave <HDL_objects>
```

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

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



TIP: The `add_wave` command accepts HDL scopes as well as HDL objects. Using `add_wave` with a scope is equivalent to the **Add To Wave Window** command in the Scope window.

Understanding HDL Objects in Waveform Configurations

When you add an HDL object to a waveform configuration, the waveform viewer creates a *wave object* of the HDL object. The wave object is linked to, but distinct from, the HDL object: you can create multiple wave objects from the same HDL object, and set the display properties of each wave object separately.

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

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

Wave objects created from HDL objects are specifically called *design wave objects*.

Wave objects display with a corresponding identifying icon.



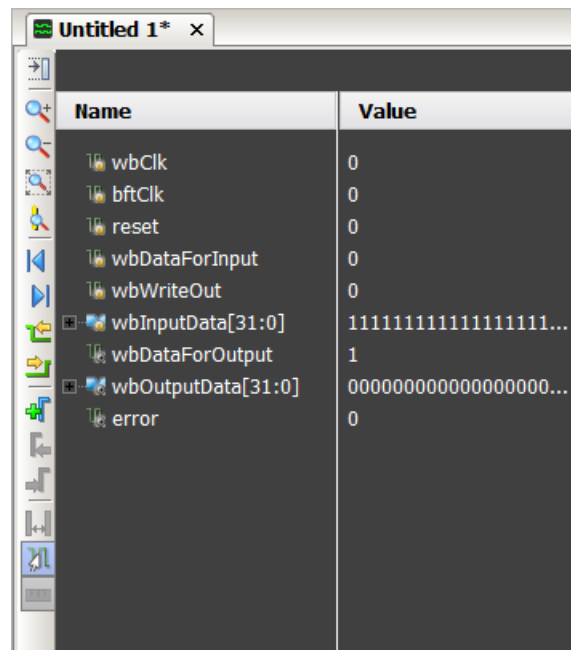
For design wave objects, the background part of the icon indicates whether the object is a scalar  or a compound  such as a Verilog vector or VHDL record.

Figure 5-5 is an example of HDL objects in the waveform configuration window.



The screenshot shows a window titled "Untitled 1*" with a table of HDL objects. The table has two columns: "Name" and "Value". The objects listed are:

Name	Value
wbClk	0
bftClk	0
reset	0
wbDataForInput	0
wbWriteOut	0
wbInputData[31:0]	111111111111111111...
wbDataForOutput	1
wbOutputData[31:0]	000000000000000000...
error	0

Figure 5-5: Waveform HDL Objects

The design objects display with **Name**, and **Value**:

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

About Wave Configurations and Waveform Windows

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

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

A wave configuration can have a name or be "**Untitled#**". The name shows on the view of the wave configuration window.

Saving a Wave Configuration

To save a wave configuration to a WCFG file:

- In the Vivado simulation GUI, select **File > Save Waveform Configuration As**, and type a name for the waveform configuration.
- In the Tcl Console, type:

```
save_wave_config <filename.wcfg>
```

The WCFG file is named as specified by the command argument and saved.

Opening a Wave Configuration and Waveform Database

If you have a waveform configuration (WCFG) file from a previous simulation run that was created in the current application version and in the same OS platform, and you want to open the WCFG and its associated simulation data (WDB), use one of the following methods to open the WCFG and WDB.

- To open a wave configuration and the waveform database, select **File > Open**, select the **.wcfg** file type from the file filter list, and select the wave configuration (WCFG) file from a previous simulation.

The static simulator displays the wave configuration, with all HDL objects previously traced, and the associated waveform database.

- To open a WDB from a previous simulation and no WCFG, select **File > Open**, choose the **.wdb** file type from the file filter list, and select the wave database (WDB) file from a previous simulation.

The static viewer displays the data from the previous simulation in the Objects window. There is no waveform data open in the waveform window.

Controlling the Display of Waveforms

You can control the waveform display using:

- Zoom feature buttons in the HDL Objects window sidebar
- Zoom combinations with the mouse wheel
- Vivado IDE X-Axis zoom gestures
- Vivado simulation Y-Axis zoom gestures. See the *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] for more information about the Vivado IDE X-Axis zoom gestures.

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

Using the Zoom Feature button

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



Zooming with the Mouse Wheel

You can also use the mouse wheel with the **Shift** or **CTRL** key in combination after clicking within the waveform to zoom in and out, emulating the operation of the dials on an oscilloscope.

Y-Axis Zoom Gestures

In addition to the zoom gestures supported for zooming in the X dimension, when over an analog waveform, additional zoom gestures are available, as shown in [Figure 5-6](#).

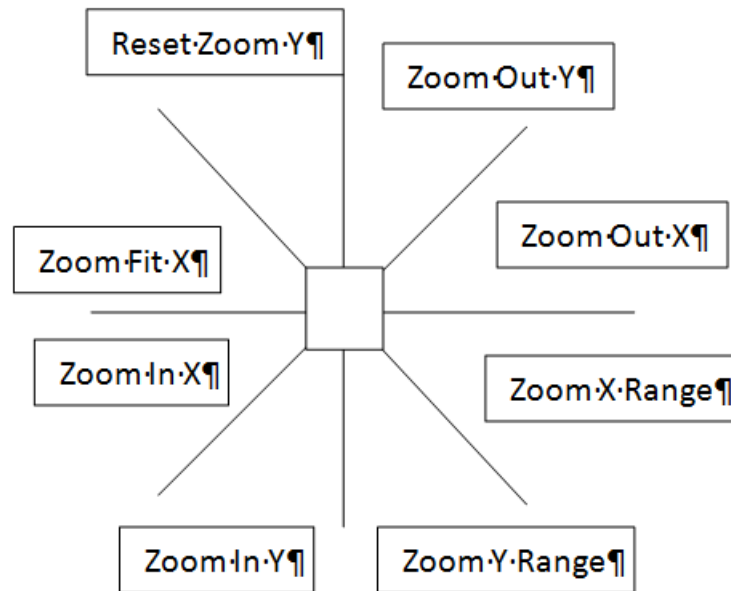


Figure 5-6: Analog Zoom Options

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

The additional zoom gestures are:

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

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

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

When viewing an HDL bus object as an analog waveform, to produce the expected waveform it is important to select a radix that matches the nature of the data in the HDL object.

For example:

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

About 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.
- Any non-0 or -1 bits cause the entire value to be 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, as determined by the settings of the Real Settings dialog box, shown in [Figure 5-7](#).

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

To set a wave object to the Real radix, access the Real Settings dialog box.

In the waveform configuration window, select an HDL object, and right-click to open the popup menu.

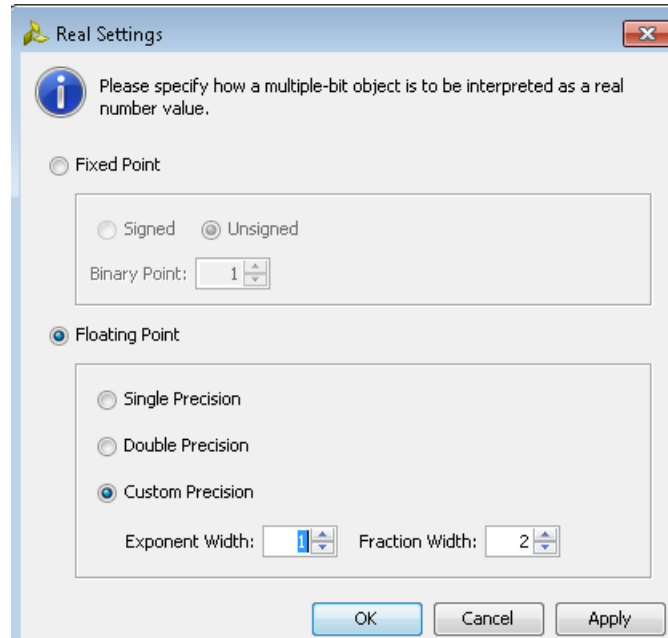


Figure 5-7: Real Settings Dialog Box

The Real Setting dialog box options are:

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

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

Other values result in <Bad Radix> values as in Fixed Point.

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



TIP: If the row indices separator lines are not visible, you can turn them on in the [Waveform Options Dialog Box, page 78](#), to make them visible.

Customizing the Appearance of Analog Waveforms

To customize the appearance of an analog waveform:

1. In the name area of a waveform window, right-click a bus to open the popup menu.
2. Select **Waveform Style** >:

- **Analog:** Sets a digital waveform to Analog.
- **Digital:** Sets an Analog waveform object to Digital.
- **Analog Settings:** Opens the Analog Setting dialog box. [Figure 5-8](#) shows the Analog Settings dialog box with the settings for analog waveform drawing.

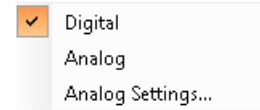


Figure 5-8: Analog Settings Dialog Box

The Analog Settings dialog box options are:

- **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).

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

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

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

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



IMPORTANT: *Analog settings are saved in a wave configuration; however, because control of zooming in the Y dimension is highly interactive, unlike other wave object properties such as radix, they do not affect the modification state of the wave configuration. Consequently, zoom settings are not saved with the wave configuration.*

Using Cursors

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



TIP: *WCFG files do not record cursor positions. For more permanent indicators, used in situations such as establishing a time-base for multiple measurements, and indicating notable events in the simulation, add markers to the waveform window instead. See [Using Markers, page 80](#) for more information.*

Placing Main and Secondary Cursors

You can place the main cursor with a single click in the waveform window.

To place a secondary cursor, **Ctrl+Click** and 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 waveform window, you see a hollow or filled-in circle if the **Snap to Transition** button is selected, which is the default behavior.

- A hollow circle ○ 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 waveform window where there is no cursor, marker, or floating ruler.


Finding the Next or Previous Transition on a Waveform

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

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.

Waveform Options Dialog Box

When you select the **Waveforms Options** button  the Waveform Options dialog box, shown in [Figure 5-9](#), opens.

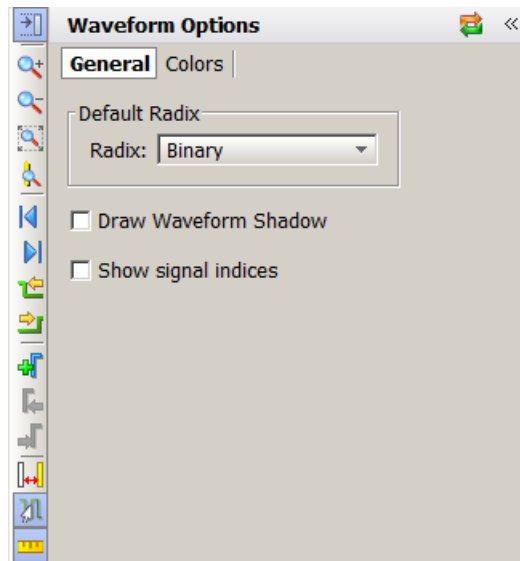


Figure 5-9: Waveform Options Dialog Box

Set the **General Waveform Options**:


- **Default Radix:** Sets the numerical format to use for newly-created design wave objects.
- **Show signal indices:** Checkbox 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.


The **Colors** page lets you set colors of items within the waveform window.

Using the Floating Ruler

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

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

The floating ruler button  and the floating ruler itself are visible only when the secondary cursor or a marker is present.

1. Do either of the following to display or hide a floating ruler:
 - Place the secondary cursor.
 - Select a marker.
2. Select **View > Floating Ruler**, or 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.

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.

Customizing the Wave Configuration

You can customize the waveform configuration using the features that are listed and briefly described in [Table 5-1](#); the feature name links to the subsection that fully describes the feature.

Note: In your PDF reader, turn on the **View > Toolbars > More Tools > Previous View** and **Next View** Buttons to navigate back and forth.


Table 5-1: Customization Features in the Wave Configuration

Feature	Description
Using Markers	You can add markers to navigate through the waveform, and to display the waveform value at a particular time.
Using Dividers	You can add a divider to create a visual separator of waveform objects.
Using Groups	You can add a group, that is a collection to which wave objects can be added in the wave configuration as a means of organizing a set of related HDL objects.
Using Virtual Buses	You can add a virtual bus to your wave configuration, to which you can add logic scalars and arrays.
Renaming Objects	You can rename waveform objects and groups.
Viewing Object Names and Changing the Name Display Mode	You can display the full hierarchical name (long name), the simple signal or bus name (short name), or a custom name for each signal.
About Radixes	The default radix controls the bus radix that displays in the wave configuration, Objects panel, and the Console panel.




Using Markers

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

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

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

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

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

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

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

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

Using Dividers

Dividers create a visual separator between HDL objects. You can add a divider to your wave configuration to create a visual separator of HDL objects, as follows:

1. In a **Name** column of the waveform window, click a signal to add a divider below that signal.
2. From the context menu, select **Edit > New Divider**, or right-click and select **New Divider**.

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

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 83](#).


Using Groups

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

To add a Group:

1. In a waveform window, select one or more wave objects to add to a group.
Note: A group can include dividers, virtual buses, and other groups.
2. Select **Edit > New Group**, or right-click and select **New Group** from the context menu.

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

A Group is represented with the **Group** icon.  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 83](#).




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

Using Virtual Buses

You can add a virtual bus to your wave configuration, which is a grouping to which you can add logic scalars and vectors. The virtual bus displays a bus waveform, whose values are composed by taking the corresponding values from the added scalars and arrays in the vertical order that they appear under the virtual bus and flattening the values to a one-dimensional vector.

To add a virtual bus:

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

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

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](#).

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



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

Renaming Objects

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

1. Select the object name in the **Name** column.
2. Select **Rename** from the popup menu.

The Rename dialog box opens.

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

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



TIP: Renaming a wave object changes the name display mode to **Custom**. To restore the original name display mode, change the display mode to **Long** or **Short**, as described in the next section.

Viewing Object Names and Changing the Name Display Mode

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

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

To change the display name:

1. Select one or more signal or bus names. Use **Shift+click** or **Ctrl+click** to select many signal names.
2. Select **Name >**:
 - **Long** to display the full hierarchical name.
 - **Short** to display the name of the signal or bus only.
 - **Custom** to display the custom name given to the signal when renamed. See [Renaming Objects, page 83](#).

Note: Long and Short names are meaningful only to design wave objects. Other wave objects (dividers, groups, and virtual buses) display their **Custom** name by default and display an **ID** string for their **Long** and **Short** names.


About Radixes

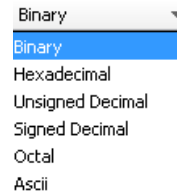
Understanding the type of data on your bus is important. You need to recognize the relationship between the radix setting and the data type to use the waveform options of Digital and Analog effectively. See [Displaying Waveforms as Analog, page 73](#) for more information about the radix setting and its effect on Analog waveform analysis.

Changing the Default Radix

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

To change the default waveform radix:

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



Changing the Radix on Individual Wave Objects

In the Objects window

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

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



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

Viewing Simulation Data from Prior Simulation Settings (Static Simulation)

When you run a simulation and display HDL objects in a waveform window, the running simulation produces a waveform database (WDB) file containing the waveform activity of the displayed HDL objects. The WDB file also stores information about all the HDL scopes and objects in the simulated design.

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

In this mode you cannot use commands that control or monitor a simulation, such as run commands, as there is no underlying "live" simulation model to control.

However, you can view waveforms and the HDL design hierarchy in a static simulation. As the simulator creates no waveform configuration by default, you must create a new waveform configuration or open a WCFG file before you can view waveforms.



IMPORTANT: *WDB files are neither backward nor cross-operating system compatible. You must open the WDB file in the same version and on the same type OS in which it was created. WCFG files are both backward and cross-OS compatible.*

Opening a WDB File for Viewing

To open a WDB file for viewing as a static simulation:

1. Open the Vivado IDE.
2. Open any Vivado project. This step is necessary to gain access to the Open Static Simulation feature.
3. In the Simulation section of the Flow Navigator, click **Open Static Simulation**.

The Specify Simulation Results dialog box opens, as shown in [Figure 5-10](#).

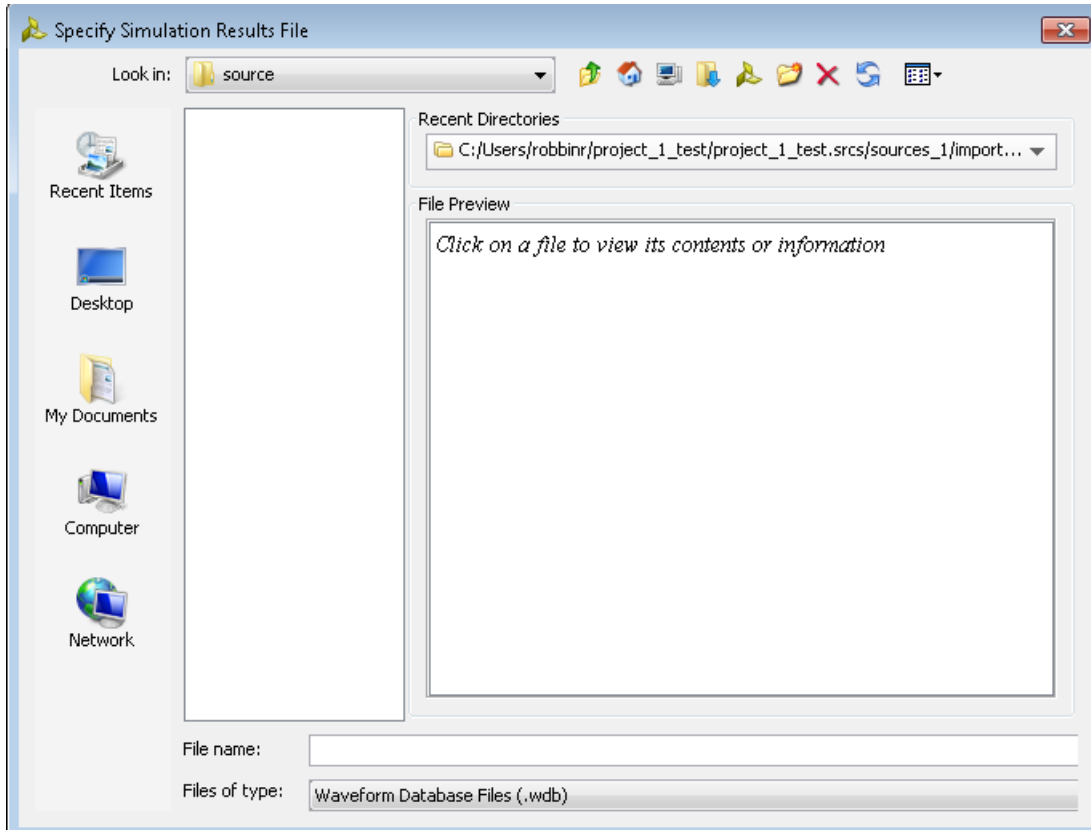


Figure 5-10: Specify Simulation Results File

4. Select a WDB file.

Simulation windows appear, with the exception of a waveform window. To view waveforms, follow the steps listed in one of the two sections below.

Opening a WCFG File

After opening a WDB file, you can open a WCFG file to use with the static simulation as follows:

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

The Specify Simulation Results dialog box, shown in [Figure 5-10](#), opens.

2. Locate and select a WCFG file that references HDL objects in the open WDB file.

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

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

Note: Although a wave configuration can contain any HDL object from the static simulation of an HDL design, only those HDL objects for which the WDB contains waveform data display waveforms. Others display blank waveforms.

Creating a New Wave Configuration

After opening a WDB file, you can create a new waveform configuration for displaying waveforms as follows:

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

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

2. Add HDL objects to the waveform configuration using the steps listed in

Note: Although you may add any HDL object from the static simulation's HDL design, only those HDL objects for which the WDB contains waveform data will display waveforms. All others will display blank waveforms.

See [Chapter 3, Running Simulation in Vivado IDE](#) for more information about creating new waveform configurations.



IMPORTANT: *Static wave configuration files are neither backward nor cross-operating system compatible. You must open the WCFG file in the same version and on the same type OS in which it was created.*


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 [Using Breakpoints, page 89](#).

Stepping Through a Simulation

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

A yellow arrow points to the currently executing line of code. 

You can also create breakpoints for additional stops while stepping through your simulation. For more information on debugging strategies in the simulator, see [Using Breakpoints, page 89](#).

1. To step through a simulation:

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

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

- From the start (0 ns), restart the simulation. Use the **Restart** command to reset time to the beginning of the test bench. See [Chapter 3, Running Simulation in Vivado IDE](#).

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

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

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

Using Breakpoints

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




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

You can set breakpoints in executable lines in your HDL file so you can run your code continuously until the source code line with the breakpoint is reached.

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, do the following:

1. Select **View > Breakpoint > Toggle Breakpoint**,  or click the **Toggle Breakpoint** button.
2. In the HDL file, click a line of code just to the right of the line number.

The **Breakpoint** button  displays next to the line.


Note: Alternatively, you can right-click a line of code, and select **Toggle Breakpoint**.

After the procedure completes, a simulation breakpoint button opens next to the line of code, and a list of breakpoints is available in the Breakpoints view.

To debug a design using breakpoints:

1. Open the HDL source file.
2. Set breakpoints on executable lines in the HDL source file, as described in [Using Breakpoints, page 89](#).
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, and the yellow arrow indicates 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. 

Running Simulation with Third Party Simulators Outside Vivado IDE

Introduction

Xilinx® supports third party simulators. Modelsim is supported through the Vivado™ Integrated Design Environment (IDE) and *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)* [Ref 3] describes its use.

This appendix provides a quick reference for running simulation outside the Vivado IDE.

See the *Xilinx Design Tools: Release Notes Guide (UG631)* [Ref 1] for information regarding supported third party simulation tools and versions.

Running RTL/Behavioral Simulation

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

1. Compile simulation libraries using the `compile_simlib` command.
2. Collect source files and create test bench. If you have IP in your design, see *Vivado User Guide: Designing with IPS (UG896)* [Ref 9].
3. If you are using Verilog compile `g1bl.v`, see [About Global Reset and Tristate for Simulation in Chapter 2](#).
4. If you have secureIP in your design, do the following:
 - a. For Modelsim: To use the precompiled libraries point to the library using the `-L` switch in `VSIM`. For example:

```
vsim -t ps -L secureip -L unisims_ver work.<testbench> work.g1bl
```
 - b. For IES: Use one of the following:
 - **IES Single Step Process**

In the single step process, you do not have to run `compplib` to compile Xilinx libraries. Only one additional switch is required, as shown in the following example:

```
-f $XILINX_PLANAHEAD/data/secureip/ncsim/ies_secureip_cell.list.f
```

IES Single Step Example

```
irun design>.v testbench>.v $XILINX_PLANAHEAD/data/verilog/src/glbl.v \
-f $XILINX_PLANAHEAD/data/secureip/ncsim/ies_secureip_cell.list.f \ \b>
-y $XILINX_PLANAHEAD/data/verilog/src/unisims +libext+.v \
-y $XILINX_PLANAHEAD/data/verilog/src/simprims +libext+.v \
+access+r+w
```

- **IES Three Step Process:**

Compile the secureIP libraries and then append them to CDS.lib and HDL.var.

- c. VCS: add the following command to the simulation command line so that the secureIP files are picked up by the simulator

```
-f $XILINX_PLANAHEAD/data/secureip/vcs/vcs_secureip_cell.list.f
```

VCS Example:

```
vcs -f $XILINX_PLANAHEAD/data/secureip/vcs/vcs_secureip_cell.list.f \
-y $XILINX_PLANAHEAD/data/verilog/src/unisims\
-y $XILINX_PLANAHEAD/data/verilog/src/xilinxcorelib\
+incdir+$XILINX_PLANAHEAD/data/verilog/src +libext+.v\
$XILINX_PLANAHEAD/data/verilog/src/glbl.v \
-Mupdate -R <testfixture>.v <design>.v
```

5. Compile and simulate the design. Refer to the user guide for the specified simulator.

Note: Ensure that you have correctly referenced the UNISIM, XilinxCorelib, SecureIP, UniMacro, and UniFast libraries for proper simulation. See [About Simulation Libraries in Chapter 2](#).

Running Netlist Simulation

The netlist simulation process involved the same steps as describe in [Running RTL/Behavioral Simulation](#).

1. Compile simulation libraries.
2. Gather files for simulation (see [Figure 4-2, page 53](#) as an example.)
 - a. The simulation testbench used for RTL simulation can be reused for the majority of designs.
 - b. Generate the simulation netlist (using `write_verilog` or `write_VHDL`).
 - For a Functional netlist, use `write_verilog -mode funcsim`.
 - For a Timing netlist, use `write_sdf`.

An example is:

```
synth_design -top top -part xc7k70tfbg676-2 -flatten_hierarchy none
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
write_verilog -mode timesim -sdf_file test.sdf test_synth_timing.v
write_sdf test.sdf
```

Now, you can use `test_synth.v` for functional simulation or `test_synth_timing.v` for timing simulation with `test.sdf`.

3. If you are using Verilog, compile `g1b1.v`. See [About Global Reset and Tristate for Simulation in Chapter 2](#).
4. If you are using the SecureIP library, see [step 4, page 91](#).
5. Compile and simulate the design. Refer to the user guide of the simulator you are using.

Note: Make sure the UNISIM, XilinxCorelib, SecureIP, and UniFast libraries are referenced correctly for proper simulation. See [About Simulation Libraries in Chapter 2](#).

Running Timing Simulation

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



IMPORTANT: *No Unimacro, Xilinxcorelib, Unifast, or UNISIM libraries are needed for timing simulation.*

The following are the command-line options to run timing simulation in supported simulators.

- MentorGraphics QuestaSim/ModelSim and Synopsys VCS

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

- Cadence IES/IUS

```
-pulse_r 0 -pulse_int_r
```



IMPORTANT: *Vivado simulation models use interconnect delays. Additional switches are necessary for timing simulation.*

Verilog and VHDL Exceptions

Introduction

The following appendix lists the exceptions to Verilog and VHDL support.

VHDL Language Support Exceptions

The Vivado™ Integrated Design Environment (IDE) supports:

- VHDL IEEE-STD-1076-1993
- Verilog IEEE-STD-1364-2001

Exceptions are noted in the **Exceptions** Column in the following tables.

Table B -1: VHDL Language Support Exceptions

Supported VHDL Construct	Exceptions
abstract_literal	Floating point expressed as based literals are not supported.
aggregate	Mixing choice directions in an aggregate is not supported.
alias_declaration	Alias to non-objects are in general not supported; particularly the following: Alias of an alias Alias declaration without subtype_indication Signature on alias declarations Operator symbol as alias_designator Alias of an operator symbol Character literals as alias designators
alias_designator	Operator_symbol as alias_designator Character_literal as alias_designator
association_element	Globally, locally static range is acceptable for taking slice of an actual in an association element.
attribute_name	Signature after prefix is not supported.

Table B -1: VHDL Language Support Exceptions (Cont'd)

Supported VHDL Construct	Exceptions
binding_indication	Binding_indication without use of entity_aspect is not supported.
bit_string_literal.	Empty bit_string_literal ("") is not supported
block_statement	Guard_expression is not supported; for example, guarded blocks, guarded signals, guarded targets, and guarded assignments are not supported.
choice	Aggregate used as choice in case statement is not supported.
concurrent_assertion_statement	Postponed is not supported.
concurrent_signal_assignment_statement	Postponed is not supported.
concurrent_statement	Concurrent procedure call containing wait statement is not supported.
conditional_signal_assignment	Keyword guarded as part of options is not supported as there is no supported for guarded signal assignment.
configuration_declaration	Non locally static for generate index used in configuration is not supported.
entity_class	Literals, unit, file and group as entity class are not supported.
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	In named parameter association in a function_call slicing, indexing or selection of formals is not supported.
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	In named parameter association in a procedure_call slicing, indexing or selection of formals is not supported.
process_statement	Postponed processes are not supported.

Table B -1: VHDL Language Support Exceptions (Cont'd)

Supported VHDL Construct	Exceptions
selected_signal_assignment	The "guarded" keyword as part of options is not supported as there is no support for guarded signal assignment.
signal_declaration	Signal_kind is not supported. Signal_kind is used for declaring guarded signals, which are not supported.
subtype_indication.	Resolved subtype of composites (arrays and records) is not supported
waveform.	Unaffected is not supported.
waveform_element	Null waveform element is not supported as it only has relevance in the context of guarded signals.

Verilog Language Support Exceptions

The following table lists the exceptions to supported Verilog language support.

Table B -2: Verilog Language Support Exceptions

Verilog Construct	Exception
Compiler Directive Constructs	
`celldefine	not supported
`endcelldefine	not supported
`undefs	Supports parameterized `define macros.
`unconnected_drive	not supported
`nounconnected_drive	not supported
Attributes	
attribute_instance	not supported
attr_spec	not supported
attr_name	not supported
Primitive Gate and Switch Types	
cmos_switchtype	not supported
mos_switchtype	not supported
pass_en_switchtype	not supported
Generated Instantiation	

Table B -2: Verilog Language Support Exceptions (Cont'd)

Verilog Construct	Exception
generated_instantiation	<p>The module_or_generate_item alternative is not supported.</p> <p>Production from 1364-2001 Verilog standard: generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_block module_or_generate_item</p> <p>Production supported by Simulator: generate_item_or_null ::= generate_conditional_statement generate_case_statement generate_loop_statement generate_blockgenerate_condition</p>
genvar_assignment	<p>Partially supported.</p> <p>All generate blocks must be named.</p> <p>Production from 1364-2001 Verilog standard: generate_block ::= begin [: generate_block_identifier] { generate_item } end</p> <p>Production supported by Simulator: generate_block ::= begin: generate_block_identifier { generate_item } end</p>
Source Text Constructs	
Library Source Text	
library_text	not supported
library_descriptions	not supported
library_declaration	not supported
include_statement	This refers to include statements within library map files (See IEEE 1364-2001, section 13.2). This does not refer to the <code>\include</code> compiler directive.
Configuration Source Text	
config_declaration	not supported
design_statement	not supported
config_rule_statement	not supported
default_clause	not supported
System Timing Check Commands	

Table B -2: Verilog Language Support Exceptions (Cont'd)

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

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at: www.xilinx.com/support

For a glossary of technical terms used in Xilinx documentation, see: www.xilinx.com/company/terms.htm.

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

[Vivado Design Suite 2012.2 Documentation](#)

1. *Xilinx Design Tools: Release Notes Guide (UG631)*
2. *Xilinx Design Tools: Installation and Licensing Guide (UG798)*
3. *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)*
4. *Vivado Tcl Command Reference Guide (UG835)*
5. *Writing Efficient Testbenches (XAPP199)*
6. *7 Series FPGA Libraries Guide for HDL Designs (UG768)*
7. *Vivado Design Suite User Guide: Using the Tcl Scripting Capabilities (UG894)*
8. *Tcl and SDC Command Tutorial (UG760)*
9. *Vivado User Guide: Designing with IPS (UG896)*