Vivado Design Suite User Guide

Logic Simulation

UG900 (v2014.1) April 23, 2014

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/23/2014	2014.1	Coded example for running a post-synthesis functional simulation from the command line corrected, page 112. Corrected information about switches in the section Compiling Simulation Libraries for VCS, page 149. Updated usage information on VCS Script Simulation, page 152. Added Appendix D, SystemVerilog Constructs Supported by the Vivado Simulator (early access). IDS_lite is obsolete; references to it have been removed. General updates made to reflect changes in version 2014.1, including SystemVerilog information, updates to graphics. Revisions and enhancements throughout, including updates to cross references and citations; addition of instructions on locating and displaying GUI features.

XILINX

Table of Contents

Chapter 1: Logic Simulation Overview

Introduction 6
Simulation Flow 6
Supported Simulators 9
Language and Encryption Support 10
OS Support and Release Changes 10

Chapter 2: Understanding Simulation Components in Vivado

Introduction 11
Using Test Benches and Stimulus Files 12
Using Xilinx Simulation Libraries 13
Compiling Simulation Libraries 24
Understanding the Simulator Language Option 25
Using the export_simulation Command 27
Recommended Simulation Resolution 29
Generating a Netlist 29
Annotating the SDF File 30
Using Global Reset and 3-State 31
Delta Cycles and Race Conditions 33
Using the ASYNC_REG Constraint 34
Simulating Configuration Interfaces 35
Disabling Block RAM Collision Checks for Simulation 39
Dumping the Switching Activity Interchange Format File for Power Analysis 40

Chapter 3: Using the Vivado Simulator from the Vivado IDE

Introduction 41
Vivado Simulator Features 41
Adding or Creating Simulation Source Files 42
Running the Vivado Simulator 51
Running Post-Synthesis Simulation 59
Running Post-Implementation Simulations 61
Identifying Between Multiple Simulation Runs 62
Pausing a Simulation 62

Saving Simulation Results 63
Closing Simulation 63
Adding a post.tcl Batch File 63
Skipping Compilation or Simulation 64
Viewing Simulation Messages 64

Chapter 4: Analyzing with the Vivado Simulator Waveforms

Introduction 66
Using Wave Configurations and Windows 66
Opening a Previously Saved Simulation Run 68
Understanding HDL Objects in Waveform Configurations 69
Customizing the Waveform 74
Controlling the Waveform Display 82
Organizing Waveforms 84
Analyzing Waveforms 86
Using Force Options 89

Chapter 5: Using Vivado Simulator Command Line and Tcl

Introduction 93
Compiling and Simulating a Design 93
Elaborating and Generating a Design Snapshot 95
Simulating the Design Snapshot 100
Example of Running Vivado Simulator in Standalone Mode 107
Project File (.prj) Syntax 108
Predefined Macros 109
Library Mapping File (xsim.ini) 109
Running Simulation Modes 111
Using Tcl Commands and Scripts 113
Tcl Property Commands 114

Chapter 6: Debugging a Design with Vivado Simulator

Introduction 118
Debugging at the Source Level 118
Generating (forcing) Stimulus 121
Power Analysis Using Vivado Simulator 126
Using the report_drivers Tcl Command 127
Using the Value Change Dump Feature 128

Chapter 7: Simulating with QuestaSim/ModelSim

Introduction 129

Simulating Xilinx Designs using QuestaSim/ModelSim 129

Chapter 8: Simulating with Cadence Incisive Enterprise Simulator (IES)

Introduction 142
Compiling Simulation Libraries for IES 142
Running Behavioral/RTL Simulation Using IES 143
IES Simulation Use models 145
Running Timing Simulation Using IES 147
Dumping SAIF for Power Analysis in IES 148

Chapter 9: Simulating with Synopsys VCS

Introduction 149
Running Behavioral/RTL Simulation using VCS 150
VCS Simulation Use Models 153

Appendix A: Value Rules in Vivado Simulator Tcl Commands

Introduction 156
String Value Interpretation 156
Vivado Simulation Logic 157

Appendix B: Vivado Simulator Mixed Language Support and Language Exceptions

Introduction 158
Using Mixed Language Simulation 158
VHDL Language Support Exceptions 163
Verilog Language Support Exceptions 165

Appendix C: Vivado Simulator Quick Reference Guide

Appendix D: SystemVerilog Constructs Supported by the Vivado Simulator

Introduction 172

Appendix E: Additional Resources and Legal Notices

Xilinx Resources 179
Solution Centers 179
Documentation References 179
Quick Take Videos 180
Links to Language and Encryption Support Standards 180
Please Read: Important Legal Notices 180

Logic Simulation Overview

Introduction

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

This chapter provides an overview of the simulation process, and the simulation options in the Vivado® Design Suite. The Vivado Design Suite Integrated Design Environment (IDE) provides an integrated simulation environment when using the Vivado simulator.

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

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

Simulation Flow

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

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

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

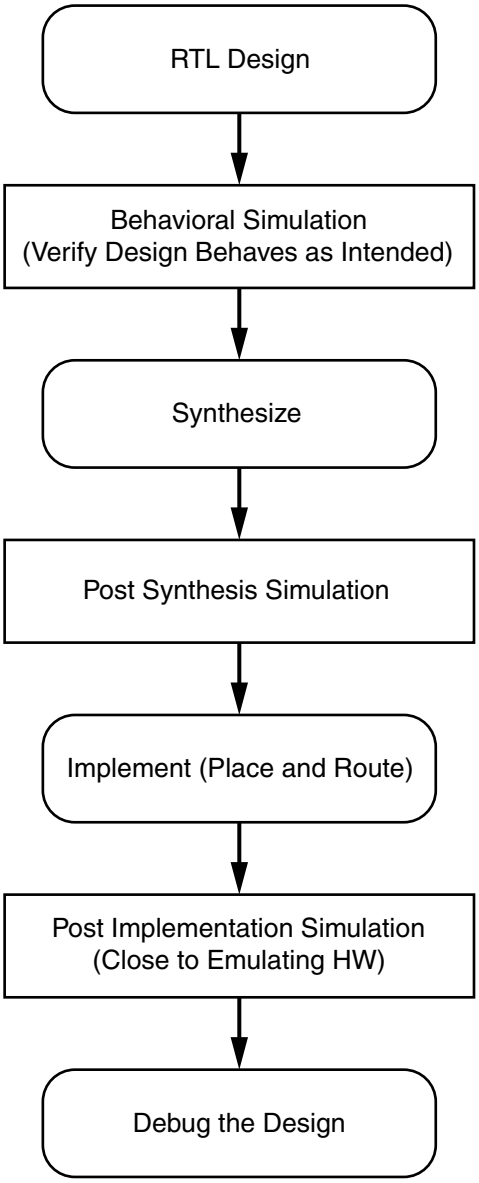


Figure 1-1: **Simulation Flow**

Behavioral Simulation at the Register Transfer Level

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

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

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

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

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

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

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

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

Post-Synthesis Simulation

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

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

These primitives are contained in the `UNISIMS_VER` library for Verilog, and the `UNISIM` library for VHDL. See UNISIM Library, page 15 for more information.

Post-Implementation Simulation

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

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

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

Supported Simulators

The Vivado Design Suite supports the following simulators:

- Vivado simulator: Integrated in the Vivado IDE. See Chapter 3, Using the Vivado Simulator from the Vivado IDE.
- Mentor Graphics QuestaSim/ModelSim: Integrated in the Vivado IDE. See Chapter 7, Simulating with QuestaSim/ModelSim for more information about integrated third-party simulators
- Cadence Incisive Enterprise Simulator (IES). See Chapter 8, Simulating with Cadence Incisive Enterprise Simulator (IES).
- Synopsys VCS and VCS MX. See Chapter 9, Simulating with Synopsys VCS.
- Aldec Active-HDL and Rivera-PRO*.

Note: Aldec offers support for Aldec simulators.

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

Language and Encryption Support

The Vivado simulator supports:

- VHDL, Verilog, and Standard Delay Format (SDF) [Ref 14]
- IEEE standards for language and encryption. For links to related IEEE standards, see the Additional Resources section of this document, Links to Language and Encryption Support Standards, page 180.

OS Support and Release Changes

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

Chapter 2

Understanding Simulation Components in Vivado

Introduction

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

The process of simulation includes:

- Creating a test bench that reflects the simulation actions you want to run
- Selecting and declaring the libraries you need to use
- Compiling your libraries (if using a third-party simulator)
- Netlist generation (if performing post-synthesis or post-implementation simulation)
- Understanding the use of global reset and 3-state in Xilinx devices

Using Test Benches and Stimulus Files

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

- Instantiates and initializes the design.
- Generates and applies stimulus to the design.
- 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 can sequentially apply stimulus to specific inputs.

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

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

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

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

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

- Always specify the `` `timescale `` in Verilog test bench files.
- Initialize all inputs to the design within the test bench at simulation time zero to properly begin simulation with known values.
- Apply stimulus data after `100ns` to account for the default Global Set/Reset (GSR) pulse used in functional and timing-based simulation.
- Begin the clock source before the Global Set/Reset (GSR) is released. For more information, see Using Global Reset and 3-State, page 31.

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

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

Using Xilinx Simulation Libraries

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

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

Similarly use variable assignment for local computation within process while signal assignment when you want data-flow across process.

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

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

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

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

Table 2-1: **Simulation Libraries**

Library Name	Description	VHDL Library Name	Verilog Library Name
UNISIM	Functional simulation of Xilinx primitives.	UNISIM	UNISIMS_VER
UNIMACRO	Functional simulation of Xilinx macros.	UNIMACRO	UNIMACRO_VER
UNIFAST	Fast simulation library.	UNIFAST	UNIFAST_VER
SIMPRIM	Timing simulation of Xilinx primitives.	N/A	SIMPRIMS_VER [a]
SECUREIP	Simulation library for both functional and timing simulation of Xilinx device features, such as the: • PCIe® IP • Gigabit Transceiver	SECUREIP	SECUREIP

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

It is important to note the following:

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

Table 2-2 lists the required simulation libraries at each simulation point.

Table 2-2: **Simulation Points and Relevant Libraries**

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

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

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

Table 2-3 lists the library locations.

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

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

The following subsections describe the libraries in more detail.

UNISIM Library

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

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

Encrypted Component Files

Table 2-4 lists the UNISIM library component files that let you call precompiled, encrypted library files when you include IP in a design. Include the path you require in your library search path. See Method 1: Using the complete UNIFAST library (Recommended), page 22 for more information.

Table 2-4: **Component Files**

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

VHDL UNISIM Library

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

- The component declarations (unisim_VCOMP.vhdp)
- Package files (unisim_VPKG.vhd)

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

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

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

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

Verilog UNISIM Library

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

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

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

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

- Using Verilog UNIFAST Library, page 21
- Chapter 8, Simulating with Cadence Incisive Enterprise Simulator (IES)
- Chapter 9, Simulating with Synopsys VCS

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

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

```
-L unisims_ver
```

UNIMACRO Library

The `UNIMACRO` library is used during functional simulation and contains macro descriptions for selective device primitives. You must specify the `UNIMACRO` library anytime you include a device macro listed in the *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* (UG953) [Ref 5].

VHDL UNIMACRO Library

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

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

Verilog UNIMACRO Library

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

The Verilog UNIMACRO library does not need to be specified in the HDL file prior to using the modules as is required in VHDL. To use the library module, specify the module name using all uppercase letters.

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 you choose.

SIMPRIM Library

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

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

TIP: *If you are a VHDL user, you can run post synthesis and post implementation functional simulation (in which case no standard default format (SDF) annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the write_vhdl Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 6].*

Specify this library as follows:

```
-L SIMPRIMS_VER
```

Where:

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

SECUREIP Simulation Library

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

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

Xilinx leverages the encryption methodology as specified in the IEEE standard *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP) (*IEEE-STD-P1735) [Ref 15]. The library compilation process automatically handles encryption.

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

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

Table 2-5: **Special Considerations for Using SECUREIP Libraries**

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

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

VHDL SECUREIP Library

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

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

Verilog SECUREIP Library

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

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

```
-L SECUREIP
```

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

UNIFAST Library

The `UNIFAST` library is an optional library that can be used during RTL behavioral simulation to speed up simulation run time.

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

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

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

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

MMCME2

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

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

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

RAMB18E1/RAMB36E1

To reduce the simulation runtimes, the fast block RAM simulation model has the following features disabled compared to the full model:

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

FIFO18E1/FIFO36E1

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

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

  Note: In some IP, such as FIFO Generator, these signals are mapped as `PROG_EMPTY` and `PROG_FULL`.

DSP48E1

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

- Pattern Detection
- OverFlow/UnderFlow
- DRP interface support

GTHE2_CHANNEL/GTHE2_COMMON

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

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

GTXE2_CHANNEL/GTXE2_COMMON

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

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

Using Verilog UNIFAST Library

There are two methods of simulating with the UNIFAST models.

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

The following subsections describe these simulation methods.

Method 1: Using the complete UNIFAST library (Recommended)

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

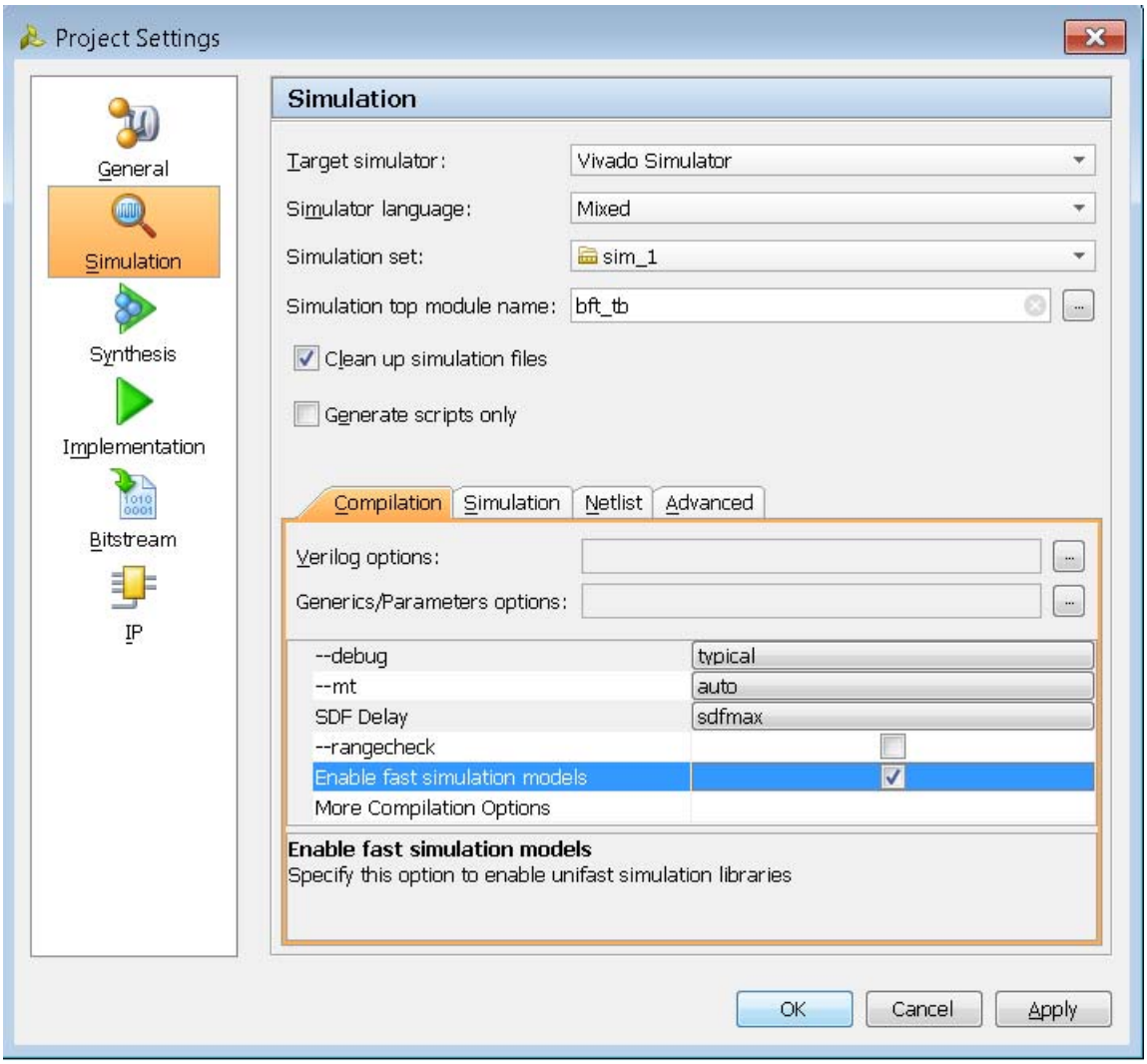


Figure 2-1: **Simulation Settings: Compilation**

See the Encrypted Component Files, page 15 for more information regarding component files.

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

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

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

Method 2: Using specific UNIFAST modules

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

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

  (For example: `instance testbench.inst.O1 use unifast_ver.MMCME2;`)

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

Example config.v

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

Using VHDL UNIFAST Library

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

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

Compiling Simulation Libraries

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

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

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

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

```
compile_simlib
```

Examples of `compile_simlib` Tcl commands include:

```
compile_simlib -simulator modelsim
```

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

Changing compile_simlib Defaults

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

- **Tcl Command**:

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

Where:

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

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

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

IMPORTANT: *The* `compile_simlib` *option compiles only Xilinx primitives. Simulation models of Xilinx Vivado IP cores are delivered as an output product when the IP is generated; consequently they are not included in the pre-compiled libraries created using* `compile_simlib`.

Understanding the Simulator Language Option

Most Xilinx IP deliver behavioral simulation models for a single language only, effectively disabling simulation for language-locked simulators if you are not licensed for the appropriate language. The `simulator_language` property ensures that an IP delivers a simulation model for any given language. (Figure 3-3, page 45 shows the location at which you can set the simulator language.)

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

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

1. Click **Window > IP Catalog**.
2. Right-click the appropriate IP and select **IP Settings** from the popup menu.
3. Click **Customize IP.**
4. In the Customize IP dialog box, click **OK**.

The Generate Output Products dialog box (shown in the figure below) appears.

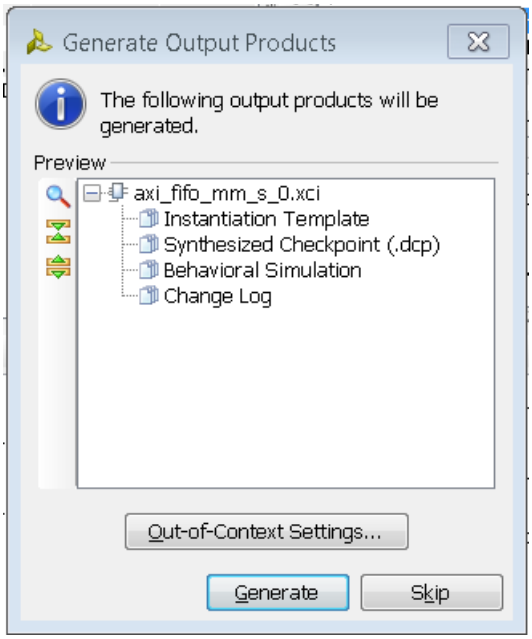


Figure 2-2: **Dialog Box Showing Generate Synthesized Checkpoint (.dcp) Option**

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

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

IP Delivered Simulation Model	simulator_language Value	Simulation Model Used
IP delivers VHDL and Verilog behavioral models	Mixed	Behavioral model (`target_language`)
	Verilog	Verilog behavioral model
	VHDL	VHDL behavioral model
IP delivers Verilog behavioral model only	Mixed	Verilog behavioral model
	Verilog	Verilog behavioral model
	VHDL	VHDL simulation netlist generated from DCP

Table 2-6: **Function of simulator_language Property *(Cont'd)***

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

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

Using the export_simulation Command

The `export_simulation` command lets you generate a standalone run file for IES and VCS.

The command exports a script and any associated data files for driving standalone simulation using the specified simulator.

When you run this command on the Vivado Tcl Console, ensure that switches are specified in the following order:

```
export_simulation  -of_objects <arg> -lib_map_path <arg>
-script_name <arg> [-absolute_path] [-32bit] [-force] -directory <arg>  -simulator
<arg> [-quiet] [-verbose]
```

The command returns: `true (0)` if successful; otherwise `false (1)`

Where:

- `[-of_objects]`
  Provides an export simulation script for the specified set of objects. The default set of objects is the active simulation top.
- `[-lib_map_path]`
  The precompiled simulation library directory path. If not specified, follow the instructions in the generated script header to manually provide the simulation library mapping information.
- `[-script_name]`
  Optional. The output shell script file name. If not specified, a file with a default name is created with the `.sh` extension.

- `[-absolute_path]`
  Makes all file paths absolute.
- `[-32bit]`
  Performs 32-bit compilation.
- `[-force]`
  Overwrites previous files.
- `-directory`
  Directory to export the simulation script.
- `-simulator`
  Simulator for which to create the simulation script (`<arg>: ies|vcs_mx`).
- `[-quiet]`
  Ignores command errors.
- `[-verbose]`
  Suspends message limits during command execution.

Examples:

The following command generates a simulation script file in the current directory for the IES simulator. The source file paths in the generated script are relative to the current directory:

```
export_simulation -simulator ies -directory "."
```

The following command generates a script file `accum_0_sim_ies.sh` for the `accum_0` IP in the specified output directory for the IES simulator:

```
export_simulation -of_objects [get_files accum_0.xci] -simulator ies -directory
"test_sim"
```

The following command generates a script file `accum_0_sim_vcs_mx.sh` for the `accum_0` IP in the specified output directory for the `VCS_MX` simulator:

```
export_simulation -of_objects [get_ips accum_0] -simulator vcs_mx -directory
"test_sim"
```

The following command includes `/sim_libs/ius/lin64/lib/cds.lib` file path in the `./test_sim/cds.lib` file ("`INCLUDE /sim_libs/ius/lin64/lib/cds.lib`") for referencing the compiled libraries for IES simulator:-

```
export_simulation -lib_map_path "/sim_libs/ius/lin64/lib" -simulator ies -directory
"test_sim"
```

Recommended Simulation Resolution

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

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

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

Generating a Netlist

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

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

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

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

Generating a Functional Netlist

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

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

These primitives are contained in the following libraries:

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

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

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

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

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

Generating a Timing Netlist

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

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

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

There are two steps to generating a timing simulation netlist:

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

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

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

- **Tcl Command**:

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

Annotating the SDF File

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

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

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

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

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

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

Using Global Reset and 3-State

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

Global Set and Reset Net

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

RECOMMENDED: *Although you can access the GSR net after configuration, recommends avoiding use of the GSR circuitry in place of a manual reset. This is because the FPGA devices offer high-speed backbone routing for high fanout signals such as a system reset. This backbone route is faster than the dedicated GSR circuitry, and is easier to analyze than the dedicated global routing that transports the GSR signal.*

In post-synthesis and post-implementation simulations, the GSR signal is automatically asserted for the first 100 ns to simulate the reset that occurs after configuration.

A GSR pulse can optionally be supplied in pre-synthesis functional simulations, but is not necessary if the design has a local reset that resets all registers.

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

Global 3-State Net

In addition to the dedicated global GSR, output buffers are set to a high impedance state during configuration mode with the dedicated Global 3-state (GTS) net. All general-purpose outputs are affected whether they are regular, 3-state, or bidirectional

outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the FPGA is configured.

In simulation, the GTS signal is usually not driven. The circuitry for driving GTS is available in the post-synthesis and post-implementation simulations and can be optionally added for the pre-synthesis functional simulation, but the GTS pulse width is set to 0 by default.

Using Global 3-State and Global Set and Reset Signals

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

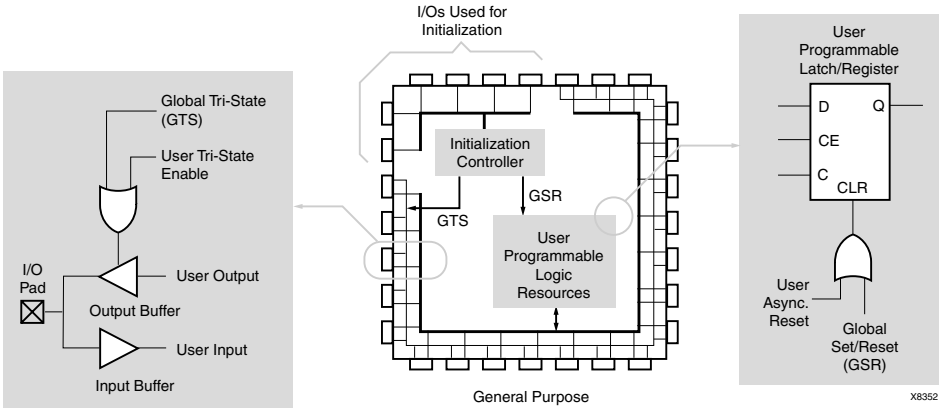


Figure 2-3: **Built-in FPGA Initialization Circuitry Diagram**

Global Set and Reset and Global 3-State Signals in Verilog

The GSR and GTS signals are defined in the <Vivado_Install_Dir>/data/verilog/src/glbl.v module.

In most cases, GSR and GTS need not be defined in the test bench.

The glbl.v file declares the global GSR and GTS signals and automatically pulses GSR for 100 ns.

Global Set and Reset and Global 3-State Signals in VHDL

The GSR and GTS signals are defined in the file: <Vivado_Install_Dir>/data/vhdl/src/unisims/primitive/GLBL_VHD.vhd.

To use the GLBL_VHD component you must instantiate it into the test bench.

The GLBL_VHD component declares the global GSR and GTS signals and automatically pulses GSR for 100 ns.

The following code snippet shows an example of instantiating the GLBL_VHD component in the test bench and changing the assertion pulse width of the Reset on Configuration (ROC) to 90 ns:

```
GLBL_VHD inst:GLBL_VHD generic map (ROC_WIDTH => 90000);
```

Delta Cycles and Race Conditions

This user guide describes event-based simulators. Event-based simulators can process multiple events at a given simulation time. While these events are being processed, the simulator cannot advance the simulation time. This event processing time is commonly referred to as *delta cycles*. There can be multiple delta cycles in a given simulation time step.

Simulation time is advanced only when there are no more transactions to process. For this reason, simulators can give unexpected results, depending on when the events are scheduled within a time step. The following VHDL coding example shows how an unexpected result can occur.

VHDL Coding Example With Unexpected Results

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

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

In this example, there are two synchronous processes:

- clk_prcs
- clk_b_prcs

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

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

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

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

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

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

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

Using the ASYNC_REG Constraint

The ASYNC_REG constraint:

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

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

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

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

The ASYNC_REG constraint is applicable to CLB and Input Output Block (IOB) registers and latches only.

RECOMMENDED: *If you cannot avoid clocking in asynchronous data, Xilinx recommends that you do so for IOB or CLB registers only. Clocking in asynchronous signals to RAM, Shift Register LUT (SRL), or other synchronous elements has less deterministic results; therefore, should be avoided. Xilinx highly recommends that you first properly synchronize any asynchronous signal in a register, latch, or FIFO before writing to a RAM, Shift Register LUT (SRL), or any other synchronous element. For more information, see the Vivado Design Suite User Guide: Using Constraints (UG903) [Ref 7].*

Disabling X Propagation for Synchronous Elements

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

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

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

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

Correcting X-Generation

To correct X-generation:

- On a synchronous path, analyze the path and fix any timing problems associated with this or other paths to ensure a properly operating circuit.
- On an asynchronous path, if you cannot otherwise avoid timing violations, disable the X propagation on synchronous elements during timing violations by the method described in Using the ASYNC_REG Constraint, page 34.

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

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

Simulating Configuration Interfaces

This section describes the simulation of the following configuration interfaces:

- JTAG simulation
- SelectMAP simulation

JTAG Simulation

BSCAN component simulation is supported on all devices.

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

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

This becomes:

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

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

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

SelectMAP Simulation

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

Table 2-7 lists the supported interfaces and devices.

Table 2-7: **Supported Configuration Devices and Modes**

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

The model handles control signal activity as well as bit file downloading. Internal register settings such as the CRC, IDCODE, and status registers are included. You can monitor the Sync Word as it enters the device and the start-up sequence as it progresses. Figure 2-4, below, illustrates how the system should map from the hardware to the simulation environment.

The configuration process is specifically outlined in the configuration user guides for each device. These guides contain information on the configuration sequence, as well as the configuration interfaces.

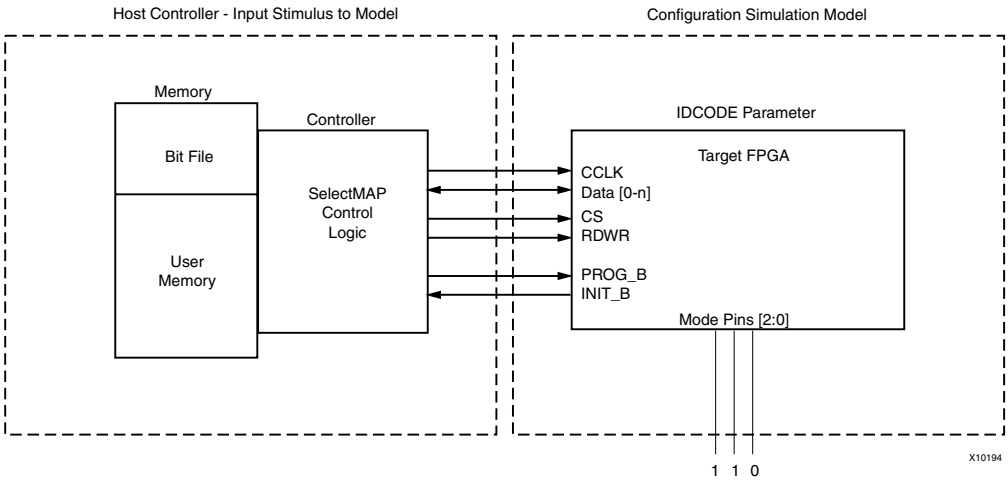


Figure 2-4: **Block Diagram of Model Interaction**

System Level Description

The SIM_CONFIGE2 model allows the configuration interface control logic to be tested before the hardware is available. It simulates the entire device, and is used at a system level for:

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

The ZIP file associated with this model is located at:

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

The ZIP file has sample test benches that simulate a processor running the SelectMAP logic. These test benches have control logic to emulate a processor controlling the SelectMAP interface, and include features such as a full configuration, ABORT, and Readback of the IDCODE and status registers.

The simulated host system must have a method for file delivery as well as control signal management. These control systems should be designed as set forth in the device configuration user guides.

The SIM_CONFIGE2 model also demonstrates what is occurring inside the device during the configuration procedure when a BIT file is loaded into the device.

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

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

Debugging with the Model

The `SIM_CONFIGE2` model provides an example of a correct configuration. You can leverage this example to assist in the debug procedure if you encounter device programming issues.

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

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

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

The `SIM_CONFIGE2` model also allows for error injection. The active CRC logic detects any issue if the data load is paused and started again with any problems. It also detects bit flips manually inserted in the BIT file, and handles them just as the device would handle this error.

Feature Support

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

The `SIM_CONFIGE2` model:

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

Table 2-8: **Model-Supported Slave SelectMAP and Serial Features**

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

Disabling Block RAM Collision Checks for Simulation

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

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

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

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

Table 2-9 shows the string options you can use with SIM_COLLISION_CHECK to control simulation behavior in the event of a collision.

Table 2-9: **SIM_COLLISION_CHECK Strings**

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

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

Dumping the Switching Activity Interchange Format File for Power Analysis

The Switching Activity Interchange Format (SAIF) is an ASCII report that assists in extracting and storing switching activity information generated by simulator tools.

This switching activity can be back-annotated into the Xilinx power analysis and optimization tools for the power measurements and estimations.

See the information about the respective simulator for more detail:

- Vivado simulator: Power Analysis Using Vivado Simulator, page 126
- Dumping SAIF in QuestaSim/ModelSim, page 140
- Dumping SAIF for Power Analysis in IES, page 148
- Dumping SAIF for Power Analysis for VCS, page 155

Chapter 3

Using the Vivado Simulator from the Vivado IDE

Introduction

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

The Vivado simulator is a Hardware Description Language (HDL) event-driven simulator that supports functional and timing simulations for VHDL, Verilog, and mixed VHDL/Verilog designs. See the "Important" note on page 60 for timing simulation support information.

See *Vivado Design Suite Tutorial: Logic Simulation* (UG937) [Ref 8] for a step-by-step demonstration of how to run Vivado simulation.

Vivado Simulator Features

The Vivado simulator supports the following features:

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

Adding or Creating Simulation Source Files

To add simulation sources to a Vivado Design Suite project:

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

   The Add Sources wizard opens.

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

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

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

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

     For a demonstration of this feature, see the Quick Take Video at www.xilinx.com/training/vivado/index.htm.

   - Add Files: Invokes a file browser so you can select simulation source files to add to the project.
   - Add Directories: Invokes directory browser to add all simulation source files from the selected directories. Files in the specified directory with valid source file extensions are added to the project.
   - Create File: Invokes the Create Source File dialog box where you can create new simulation source files. See the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 2] for more information about project source files.
   - Buttons on the side of the dialog box let you do the following:
     - Remove: Removes the selected source files from the list of files to be added.
     - Move Selected File Up: Moves the file up in the list order.
     - Move Selected File Down: Moves the file down in the list order.
   - Check boxes in the wizard provide the following options:
     - Scan and add RTL include files into project: Scans the added RTL file and adds any referenced include files.
     - Copy sources into project: Copies the original source files into the project and uses the local copied version of the file in the project.

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

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

Working with Simulation Sets

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

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

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

To edit a simulation set:

1. In the Sources window popup menu, select **Simulation Sources > Edit Simulation Sets**, as shown in Figure 3-1.

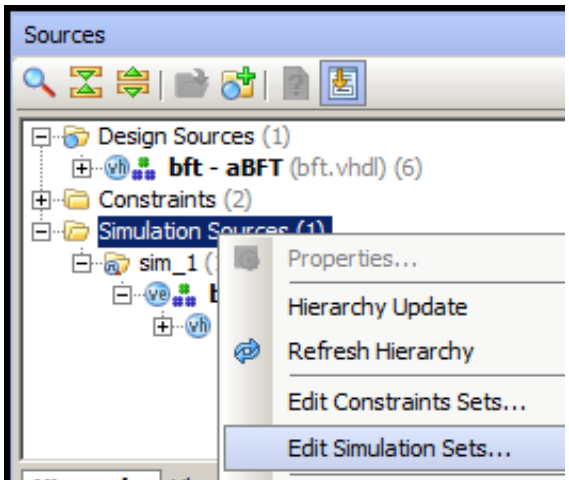


Figure 3-1: **Edit Simulation Sets Option**

The Add or Create Simulation Sources wizard opens.

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

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

3. Add additional files as needed.

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

Using Simulation Settings

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

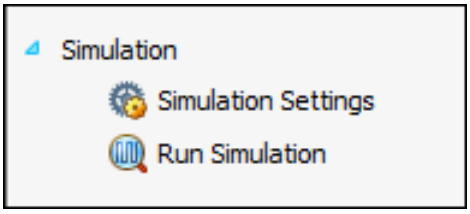


Figure 3-2: **Flow Navigator Simulation Options**

- Simulation Settings: Opens the Simulation Settings dialog box where you can select and configure the Vivado simulator.
- Run Simulation: Sets up the command options to compile, elaborate, and simulate the design based on the simulation settings, then launches the Vivado simulator. When you run simulation prior to synthesizing the design, the Vivado simulator runs a behavioral simulation, and opens a waveform window, (see Figure 3-12, page 56) that shows the HDL objects with the signal and bus values in either digital or analog form.

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

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

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

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

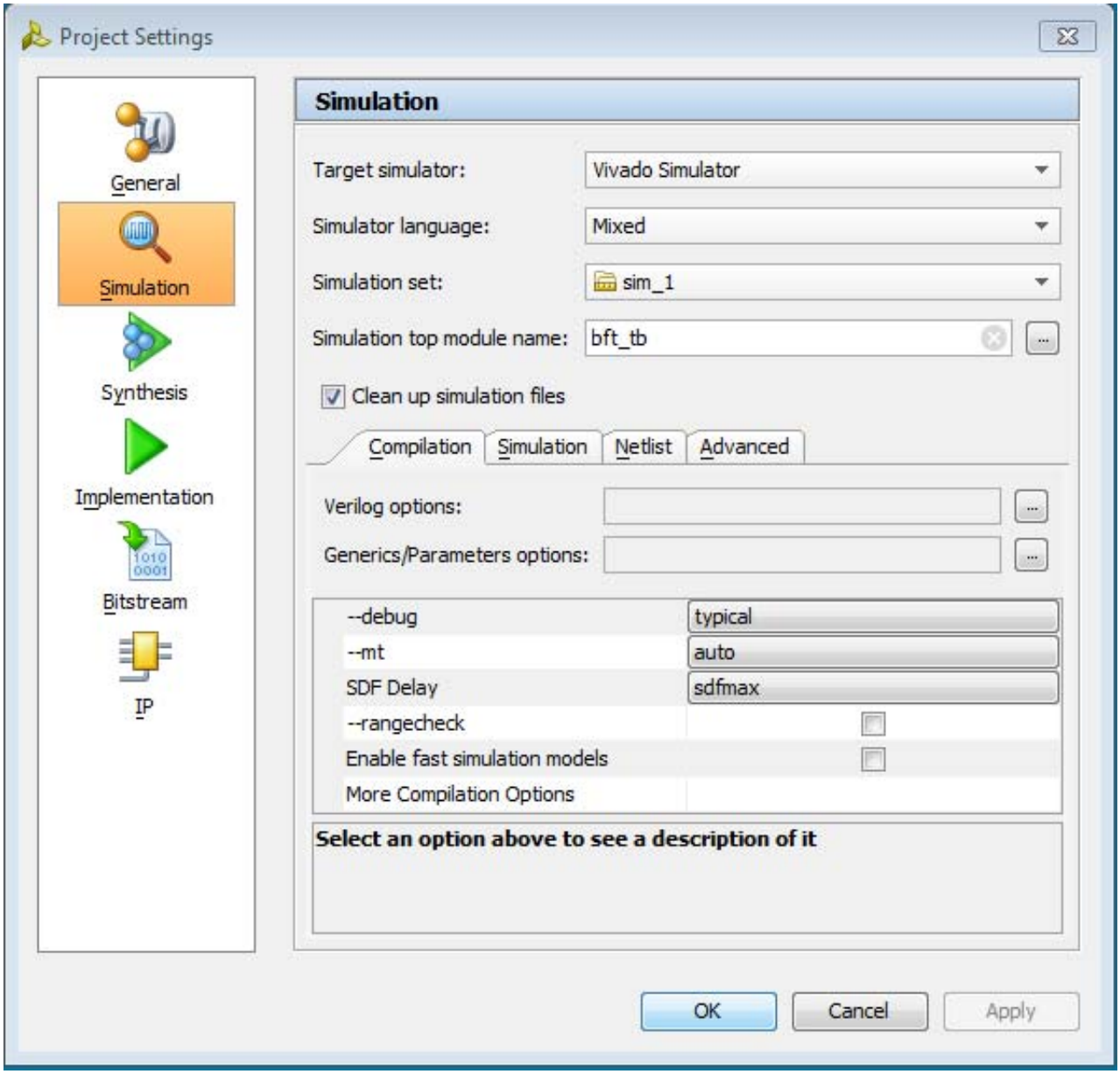


Figure 3-3: **Simulation Settings Dialog Box**

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

Simulation Dialog Box Options

Navigation: Flow Navigator > Simulation > Simulation Settings

Note: Available options in the dialog box are based on the target simulator selected.

Target simulator

Specifies the simulator to be launched for behavioral or timing simulation. The available options are:

- Vivado Simulator: Specifies the target simulator is the Vivado simulator.

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

- QuestaSim/ModelSim: Specifies that QuestaSim/ModelSim is the target simulator.

  See Chapter 7, Simulating with QuestaSim/ModelSim for third-party simulation information.

Simulator language

- Options are VHDL, Verilog, or Mixed.

Simulation set

- Selects the active simulation set using an existing simulation set or by using the Create simulation set option, described in Working with Simulation Sets, page 43.

Simulation top module name

- Sets the simulation top module.

Clean up simulation files (check box)

Selects the option to delete all content of the `\simset` simulation directory prior to running the simulation.

Generate scripts only (check box)

Sets the option to generate only the target simulator project and scripts. When selected, the simulation tool is not launched, and simulation is not performed.

Compiled library location

Selects the precompiled library location when QuestaSim or ModelSim is the target simulator.

- See the Using Xilinx Simulation Libraries, page 13 for information about how to specify simulation libraries.
- See Chapter 7, Simulating with QuestaSim/ModelSim, for more information regarding QuestaSim/ModelSim.

Compilation Tab

Navigation: Flow Navigator > Simulation > Simulation Settings > Simulation Dialog Box > Simulation tab

Shown in the figure below, the Compilation tab provides the frequently-used compilation options based on the selected target simulator.

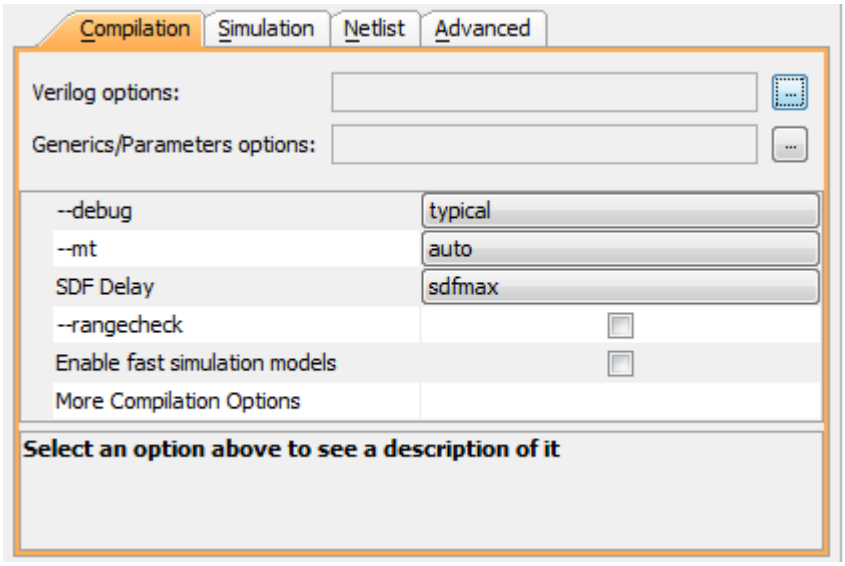


Figure 3-4: **Project Settings, Simulation Dialog Box, Compilation Tab**

- Verilog options: Provides the option to specify file search paths and `` `define `` values.
- Generics/Parameters options: Provides the option to set and override top-level generic/parameter values.
- Command options:
  - `-debug`: Sets the simulation debug visibility level. Is set to `typical` by default for a faster simulation run. Other options are `off` and `all`. The `typical` setting includes `wave` and `line` options.

- `-mt`: Is set to `auto` by default. This option lets you specify the number of sub-compilation job to run in parallel. When you select the option, a drop-down menu displays the following selections, as shown in Figure 3-4, above: `off`, `2`, `4`, `8`, `16`, and `32`.

  You can also set the `-mt` level using the Tcl command:
  `set_property XELAB.MT_LEVEL off [get_filesets sim_1]`

- SDF Delay:

  The Standard Delay Format (SDF) provides for maximum, typical, and minimum delay values. This option specifies the delay type to be used during timing simulation. This option value is set to `sdfmax` (setup verification) by default. Optionally, you can select `sdfmin` from the drop-down menu. The typical SDF delay values are not characterized by Xilinx, so the `typical` option is not available.

- `-rangecheck`:

  This check box is disabled by default for a faster simulation run. This option lets you enable or disable a run time value range check for VHDL.

- Enable fast simulation models:

  The Xilinx simulation libraries use gate-level simulation models for hard IP components by default. When selected, this option enables the use of behavioral UNIFAST library models for simulation. These models allow for faster simulation speeds at the expense of simulation accuracy. See UNIFAST Library, page 19 for more information regarding this library.

Simulation Tab

Navigation: Flow Navigator > Simulation > Simulation Settings > Simulation Dialog Box > Simulation tab

As shown in Figure 3-5, below, the Simulation tab provides available simulation options, based on the target simulator selected. You can select an option to see a description. Select from the options described below.

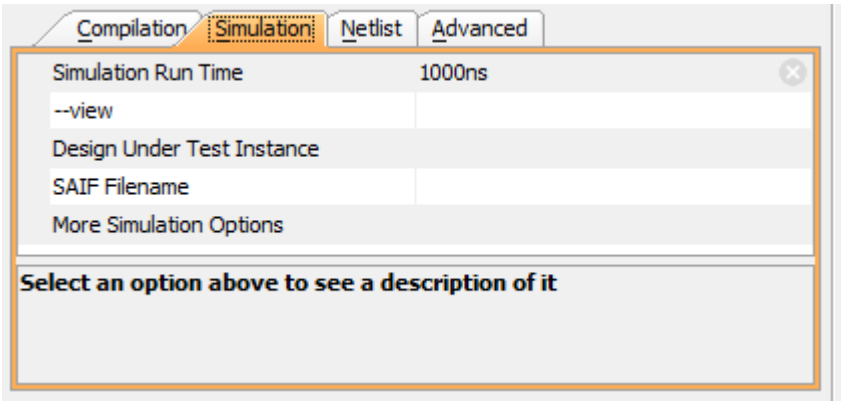


Figure 3-5: **Project Settings, Simulation Dialog Box, Simulation Tab**

- Simulation Run Time: Specifies the amount of simulation time to run automatically when the simulation is launched. The defaults is 1,000 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.
- Design Under Test Instance: Lets you specify the instance name of the design under test. The default is `/uut`.
- SAIF Filename: Lets you specify an SAIF file name if one was created for power optimization and estimation (timing simulation only).

TIP: *When you select an option, a tool tip 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 Vivado IDE open the WCFG file using the* `-view` *option in later runs, which can save time on setting up simulation waveforms.*

Netlist Tab

Navigation: Flow Navigator > Simulation > Simulation Settings > Simulation Dialog Box > Netlist tab

As shown in Figure 3-6, below, the Netlist tab provides the simulation `write_verilog` netlist settings options. Select an option to see its description.

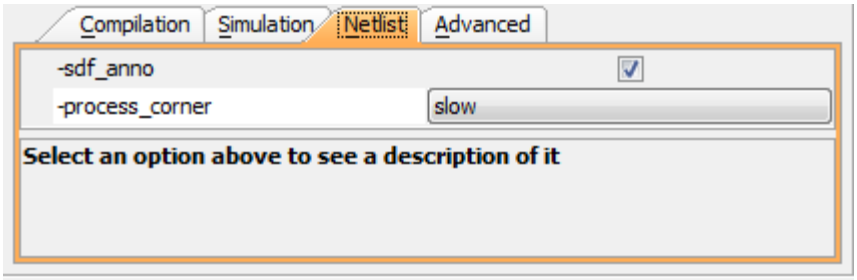


Figure 3-6: **Simulation Settings: Netlist View**

Netlist options include:

- `-sdf_anno`: check box enables SDF annotation.

  Enable the `-sdf_anno`: check box select the Tcl command:
  `write_sdf -process_corner <fast|slow> test.sdf`

- `-process_corner`: You can specify the `-process_corner` as `fast` or `slow`. Default setting: `slow`.

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.*

Advanced Tab

Navigation: Flow Navigator > Simulation > Simulation Settings > Simulation Dialog Box > Advanced tab

Shown in Figure 3-7, below, the Advanced tab provides an option to include all design sources for simulation.

CAUTION! *This is an advanced user feature. Unchecking the box could produce unexpected results.*

- The *Include all design sources for simulation* check box is selected by default. As long as the check box is selected, the simulation set includes Out-of-Context (OOC) IP, IP Integrator files, and DCP.
- Unchecking the box gives you the flexibility to include only the files you want to simulate, but, as stated above, you might experience unexpected results.

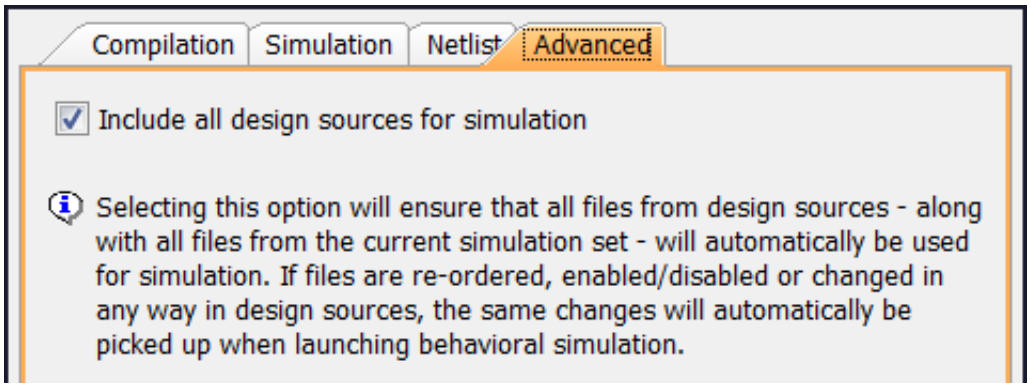


Figure 3-7: **Advanced Simulation Settings View**

Running the Vivado Simulator

From the Flow Navigator, select **Run Simulation** to display the Vivado simulator GUI, shown in Figure 3-8.

The main components of the Vivado simulator GUI are:

1. Main Toolbar
2. Run Menu
3. Objects Window
4. Simulation Toolbar
5. Wave Objects
6. Waveform Window
7. Scopes Window
8. Sources Window

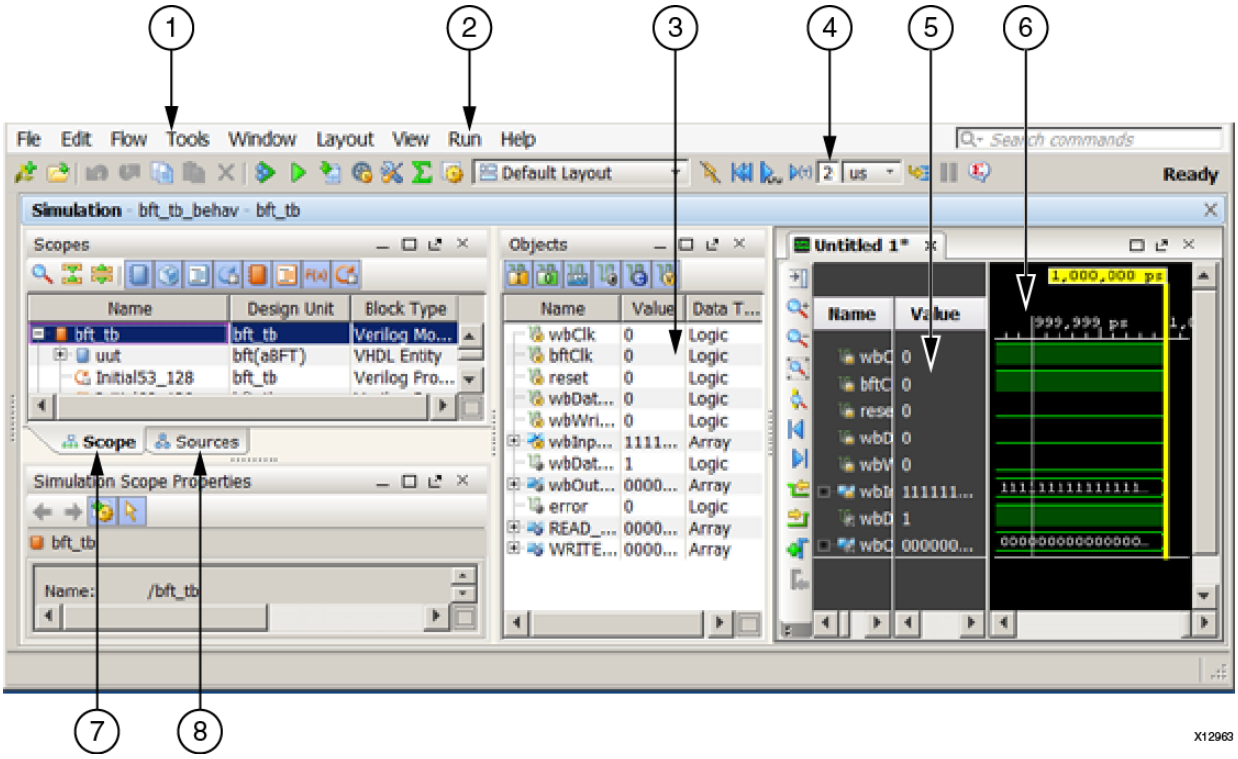


Figure 3-8: **Vivado Simulator GUI**

Main Toolbar

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

Run Menu

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

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

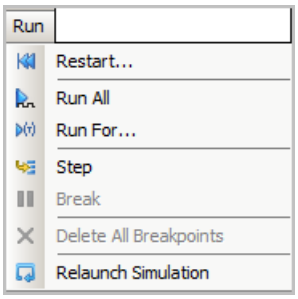


Figure 3-9: **Simulation Run Menu Options**

The Vivado simulator Run menu options:

- Restart: Lets you restart an existing simulation from 0.
  Tcl Command: `restart`
- Run All: Lets you run an open simulation to completion.
  Tcl Command: `run all`
- Run For: Lets you specify a time for the simulation to run.
  Tcl Command: `run <time>`
- Step: Runs the simulation up to the next HDL source line.
- Break: Lets you interrupt a running simulation.
- Delete All Breakpoints: Deletes all breakpoints.
- Relaunch Simulation: Recompiles the simulation files and relaunches the run.

Objects Window

The HDL Objects window displays the HDL objects in the design, as shown in the figure below.

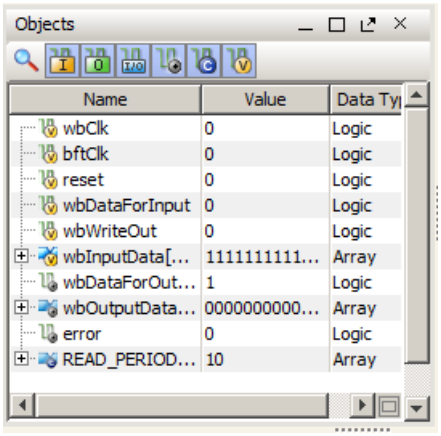


Figure 3-10: **HDL Objects Window**

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

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

```
get_value <hdl_object>
```

Table 3-1 briefly describes the buttons at the top of the Objects window as follows:

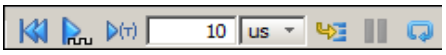
Table 3-1: **HDL Object Buttons**

Button	Description
	The **Search** button, when selected, opens a field in which you can enter an object name on which to search.
	Input signal.
	Output signal.
	Input/Output signal.
	Internal signal.
	Constant signal.
	Variable signal.

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

Simulation Toolbar

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



When you hover your mouse over the toolbar buttons, a tool tip describes the button function. The buttons are also labeled in Figure 3-9, page 53.

Wave Objects

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

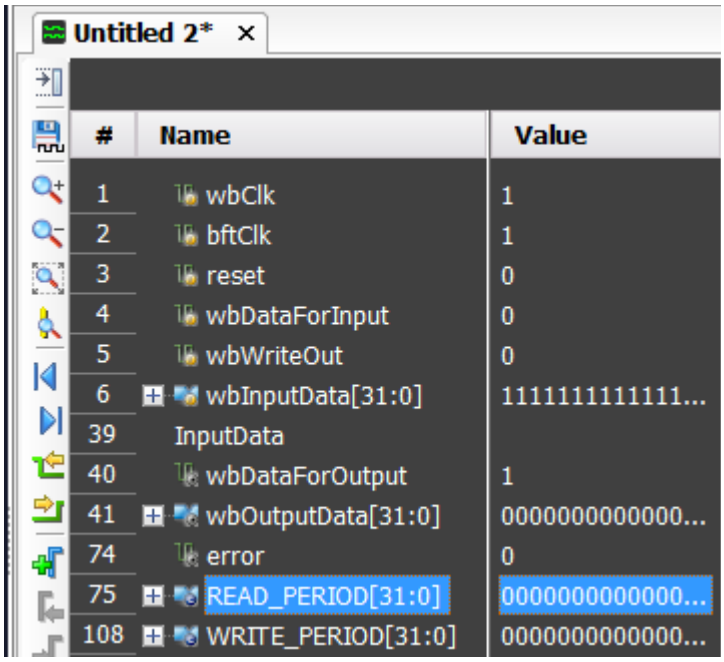


Figure 3-11: **HDL Objects in Waveform**

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

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

The Vivado IDE traces the HDL object in the waveform configuration during simulation, and you use the wave configuration to examine the simulation results.

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

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

Waveform Window

When you invoke the simulator, by default, it opens a waveform window that displays a new wave configuration consisting of the traceable top-module of HDL objects in the simulation as shown in Figure 3-12.

TIP: *Displaying the Waveform Window: on closing and reopening a project, you must rerun simulation to invoke the Waveform Window.*

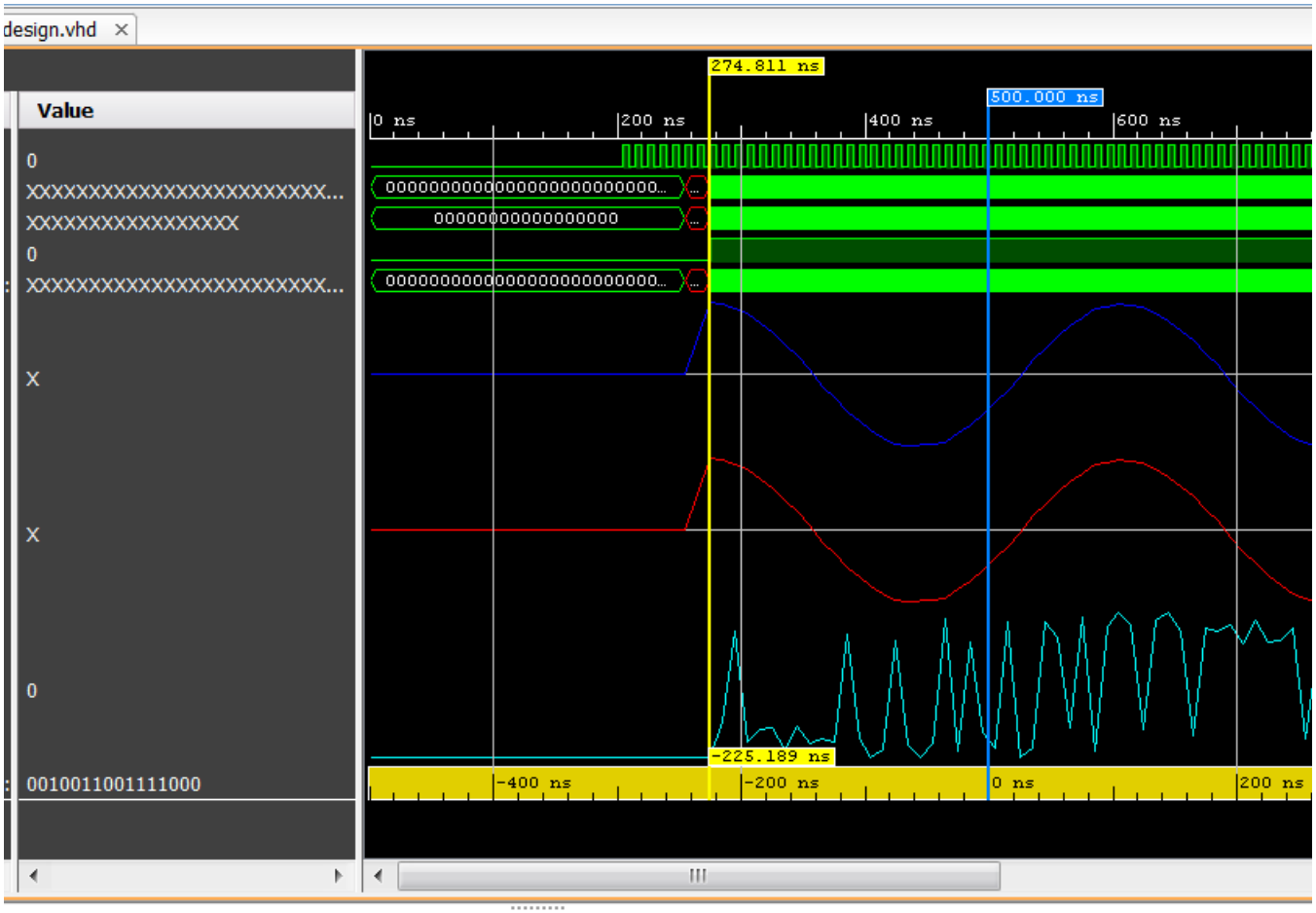


Figure 3-12: **Waveform Window**

To add an individual HDL object or set of objects to the waveform window, You can right-click the object and select the **Add to Wave Window** option from the context menu.

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

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

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

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

Saving a Waveform

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

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

The specified command argument names and saves the WCFG file.

Creating and Using Multiple Waveform Configurations

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

You can set a different waveform window to be the *active* window by clicking the title of the window. See Identifying Between Multiple Simulation Runs, page 62 and Creating a New Wave Configuration, page 67 for more information.

Scopes Window

Figure 3-13 shows the Scopes Window, where you can view and filter HDL objects by type using the filter buttons at the top of the window. Hover over a button for a tool tip description of what object type the filter button represents.

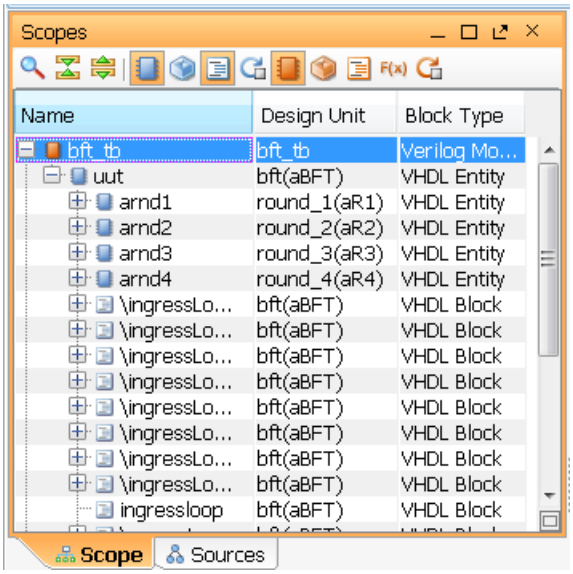


Figure 3-13: **Scopes Window**

Sources Window

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

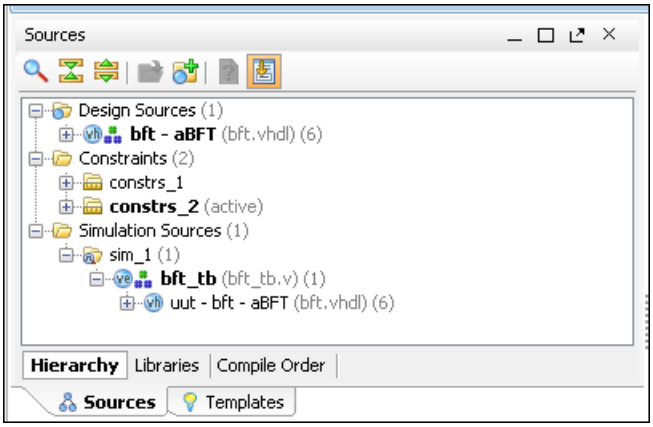
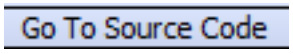


Figure 3-14: **Sources Window**

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

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

Additional Scopes and Sources Options

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

See Using the Scopes Window, page 70 for more detail on how to use the Scopes window in the Vivado simulator. As an equivalent to using the Scopes and Objects windows, you can navigate the HDL design by typing the following in the Tcl Console:

```
get_scopes
current_scope
report_scopes
report_values
```

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

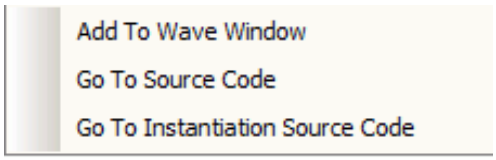
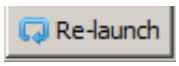


Figure 3-15: **Objects Context Menu**

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

Running Post-Synthesis Simulation

Post-Synthesis and Post-Implementation timing simulation can optionally include one of the following:

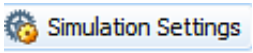
- Gate-level netlist containing `SIMPRIMS` library components
- `SECUREIP`
- Standard Delay Format (SDF) files

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

You use Post-Implementation timing simulation after your design is completely through the implementation (Place and Route) process in Vivado IDE. You can now begin to observe how your design behaves in the actual circuit.

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

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

You control these options using Simulation Settings 

as described in Using Simulation Settings, page 44.

IMPORTANT: *Post-Synthesis and Post-Implementation timing simulations are supported for Verilog only. There is no support for VHDL timing simulation. If you are a VHDL user, you can run post synthesis and post implementation functional simulation (in which case no SDF annotation is required and the simulation netlist uses the UNISIM library). You can create the netlist using the write_vhdl Tcl command. For usage information, refer to the Vivado Design Suite Tcl Command Reference Guide (UG835) [Ref 6].*

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

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

Running Post-Synthesis Functional Simulation

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

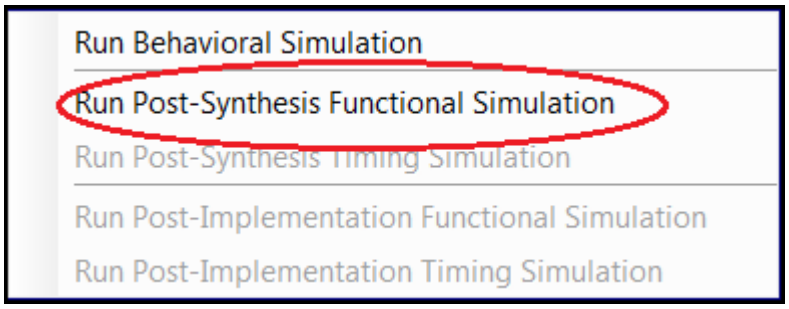


Figure 3-16: **Run Post-Synthesis Functional Simulation**

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

Running Post-Synthesis Timing Simulation

When synthesis is run successfully, the **Run Simulation > Post-Synthesis Timing Simulation** option becomes available, as shown in the figure below.

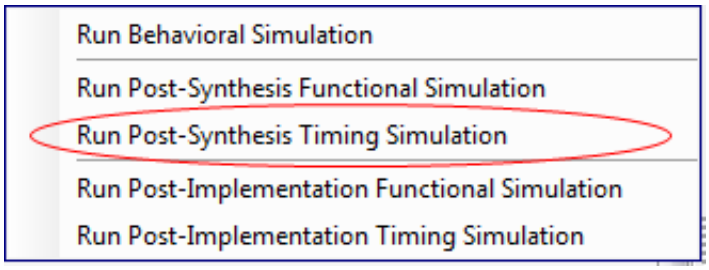


Figure 3-17: **Run Post-Synthesis Timing Simulation**

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

Running Post-Implementation Simulations

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

Running Post-Implementation Functional Simulations

When implementation is successful, the **Run Simulation > Post-Implementation Functional Simulation** option is available, as shown in the figure below.

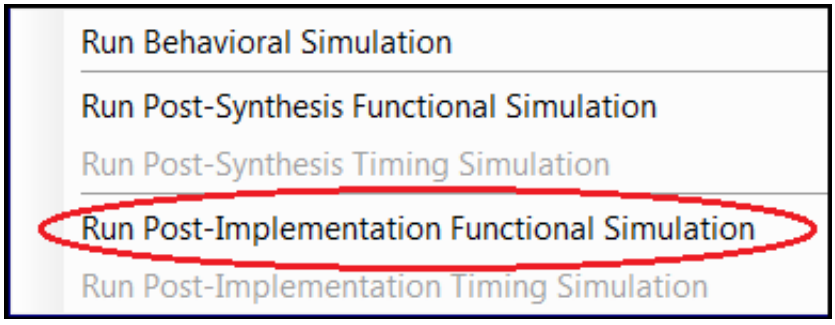


Figure 3-18: **Run Post-Implementation Functional Simulation**

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

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

Running Post-Implementation Timing Simulations

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

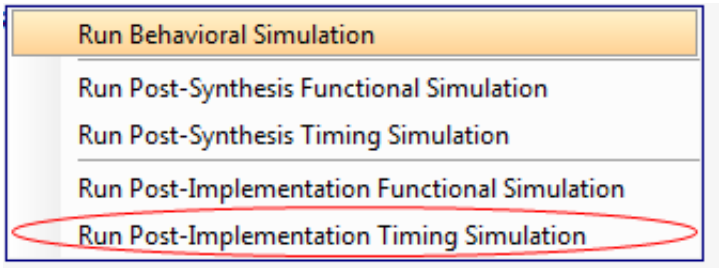


Figure 3-19: **Run Post-Implementation Timing Simulation**

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

Identifying Between Multiple Simulation Runs

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

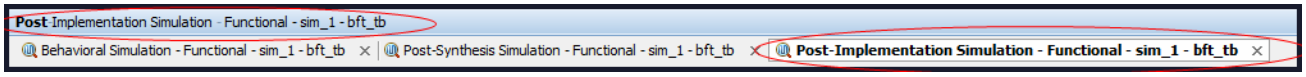


Figure 3-20: **Active Simulation Type**

Pausing a Simulation

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

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

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

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

Resume the simulation any time using the Run All, Run, or Step commands. See Stepping Through a Simulation, page 119 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; are saved to the wave configuration (WCFG) file upon demand. See Chapter 4, Analyzing with the Vivado Simulator Waveforms.

Closing Simulation

To close a simulation, in the Vivado IDE:

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

To close a simulation from the Tcl Console, type:

```
close_sim
```

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

Adding a post.tcl Batch File

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

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

   ```
   run 5us
   ```

2. Name the file `post.tcl`, and place it in an available location.
3. Use the **Add Sources** button to invoke the Add Sources wizard, and select **Add or Create Simulation Sources**.

4. Add the `post.tcl` file to your Vivado Design Suite project as a simulation source. The `post.tcl` file displays in the Simulation Sources folder, as shown in the figure below.

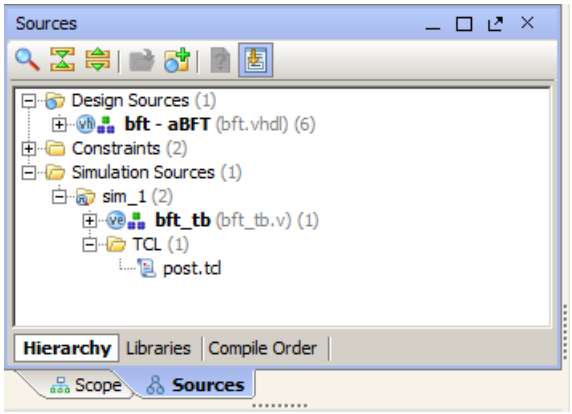


Figure 3-21: **Using the post.tcl File in a Design**

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

   Simulation runs again, with the additional time you specified in the `post.tcl` file added to the originally specified time. Notice that the Vivado simulator automatically sources the `post.tcl` file and the resulting simulation runs for the additional time.

Skipping Compilation or Simulation

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

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

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

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

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

The Vivado tools skip the execution of simulation step.

Viewing Simulation Messages

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

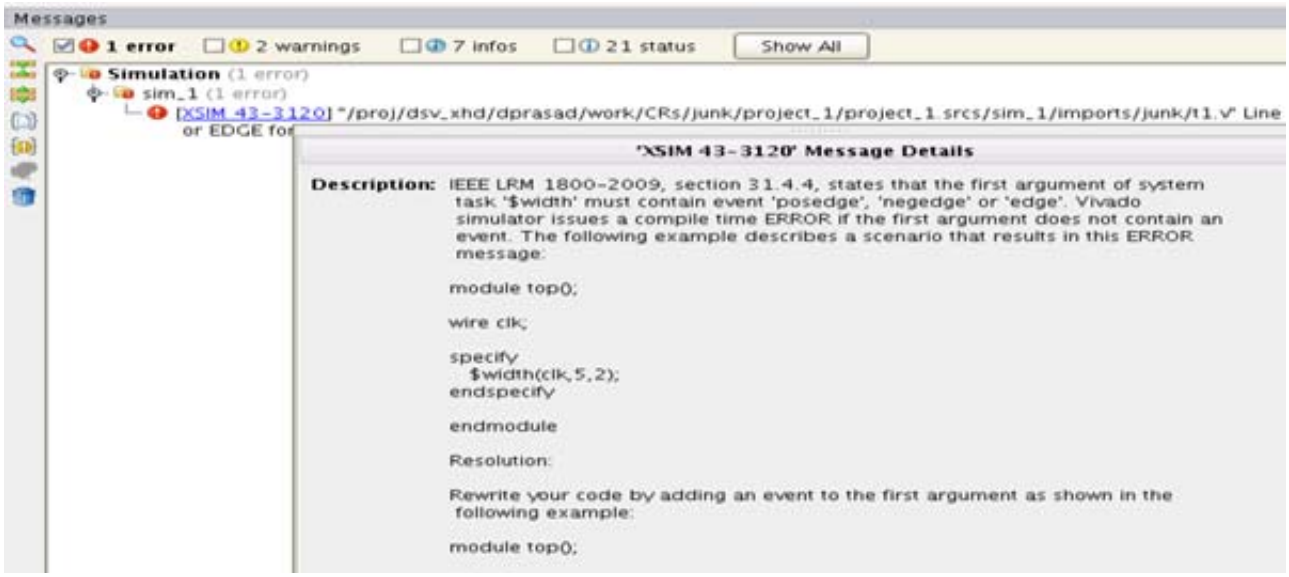


Figure 3-22: **Simulator Message Description and Resolution Information**

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

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

An example of such a command is as follows:

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

Analyzing with the Vivado Simulator Waveforms

Introduction

In the Vivado® simulator GUI, you can work 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 4].

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

Using Wave Configurations and Windows

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

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

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

Creating a New Wave Configuration

Create a new waveform configuration for displaying waveforms as follows:

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

   A new waveform window opens and displays a new, untitled waveform configuration.
   `add_wave <HDL_Object>`.

2. Add HDL objects to the waveform configuration using the steps listed in Understanding HDL Objects in Waveform Configurations, page 69.

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

See Chapter 3, Using the Vivado Simulator from the Vivado IDE for more information about creating new waveform configurations. Also see Creating and Using Multiple Waveform Configurations, page 57 for information on multiple waveforms.

Opening a WCFG File

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

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

   The Specify Simulation Results dialog box opens.

2. Locate and select a WCFG file.

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

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

   Tcl command: `open_wave_config <waveform_name>`

Saving a Wave Configuration

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

Tcl command: `save_wave_config <waveform_name>`

Opening a Previously Saved Simulation Run

There are two methods for opening a previously saved simulation in Vivado: an interactive method and a programmatic method.

Interactive Method

- If a Vivado project is loaded, click **View > Open Static Simulation** and select the WDB file containing the waveform from the previously run simulation.
- Alternatively, in the Tcl console, run: `open_wave_database <name>.wdb`.

Programmatic Method

Create a Tcl file (for example, `design.tcl`) with contents: `current_fileset; open_wave_database <name>.wdb`. Then run it as `vivado -source design.tcl`.

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

The WDB file also stores information about all the HDL scopes and objects in the simulated design.

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

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.

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.*

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

Understanding HDL Objects in Waveform Configurations

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

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

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

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

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

Figure 4-1 shows an example of HDL objects in the waveform configuration window. The design objects display Name and Value.

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

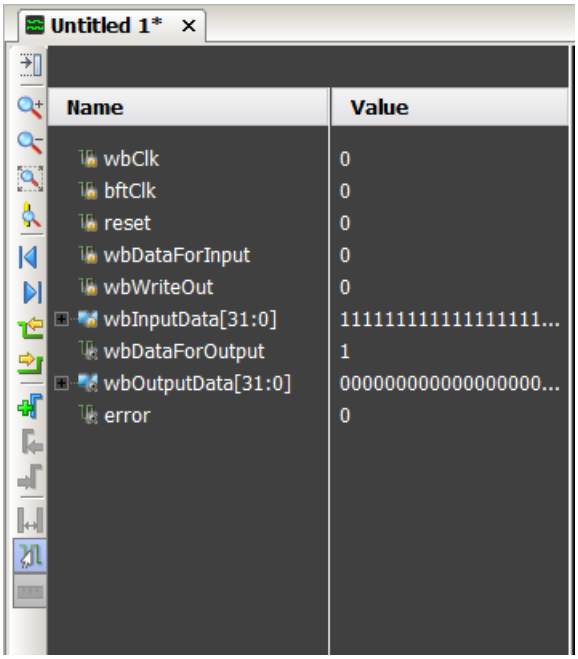


Figure 4-1: **Waveform HDL Objects**

Using the Scopes Window

Figure 4-2 shows the Vivado simulator Scopes window.

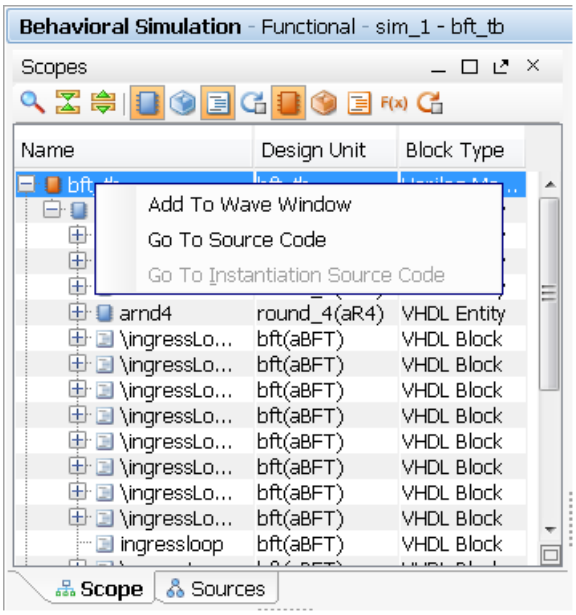


Figure 4-2: **Scopes Window**

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 **Search** button. and type the string in the text box.

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

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

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

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

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

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

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

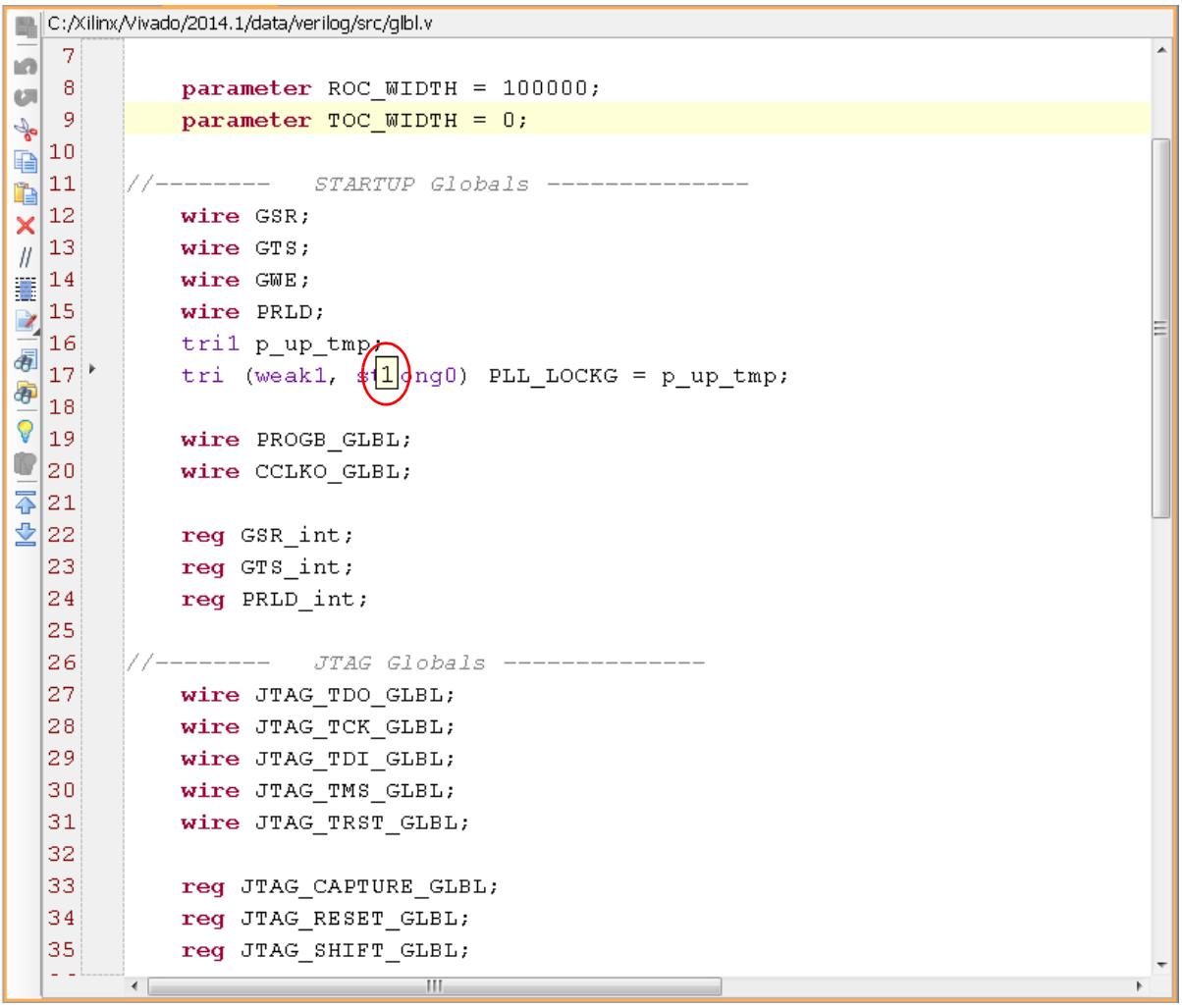


Figure 4-3: **Source Code with Identifier Value Displayed**

Using the Objects Window

Figure 4-4 shows the Vivado simulator Objects window.

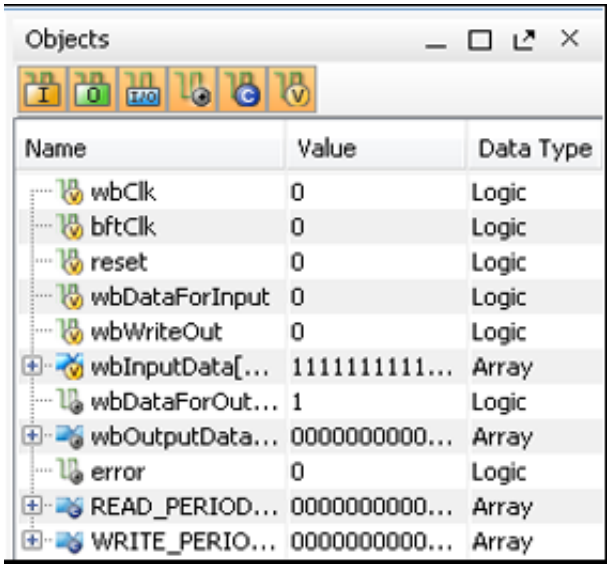
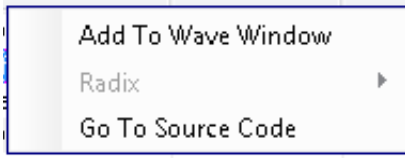


Figure 4-4: **Objects Window**

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

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



- Add to Wave Window: Add the selected object to the waveform configuration.

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

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

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

Customizing the Waveform

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

Using Analog Waveforms

The following subsections describe the features and requirements around using analog waveforms.

Using Radixes and Analog Waveforms

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

- Binary, octal, hexadecimal, ASCII, and unsigned decimal 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, based on settings of the Real Settings dialog box.

To set a wave object to the Real radix:

1. Open the **Real Settings** dialog box, shown in Figure 4-5.
2. In the waveform configuration window, select an HDL object, and right-click to open the popup menu.

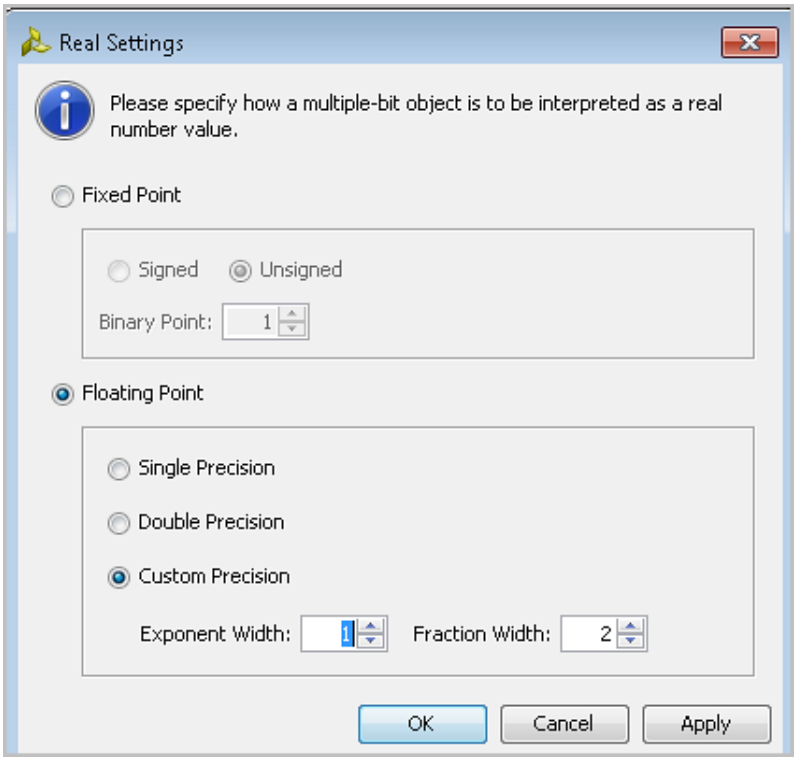


Figure 4-5: **Real Settings Dialog Box**

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

The Real Setting dialog box options are:

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

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

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

TIP: *If the row indices separator lines are not visible, you can turn them on in the Using the Waveform Options Dialog Box, page 81, to make them visible.*

Displaying Waveforms as Analog

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

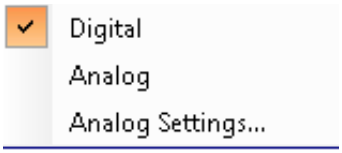
For example:

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

Customizing the Appearance of Analog Waveforms

To customize the appearance of an analog waveform:

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



   - Analog: Sets a Digital waveform to Analog.
   - Digital: Sets an Analog waveform object to Digital.
   - Analog Settings: Figure 4-6 shows the Analog Settings dialog box with the settings for analog waveform drawing.

Figure 4-6: **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, so a waveform that reaches the upper- or lower-bound of the waveform area follows the bound as a horizontal line until values are once again within the range.
  - Overlap: Specifies that the waveform be drawn wherever its values are, even if they lie outside the bounds of the waveform area and overlap other waveforms, up to the limits of the waveform window itself.
- Horizontal Line: Specifies whether to draw a horizontal rule at the given value. If the check-box is on, a horizontal grid line is drawn at the vertical position of the specified Y value, if that value is within the Y range of the waveform.

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

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

About Radixes

Understanding the type of data on your bus is important. You need to recognize the relationship between the radix setting and the data type to use the waveform options of Digital and Analog effectively. See Displaying Waveforms as Analog, page 76 for more information about the radix setting and its effect on Analog waveform analysis.

Changing the Default Radix

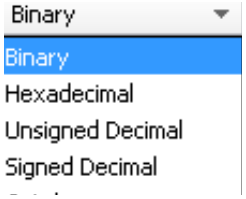
The default waveform radix controls the numerical format of values for all wave objects whose radix you did not explicitly set. The default waveform radix defaults to **`binary`**.

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

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

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

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

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

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

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

Waveform Object Naming Styles

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

Renaming Objects

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

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

   The Rename dialog box opens.

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

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

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

Changing the Object Name Display

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

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

To change the display name:

1. Select one or more signal or bus names. Use Shift+click or Ctrl+click to select many signal names.
2. Select **Name >**:
   - **Long** to display the full hierarchical name.
   - **Short** to display the name of the signal or bus only.
   - **Custom** to display the custom name given to the signal when renamed. See Renaming Objects, page 79.

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

Reversing the Bus Bit Order

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

To reverse the bit order:

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

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

Using the Waveform Options Dialog Box

Select the **Waveforms Options** button to open the Waveform Options dialog box, shown in the figure below.

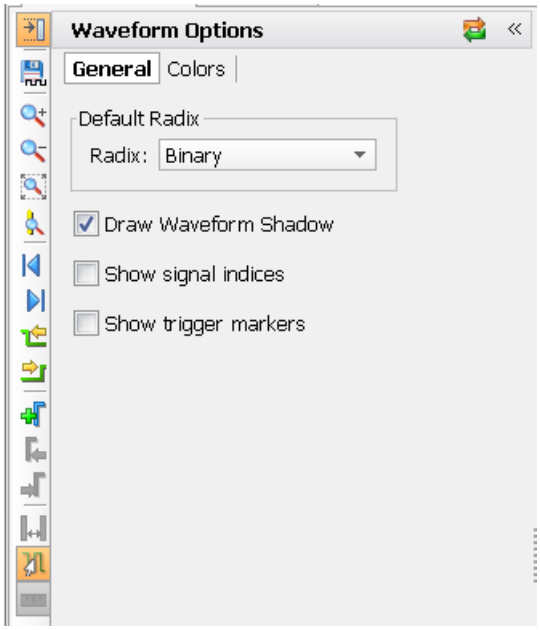


Figure 4-7: **Waveform Options Dialog Box**

The General Waveform Options are:

- Default Radix: Sets the numerical format to use for newly-created design wave objects.
- Draw Waveform Shadow: Creates a shaded representation of the waveform.
- Show signal indices: check box displays the row numbers to the left of each wave object name. You can drag the lines separating the row numbers to change the height of a wave object.
- Show trigger markers: Has not effect on Vivado Simulator waveforms.
- The Colors page lets you set colors of items within the waveform.

Controlling the Waveform Display

You can control the waveform display using:

- Zoom feature buttons in the HDL Objects window sidebar
- Zoom combinations with the mouse wheel
- Vivado IDE Y-Axis zoom gestures
- Vivado simulation X-Axis zoom gestures. See the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 2] for more information about the Vivado IDE X-Axis zoom gestures.

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

Using the Zoom Feature Button

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

Zooming with the Mouse Wheel

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

Y-Axis Zoom Gestures

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

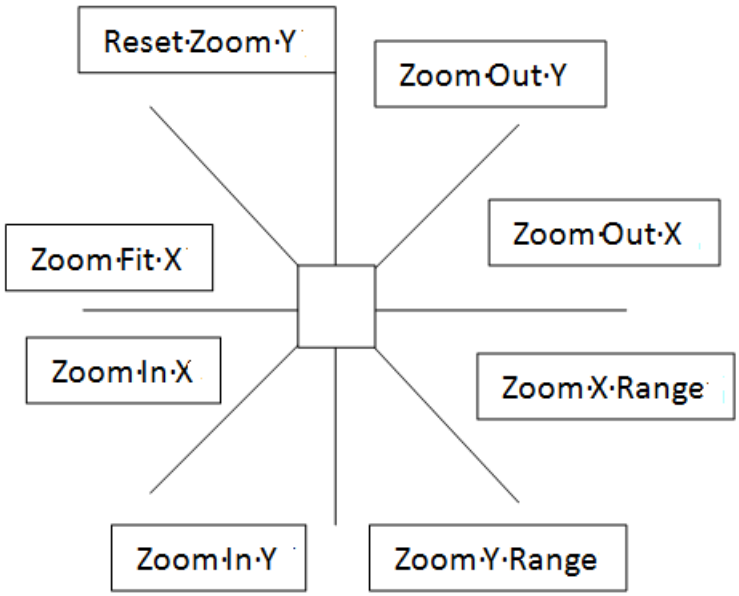


Figure 4-8: **Analog Zoom Options**

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

The additional zoom gestures are:

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

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

Be aware of the following limitations:

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

Organizing Waveforms

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

Using Groups

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

To add a Group:

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

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

2. Select **Edit > New Group**, or right-click and select **New Group** from the context menu.

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

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

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

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

You can move or remove Groups as follows:

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

Groups can be renamed also; see Renaming Objects, page 79.

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

Using Dividers

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

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

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

   Tcl command: `add_wave_divider`

You can move or delete Dividers as follows:

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

Dividers can be renamed also; see Renaming Objects, page 79.

Using Virtual Buses

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

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

To add a virtual bus:

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

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

   Tcl Command: `add_wave_virtual_bus`

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

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

You can rename a virtual bus; see Renaming Objects, page 79.

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

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

Analyzing Waveforms

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

Using Cursors

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

TIP: *WCFG files do not record cursor positions. For more permanent indicators, used in situations such as establishing a time-base for multiple measurements, and indicating notable events in the simulation, add markers to the waveform window instead. See Using Markers, page 88 for more information.*

Placing Main and Secondary Cursors

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

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

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

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

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 next or previous transition of selected waveform or from the current position of the cursor.

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

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

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

2. Click the **Next Transition** or **Previous Transition** sidebar button, or use the right or left keyboard arrow key to move to the next or previous transition, respectively.

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

Using the Floating Ruler

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

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

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

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

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

   Select the command again to hide the floating ruler.

Using Markers

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

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

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

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

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

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

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

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

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

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

Using Force Options

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

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

- Force Constant
- Force Clock
- Remove Force

Force Constant

The Force Constant option lets you assign a new constant force that overrides the assignments made within HDL code or previously applied constant or clock force.

Force Constant is an option on the Objects window context menu, as shown in the figure below:

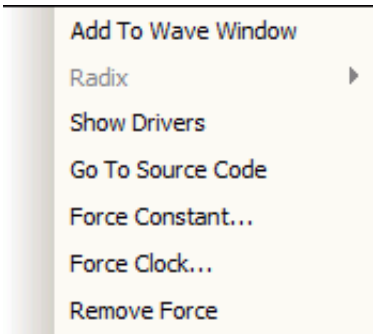


Figure 4-9: **Force Options**

When you select the Force Constant option, the Force Constant dialog box opens where you can enter the relevant values, as shown in the figure below.

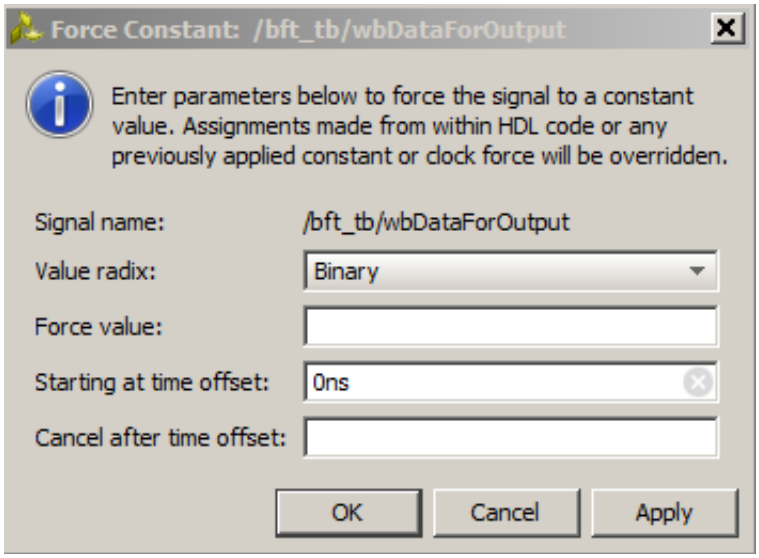


Figure 4-10: **Force Selected Signal Dialog Box**

The Force Constant options descriptions:

- Signal name: Displays the default signal name, that is, the full path name of the selected item.
- Value radix: Displays the current radix setting of the selected signal. You can choose one of the supported radix types: Binary, Hexadecimal, Unsigned Decimal, Signed Decimal, Octal, and ASCII. The GUI then disallows entry of the values based on the Radix setting. For example: if you choose Binary, no numerical values other than 0 and 1 are allowed.
- Force value: Specifies a force constant value using the defined radix value.

- Starting at time offset: Starts after the specified time. The default starting time is 0. Time can be a string, such as 10 or 10 ns. When you enter a number without a unit, the Vivado simulator uses the default.
- Cancel after time offset: Cancels after the specified time. Time can be a string such as 10 or 10 ns. If you enter a number without a unit, the default simulation time unit is used.

  TCL commands:

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

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

Force Clock

When you select the **Force Clock** option in the Objects window menu, the Force Clock dialog box opens, as shown in the figure below:

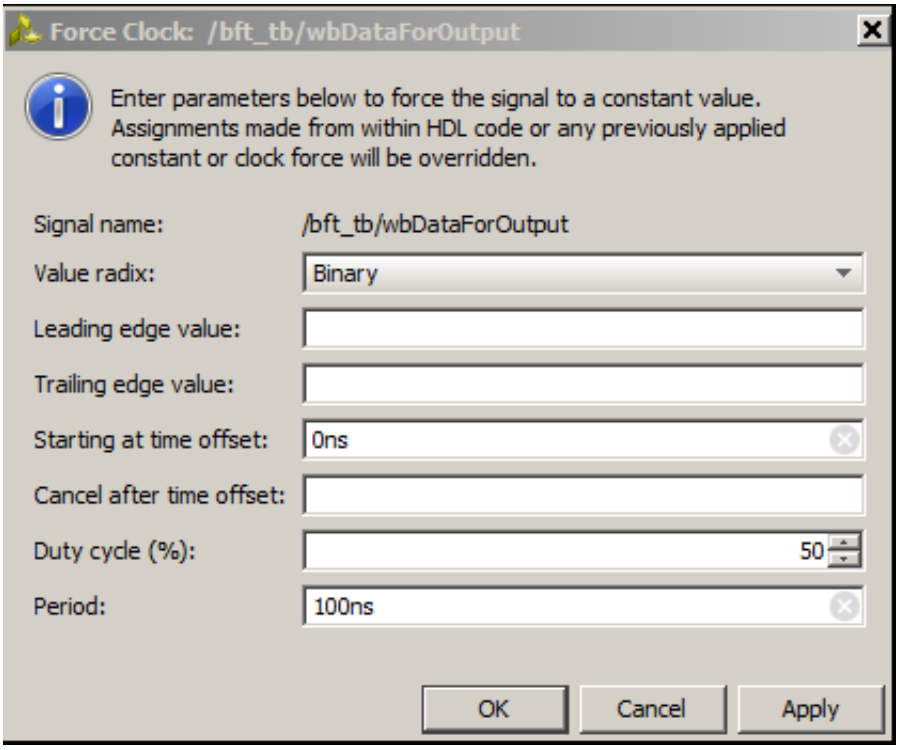


Figure 4-11: **Force Clock Dialog Box**

The options in the Force Clock dialog box are:

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

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

- Value radix: Displays the current radix setting of the selected signal. Select one of the displayed radix types from the drop-down menu: `Binary`, `Hexadecimal`, `Unsigned Decimal`, `Signed Decimal`, `Octal`, or `ASCII`.
- Leading edge value: Specifies the first edge of the clock pattern. The leading edge value uses the radix defined in Value radix field.
- Trailing edge value: Specifies the second edge of the clock pattern. The trailing edge value uses the radix defined in the Value radix field.
- Starting at time offset: Starts the force command after the specified time from the current simulation. The default starting time is 0. Time can be a string, such as `10` or `10 ns`. If you enter a number without a unit, the Vivado simulator uses the default user unit.
- Cancel after time offset: Cancels the force command after the specified time from the current simulation time. Time can be a string, such as 10 or 10 ns. When you enter a number without a unit, the Vivado simulator uses the default simulation time unit.
- Duty cycle (%): Specifies the percentage of time that the clock pulse is in an active state. The acceptable value is a range from 0 to 100.
- Period: Specifies the length of the clock pulse, defined as a time value. Time can be a string, such as 10 or 10 ns.

Remove Force

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

```
remove_force <force object>
```

Chapter 5

Using Vivado Simulator Command Line and Tcl

Introduction

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

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

```
help -category sim
```

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

Compiling and Simulating a Design

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

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

The following subsections describe these steps.

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

- Generating a Timing Netlist in Chapter 2
- Running Post-Synthesis and Post-Implementation Simulations, page 112

Parsing Design Files

The `xvhdl` and `xvlog` commands parse VHDL and Verilog files, respectively. Descriptions for each option are available in Table 5-2, page 103.

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

xvhdl

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

xvhdl Syntax

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

This command parses the VHDL source file(s) and stores the parsed dump into a HDL library on disk.

xvhdl Examples

```
xvhdl file1.vhd file2.vhd
xvhdl -work worklib file1.vhd file2.vhd
xvhdl -prj files.prj
```

xvlog

The xvlog command is the Verilog parser. The xvlog command parses the Verilog source file(s) and stores the parsed dump into a HDL library on disk.

xvlog Syntax

```
xvlog
[-d [define] <name>[=<val>]]
[-encryptdumps]
[-f [-file] <filename>]
[-h [-help]]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=<library_dir>]]
[-log <filename>]
[-nolog]
[-relax]
[-prj <filename>]
[-sourcelibdir <sourcelib_dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-sv]
[-v [verbose] [0|1|2]]
[-version]
[-work <library_name> [=<library_dir>]
```

xvlog Examples

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

Elaborating and Generating a Design Snapshot

Simulation with the Vivado simulator happens in two phases:

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

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

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

xelab

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

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

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

TIP: `xelab` *can implicitly call the parsing commands,* `xvlog` *and* `xvhdl`*. You can incorporate the parsing step by using the* `xelab -prj` *option. See* *Project File (.prj) Syntax, page 108* *for more information about project files.*

xelab Command Syntax Options

Descriptions for each option are available in Table 5-2, page 103.

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

```
xelab
[-d [define] <name>[=<val>]
[-debug <kind>]
[-f [-file] <filename>]
[-generic_top <value>]
[-h [-help]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-log <filename>]
[-L [-lib] <library_name> [=<library_dir>]
[-maxdesigndepth arg]
[-mindelay]
[-typdelay]
[-maxdelay]
[-mt arg]
[-nolog]
[-notimingchecks]
[-nosdfinterconnectdelays]
[-nospecify]
```

```
[-O arg]
[-Odisable_acceleration arg]
[-Odisable_process_opt]
[-Oenable_cdfg]
[-Odisable_cdfg]
[-Oenable_unused_remova]
[-Odisable_unused_removal]
[-override_timeunit]
[-override_timeprecision]
[-prj <filename>]
[-pulse_e arg]
[-pulse_r arg]
[-pulse_int_e arg]
[-pulse_int_r arg]
[-pulse_e_style arg]
[-r [-run]]
[-R [-runall
[-rangecheck]
[-relax]
[-s [-snapshot] arg]
[-sdfnowarn]
[-sdfnoerror]
[-sdfroot <root_path>]
[-sdfmin arg]
[-sdftyp arg]
[-sdfmax arg]
[-sourcelibdir <sourcelib_dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-stat]
[-timescale]
[-timeprecision_vhdl arg]
[-trace_limit arg]
[-transport_int_delays]
[-v [verbose] [0|1|2]]
[-version]
```

xelab Examples

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

Verilog Search Order

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

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

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

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

Verilog Instantiation Unit

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

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

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

IMPORTANT: *Connecting a whole VHDL record object to a Verilog object is unsupported.*

VHDL Instantiation Unit

When a VHDL design instantiates a component, the `xelab` command treats the component name as a VHDL unit and searches for it in the logical `work` library.

- If a VHDL unit is found, the `xelab` command binds it and the search stops.
- If `xelab` does not find a VHDL unit, it treats the case-preserved component name as a Verilog module name and continues a case-sensitive search in the user-specified list and order of unified logical libraries. The command selects the first matching the name, then stops the search.
- If case sensitive search is not successful, `xelab` performs a case-insensitive search for a Verilog module in the user-specified list and order of unified logical libraries. If a unique binding is found for any one library, the search stops.

`uselib Verilog Directive

The Verilog `` `uselib `` directive is supported, and sets the library search order.

`uselib Syntax

```
<uselib compiler directive> ::= `uselib [<Verilog-XL uselib directives>|<lib
directive>]
<Verilog-XL uselib directives> :== dir = <library_directory> | file = <library_file>
| libext = <file_extension>
<lib directive>::= <library reference> {<library reference>}
<library reference> ::= lib = <logical library name>
```

`uselib Lib Semantics

The `` `uselib lib `` directive cannot be used with any of the Verilog-XL `` `uselib `` directives. For example, the following code is illegal:

```
`uselib dir=./ file=f.v lib=newlib
```

Multiple libraries can be specified in one `` `uselib `` directive.

The order in which libraries are specified determines the search order. For example:

```
`uselib lib=mylib lib=yourlib
```

Specifies that the search for an instantiated module is made in `mylib` first, followed by `yourlib`.

Like the directives, such as `` `uselib dir ``, `` `uselib file ``, and `` `uselib libext ``, the `` `uselib lib `` directive is persistent across HDL files in a given invocation of parsing and analyzing, just like an invocation of parsing is persistent. Unless another `` `uselib `` directive is encountered, a `` `uselib `` (including any Verilog XL `` `uselib ``) directive in the HDL source

remains in effect. A `` `uselib `` without any argument removes the effect of any currently active `` `uselib <lib|file|dir|libext> ``.

The following module search mechanism is used for resolving an instantiated module or UDP by the Verific Verilog elaboration algorithm:

- First, search for the instantiated module in the ordered list of logical libraries of the currently active `` `uselib lib `` (if any).
- If not found, search for the instantiated module in the ordered list of libraries provided as search libraries in `xelab` command line.
- If not found, search for the instantiated module in the library of the parent module. For example, if module A in library `work` instantiated module B of library `mylib` and B instantiated module C, then search for module C in the /`mylib`, library, which is the library of 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

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

Simulating the Design Snapshot

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

xsim Executable Syntax

The command syntax is as follows:

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

Where:

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

xsim Executable Options

Table 5-1: **xsim Executable Command Options**

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

Table 5-1: **xsim Executable Command Options *(Cont'd)***

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

TIP: *When running the xelab, xsc, xsim, xvhdl, or xvlog commands in batch files or scripts, it might also be necessary to define the XILINX_VIVADO environment variable to point to the installation hierarchy of the Vivado Design Suite. To set the XILINX_VIVADO variable, add one of the following to your script or batch file:*

On Windows: `set XILINX_VIVADO=<vivado_install_area>/Vivado/2014.1`

On Linux: `setenv XILINX_VIVADO vivado_install_area>/Vivado/2014.1`

xelab, xvhd, and xvlog Command Options

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

Table 5-2: **xelab, xvhd, and xvlog Command Options**

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

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

Command Option	Description	Used by Command
`-L [-lib] <library_name> [=<library_dir>]`	Specify search libraries for the instantiated non-VHDL design units; for example, a Verilog design unit. Use `-L|--lib` for each search library. The format of the argument is `<name>[=<dir>]` where `<name>` is the logical name of the library and `<library_dir>` is an optional physical directory of the library.	xelab xvhdl xvlog
`-log <filename>`	Specify the log file name. Default: `<xvlog|xvhdl|xelab|xsim>.log`.	xelab xsim xvhdl xvlog
`-maxarraysize arg`	Set maximum vhdl array size to be 2**n (Default: n = 28, which is 2**28)	xelab
`-maxdelay`	Compile Verilog design units with maximum delays.	xelab
`-maxdesigndepth arg`	Override maximum design hierarchy depth allowed by the elaborator (Default: 5000).	xelab
`-maxlogsize arg (=-1)`	Set the maximum size a log file can reach in MB. The default setting is unlimited.	xsim
`-mindelay`	Compile Verilog design units with minimum delays.	xelab
`-mt arg`	Specifies the number of sub-compilation jobs which can be run in parallel. Possible values are `auto`, `off`, or an `integer` greater than 1. If `auto` is specified, `xelab` selects the number of parallel jobs based on the number of CPUs on the host machine. (Default = `auto`). To further control the `-mt` option, an advanced user can set the Tcl property as follows: `set_property XELAB.MT_LEVEL off|N [get_filesets sim_1]`	xelab
`-nolog`	Suppress log file generation.	xelab xvhdl xvlog
`-noieeewarnings`	Disable warnings from VHDL IEEE functions.	
`-notimingchecks`	Ignore timing check constructs in Verilog specify block(s).	xelab
`-nosdfinterconnectdelays`	Ignore SDF port and interconnect delay constructs in SDF.	xelab
`-nospecify`	Ignore Verilog path delays and timing checks.	xelab

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

Command Option	Description	Used by Command
`-O arg`	Enable or disable optimizations. -O0 = Disable optimizations -O1 = Enable basic optimizations -O2 = Enable most commonly desired optimizations (Default) -O3 = Enable advanced optimizations ***Note:*** A lower value speeds compilation at expense of slower simulation: a higher value slows compilation but simulation runs faster.	xelab
`-Odisable_acceleration arg`	Turn off acceleration for the specified HDL package. Choices are: `all`, `math_real`, `math_complex`, `numeric_std`,, std_logic_signed, std_logic_unsigned (default: acceleration is on)	xelab
`-Odisable_process_opt`	Turn off the process-level optimization (default on)	xelab
`-Oenable_cdfg` `-Odisable_cdfg`	Turn on (enable) or off (disable) the building of the control+data flow graph (default: on)	xelab
`-Oenable_unused_removal` `-Odisable_unused_removal`	Turn on (enable or off (disable) the optimization to remove unused signals and statements (default: on)	xelab
`-override_timeunit`	Override timeunit for all Verilog modules, with the specified time unit in `-timescale` option.	xelab
`-override_timeprecision`	Override time precision for all Verilog modules, with the specified time precision in -timescale option.	xelab
`-pulse_e arg`	Path pulse error limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).	xelab
`-pulse_r arg`	Path pulse reject limit as percentage of path delay. Allowed values are 0 to 100 (Default is 100).	xelab
`-pulse_int_e arg`	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).	xelab
`-pulse_int_r arg`	Interconnect pulse reject limit as percentage of delay. Allowed values are 0 to 100 (Default is 100).	xelab
`-pulse_e_style arg`	Specify when error about pulse being shorter than module path delay should be handled. Choices are: `ondetect`: report error right when violation is detected `onevent`: report error after the module path delay. Default: `onevent`	xelab

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

Command Option	Description	Used by Command
`-prj <filename>`	Specify XSim project file containing one or more entries of `vhdl|verilog <work lib> <HDL file name>`.	xelab xvhdl xvlog
`-r [-run]`	Run the generated executable snapshot in command-line interactive mode.	xelab
`-rangecheck`	Enable run time value range check for VHDL.	xelab
`-R [-runall`	Run the generated executable snapshot until the end of simulation.	xelab
`-relax`	Relax strict language rules.	xelab xvhdl xvlog
`-s [-snapshot] arg`	Specify the name of output simulation snapshot. Default is `<worklib>.<unit>`; for example: `work.top`. Additional unit names are concatenated using #; for example: `work.t1#work.t2`.	xelab
`-sdfnowarn`	Do not emit SDF warnings.	xelab
`-sdfnoerror`	Treat errors found in SDF file as warning.	xelab
`-sdfmin arg`	`<root=file>` SDF annotate `<file>` at `<root>` with minimum delay.	xelab
`-sdftyp arg`	`<root=file>` SDF annotate `<file>` at `<root>` with typical delay.	xelab
`-sdfmax arg`	`<root=file>` SDF annotate `<file>` at `<root>` with maximum delay.	xelab
`-sdfroot <root_path>`	Default design hierarchy at which SDF annotation is applied.	xelab
`-sourcelibdir <sourcelib_dirname>`	Directory for Verilog source files of uncompiled modules. Use `-sourcelibdir <sourcelib_dirname>` for each source directory.	xelab
`-sourcelibext <file_extension>`	File extension for Verilog source files of uncompiled modules. Use `-sourcelibext <file extension>` for source file extension	xelab
`-sourcelibfile <filename>`	File name of a Verilog source file with uncompiled modules. Use `-sourcelibfile <filename>`.	xelab
`-stat`	Print tool CPU and memory usages, and design statistics.	xelab
`-sv`	Compile input files in System Verilog mode.	xvlog
`-timescale`	Specify default timescale for Verilog modules. Default: `1ns/1ps`.	xelab

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

Command Option	Description	Used by Command
`-timeprecision_vhdl arg`	Specify time precision for vhdl designs. Default: `1ps`.	xelab
`-trace_limit arg (=65536)`	Maximum allowable signal width that can be traced for waveform viewing	xelab
`-transport_int_delays`	Use transport model for interconnect delays.	xelab
`-typdelay`	Compile Verilog design units with typical delays (Default).	xelab
`-v [verbose] [0|1|2]`	Specify verbosity level for printing messages. Default = 0.	xelab
`-version`	Print the compiler version to screen.	xelab xsim xvhdl xvlog
`-work <library_name> [=<library_dir>]`	Specify the work library. The format of the argument is `<name>[=<dir>]` where: • `<name>` is the logical name of the library. • `<library_dir>` is an optional physical directory of the library.	xvhdl xvlog

Example of Running Vivado Simulator in Standalone Mode

When running the Vivado simulator in standalone mode, you can execute commands to:

- Analyze the design file
- Elaborate the design and create a snapshot
- Open the Vivado simulator GUI and wave configuration file(s) and run simulation

Step1: Analyzing the Design File

To begin, analyze your HDL source files by type, as shown in the table below. Each command can take multiple files.

Table 5-3: **File Types and Associated Commands for Design File Analysis**

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

Step2: Elaborating and Creating a Snapshot

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

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

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

Step 3: Running Simulation

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

To run the Vivado simulator GUI, use the following command:

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

To open the wave config file:

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

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

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

Project File (.prj) Syntax

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

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

Where:

`<work_library>`: Is the library into which the HDL files on the given line are to be compiled.

`<file_names>`: Are Verilog source files. You can specify multiple Verilog files per line.

`<file_name>`: Is a VHDL source file; specify only one VHDL file per line.

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

Predefined Macros

XILINX_SIMULATOR is a Verilog predefined-macro. The value of this macro is 1. Predefined macros perform tool-specific functions, or identify which tool to use in a design flow. The following is an example usage:

```
`ifdef VCS
     // VCS specific code
`endif
`ifdef INCA
     // NCSIM specific code
`endif
`ifdef MODEL_TECH
     // MODELSIM specific code
`endif
`ifdef XILINX_ISIM
     // ISE Simulator (ISim) specific code
`endif
`ifdef XILINX_SIMULATOR
     // Vivado Simulator (XSim) specific code
`endif
```

Library Mapping File (xsim.ini)

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

The compilers attempt to read xsim.ini from these locations in the following order:

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

The xsim.ini file has the following syntax:

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

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

```
std=<Vivado_Install_Area>/xsim/vhdl/std
ieee=<Vivado_Install_Area>/xsim/vhdl/ieee
vl=<Vivado_Install_Area>/xsim/vhdl/vl
synopsys=<Vivado_Install_Area>/xsim/vhdl/synopsys
unisim=<Vivado_Install_Area>/xsim/vhdl/unisim
unimacro=<Vivado_Install_Area>/xsim/vhdl/unimacro
unifast=<Vivado_Install_Area>/xsim/vhdl/unifast
simprims_ver=<Vivado_Install_Area>/xsim/verilog/simprims_ver
unisims_ver=<Vivado_Install_Area>/xsim/verilog/unisims_ver
unimacro_ver=<Vivado_Install_Area>/xsim/verilog/unimacro_ver
unifast_ver=<Vivado_Install_Area>/xsim/verilog/unifast_ver
secureip=<Vivado_Install_Area>/xsim/verilog/secureip
work=./work
```

The `xsim.ini` file has the following features and limitations:

- There must be no more than one library path per line inside the `xsim.ini` file.
- If the directory corresponding to the physical path does not exist, `xvhd` or `xvlog` creates it when the compiler first tries to write to that path.
- You can describe the physical path in terms of environment variables. The environment variable must start with the `$` character.
- The default physical directory for a logical library is `xsim/<language>/<logical_library_name>`, for example, a logical library name of:

  ```
  <Vivado_Install_Area>/xsim/vhdl/unisim
  ```

- File comments must start with --

Running Simulation Modes

You can run any mode of simulation from the command line. The following subsections illustrate and describe the simulation modes when run from the command line.

Behavioral Simulation

Figure 5-1 illustrates the behavioral simulation process:

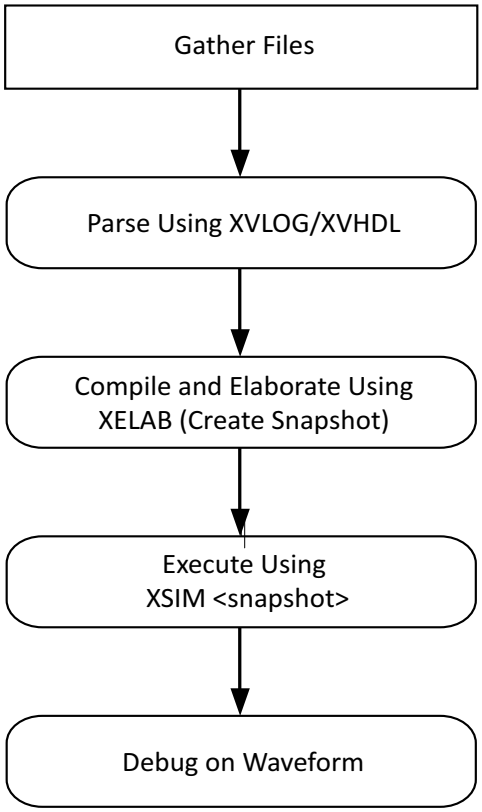


Figure 5-1: **Behavioral Simulation Process**

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

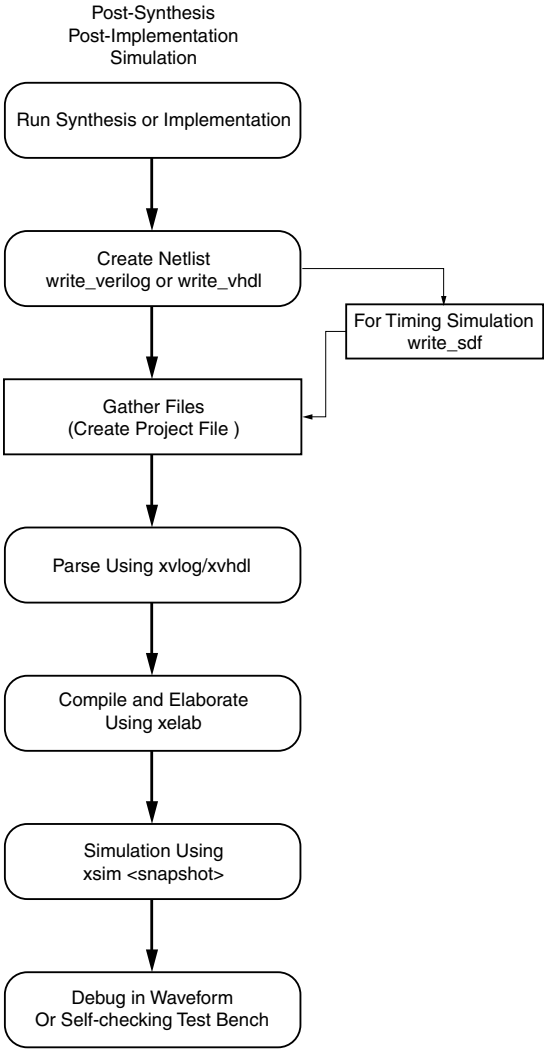


Figure 5-2: **Post-Synthesis and Post-Implementation Simulation**

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

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

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

Using Tcl Commands and Scripts

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

Using a -tclbatch File

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

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

```
run 20ns
current_time
quit
```

Then launch simulation as follows:

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

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

Launching Vivado Simulator from the Tcl Console

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

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

Tcl Property Commands

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

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

report_property Command

- **Tcl Command:**

```
report_property  [-all] [-class <arg>] [-return_string] [-file <arg>] [-append]
[-regexp] [-quiet] [-verbose] [<object>] [<pattern>]
```

Related commands are as follows:

- create_property
- get_cells
- list_property Command
- list_property_value
- reset_property
- set_property
- get_property

list_property Command

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

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

The command returns a list of property names.

get_property

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

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

The command returns the property value.

Example:

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

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

Property Command Options

Table 5-4 lists the Property options.

Table 5-4: **Property Options**

Property	Type	Read-only	Visible	Value
CLASS	string	TRUE	TRUE	fileset
FILESET_TYPE	string	TRUE	TRUE	Simulation sources
GENERIC	string	FALSE	TRUE	
INCLUDE_DIRS	string	FALSE	TRUE	
NAME	string	FALSE	TRUE	sim_1
NEEDS_REFRESH	bool	TRUE	TRUE	0
NL.CELL	string	FALSE	TRUE	
NL.INCL_UNISIM_MODELS	bool	FALSE	TRUE	0
NL.MODE	string	FALSE	FALSE	timesim
NL.NOLIB	bool	FALSE	FALSE	1

Table 5-4: **Property Options *(Cont'd)***

Property	Type	Read-only	Visible	Value
NL.PROCESS_CORNER	string	FALSE	TRUE	slow
NL.RENAME_TOP	string	FALSE	TRUE	
NL.SDF_ANNO	bool	FALSE	TRUE	1
NL.WRITE_ALL_OVERRIDES	bool	FALSE	TRUE	0
RUNTIME	string	FALSE	TRUE	1000 ns
SIM_MODE	string	FALSE	FALSE	post-implementation
SKIP_COMPILATION	bool	FALSE	FALSE	0
SKIP_SIMULATION	bool	FALSE	FALSE	0
SOURCE_SET	string	FALSE	TRUE	sources_1
TOP	string	FALSE	TRUE	bft_tb
TOP_ARCH	string	FALSE	FALSE	
TOP_AUTO_SET	bool	FALSE	FALSE	1
TOP_FILE	string	FALSE	FALSE	Example: C:/scripts/project_17/project_17.srcs/sim_1/imports/sources/bft_tb.v
TOP_LIB	string	FALSE	FALSE	work
UNIT_UNDER_TEST	string	FALSE	TRUE	
VERILOG_DEFINE	string	FALSE	TRUE	
VERILOG_DIR	string	FALSE	FALSE	
VERILOG_UPPERCASE	bool	FALS	TRUE	0
VHDL_GENERIC	string	FALSE	FALSE	
XELAB.DEBUG_LEVEL	enum	FALSE	TRUE	typical
XELAB.DLL	bool	FALSE	TRUE	0
XELAB.LOAD_GLBL	bool	FALSE	TRUE	1
XELAB.MORE_OPTIONS	string	FALSE	TRUE	

Table 5-4: **Property Options *(Cont'd)***

Property	Type	Read-only	Visible	Value
XELAB.MT_LEVEL	enum	FALSE	TRUE	auto
XELAB.RANGECHECK	bool	FALSE	TRUE	0
XELAB.RELAX	bool	FALSE	TRUE	0
XELAB.SDF_DELAY	enum	FALSE	TRUE	sdfmax
XELAB.SNAPSHOT	string	FALSE	TRUE	
XELAB.UNIFAST	bool	FALSE	TRUE	0
XSIM.MORE_OPTIONS	string	FALSE	TRUE	
XSIM.SAIF	string	FALSE	TRUE	
XSIM.TCLBATCH	string	FALSE	TRUE	
XSIM.VIEW	string	FALSE	TRUE	
XSIM.WDB	string	FALSE	TRUE	

Chapter 6

Debugging a Design with Vivado Simulator

Introduction

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

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

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

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

See Adding Conditions and Outputting Diagnostic Messages for Debugging, page 121 for more information.

Debugging at the Source Level

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

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

You can use the Step, Run All, or Run For command to advance the simulation after a stop. For more information, see the section, Using Breakpoints, below.

Stepping Through a Simulation

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

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

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

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

     The HDL associated with the top design unit opens as a new view in the waveform window.
   - From the start (0 ns), restart the simulation. Use the **Restart** command to reset time to the beginning of the test bench. See Chapter 3, Using the Vivado Simulator from the Vivado IDE.
2. Select **Window > Tile Horizontally (or Window > Tile Vertically**) to simultaneously see the waveform and the HDL code.
3. Repeat the **Step** action until debugging is complete.

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

Using Breakpoints

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

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

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

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

To set a breakpoint In the GUI:

1. To set a breakpoint, run a simulation.
2. Go to your source file and click the triangle to the left of the source line of interest. A red dot confirms the breakpoint is set correctly.

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

To set a breakpoint in the Tcl Console:

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

   This command adds a breakpoint at `<line_number>` of `<file_name>`. See the Vivado help or the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 5] for command usage.

To debug a design using breakpoints:

1. Open the HDL source file.
2. Set breakpoints on executable lines in the HDL source file.
3. Repeat steps 1 and 2 until all breakpoints are set.
4. Run the simulation, using a Run option:
   - To run from the beginning, use the **Run > Restart** command.
   - Use the **Run > Run All or Run > Run for Specified Time** command.

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

   The HDL source file displays, 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.

To delete all breakpoints:

- Type the Tcl command `remove_bp`.

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

- Type the Tcl command `report_bp <list>`.

Adding Conditions and Outputting Diagnostic Messages for Debugging

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

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

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

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

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

Generating (forcing) Stimulus

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

```
add_force
```

Command syntax:

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

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

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

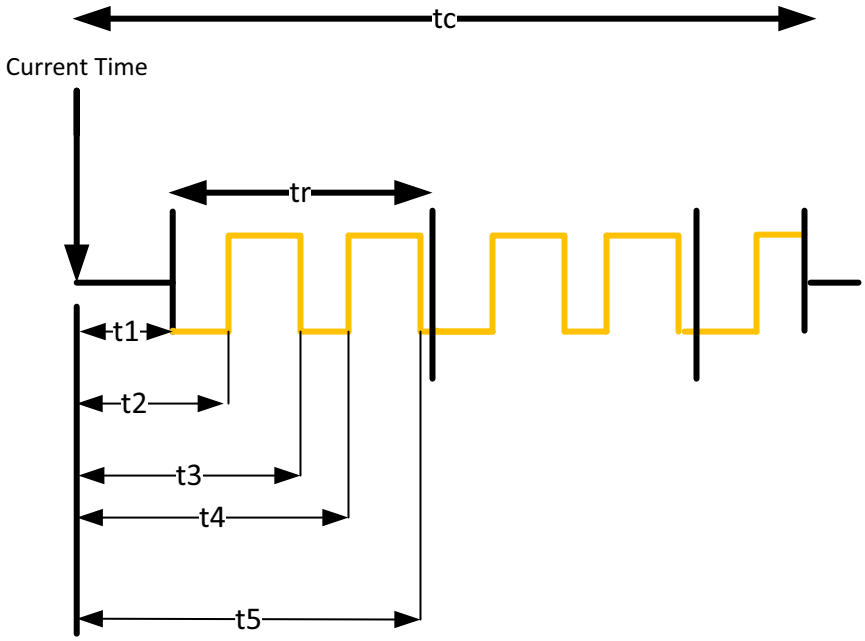


Figure 6-1: **Illustration of -add_force Functionality**

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

```
add_force -help
```

Adding Force in Verilog Code

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

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

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

Command Examples

You can invoke the following commands to observe the effect of `add_force`:

```
xelab -vlog tmp.v -debug all
xsim work.top
```

At the command prompt, type:

```
add_force /top/I1/sel 1
run 10
add_force /top/I1/sel 0
run all
```

Using add_force with remove_force

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

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

reg [4:0] r1;

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

assign out1 = r1;
endmodule

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

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

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

initial
begin
    updown = 1; clk = 0;
end

initial
    #500 $finish;

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

Running add_force and remove_force in a Tcl Batch File

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

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

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

```
add_force clk {0 1} {1 2} -repeat_every 3 -cancel_after 500
add_force updown {0 10} {1 20} -repeat_every 30
run 100
```

   Observe that the value of `out1` increments as well as decrements in the Waveform window.

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

4. In the Tcl Console, type:

```
remove_force force2
```

   Observe that the value of signal `updown` is now the default value present in design.

5. In the Tcl Console, type:

```
run 100
```

   Observe that only the value of `out1` increments.

6. In the Tcl Console, type:

```
remove_force force1
run 100
```

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

Power Analysis Using Vivado Simulator

The SAIF dumping is optimized for Xilinx® Power tools and dumps the following HDL types:

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

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

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

Generating SAIF Dumping

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

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

   ```
   xelab -debug typical top -s mysim
   ```

2. Use the following Tcl command to start SAIF dumping:

   ```
   open_saif <saif_file_name>
   ```

3. Add the scopes and signals to be generated by typing one of the following Tcl commands:

   ```
   log_saif [get_objects]
   ```

   To recursively log all instances, the Tcl command:

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

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

   ```
   read_saif
   ```

5. Run the simulation (use any of the run commands).

6. Close the SAIF file, by typing the Tcl command:

```
close_saif
```

Example SAIF Tcl Commands

To log SAIF for:

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

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

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

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

Dumping SAIF using a Tcl Simulation Batch File

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

Using the report_drivers Tcl Command

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

```
report_drivers <hdl_object>
```

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

Using the Value Change Dump Feature

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

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

Table 6-1: **Tcl Commands for VCD**

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

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

```
<command> -help
```

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

Simulating with QuestaSim/ModelSim

Introduction

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

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

Simulating Xilinx Designs using QuestaSim/ModelSim

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

The *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 1] describes the use of the Vivado IDE.

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

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

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

Setting QuestaSim/ModelSim for Use in the Vivado IDE

The following subsections describe two vital steps required for invoking Questasim/Modelsim from the Vivado IDE: pointing to the QuestaSim/ModelSim executable and compiling the simulation libraries.

Pointing to the QuestaSim/ModelSim Simulator Install Location

To define the ModelSim/QuestaSim installation path:

1. Select **Tools > Options > General**
2. In the Vivado Options, General dialog box, locate the **3rd Party Tools** field, shown in the figure below, and browse to the QuestaSim or ModelSim installation path.

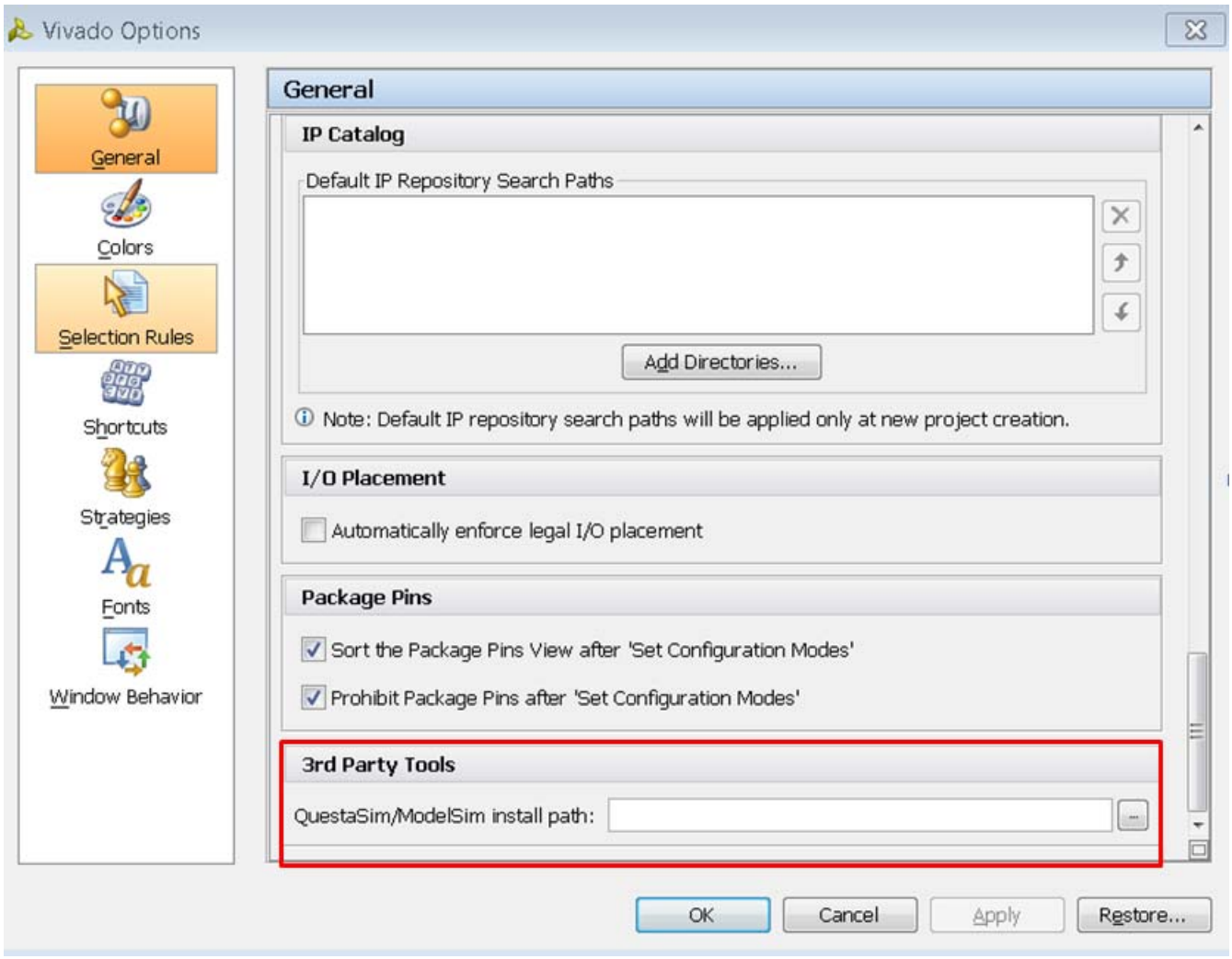


Figure 7-1: **Vivado Options, General Dialog Box Showing QuestaSim/ModelSim Install Path Field**

Compiling Simulation Libraries for QuestaSim/ModelSim

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

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

Example: compiling the libraries for QuestaSim to a specific directory:

```
compile_simlib -simulator questa -directory <library_output_directory>
```

Other switches, such as `-directory <library_output_directory>` and `-simulator_exec_path <questa_install_location>` are derived from the simulation settings in your local environment.

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 6] for more information about the Tcl command, or type `compile_simlib -h` in the Tcl Console.

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

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

After the libraries are compiled, the simulator references these compiled libraries using the `modelsim.ini` file. The `compile_simlib` command copies the `modelsim.ini` file to the `<library_output_directory>`.

IMPORTANT: *If you have IP generated from the Vivado IP catalog (`IP.xci`), you do not need the `XilinxCoreLib` library for simulation. Simulation models for Vivado Design Suite IP cores are delivered as an output product when the IP is generated. Consequently, the they are not included in the pre-compiled `XilinxCoreLib` libraries created using `compile_simlib`. The `XilinxCorelib` library is available for simulating legacy ISE® Design Suite CORE Generator™ IP used in the Vivado Design Suite.*

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

Running QuestaSim/ModelSim Simulation

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

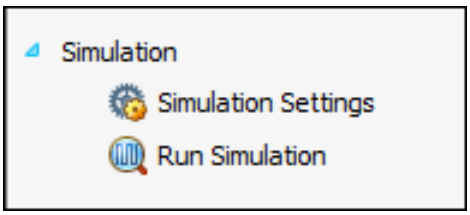


Figure 7-2: **Flow Navigator Simulation Options**

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

QuestaSim/ModelSim Simulation Options

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

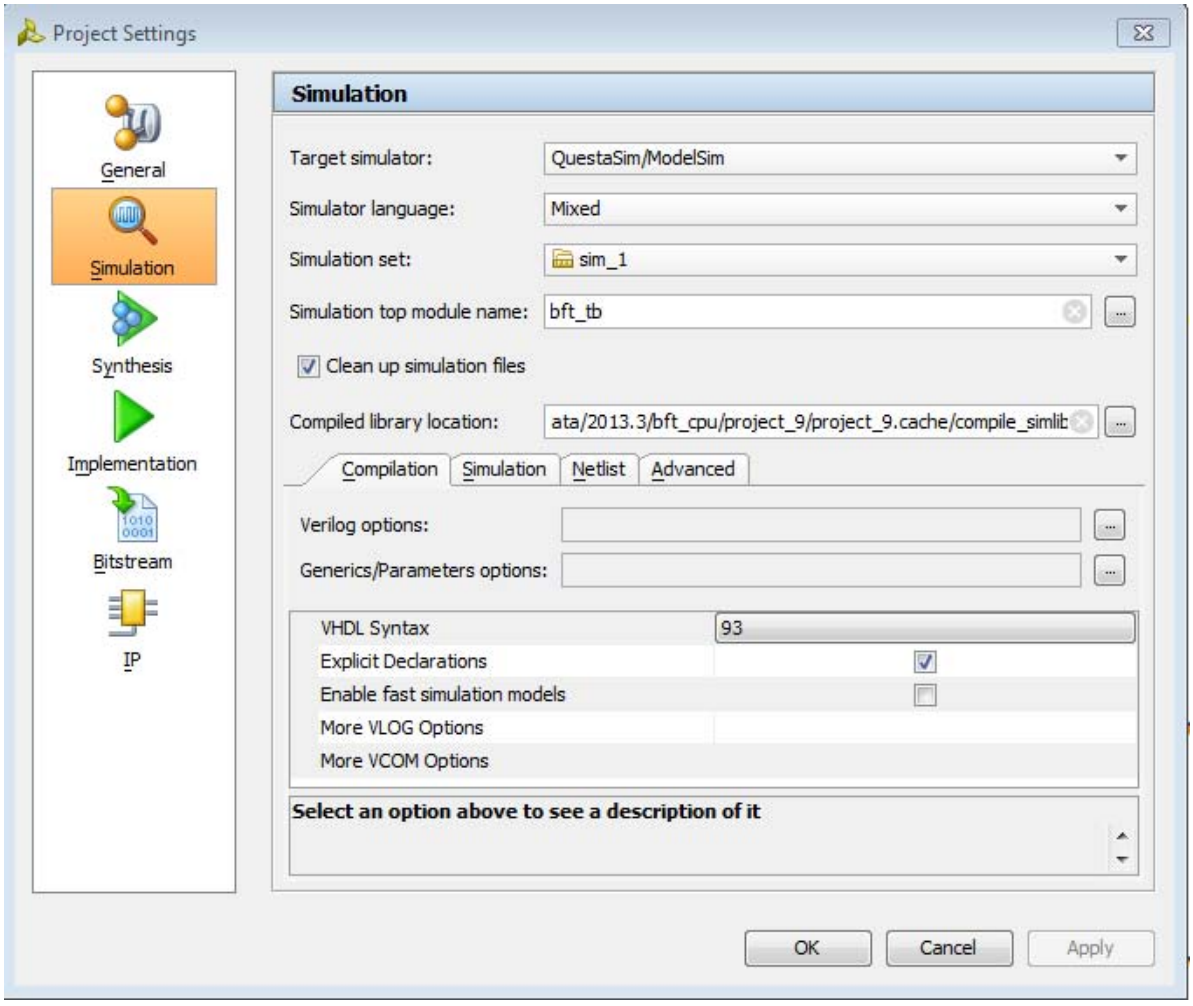


Figure 7-3: **Compilation Simulation Options**

Option Descriptions:

- Target simulator: Displays the selected simulation tool.
- Simulator language: Determines the language in which Xilinx generated output products for simulation. For more information refer to Understanding the Simulator Language Option in Chapter 2.
- Simulation set: Displays the selected simulation set.

- Simulation top module name: Points to the top-level simulation module.
- Clean up simulation files: Removes unnecessary simulation files.
- Compiled library location: Points to the location of the compiled libraries. Useful if you have previously compiled the libraries using `compile_simlib`.

Selecting Compilation Options

The available compilation options are:

- Path selection options for Verilog and Generics/Parameters.
- VHDL Syntax: Lets you select the VHDL language version. The options are 93, 87, 2002, and 2008.
- Explicit Declarations: This check box turns explicit declarations on and off. The check box is on by default.
- Enable fast simulation options: Enables or disables fast simulation models.
- More VLOG Options: Let you enter additional VLOG options.
- More VCOM Options: Let you enter additional VCOM options.

Selecting Simulation Options

The Simulation view is shown in the figure below:

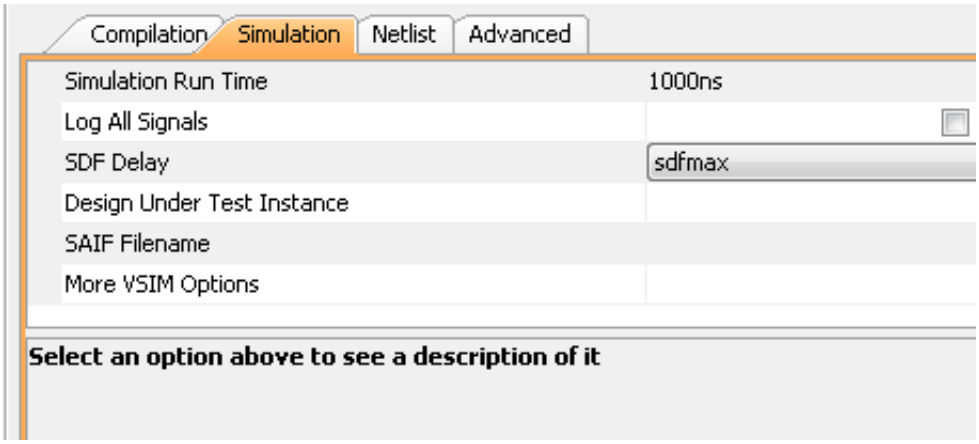


Figure 7-4: **Simulation Options**

The available options are:

- Simulation Run Time: Shows the default simulation run time.
- Log All Signals: Instructs the software to generate a log of all simulation signals.

  Note: If you log all signals, this slows simulation and increases the size of the waveform database.

- SDF Delay: Options in the drop-down are: `sdfmax` (default) and `sdfmin`. See Annotating an SDF File in QuestaSim/ModelSim, page 141, for more information.
- Design Under Test Instance: Lets you enter a name for the design.
- SAIF Filename: Lets you enter the name of the SAIF file to use. See Dumping SAIF in QuestaSim/ModelSim, page 140 for more information.
- More VSIM Options: Lets you enter more QuestaSim/ModelSim options.

Selecting Netlist Options

The available netlist options are shown in the figure below.

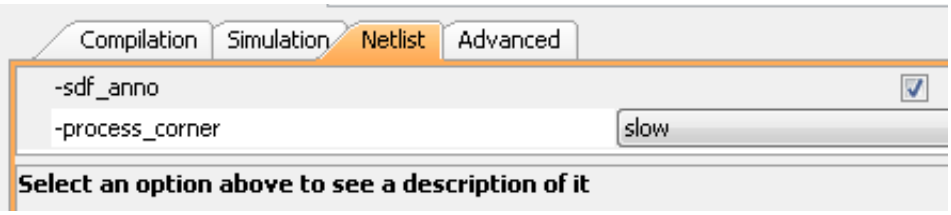


Figure 7-5: **Netlist Options**

The options are:

- `-sdf_anno`: a check box is available to select the command. This option is enabled by default.
- `-process_corner`: You can specify the `-process_corner` as `fast` or `slow`.

See Annotating an SDF File in QuestaSim/ModelSim, page 141 for more information about SDF.

Selecting Advanced Simulation Options

The Advanced view is shown in the figure below.

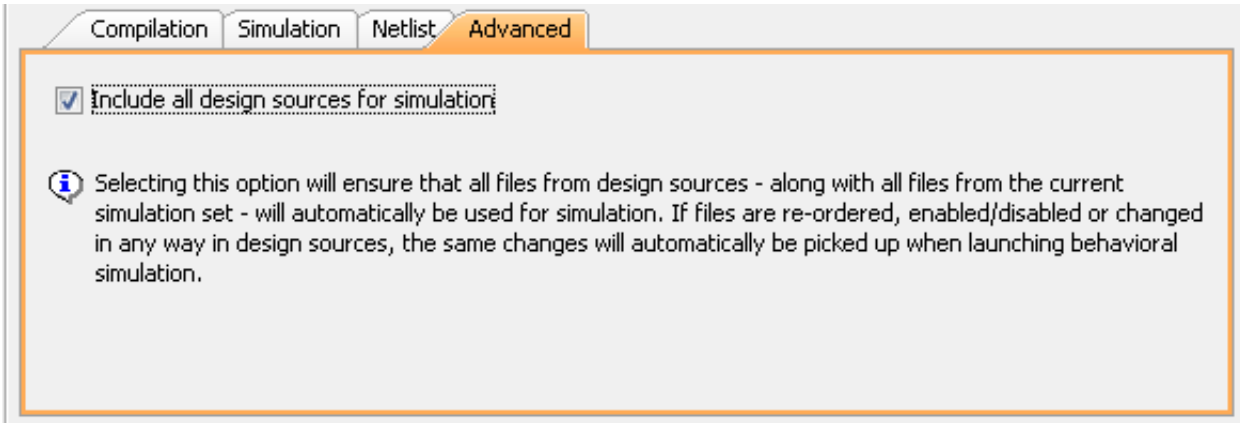


Figure 7-6: **Advanced Option**

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

CAUTION! *It is best not to uncheck this option!*

Adding or Creating Simulation Source Files

To add simulation sources to a project:

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

   The Add Sources wizard opens.

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

   The Add or Create Simulation Sources dialog box options are:

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

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

   - Add Files: Invokes a file browser so you can select simulation source files to add to the project.
   - Add Directories: Invokes directory browser to add all simulation source files from the selected directories. Files in the specified directory with valid source file extensions are added to the project.

- Create File: Invokes the Create Source File dialog box where you can create new simulation source files.

Buttons on the side of the dialog box let you do the following:

- Remove: Removes the selected source files from the list of files to be added.
- Move Selected File Up: Moves the file up in the list order.
- Move Selected File Down: Moves the file down in the list order.

Check boxes in the wizard provide the following options:

- Scan and add RTL include files into project: Scans the added RTL file and adds any referenced include files.
- Copy sources into project: Copies the original source files into the project and uses the local copied version of the file in the project.
- If you selected to add directories of source files using the Add Directories command, the directory structure is maintained when the files are copied locally into the project.
- Add sources from subdirectories: Adds source files from the subdirectories of directories specified in the Add Directories option.
- Include all design sources for simulation: Includes all the design sources for simulation.

Working with Simulation Sets

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

The simulation set lets you define different sources for different stages of the design.

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

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

To edit a simulation set:

1. In the Sources window popup menu, select **Simulation Sources > Edit Simulation Sets**, as shown in the figure below.

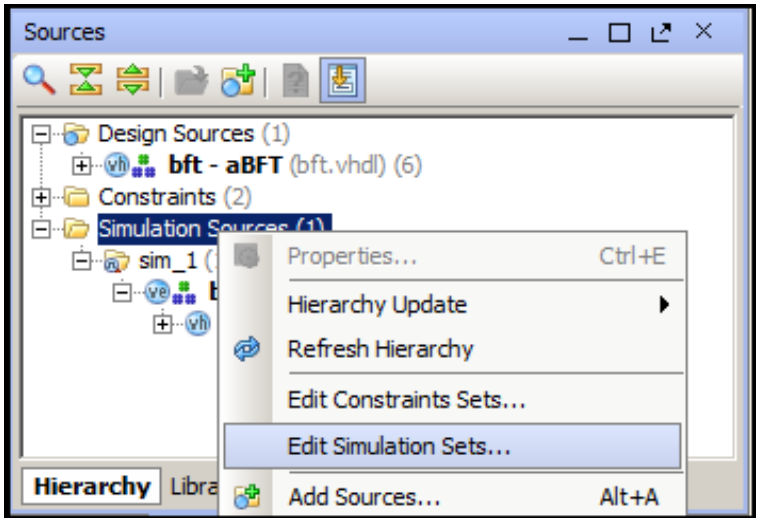


Figure 7-7: **Edit Simulation Sets Option**

The Add or Create Simulation Sources wizard opens.

2. From the Add or Create Simulation Sources wizard, select **Add Files**. This adds the sources associated with the project to the newly-created simulation set.
3. Add additional files as needed.

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

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

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

IMPORTANT: *Confirm the library compilation order before running a third-party simulation.*

Running QuestaSim/ModelSim Simulation

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

When you run simulation prior to synthesizing the design, the QuestaSim/ModelSim simulator runs a behavioral simulation. Following each successful design step (synthesis and implementation), the option to run a functional or timing simulation becomes

available. You can initiate a simulation run from the Flow Navigator or by typing in a Tcl command.

From the Flow Navigator, click **Run Simulation**, as shown in the figure below.

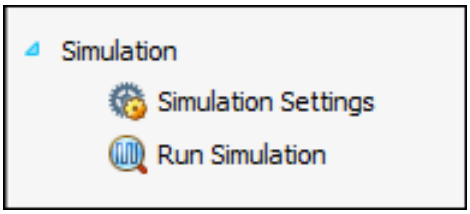


Figure 7-8: **Flow Navigator Simulation Options**

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

TIP: *This command provides a -scripts_only option that can be used to write a `DO` file. Use the DO file to run QuestaSim/ModelSim simulations outside of the IDE.*

Using a Custom DO File during a ModelSim Run

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

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

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

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

Running Post-Synthesis Functional Simulation in QuestaSim/ModelSim

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

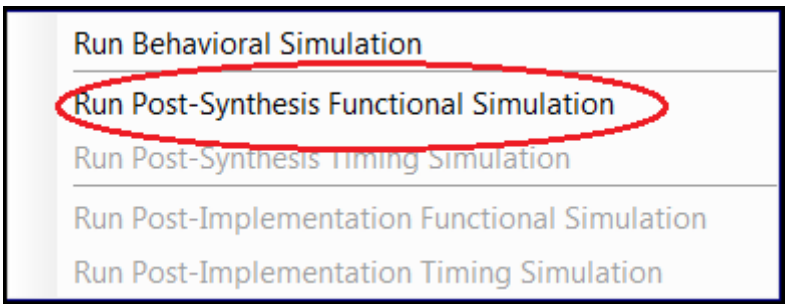


Figure 7-9: **Run Post-Synthesis Functional Simulation**

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

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

Running Post-Synthesis Timing Simulation in QuestaSim/ModelSim

When synthesis is run successfully, the **Run Simulation > Post-Synthesis Timing Simulation** option becomes available, as shown in the figure below.

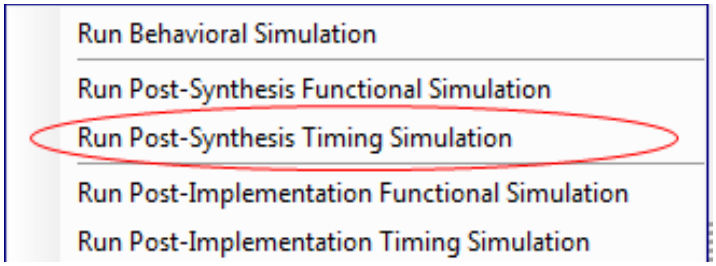


Figure 7-10: **Run Post-Synthesis Timing Simulation**

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

Using ModelSim in Command Line Mode

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

Running RTL/Behavioral Simulation in QuestaSim/ModelSim

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

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

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

Running Timing Simulation in QuestaSim/ModelSim

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

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

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

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

Running Netlist Simulation in QuestaSim/ModelSim

The netlist simulation process involved the same steps as described in the section Running Timing Simulation in QuestaSim/ModelSim, page 139.

1. Compile simulation libraries.
2. Gather files for simulation:
   a. You can reuse the RTL simulation test bench for the majority of designs.
   b. Generate the simulation netlist (as described in the section Generating a Netlist, page 29).
      - If you are using Verilog, compile `glbl.v`. See Using Global Reset and 3-State in Chapter 2.
      - If you have `SECUREIP` in your design, use the precompiled libraries and point to the library using the `-L` switch in `VSIM`. For example:

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

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

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

Dumping SAIF in QuestaSim/ModelSim

See Dumping the Switching Activity Interchange Format File for Power Analysis, page 40 for more information about Switching Activity Interchange Format (SAIF).

QuestaSim/ModelSim uses explicit power commands to dump an SAIF file, as follows:

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

```
power add <hdl_objects>
```

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

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

For more detailed usage or information about each commands, see the *ModelSim User Guide*.

Example do File

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

Annotating an SDF File in QuestaSim/ModelSim

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

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

To run a setup check, create an SDF with `–process` corner slow, and use the max column from the SDF, then specify:

```
–sdfmax
```

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

```
-sdfmin
```

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

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

Using Verilog UNIFAST Library with QuestaSim/ModelSim

There are two methods of simulating with the UNIFAST models.

Method 1

Recommended method for simulating with all the UNIFAST models.

Select the **Simulation Settings > Enable fast simulation models** check box to enable UNIFAST support in a Vivado project environment for ModelSim. See UNIFAST Library, page 19 for more information.

Method 2

Recommended for more advanced who want to determine which modules to use with the `UNIFAST` models. See Method 2: Using specific UNIFAST modules, page 23 for the description of this method.

Simulating with Cadence Incisive Enterprise Simulator (IES)

Introduction

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

IMPORTANT: Xilinx® recommends using supported versions of third-party simulators. For more information on supported simulators and operating systems, see the "Compatible Third-Party Tools" table in the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973) [Ref 1].

Compiling Simulation Libraries for IES

Before you begin simulation, you must run the `compile_simlib` Tcl command from the Vivado TCL console to compile the Xilinx simulation Libraries for the target simulator.

IMPORTANT: *With any change with the version of Vivado or Simulator, libraries must be recompiled.*

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

Tcl command: `compile_simlib -simulator ies`

Other switches, such as `-directory <library_output_directory>` and `-simulator_exec_path <ies_install_location>` are derived from the simulation settings.

See the *Vivado Design Suite Tcl Command Reference* (UG835) [Ref 6] for more information about the Tcl command, or type `compile_simlib -h` in the Tcl Console.

After the libraries are compiled, the simulator references the compiled libraries using the `CDS.lib` file. The `compile_simlib` command copies the `cds.lib` and `HDL.var` file to the `<library_output_directory>`.

IMPORTANT: *The `compile_simlib` option compiles only Xilinx primitives and legacy ISE® Design Suite Xilinx cores. Simulation models of Vivado Design Suite IP cores are delivered as an output product when the IP is generated. Consequently, they are not included in the pre-compiled libraries created with `compile_simlib`.*

Note: The `cds.lib` file created by `compile_simlib` specifies reference library paths for Xilinx primitives and includes a reference to the master CDS.lib. The master CDS.lib has mapping to standard VHDL libraries. If the correct `cds.lib` file is not found in the path, you cannot run simulation.

IMPORTANT: *If you have a simulator language preference, refer to Understanding the Simulator Language Option in Chapter 2 before proceeding.*

Running Behavioral/RTL Simulation Using IES

IES simulator is not integrated into the Vivado IDE, but a Tcl command called `export_simulation` is provided. The `export_simulation` command writes a complete behavioral/RTL simulation script that can be used as a standalone or that can be plugged into your custom simulation environment.

The following are the steps for simulating a design with IES:

1. Open a Vivado Design Suite project.
2. If you have not done so, compile the simulation libraries.
3. From the Tcl Console, run the `export_simulation` command.

   The `export_simulation` command generates a script that you can use on the Tcl Console. It also creates `DO` file.

4. Execute the script.

For more information see:

- Using the export_simulation Command in Chapter 2.
- *Vivado Design Suite Tcl Command Reference* (UG835) [Ref 6]

IMPORTANT: *Xilinx recommends you always use the `-lib_map_path` switch with the `export_simulation` TCL command.*

Understanding the `export_simulation` Script

The `export_simulation` creates the following files:

- A script file to run simulation.

  Optionally, you can specify the name of the shell script using the following Tcl command:

  ```
  export_simulation -script_name <arg>
  ```

  Where `-script_name` lets you specify the name of the shell script.

  If you do not specify this option, the file name is generated based on the object type selected with `-of_objects` switch, with the following form:

  - `<simulation_top_name>_sim_simulator.sh`
  - `<ip_name>_sim_simulator.sh`

- A DO file for post-processing operations (such as `run` and `quit`).
- A data file that copies files (such as `MIF`, `COE`, `DAT`) required for running the simulation.

IMPORTANT: *If you move the script generated by `export_simulation` to a different directory location, makes sure you copy all the `export_simulation` generated files as well (for example, a DAT file). Failing to move the file with the script causes simulation failures.*

The following subsections list the major sections of the script.

Script Setup

The Setup section does the following:

- Cleans up any stale libraries that might have been created by a previous simulation run.
- Creates design library directories
- Creates a mapping for design libraries in the `CDS.lib` file.

Script Compilation

The Compilation section of the script saves VHDL and Verilog compile options to `ncvhdl_opts` and `ncvlog_opts` variables, respectively, as follows:

```
ncvhdl_opts="-64bit -V93 -RELAX -logfile ncvhdl.log -append_log"
ncvlog_opts="-64bit -messages -logfile ncvlog.log -append_log"
```

To make a compile option change to all files, edit the required variable.

Script Elaboration

In the Elaboration section of the script, the export_simulation command saves the following:

- Elaboration options to the `ncelab_opts` variable
- Library mappings to the `design_libs_elab` variable

Edit either of the options to fit your design requirements.

Script Simulation

In the Simulation section, the script save all simulation options to the `ncsim_opts` variable

You can also add post-processing options by editing the `DO` file.

The script generated by `export_simulation` provides three options:

- Usage:

  ```
  tb_sim_ies.sh -help
  ```

  Print help.

- Usage:

  ```
  tb_sim_ies.sh -noclean_files
  ```

  Do not remove simulator generated files from the previous run.

- Usage:

  ```
  tb_sim_ies.sh -reset_run
  ```

  Recreate simulator setup files and library mappings for a clean run. The generated files from the previous run are removed automatically.

IES Simulation Use models

The following subsections describe use models for simulation in IES. There are four major use models:

- Use Model 1: You have generated an IP from the Vivado IP catalog, and you would like to collect the files (`RTL, DAT, MIF, COE` files) and add them to your simulation environment.
- Use Model 2: You have a complete RTL project, and you would like to simulate the project using IES.

- Use Model 3: You have a project that has RTL and block design.The project also uses an ELF file for a processor system. You would like to simulate the entire design in IES.
- Use Model 4: You have a AXI BFM or a Zynq® device BFM in your project.

Use Model 1: Generating Xilinx IP Simulation Files

In this use model, the `accum_0.xci` file is the IP you generated from the Vivado IP catalog. Now you must collect all the files needed for simulation. The easiest way to do this is to run `export_simulation`.

The following command generates an `accum_0_sim_ies.sh` script file for the `accum_0` IP in the specified output directory for the IES simulator:

```
export_simulation -lib_map_path "/sim_libs/ies/lin64/lib" \
-of_objects [get_files accum_0.xci] -simulator ies \
-directory test_sim
```

Now that you have `accum_0_sim_ies.sh`, you can either incorporate the script into your simulation environment or parse the compile commands and add all the RTL files directly to your script. Make sure to copy all data files generated by `export_simulation`.

Use Model 2: Simulating an RTL Design

The following Tcl command generates a `top_tb_sim_ies.sh` script file in the `./test_sim` directory for a project with simulation top set to `top_tb`. The command also copies data files (such as `MIF`, `COE`, `DAT`) to the `./test_sim` directory:

```
export_simulation -simulator ies -lib_map_path "/sim_libs/ies/lin64/lib" \
-directory ./test_sim
```

Use Model 3: Simulating a Vivado IP Integrator Design that Contains an ELF File

To simulate a Vivado IP integrator design in IES, use the Tcl command:

```
export_simulation -simulator ies -lib_map_path "/sim_libs/ies/lin64/lib" \
-directory ./test_sim
```

Use Model 4: Simulating Design with AXI Bus Functional Models

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

1. Set the `LD_LIBRARY_PATH` environment variable, using the following syntax:

```
setenv LD_LIBRARY_PATH $XILINX_VIVADO/lib/lnx64.o/:$LD_LIBRARY_PATH
```

2. Add the following switch to `ncsim` options file:

```
-loadvpi $XILINX_VIVADO/lib/lnx64.o/:xilinx_register_systf.xilinx_register_systf
```

Code Example:

```
simulate()
{
  ncsim_opts="-loadvpi
$XILINX_VIVADO/lib/lnx64.o/:xilinx_register_systf.xilinx_register_systf \
 -64bit -logfile tb_sim.log"
ncsim $ncsim_opts work.tb -input tb.do
}
```

Note: The `export_simulation` command always applies OS defaults. So, if you are using 64-bit Vivado tools, the scripts generated by the Tcl command will be for 64-bit simulation. If you want to use 32-bit simulation, you must edit the compile, elaborate, and simulate options. You must also point to a 32-bit location for the `LD_LIBRARY_PATH` and `-loadvpi` switch (change `lin64` to `lin`).

IMPORTANT: *Ensure that the* `-lib_map_path` *points to the directory that contains the* `cds.lib`.

Running Timing Simulation Using IES

Timing simulation includes the following steps:

1. Generating the simulation netlist (`timesim.v` generation).
2. Annotating timing information to the netlist (SDF file generation).
3. Analyzing, elaborating, and simulating the timing netlist and SDF using IES.

Generating a Timing Netlist and SDF Generation in the Vivado Design Suite

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

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

IES Timing run Simulation Tcl commands:

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

Dumping SAIF for Power Analysis in IES

IES provides power commands to generate SAIF with specific requirements.

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

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

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

   ```
   dumpsaif -end
   ```

For more detailed usage or information on IES commands, see the IES documentation.

Simulating with Synopsys VCS

Introduction

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

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

Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)

Compiling Simulation Libraries for VCS

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

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

Tcl command: `compile_simlib -simulator vcs_mx`

Other switches, such as `-directory <library_output_directory>` and `-simulator_exec_path <vcs_mx_install_location>` are derived from the simulation settings.

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 6] for more information about the Tcl command, or type `compile_simlib -h` in the Tcl Console.

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

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

IMPORTANT: *The `compile_simlib` option compiles only Xilinx primitives and legacy ISE® Design Suite Xilinx cores. Simulation models of Xilinx Vivado IP cores are delivered as an output product when the IP is generated; consequently they are not included in the pre-compiled libraries created using `compile_simlib`.*

Note: The `synopsys_sim.setup` file created by `compile_simlib` specifies the default initialization file and contains control variables that specify reference library paths, optimization, compiler, and simulator settings.

IMPORTANT: *If you have a simulator language preference, refer to Understanding the Simulator Language Option in Chapter 2 before proceeding.*

Running Behavioral/RTL Simulation using VCS

VCS simulator is not integrated in the Vivado IDE, but the Vivado Design Suite provides an `export_simulation` Tcl command. The command writes a complete Behavioral/RTL simulation script that can be used standalone or plugged into your custom simulation environment.

The script uses the Multi-Step process with precompiled libraries (see Compiling Simulation Libraries for VCS, page 149). The generated script contains simulator commands for compiling, elaborating, and simulating the design.

The command retrieves the simulation compile order of specified objects, and exports this information in a text file with the compiler commands and default options for the target simulator. The specified object can be either a simulation *fileset* or an IP. If the object is not specified, the `export_simulation` command generates the script for the simulation top.

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 6] for more information about the `compile_simlib` command.

Follow the steps below to simulate a design with VCS:

1. Open a Vivado Design Suite project.
2. If you have not done so, compile the simulation libraries.
3. From the Tcl Console, run the `export_simulation` command.

   The `export_simulation` command generates a script that you can use on the Tcl Console. It also creates `.do` file.

4. Execute the script.

For more information see Using the export_simulation Command in Chapter 2 and the *Vivado Design Suite Tcl Command Reference* (UG835) [Ref 6].

IMPORTANT: *Always use the `-lib_map_path` switch with the `export_simulation` Tcl command.*

Understanding the `export_simulation` Script

The `export_simulation` creates the following files:

- A script file to run simulation.

  Optionally, you can specify the name of the shell script using the following Tcl command:

  ```
  export_simulation -script_name <arg>
  ```

  Where `-script_name` lets you specify the name of the shell script.

  If you do not specify this option, the file name is generated based on the object type selected with `-of_objects` switch, with the following form:

  - `<simulation_top_name>_sim_simulator.sh`
  - `<ip_name>_sim_simulator.sh`

- A `DO` file for post-processing operations (such as `run` and `quit`).
- A data file that copies files (such as `MIF`, `COE`, `DAT`) required for running the simulation.

IMPORTANT: *If you move the script generated by `export_simulation` to a different directory location, makes sure you copy all the `export_simulation` generated files as well (for example, a DAT file). Failing to move the file with the script causes simulation failures.*

The following subsections list the major sections of the script.

Script Setup

The Setup section does the following:

- Cleans up any stale libraries that might have been created by a previous simulation run.
- Creates design library directories
- Creates a mapping for design libraries in the `synopsys_sim.setup` file.

Script Compilation

The Compilation section of the script saves VHDL and Verilog compile options to `vhdlan_opts` and `vlogan_opts` variables. To make a compile option change to all files, edit the required variables, respectively, as follows:

```
vhdlan_opts="-full64 -l vhdlan.log"

vlogan_opts="-full64 -l vlogan.log"
```

Script Elaboration

In the Elaboration section of the script, the `export_simulation` command saves Elaboration options to the `vcs_opts` variable. Edit options according to your design requirements.

Script Simulation

In the Simulation section, the script saves simulation options to the `<top>_simv_opts` variable

You can also add post-processing options by editing the `.do` file.

The script generated by `export_simulation` provides three options:

- Usage:

  ```
  tb_sim_vcs.sh -help
  ```

  Print help.

- Usage:

  ```
  tb_sim_vcs.sh -noclean_files
  ```

  Do not remove simulator generated files from the previous run.

- Usage:

  ```
  tb_sim_vcs.sh -reset_run
  ```

  Recreate simulator setup files and library mappings for a clean run. The generated files from the previous run are removed automatically.

VCS Simulation Use Models

The following subsections describe use models for simulation in VCS. There are four major use models:

- Use Model 1: You have generated an IP from the Vivado IP catalog, and you would like to collect the files (`RTL, DAT, MIF, COE` files) and add them to your simulation environment.
- Use Model 2: You have a complete RTL project, and you would like to simulate the project using VCS.
- Use Model 3: You have a project that has RTL and block design. The project also uses an ELF file for a processor system. You would like to simulate the entire design in VCS.
- Use Model 4: You have a AXI BFM or a Zynq® device BFM in your project.

Use Model 1: Generating Simulation Files for an IP

In this project `accum_0.xci` file is the IP you generated from the Vivado IP catalog. Now you must collect all the files needed for simulation. The easiest way to do this is to run the `export_simulation`.

The following command generates a script file `accum_0_sim_vcs.sh` for the `accum_0` IP in the specified output directory for the VCS simulator:

```
export_simulation -lib_map_path "/sim_libs/vcs/lin64/lib" \
  -of_objects [get_files accum_0.xci] -simulator vcs_mx \
  -directory test_sim
```

IMPORTANT: *The synopsys_sim.setup file should have library_scan values set to TRUE. This lets VCS search for libraries across language boundaries. Failing to change this value results in UNBOUND component errors.*

Now that you have `accum_0_sim_vcs.sh`, you can either incorporate the script into your simulation environment or parse the compile commands and add all the RTL files directly to your script. Make sure to copy all data files generated by `export_simulation`

Use Model 2: Simulating an RTL Design

The following command generates a script file named `top_tb_sim_vcs.sh` in the `./test_sim` directory for a project with simulation top set to `top_tb`. The command also copies any data files (such as `MIF`, `COE`, `DAT`) to the `./test_sim` directory:

```
export_simulation -simulator vcs_mx -lib_map_path "/sim_libs/vcs/lin64/lib" \
                  -directory ./test_sim
```

Use Model 3: Simulating an IP Integrator Design Containing an ELF file

The ELF files used in the projects are converted to MEM files, and the block memory models use the MEM files for simulations. The `export_simulation` command has the ability to generate MEM files.

To simulate an IP Integrator design with the VCS simulator, use the following Tcl command:

```
export_simulation -simulator vcs_mx -lib_map_path "/sim_libs/vcs/lin64/lib" \
                  -directory ./test_sim
```

Use Model 4: Simulating an AXI Design with a Bus Functional Model

If you have an AXI Bus Functional Model (BFM) in any of the use models add the following steps:

1. Set the `LD_LIBRARY_PATH` to point to the Xilinx security libraries in the shell, as shown:

   ```
   setenv LD_LIBRARY_PATH $XILINX_VIVADO/lib/lnx64.o/:$LD_LIBRARY_PATH
   ```

2. Modify the `.sh` script that was written by the export_simulation command by adding the following line to the `vcsopts`:

   ```
   vcs_opts="-load $XILINX_VIVADO/lib/lnx64.o/libxil_vcs.so:xilinx_register_systf
   ```

Example:

```
elaborate()
{
  vcs_opts="-load $XILINX_VIVADO/lib/lnx64.o/libxil_vcs.so:xilinx_register_systf -full64 \
-debug_pp -t ps -licwait -60 -l design_1_wrapper_comp.log"

  vcs $vcs_opts work.design work.glbl -o design_simv
}
```

Note: The `export_simulation` command always applies OS defaults. So, if you are using 64-bit Vivado Design Suite tools, the scripts generated by the Tcl command will be for 64-bit simulation. If you want to use 32- bit simulation you must edit the compile, elaborate, and simulate options. You must also point to a 32- bit location for the `LD_LIBRARY_PATH` and `-loadvpi` switch (change `lin64` to `lin`).

IMPORTANT: *Ensure that the -lib_map_path switch points to the directory that contains the synopsys_sim.setup file.*

VCS Timing Simulation

VCS Timing simulation consists of the following steps:

1. Generating the simulation netlist (`timesim.v` generation).
2. Annotating timing information to the netlist (SDF f ile generation).
3. Analyzing, elaborating, and simulating the timing netlist and SDF using VCS.

Timing Netlist/SDF Generation in the Vivado Design Suite

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

- **Tcl Command:**

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

VCS Timing Simulation Command

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

Dumping SAIF for Power Analysis for VCS

VCS provides power commands to generate SAIF with specific requirements.

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

```
power <hdl_objects>
```

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

```
power -enable
```

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

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

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

Value Rules in Vivado Simulator Tcl Commands

Introduction

This appendix contains the value rules that apply to both the `add_force` and the `set_value` Tcl commands.

String Value Interpretation

The interpretation of the value string is determined by the declared type of the HDL object and the `-radix` command line option. The `-radix` always overrides the default radix determined by the HDL object type.

- For HDL objects of type `logic`, the value is or a one-dimensional array of the `logic` type or the value is a string of digits of the specified radix.
  - If the string specifies less bits than the type expects, the string is implicitly zero-extended (not sign-extended) to match the length of the type.
  - If the string specifies more bits than the type expects, the extra bits on the MSB side must be zero; otherwise the command generates a size mismatch error.

    For example, with radix hex and a 6 bit `logic` array, the value `3F` specifies 8 bits (4 per hex digit), equivalent to binary 0011 1111. But, because the upper two bits of `3` are zero, the value can be assigned to the HDL object. In contrast, the value `7F` would generate an error, because the upper two bits are not zero.
  - A scalar (not array or record) `logic` HDL object has an implicit length of 1 bit.
  - For a `logic` array declared as `a [left:right]` (Verilog) or `a(left TO/DOWNTO right)`, the left-most value bit (after extension/truncation) is assigned to `a[left]` and the right-most value bit is assigned to `a[right]`.

Vivado Simulation Logic

The logic is not a concept defined in HDL but is a heuristic introduced by the Vivado® simulator.

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

XILINX
ALL PROGRAMMABLE™

Appendix B

Vivado Simulator Mixed Language Support and Language Exceptions

Introduction

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

- VHDL, see *IEEE Standard VHDL Language Reference Manual* (IEEE-STD-1076-1993) [Ref 15]
- Verilog, see *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1364-2001) [Ref 12]
- Verilog with IP, see *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)* (IEEE-STD-P1735) [Ref 15]

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

Using Mixed Language Simulation

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

Restrictions on Mixed Language in Simulation

- A VHDL design can instantiate Verilog modules and a Verilog design can instantiate VHDL components. Component instantiation-based default binding is used for binding a Verilog module to a VHDL component. Specifically, configuration specification and direct instantiation are not supported for a Verilog module instantiated inside a VHDL component. Any other kind of mixed use of VHDL and Verilog, such as VHDL process calling a Verilog function, is not supported.
- A subset of VHDL types, generics, and ports are allowed on the boundary to a Verilog module. Similarly, a subset of Verilog types, parameters and ports are allowed on the boundary to VHDL components. See Table B-2, page 161.

IMPORTANT: *Connecting whole VHDL record object to a Verilog object is unsupported; however, VHDL record elements of a supported type can be connected to a compatible Verilog port.*

- A Verilog hierarchical reference cannot refer to a VHDL unit nor can a VHDL expanded or selected name refer to a Verilog unit.

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

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

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

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

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

-- Top Verilog Module
module top(input in1,output reg out1);
mid I1();
always@(in1)
begin

// This hierarchical reference into a VHDL instance is not allowed
  if(I1.const1 >= 10) out1 = in1;
// This hierarchical reference into a Verilog instance traversing through a
// VHDL instance is allowed
  if (I1.I2.data == 1)out1 = ~in1;
end
endmodule
```

Key Steps in a Mixed Language Simulation

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

Note: The library search order specified by `-L` is used for binding Verilog modules to other Verilog modules as well.

Mixed Language Binding and Searching

When you instantiate a VHDL component in a Verilog module or a Verilog module in a VHDL architecture, the `xelab` command:

- First searches for a unit of the same language as that of the instantiating design unit.
- If a unit of the same language is not found, `xelab` searches for a cross-language design unit in the libraries specified by the `-L` option.

The search order is the same as the order of appearance of libraries on the `xelab` command line. See Method 1: Using the complete UNIFAST library (Recommended), page 22 for more information.

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

Instantiating Mixed Language Components

In a mixed language design, you can instantiate a Verilog module in a VHDL architecture or a VHDL component in a Verilog module as described in the following subsections.

To ensure that you are correctly matching port types, review the Port Mapping and Supported Port Types, page 161.

Instantiating a Verilog Module in a VHDL Design Unit

1. Declare a VHDL component with the same name and in the same case as the Verilog module that you want to instantiate. For example:

```
COMPONENT MY_VHDL_UNIT PORT (
   Q : out   STD_ULOGIC;
   D : in    STD_ULOGIC;
   C : in    STD_ULOGIC );
END COMPONENT;
```

2. Use named or positional association to instantiate the Verilog module. For example:

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

Instantiating a VHDL Component in a Verilog Design Unit

To instantiate a VHDL component in a Verilog design unit, instantiate the VHDL component 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);
);
```

Port Mapping and Supported Port Types

Table B-1 lists the supported port types.

Table B-1: **Supported Port Types**

VHDL [1]	Verilog [2]
IN	INPUT
OUT	OUTPUT
INOUT	INOUT

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

Table B-2 shows the supported VHDL and Verilog data types for ports on the mixed language design boundary.

Table B-2: **Supported VHDL and Verilog data types**

VHDL Port	Verilog Port
`bit`	net
`std_logic`	net
`std_logic`	net
`bit_vector`	vector net
`signed`	vector net
`unsigned`	vector net
`std_ulogic_vector`	vector net
`std_logic_vector`	vector net

Note: Verilog output port of type `reg` is supported on the mixed language boundary. On the boundary, an output `reg` port is treated as if it were an output net (wire) port. Any other type found on mixed language boundary is considered an error.

Note: `XSim` supports the record element as an actual in the port map of a Verilog module that is instantiated in the mixed domain. All those types that are supported as VHDL port (listed in Table B-2) are also supported as a record element.

Generics (Parameters) Mapping

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

- integer
- real
- string
- boolean

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

VHDL and Verilog Values Mapping

Table B-3 lists the Verilog states mappings to `std_logic` and `bit`.

Table B-3: **Verilog States mapped to std_logic and bit**

Verilog	std_logic	bit
Z	Z	0
0	0	0
1	1	1
X	X	0

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

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

Table B-4: **VHDL bit Mapping to Verilog States**

bit	Verilog
0	0
1	1

Table B-5 lists the VHDL type `std_logic` mappings to Verilog states.

Table B-5: **VHDL std_logic mapping to Verilog States**

std_logic	Verilog
U	X
X	X
0	0
1	1
Z	Z

Table B-5: **VHDL std_logic mapping to Verilog States *(Cont'd)***

std_logic	Verilog
W	X
L	0
H	1
-	X

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

VHDL Language Support Exceptions

Certain language constructs are not supported by the Vivado simulator. Table B-6 lists the VHDL language support exceptions.

Table B-6: **VHDL Language Support Exceptions**

Supported VHDL Construct	Exceptions
`abstract_literal`	Floating point expressed as based literals are not supported.
`alias_declaration`	Alias to non-objects are in general not supported; particularly the following: Alias of an alias Alias declaration without subtype_indication Signature on alias declarations Operator symbol as alias_designator Alias of an operator symbol Character literals as alias designators
`alias_designator`	Operator_symbol as alias_designator Character_literal as alias_designator
`association_element`	Globally, locally static range is acceptable for taking slice of an actual in an association element.
`attribute_name`	Signature after prefix is not supported.
`binding_indication`	Binding_indication without use of entity_aspect is not supported.
`bit_string_literal.`	Empty bit_string_literal (" ") is not supported
`block_statement`	Guard_expression is not supported; for example, guarded blocks, guarded signals, guarded targets, and guarded assignments are not supported.
`choice`	Aggregate used as choice in case statement is not supported.

Table B-6: **VHDL Language Support Exceptions *(Cont'd)***

Supported VHDL Construct	Exceptions
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	Slicing, indexing, and selection of formals is not supported in a named parameter association within a function_call.
instantiated_unit	Direct configuration instantiation is not supported.
mode	Linkage and Buffer ports are not supported completely.
options	Guarded is not supported.
primary	At places where primary is used, allocator is expanded there.
procedure_call	Slicing, indexing, and selection of formals is not supported in a named parameter association within a procedure_call.
process_statement	Postponed processes are not supported.
selected_signal_assignment	The guarded keyword as part of options is not supported as there is no support for guarded signal assignment.
signal_declaration	The signal_kind is not supported. The signal_kind is used for declaring guarded signals, which are not supported.
subtype_indication	Resolved subtype of composites (arrays and records) is not supported
waveform	Unaffected is not supported.
waveform_element	Null waveform element is not supported as it only has relevance in the context of guarded signals.

Verilog Language Support Exceptions

Table B-7 lists the exceptions to supported Verilog language support.

Table B-7: **Verilog Language Support Exceptions**

Verilog Construct	Exception
Compiler Directive Constructs	
`unconnected_drive	not supported
`nounconnected_drive	not supported
Attributes	
attribute_instance	not supported
attr_spec	not supported
attr_name	not supported
Primitive Gate and Switch Types	
cmos_switchtype	not supported
mos_switchtype	not supported
pass_en_switchtype	not supported
Generated Instantiation	
generated_instantiation	The module_or_generate_item alternative is not supported. Production from standard (see *IEEE Standard Verilog Hardware Description Language (*IEEE 1364-2001) section 13.2 [Ref 12]: generate_item_or_null ::= generate_conditonal_statement | generate_case_statement | generate_loop_statement | generate_block | module_or_generate_item Production supported by Simulator: generate_item_or_null ::= generate_conditional_statement| generate_case_statement | generate_loop_statement | generate_blockgenerate_condition

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

Verilog Construct	Exception

genvar_assignment	Partially supported.
	All generate blocks must be named.
	Production from standard (see *IEEE Standard Verilog Hardware Description Language (*IEEE 1364-2001) section 13.2 [Ref 12]:
	`generate_block ::=`
	`begin`
	`[ : generate_block_identifier ]`
	`{ generate_item }`
	`end`
	`Production supported by Simulator:`
	`generate_block ::=`
	`begin:`
	`generate_block_identifier {`
	`generate_item }`
	`end`

Source Text Constructs	
Library Source Text	
library_text	not supported
library_descriptions	not supported
library_declaration	not supported
include_statement	This refers to include statements within library map files (See *IEEE Standard Verilog Hardware Description Language (*IEEE 1364-2001) section 13.2 [Ref 12]. This does not refer to the `include compiler directive.
System Timing Check Commands	
$skew_timing_check	not supported
$timeskew_timing_check	not supported
$fullskew_timing_check	not supported
$nochange_timing_check	not supported
System Timing Check Command Argument	
checktime_condition	not supported
PLA Modeling Tasks	
$async$nand$array	not supported
$async$nor$array	not supported
$async$or$array	not supported
$sync$and$array	not supported
$sync$nand$array	not supported
$sync$nor$array	not supported

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

Verilog Construct	Exception
`$sync$or$array`	not supported
`$async$and$plane`	not supported
`$async$nand$plane`	not supported
`$async$nor$plane`	not supported
`$async$or$plane`	not supported
`$sync$and$plane`	not supported
`$sync$nand$plane`	not supported
`$sync$nor$plane`	not supported
`$sync$or$plane`	not supported
Value Change Dump (VCD) Files	

`$dumpportson`
`$dumpports`
`$dumpportsoff`
`$dumpportsflush`
`$dumpportslimit`
`$vcdplus`
	not supported

XILINX
ALL PROGRAMMABLE

Appendix C

Vivado Simulator Quick Reference Guide

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

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

Parsing HDL Files		
Vivado Simulator supports three HDL file types: Verilog, SystemVerilog and VHDL. You can parse the supported files using XVHDL and XVLOG commands.		

Parsing VHDL files	`xvhdl file1.vhd file2.vhd`
	`xvhdl -work worklib file1.vhd file2.vhd`
	`xvhdl -prj files.prj`	

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

Parsing SystemVerilog files	`xvlog -sv file1.v file2.v`
	`xvlog -work worklib -sv file1.v file2.v`
	`xvlog -prj files.prj`
	Note: For information about the PRJ file format, see Project File (.prj) Syntax in Chapter 5.	

Additional xvlog and xvhdl Options		
xvlog and xvhdl Key Options	See Table 5-2, page 103 for a complete list of command options. The following are key options for `xvlog` and `xvhdl`:	
	Key Option	Applies to:
	`-d [define] <name>[=<val>]`	`xvlog, xvhdl`
	`-h [-help]`	`xvlog, xvhdl`
	`-i [include] <directory_name>`	`xvlog, xvhdl`
	`-initfile <init_filename>`	`xvlog, xvhdl`
	`-L [-lib] <library_name> [=<library_dir>]`	`xvlog, xvhdl`
	`-log <filename>`	`xvlog, xvhdl`
	`-prj <filename>`	`xvlog, xvhdl`
	`-relax`	`xvhdl`
	`-work <library_name> [=<library_dir>]`	`xvlog, xvhdl`

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

Elaborating and Generating an Executable Snapshot		
After parsing, you can elaborate the design in Vivado Simulator using the XELAB command. Xelab generates an executable snapshot. ***Note:*** You can skip the parser stage, directly invoke the `XELAB` command, and pass the PRJ file. `XELAB` calls `XVLOG` and `XVHDL` for parsing the files.		
Usage	`xelab top1 top2`	Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units are compiled in the `/work` library.
	`xelab lib1.top1 lib2.top2`	Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units have are compiled in `lib1` and `lib2`, respectively

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

	`xelab top1 top2 -s top`	Elaborates a design that has two top design units: `top1` and `top2`. In this example, the design units are compiled in the `/work` library. After compilation, `xelab` generates an executable snapshot with the name `top`. Without the `-s` top switch, `xelab` creates the snapshot by concatenating the unit names.
Command Line Help and xelab Options	`xelab -help` xelab, xvhd, and xvlog Command Options, page 103	
Running Simulation		
After parsing, elaboration and compilation stages are successful; xsim generates an executable snapshot to run simulation.		
Usage	`xsim top -R`	Simulates the design to through completion.
	`xsim top -gui`	Opens the Vivado simulator GUI.

	`xsim top`	Opens the Vivado Design Suite command prompt in Tcl mode. From there, you can invoke such options as:
		`run -all`
		`run 100 ns`

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

	Three Stage	`xvlog bot.v`
		`xvhdl top.vhd`
		`xelab work.top -s top`
		`xsim top -R`

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

	Two Stage	`xelab -prj my_prj.prj work.top -s top`
		`xsim top -R`
		
		where `my_prj.prj` file contains:
		
		`verilog work bot.v`
		`vhdl work top.vhd`

	Single Stage	`xelab -prj my_prj.prj work.top -s top -R`
		where `my_prj.prj` file contains:
		`verilog work bot.v`
		`vhdl work top.vhd`

Vivado Simulation Tcl Commands		
The following are commonly used Tcl commands. For a complete list, invoke following commands in the Tcl Console: • `load_features simulator` • `help -category simulation` For information on any Tcl Command, type: `-help <Tcl_command>`		

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

Common Vivado Simulator Tcl Commands:	`add_bp`	Add break point at a line of HDL source. See TCL commands:, page 91 for more information
	`add_force`	Force the value of a signal, wire, or register to a specified value.
	`current_time` `now`	Report current simulation time. See current_time, page 113 for an example of this command within a Tcl script.
	`current_scope`	Report or set the current, working HDL scope. See Additional Scopes and Sources Options, page 59 for more information.
	`get_objects`	Get a list of HDL objects in one or more HDL scopes, per the specified pattern. See log_saif [get_objects -filter {type == in_port || type == out_port || type == inout_port || type == port } /tb/UUT/*], page 127 for an example use of this command.
	`get_scopes`	Get a list of child HDL scopes. See Additional Scopes and Sources Options, page 59 for more information.
	`get_value`	Get the current value of the selected HDL object (variable, signal, wire, register). Type `get_value -help` in Tcl Console for more information.
	`launch_modelsim`	Launch simulation using the ModelSim simulator. See Pointing to the QuestaSim/ModelSim Simulator Install Location, page 130 and To use the corresponding Tcl command, type: launch_modelsim, page 138 for more information.
	`launch_xsim`	Launch simulation using the Vivado simulator. See To use the corresponding Tcl command, type: launch_xsim, page 44 for more information.
	`remove_bps`	Remove breakpoints from a simulation. See Type the Tcl command remove_bp., page 121 for more information.
	`report_drivers`	Print drivers along with current driving values for an HDL wire or signal object. See Using the report_drivers Tcl Command, page 127 for more information.
	`report_values`	Print current simulated value of given HDL objects (variables, signals, wires, or registers). See report_drivers <hdl_object>, page 127 for more information.
	`restart`	Rewind simulation to post loading state (as though the design was reloaded); time is set to 0. See Restart: Lets you restart an existing simulation from 0. Tcl Command: restart, page 53.
	`set_value`	Set the HDL object (variable, signal, wire, or register) to a specified value. Appendix A, Value Rules in Vivado Simulator Tcl Commands.
	`step`	Step simulation to the next statement. See Simulation Toolbar, page 55.

Appendix D

SystemVerilog Constructs Supported by the Vivado Simulator

IMPORTANT: *In 2014.1, SystemVerilog support is early access only.*

Introduction

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

Targeting SystemVerilog for a Specific File

By default, the Vivado Simulator tool compiles V files with the Verilog 2001 syntax and SV files with the SystemVerilog syntax.

To target SystemVerilog for a specific V file in the Vivado IDE:

1. Right-click the file and select **Source Node Properties**.
2. In the Source Node Properties window, change the Type from Verilog to SystemVerilog and click **Apply**.

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

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

Running SystemVerilog in Standalone or prj Mode

Standalone Mode

A new `-sv` flag has been introduced to `xvlog`, so if you want to read any SystemVerilog file, you can use following command:

```
xvlog  -sv <Design file list>
xvlog -sv  -work <LibraryName> <Design File List>
xvlog -sv -f  <FileName> [Where FileName contain path of test cases]
```

prj Mode

If you want to run the Vivado simulator in the `prj`-based flow, use `sv` as the file type, as you would `verilog` or `vhdl`.

```
xvlog -prj <prj File>
xelab -prj <prj File>  <topModuleName> <other options>
```

. . .where the entry in `prj` file appears as follows:

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

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

Primary construct	Secondary construct	LRM section	Status
Data type		6	
	Singular and aggregate types	6.4	Supported
	Nets and variables	6.5	Supported
	Variable declarations	6.8	Supported
	Vector declarations	6.9	Supported
	2-state (two-value) and 4-state (four-value) data types	6.11.2	Supported
	Signed and unsigned integer types	6.11.3	Supported
	Real, shortreal and realtime data types	6.12	Supported
	User-defined types	6.18	Supported
	Enumerations	6.19	Supported
	Defining new data types as enumerated types	6.19.1	Supported
	Enumerated type ranges	6.19.2	Supported
	Type checking	6.19.3	Supported

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

Primary construct	Secondary construct	LRM section	Status
	Enumerated types in numerical expressions	6.19.4	Supported
	Enumerated type methods	6.19.5	Supported
	Type parameters	6.20.3	Supported
	Const constants	6.20.6	Supported
	Type operator	6.23	Supported
	Cast operator	6.24.1	Supported
	`$cast` dynamic casting	6.24.2	Not Supported
	Bitstream casting	6.24.3	Supported
Aggregate data types		7	
	Structures	7.2	Supported
	Packed/Unpacked structures	7.2.1	Supported
	Assigning to structures	7.2.2	Supported
	Unions	7.3	Supported
	Packed/Unpacked unions	7.3.1	Supported
	Tagged unions	7.3.2	Not Supported
	Packed arrays	7.4.1	Supported
	Unpacked arrays	7.4.2	Supported
	Operations on arrays	7.4.3	Supported
	Multidimensional arrays	7.4.5	Supported
	Indexing and slicing of arrays	7.4.6	Supported
	Array assignments	7.6	Supported
	Arrays as arguments to subroutines	7.7	Supported
	Array querying functions	7.11	Supported
	Array manipulation methods (those that do not return queue type)	7.12	Supported
Processes		9	
	Combinational logic `always_comb` procedure	9.2.2	Supported
	Implicit `always_comb` sensitivities	9.2.2.1	Supported
	Latched logic `always_latch` procedure	9.2.2.3	Supported
	Sequential logic `always_ff` procedure	9.2.2.4	Supported
	Sequential blocks	9.3.1	Supported

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

Primary construct	Secondary construct	LRM section	Status
	Parallel blocks	9.3.2	Not Supported
	Procedural timing controls	9.4	Supported
	Conditional event controls	9.4.2.3	Supported
	Sequence events	9.4.2.4	Not Supported
Assignment statement		10	
	The continuous assignment statement	10.3.2	Supported
	Variable declaration assignment (variable initialization)	10.5	Supported
	Assignment-like contexts	10.8	Supported
	Array assignment patterns	10.9.1	Supported
	Structure assignment patterns	10.9.2	Supported
	Unpacked array concatenation	10.10	Supported
	Net aliasing	10.11	Not Supported
Operators and expressions		11	
	Constant expressions	11.2.1	Supported
	Aggregate expressions	11.2.2	Supported
	Operators with real operands	11.3.1	Supported
	Operations on logic (4-state) and bit (2-state) types	11.3.4	Supported
	Assignment within an expression	11.3.6	Supported
	Assignment operators	11.4.1	Supported
	Increment and decrement operators	11.4.2	Supported
	Arithmetic expressions with unsigned and signed types	11.4.3.1	Supported
	Wildcard equality operators	11.4.6	Supported
	Concatenation operators	11.4.12	Supported
	Set membership operator	11.4.13	Supported
	Concatenation of `stream_expressions`	11.4.14.1	Supported
	Re-ordering of the generic stream	11.4.14.2	Supported

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

Primary construct	Secondary construct	LRM section	Status
	Streaming concatenation as an assignment target (unpack)	11.4.14.3	Not Supported
	Streaming dynamically sized data	11.4.14.4	Not Supported
Procedural programming statement		12	
	`Unique-if`, `unique0-if` and `priority-if`	12.4.2	Supported
	Violation reports generated by `B-if`, `unique0-if`, and priority-if constructs	12.4.2.1	Supported
	If statement violation reports and multiple processes	12.4.2.2	Supported
	`unique-case`, `unique0-case`, and `priority-case`	12.5.3	Supported
	Violation reports generated by `unique-case`, `unique0-case`, and `priority-case` construct	12.5.3.1	Supported
	Case statement violation reports and multiple processes	12.5.3.2	Supported
	Set membership case statement	12.5.4	Supported
	Pattern matching conditional statements	12.6	Not Supported
	Loop statements	12.7	Supported
	Jump statement	12.8	Supported
Tasks		13.3	
	Static and Automatic task	13.3.1	Supported
	Tasks memory usage and concurrent activation	13.3.2	Supported
Function		13.4	
	Return values and void functions	13.4.1	Supported
	Static and Automatic function	13.4.2	Supported

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

Primary construct	Secondary construct	LRM section	Status
	Constant function	13.4.3	Supported
	Background process spawned by function call	13.4.4	Not Supported
Subroutine calls and argument passing		13.5	
	Pass by value	13.5.1	Supported
	Pass by reference	13.5.2	Not Supported
	Default argument value	13.5.3	Supported
	Argument binding by name	13.5.4	Supported
	Optional argument list	13.5.5	Supported
	Import and Export function	13.6	Not Supported
	Task and function name	13.7	Supported
Utility system tasks and system functions (only synthesizable set)		20	Supported
I/O system tasks and system functions (only synthesizable set)		21	Supported
Compiler directives		22	Supported
Modules and hierarchy		23	
	Default port values	23.2.2.4	Supported
	Top-level modules and `$root`	23.3.1	Supported
	Module instantiation syntax	23.3.2	Supported
	Nested modules	23.4	Supported
	Extern modules	23.5	Supported
	Hierarchical names	23.6	Supported
	Member selects and hierarchical names	23.7	Supported
	Upwards name referencing	23.8	Supported
	Overriding module parameters	23.10	Supported
	Binding auxiliary code to scopes or instances	23.11	Not Supported
Interfaces		25	
	Interface syntax	25.3	Supported
	Ports in interfaces	25.4	Supported

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

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

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

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

Documentation References

1. Vivado® Design Suite Documentation: (*Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
2. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
3. *Vivado Design Suite User Guide: Using the TCL Scripting Capabilities* (UG894)
4. *Writing Efficient Testbenches* (XAPP199)
5. *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide* (UG953)
6. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
7. *Vivado Design Suite User Guide: Using Constraints* (UG903)
8. Vivado Design Suite Tutorial: Simulation (UG937)
9. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)

10. Answer Record "How do I perform NCSim or VCS simulation using IP in Vivado?" AR# 56487: http://www.xilinx.com/support/answers/56487.html

Quick Take Videos

http://www.xilinx.com/training/vivado/index.htm

Links to Language and Encryption Support Standards

11. *IEEE Standard VHDL Language Reference Manual* (IEEE-STD-1076-2008)
12. *IEEE Standard Verilog Hardware Description Language* (IEEE-STD-1364-2001)
13. *IEEE Standard System Verilog Hardware Description Language* (IEEE-STD-1800-2009)
14. *Standard Delay Format Specification (SDF)* (Version 2.1)
15. *Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)* (IEEE-STD-P1735).

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

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