



Questa™ SV/AFV Tutorial

Software Version 6.4c

© 1991-2008 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:

Mentor Graphics Corporation

8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.

Telephone: 503.685.7000

Toll-Free Telephone: 800.592.2210

Website: www.mentor.com

SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/user/feedback_form.cfm

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other third parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the respective third-party owner. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/terms_conditions/trademarks.cfm.

Table of Contents

Chapter 1

Introduction.....	15
Assumptions.....	15
Where to Find Our Documentation	15
Download a Free PDF Reader With Search	16
Mentor Graphics Support.....	16
Before you Begin.....	17
Example Designs	17

Chapter 2

Conceptual Overview	19
Design Optimizations.....	19
Basic Simulation Flow.....	19
Project Flow.....	20
Multiple Library Flow	21
Debugging Tools	22

Chapter 3

Basic Simulation	25
Create the Working Design Library.....	26
Run the Simulation	30
Set Breakpoints and Step through the Source	32
Navigating the Interface.....	35

Chapter 4

Projects.....	41
Create a New Project	41
Add Objects to the Project	42
Changing Compile Order (VHDL).....	44
Compile the Design.....	45
Load the Design	46
Organizing Projects with Folders.....	47
Add Folders.....	47
Moving Files to Folders	49
Simulation Configurations.....	50

Chapter 5

Working With Multiple Libraries.....	55
Creating the Resource Library.....	55
Creating the Project	57
Linking to the Resource Library	58

Linking in Verilog	59
Linking in VHDL	60
Permanently Mapping VHDL Resource Libraries	62
Chapter 6	
Simulating Designs With SystemC	65
Setting up the Environment	66
Preparing an OSCI SystemC design	66
Compiling a SystemC-only Design	70
Mixed SystemC and HDL Example	70
Viewing SystemC Objects in the GUI	74
Setting Breakpoints and Stepping in the Source Window	75
Examining SystemC Objects and Variables	77
Removing a Breakpoint	79
Chapter 7	
Analyzing Waveforms	81
Loading a Design	82
Add Objects to the Wave Window	82
Zooming the Waveform Display	84
Using Cursors in the Wave Window	84
Working with a Single Cursor	85
Working with Multiple Cursors	86
Saving and Reusing the Window Format	87
Chapter 8	
Creating Stimulus With Waveform Editor	89
Load a Design Unit	89
Create Graphical Stimulus with a Wizard	90
Edit Waveforms in the Wave Window	93
Save and Reuse the Wave Commands	96
Exporting the Created Waveforms	97
Simulating with the Testbench File	98
Importing an EVCD File	100
Chapter 9	
Debugging With The Dataflow Window	103
Exploring Connectivity	104
Tracing Events	106
Tracing an X (Unknown)	110
Displaying Hierarchy in the Dataflow Window	112
Chapter 10	
Viewing And Initializing Memories	115
View a Memory and its Contents	116
Navigate Within the Memory	119
Export Memory Data to a File	121

Table of Contents

Initialize a Memory	123
Interactive Debugging Commands	126
Chapter 11	
Analyzing Performance With The Profiler	131
View Profile Details.....	136
Filtering and Saving the Data	137
Chapter 12	
Simulating With Code Coverage.....	141
Coverage Statistics in the Main window	145
Coverage Statistics in the Source Window	146
Toggle Statistics in the Objects Pane.....	148
Excluding Lines and Files from Coverage Statistics.....	149
Creating Code Coverage Reports.....	150
Chapter 13	
Debugging With PSL Assertions.....	153
Compile the Example Design	153
Load and Run Without Assertions.....	154
Using Assertions to Speed Debugging	155
Debugging the Assertion Failure	160
Chapter 14	
SystemVerilog Assertions and Functional Coverage.....	165
Design Files for this Lesson	165
Understanding the Interleaver Design	165
Related Reading	168
Run the Simulation without Assertions.....	168
Run the Simulation with Assertions	169
Debugging with Assertions	171
Exploring Functional Coverage.....	179
Creating Functional Coverage Reports	190
Lesson Wrap-Up	192
Chapter 15	
Using the SystemVerilog DPI.....	193
Chapter 16	
Using SystemVerilog DPI for Data Passing	203
Mapping Verilog and C	203
Chapter 17	
Comparing Waveforms	215
Creating the Reference Dataset	216
Creating the Test Dataset.....	216

Comparing the Simulation Runs	218
Viewing Comparison Data.	219
Comparison Data in the Wave Window	219
Comparison Data in the List Window	220
Saving and Reloading Comparison Data	221
 Chapter 18	
Automating Simulation	225
Creating a Simple DO File.	225
Running in Command-Line Mode.	226
Using Tcl with the Simulator.	229
 Index	
 End-User License Agreement	

List of Examples

Example 14-1. Assertion Property Definition	173
---	-----

List of Figures

Figure 2-1. Basic Simulation Flow - Overview Lab	20
Figure 2-2. Project Flow	21
Figure 2-3. Multiple Library Flow.....	22
Figure 3-1. Basic Simulation Flow - Simulation Lab	25
Figure 3-2. The Create a New Library Dialog.....	26
Figure 3-3. work Library in the Workspace.....	27
Figure 3-4. Compile Source Files Dialog	28
Figure 3-5. Verilog Modules Compiled into work Library	28
Figure 3-6. Workspace sim Tab Displays Design Hierarchy	29
Figure 3-7. Object Pane Displays Design Objects.....	30
Figure 3-8. Using the Popup Menu to Add Signals to Wave Window	31
Figure 3-9. Waves Drawn in Wave Window.....	32
Figure 3-10. Setting Breakpoint in Source Window	33
Figure 3-11. Setting Restart Functions	34
Figure 3-12. Blue Arrow Indicates Where Simulation Stopped.....	34
Figure 3-13. Values Shown in Objects Window	35
Figure 3-14. Parameter Name and Value in Source Examine Window	35
Figure 3-15. The Main Window	36
Figure 3-16. Window/Pane Control Icons.....	37
Figure 3-17. zooming in on Workspace Pane	38
Figure 3-18. Panes Rearranged in Main Window	39
Figure 4-1. Create Project Dialog - Project Lab	42
Figure 4-2. Adding New Items to a Project.....	43
Figure 4-3. Add file to Project Dialog	43
Figure 4-4. Newly Added Project Files Display a “?” for Status	44
Figure 4-5. Compile Order Dialog.....	45
Figure 4-6. Library Tab with Expanded Library	46
Figure 4-7. Structure Tab for a Loaded Design.....	47
Figure 4-8. Adding New Folder to Project	48
Figure 4-9. A Folder Within a Project.....	48
Figure 4-10. Creating Subfolder	48
Figure 4-11. A folder with a Sub-folder	49
Figure 4-12. Changing File Location via the Project Compiler Settings Dialog.....	49
Figure 4-13. Simulation Configuration Dialog	51
Figure 4-14. A Simulation Configuration in the Project Tab	52
Figure 4-15. Transcript Shows Options for Simulation Configurations	52
Figure 5-1. Creating New Resource Library	56
Figure 5-2. Compiling into the Resource Library	57
Figure 5-3. VHDL Simulation Warning Reported in Main Window	59
Figure 5-4. Specifying a Search Library in the Simulate Dialog.....	60

List of Figures

Figure 5-5. Mapping to the parts_lib Library	61
Figure 5-6. Adding LIBRARY and USE Statements to the Testbench.....	62
Figure 6-1. SystemC Code Before and After Modifications	68
Figure 6-2. Editing the SystemC Header File.....	69
Figure 6-3. The ringbuf.h File.....	72
Figure 6-4. The test_ringbuf.cpp File	72
Figure 6-5. The test_ringbuf Design	73
Figure 6-6. SystemC Objects in the work Library.....	74
Figure 6-7. SystemC Objects in the sim Tab of the Workspace	75
Figure 6-8. Active Breakpoint in a SystemC File	76
Figure 6-9. Simulation Stopped at Breakpoint	77
Figure 6-10. Stepping into a Separate File.....	77
Figure 6-11. Output of show Command	78
Figure 6-12. SystemC Primitive Channels in the Wave Window	79
Figure 7-1. Panes of the Wave Window	81
Figure 7-2. Undocking the Wave Window	83
Figure 7-3. Zooming in with the Mouse Pointer	84
Figure 7-4. Working with a Single Cursor in the Wave Window	85
Figure 7-5. Renaming a Cursor	86
Figure 7-6. Interval Measurement Between Two Cursors.....	87
Figure 7-7. A Locked Cursor in the Wave Window	87
Figure 8-1. Initiating the Create Pattern Wizard from the Objects Pane.....	91
Figure 8-2. Create Pattern Wizard	91
Figure 8-3. Specifying Clock Pattern Attributes	92
Figure 8-4. The <i>clk</i> Waveform.....	92
Figure 8-5. The <i>reset</i> Waveform	93
Figure 8-6. Edit Insert Pulse Dialog	93
Figure 8-7. Signal <i>reset</i> with an Inserted Pulse	94
Figure 8-8. Edit Stretch Edge Dialog.....	94
Figure 8-9. Stretching an Edge on the <i>clk</i> Signal.....	95
Figure 8-10. Deleting an Edge on the <i>clk</i> Signal	95
Figure 8-11. The Export Waveform Dialog.....	97
Figure 8-12. The counter Waveform Reacts to Stimulus Patterns.....	98
Figure 8-13. The <i>export</i> Testbench Compiled into the work Library	99
Figure 8-14. Waves from Newly Created Testbench.....	99
Figure 8-15. EVCD File Loaded in Wave Window	100
Figure 8-16. Simulation results with EVCD File	101
Figure 9-1. A Signal in the Dataflow Window	105
Figure 9-2. Expanding the View to Display Connected Processes	105
Figure 9-3. The test Net Expanded to Show All Drivers.....	106
Figure 9-4. The embedded wave viewer pane	107
Figure 9-5. Signals Added to the Wave Viewer Automatically	108
Figure 9-6. Cursor in Wave Viewer Marks Last Event	109
Figure 9-7. Tracing the Event Set	109
Figure 9-8. A Signal with Unknown Values	110

Figure 9-9. ChaseX Identifies Cause of Unknown on t_out	111
Figure 9-10. Displaying Hierarchy in the Dataflow Window	113
Figure 10-1. The mem Tab in the MDI Frame Shows Addresses and Data	117
Figure 10-2. The Memory Display Updates with the Simulation	117
Figure 10-3. Changing the Address Radix.	118
Figure 10-4. New Address Radix and Line Length.	119
Figure 10-5. Goto Dialog.	119
Figure 10-6. Editing the Address Directly.	120
Figure 10-7. Searching for a Specific Data Value.	120
Figure 10-8. Export Memory Dialog.	122
Figure 10-9. Import Memory Dialog.	124
Figure 10-10. Initialized Memory from File and Fill Pattern	125
Figure 10-11. Data Increments Starting at Address 251	126
Figure 10-12. Original Memory Content.	127
Figure 10-13. Changing Memory Content for a Range of Addresses	127
Figure 10-14. Random Content Generated for a Range of Addresses.	128
Figure 10-15. Changing Memory Contents by Highlighting.	128
Figure 10-16. Entering Data to Change.	129
Figure 10-17. Changed Memory Contents for the Specified Addresses	129
Figure 11-1. Sampling Reported in the Transcript	133
Figure 11-2. The Profile Window	134
Figure 11-3. Design Unit Performance Profile	135
Figure 11-4. Expand the Hierarchical Function Call Tree.	136
Figure 11-5. The Source Window Showing a Line from the Profile Data	136
Figure 11-6. Profile Details of the Function <i>Tcl_Close</i>	137
Figure 11-7. Profile Details of Function <i>sm_0</i>	137
Figure 11-8. The Profiler Toolbar	138
Figure 11-9. The Filtered Profile Data.	138
Figure 11-10. The Profile Report Dialog.	139
Figure 11-11. The <i>calltree.rpt</i> Report	140
Figure 12-1. Code Coverage Columns in the Main Window Workspace	143
Figure 12-2. Missed Coverage Pane	143
Figure 12-3. Instance Coverage Pane	144
Figure 12-4. Details Pane.	144
Figure 12-5. Current Exclusions Pane	144
Figure 12-6. Right-click a Column Heading to Show Column List	145
Figure 12-7. Coverage Statistics in the Source Window	147
Figure 12-8. Coverage Numbers Shown by Hovering the Mouse Pointer	148
Figure 12-9. Toggle Coverage in the Objects Pane	149
Figure 12-10. Excluding a File Using Menus in the Workspace.	150
Figure 12-11. Coverage Text Report Dialog	151
Figure 12-12. Coverage HTML Report Dialog	152
Figure 12-13. Coverage Exclusions Report Dialog	152
Figure 13-1. Transcript After Running Simulation Without Assertions	155
Figure 13-2. Change Assertions Dialog.	157

List of Figures

Figure 13-3. Assertion Failure Indicated in Wave Window	158
Figure 13-4. The Assertion Debug Pane Shows Failed Assertion Details	159
Figure 13-5. Assertion failure indicated in the Analysis pane	160
Figure 13-6. Source Code for Failed Assertion	161
Figure 13-7. Examining <i>we_n</i> With Respect to <i>mem_state</i>	162
Figure 13-8. Dataflow Options Dialog	163
Figure 13-9. Viewing <i>we_n</i> in the Dataflow Window	163
Figure 13-10. Finding the Bug in the Source Code	164
Figure 14-1. Incoming Data	165
Figure 14-2. Block Diagram of the Intellexer	166
Figure 14-3. Block Diagram of the Testbench	167
Figure 14-4. First Simulation Stops at Error	169
Figure 14-5. Enabling Assertion Failure Tracking and Action	170
Figure 14-6. Assertions Set to Break on Failure	171
Figure 14-7. Assertions in Wave Window	171
Figure 14-8. Assertion Failure Message in the Transcript	172
Figure 14-9. Assertions Tab Shows Failure Count	173
Figure 14-10. Source Pane Pointer Shows Where Simulation Stopped	173
Figure 14-11. The Inverted Red Triangle Indicates an Assertion Failure	174
Figure 14-12. Setting the Radix	175
Figure 14-13. Diagnosing Assertion Failure in the Wave Window	176
Figure 14-14. The <i>wadder11</i> Signal in the Dataflow Window	177
Figure 14-15. Source Code for the ALWAYS Block	178
Figure 14-16. Source Code for <i>waddr[11]</i>	178
Figure 14-17. Covergroup Code	179
Figure 14-18. Covergroup Bins	180
Figure 14-19. Covergroup <i>sm_svg</i>	181
Figure 14-20. Bins for the <i>sm_cvg</i> Covergroup	182
Figure 14-21. Viewing the Source Code for a Covergroup	183
Figure 14-22. Source Code for <i>ram_cvg</i> Covergroup	183
Figure 14-23. Covergroup Instances for <i>ram_cvg</i>	184
Figure 14-24. Cover Directive for the Interleaver Design	185
Figure 14-25. Source Code for the Cover Directive	185
Figure 14-26. Scoreboard Information in the Transcript	186
Figure 14-27. Covergroup Coverage in the Analysis Window	187
Figure 14-28. Cover Directive Counts State Transitions	188
Figure 14-29. Changing the Cover Directive View to Count View	188
Figure 14-30. First Temporal and Count Mode Views of Cover Directive	189
Figure 14-31. Second Temporal and Count Mode Views of Cover Directive	189
Figure 14-32. Functional Coverage Report Dialog	190
Figure 14-33. The Functional Coverage Report	191
Figure 15-1. Source Code for Module <i>test.sv</i>	194
Figure 15-2. Source Code for the <i>foreign.c</i> File - DPI Lab	196
Figure 15-3. The <i>sv_YellowLight</i> Function in the <i>test.sv</i> File	196
Figure 15-4. The <i>sv_WaitForRed</i> Task in the <i>test.sv</i> File	197

Figure 15-5. The <i>sv_RedLight</i> Function in the <i>test.sv</i> File	197
Figure 15-6. Function Calls in the <i>test.sv</i> File	198
Figure 15-7. Makefile for Compiling and Running on UNIX or Linux Platforms	198
Figure 15-8. The <i>windows.bat</i> File for Compiling and Running in Windows - DPI Lab . . .	199
Figure 15-9. The <i>light</i> Signal in the Objects Pane	201
Figure 15-10. The <i>light</i> Signal in the Wave Window	201
Figure 15-11. Source Code for <i>test.sv</i>	202
Figure 16-1. Source Code for the <i>foreign.c</i> File - Data Passing Lab	204
Figure 16-2. Source Code for the <i>test.sv</i> Module	205
Figure 16-3. Makefile for Compiling and Running on UNIX and Linux Platforms	206
Figure 16-4. The <i>windows.bat</i> File for Compiling and Running in Windows - Data Passing Lab	207
Figure 16-5. Line 12 of <i>test.sv</i> in the Source Window	209
Figure 16-6. The Value of <i>int_var</i> is Currently 0	209
Figure 16-7. The Value of <i>int_var</i> Printed to the Transcript Window	210
Figure 16-8. The Value of <i>bit_var</i> is 0.	210
Figure 16-9. Transcript Shows the Value Returned for <i>bit_var</i>	210
Figure 16-10. The <i>dpi_types.h</i> File	211
Figure 16-11. The Transcript Shows the Correct Value of logic X.	213
Figure 17-1. First dialog of the Waveform Comparison Wizard	218
Figure 17-2. Second dialog of the Waveform Comparison Wizard	218
Figure 17-3. Comparison information in the Workspace and Objects panes	219
Figure 17-4. Comparison objects in the Wave window	220
Figure 17-5. The compare icons	220
Figure 17-6. Compare differences in the List window	221
Figure 17-7. Coverage data saved to a text file	222
Figure 17-8. Displaying Log Files in the Open dialog	223
Figure 17-9. Reloading saved comparison data.	223
Figure 18-1. A Dataset in the Main Window Workspace	228
Figure 18-2. Buttons Added to the Main Window Toolbar	230

List of Tables

Table 1-1. Documentation List	15
Table 3-1. The Main Window	36
Table 6-1. Supported Operating Systems for SystemC	66
Table 11-1. Columns in the Profile Window	134
Table 12-1. Coverage Icons in the Source Window	147

Chapter 1

Introduction

Assumptions

We assume that you are familiar with the use of your operating system. You should also be familiar with the window management functions of your graphic interface: OpenWindows, OSF/Motif, CDE, KDE, GNOME, or Microsoft Windows 2000/XP.

We also assume that you have a working knowledge of the language in which your design and/or testbench is written (i.e., VHDL, Verilog, etc.). Although QuestaSim™ is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Where to Find Our Documentation

QuestaSim documentation is available from our website at

www.mentor.com/supportnet

or from the tool by selecting **Help** :

Table 1-1. Documentation List

Document	Format	How to get it
<i>Installation & Licensing Guide</i>	PDF	Help > PDF Bookcase
	HTML and PDF	Help > InfoHub
<i>Quick Guide</i> (command and feature quick-reference)	PDF	Help > PDF Bookcase and Help > InfoHub
<i>Tutorial</i>	PDF	Help > PDF Bookcase
	HTML and PDF	Help > InfoHub
<i>User's Manual</i>	PDF	Help > PDF Bookcase
	HTML and PDF	Help > InfoHub
<i>Reference Manual</i>	PDF	Help > PDF Bookcase
	HTML and PDF	Help > InfoHub

Table 1-1. Documentation List

Document	Format	How to get it
<i>Foreign Language Interface Manual</i>	PDF	Help > PDF Bookcase
	HTML	Help > InfoHub
Command Help	ASCII	type help [command name] at the prompt in the Transcript pane
Error message help	ASCII	type verror <msgNum> at the Transcript or shell prompt
Tcl Man Pages (Tcl manual)	HTML	select Help > Tcl Man Pages , or find <i>contents.htm</i> in <i>\modeltech\docs\tcl_help_html</i>
Technotes	HTML	available from the support site

Download a Free PDF Reader With Search

QuestaSim PDF documentation requires an Adobe Acrobat Reader for viewing. The Reader is available without cost from Adobe at

www.adobe.com.

Mentor Graphics Support

Mentor Graphics software support includes software enhancements, technical support, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service. For details, see:

<http://supportnet.mentor.com/about/>

If you have questions about this software release, please log in to SupportNet. You may search thousands of technical solutions, view documentation, or open a Service Request online at:

<http://supportnet.mentor.com/>

If your site is under current support and you do not have a SupportNet login, you may easily register for SupportNet by filling out the short form at:

<http://supportnet.mentor.com/user/register.cfm>

All customer support contact information can be found on our web site at:

<http://supportnet.mentor.com/contacts/supportcenters/>

Before you Begin

Preparation for some of the lessons leaves certain details up to you. You will decide the best way to create directories, copy files, and execute programs within your operating system. (When you are operating the simulator within QuestaSim's GUI, the interface is consistent for all platforms.)

Examples show Windows path separators - use separators appropriate for your operating system when trying the examples.

Example Designs

QuestaSim comes with Verilog and VHDL versions of the designs used in these lessons. This allows you to do the tutorial regardless of which license type you have. Though we have tried to minimize the differences between the Verilog and VHDL versions, we could not do so in all cases. In cases where the designs differ (e.g., line numbers or syntax), you will find language-specific instructions. Follow the instructions that are appropriate for the language you use.

Chapter 2

Conceptual Overview

Introduction

QuestaSim is a verification and simulation tool for VHDL, Verilog, SystemVerilog, and mixed-language designs.

This lesson provides a brief conceptual overview of the QuestaSim simulation environment. It is divided into five topics, which you will learn more about in subsequent lessons.

- Design Optimizations — Refer to the [Optimizing Designs with vopt](#) chapter in the User's Manual.
- Basic simulation flow — Refer to [Chapter 3 Basic Simulation](#).
- Project flow — Refer to [Chapter 4 Projects](#).
- Multiple library flow — Refer to [Chapter 5 Working With Multiple Libraries](#).
- Debugging tools — Refer to remaining lessons.

Design Optimizations

Before discussing the basic simulation flow, it is important to understand design optimization. By default, QuestaSim optimizations are automatically performed on all designs. These optimizations are designed to maximize simulator performance, yielding improvements up to 10X, in some Verilog designs, over non-optimized runs.

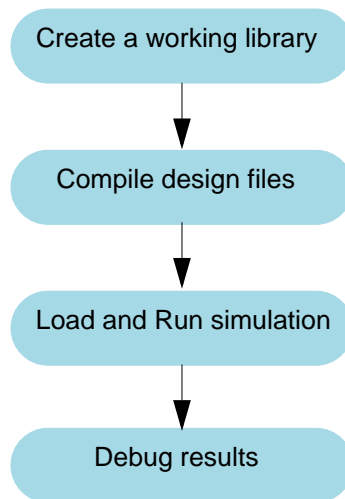
Global optimizations, however, may have an impact on the visibility of the design simulation results you can view – certain signals and processes may not be visible. If these signals and processes are important for debugging the design, it may be necessary to customize the simulation by removing optimizations from specific modules.

It is important, therefore, to make an informed decision as to how best to apply optimizations to your design. The tool that performs global optimizations in QuestaSim is called [vopt](#). Please refer to the [Optimizing Designs with vopt](#) chapter in the QuestaSim User's Manual for a complete discussion of optimization trade-offs and customizations. For details on command syntax and usage, please refer to [vopt](#) in the Reference Manual.

Basic Simulation Flow

The following diagram shows the basic steps for simulating a design in QuestaSim.

Figure 2-1. Basic Simulation Flow - Overview Lab



- **Creating the Working Library**

In QuestaSim, all designs are compiled into a library. You typically start a new simulation in QuestaSim by creating a working library called "work". "Work" is the library name used by the compiler as the default destination for compiled design units.

- **Compiling Your Design**

After creating the working library, you compile your design units into it. The QuestaSim library format is compatible across all supported platforms. You can simulate your design on any platform without having to recompile your design.

- **Loading the Simulator with Your Design and Running the Simulation**

With the design compiled, you load the simulator with your design by invoking the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL).

Assuming the design loads successfully, the simulation time is set to zero, and you enter a run command to begin simulation.

- **Debugging Your Results**

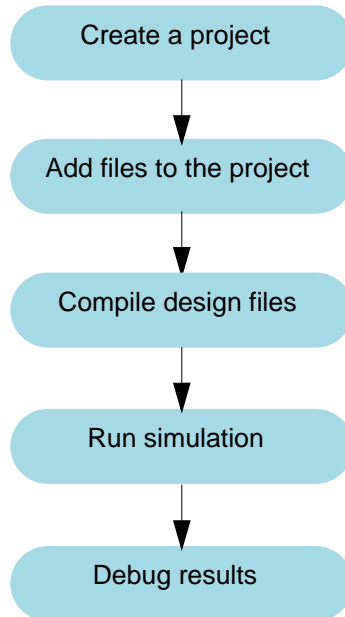
If you don't get the results you expect, you can use QuestaSim's robust debugging environment to track down the cause of the problem.

Project Flow

A project is a collection mechanism for an HDL design under specification or test. Even though you don't have to use projects in QuestaSim, they may ease interaction with the tool and are useful for organizing files and specifying simulation settings.

The following diagram shows the basic steps for simulating a design within a QuestaSim project.

Figure 2-2. Project Flow



As you can see, the flow is similar to the basic simulation flow. However, there are two important differences:

- You do not have to create a working library in the project flow; it is done for you automatically.
- Projects are persistent. In other words, they will open every time you invoke QuestaSim unless you specifically close them.

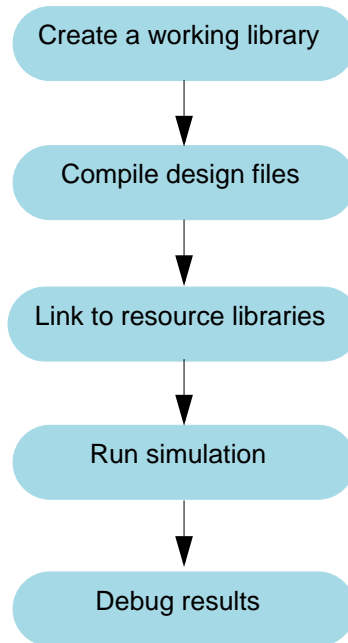
Multiple Library Flow

QuestaSim uses libraries in two ways: 1) as a local working library that contains the compiled version of your design; 2) as a resource library. The contents of your working library will change as you update your design and recompile. A resource library is typically static and serves as a parts source for your design. You can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor).

You specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched. A common example of using both a working library and a resource library is one where your gate-level design and testbench are compiled into the working library, and the design references gate-level models in a separate resource library.

The diagram below shows the basic steps for simulating with multiple libraries.

Figure 2-3. Multiple Library Flow



You can also link to resource libraries from within a project. If you are using a project, you would replace the first step above with these two steps: create the project and add the testbench to the project.

Debugging Tools

QuestaSim offers numerous tools for debugging and analyzing your design. Several of these tools are covered in subsequent lessons, including:

- Using projects
- Working with multiple libraries
- Simulating with SystemC
- Setting breakpoints and stepping through the source code
- Viewing waveforms and measuring time
- Exploring the "physical" connectivity of your design
- Viewing and initializing memories
- Creating stimulus with the Waveform Editor

- Analyzing simulation performance
- Testing code coverage
- Comparing waveforms
- Debugging with PSL assertions
- Using SystemVerilog assertions and cover directives
- Using the SystemVerilog DPI
- Automating simulation

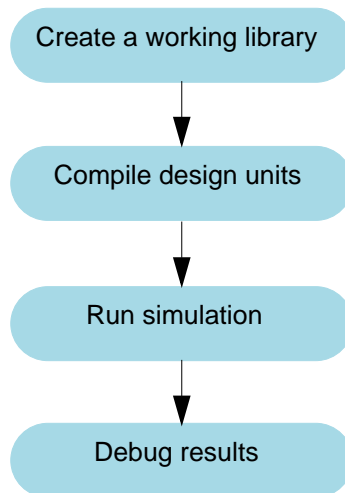
Chapter 3

Basic Simulation

Introduction

In this lesson you will go step-by-step through the basic simulation flow:

Figure 3-1. Basic Simulation Flow - Simulation Lab



Design Files for this Lesson

The sample design for this lesson is a simple 8-bit, binary up-counter with an associated testbench. The pathnames are as follows:

Verilog – `<install_dir>/examples/tutorials/verilog/basicSimulation/counter.v` and `tcounter.v`

VHDL – `<install_dir>/examples/tutorials/vhdl/basicSimulation/counter.vhd` and `tcounter.vhd`

This lesson uses the Verilog files `counter.v` and `tcounter.v`. If you have a VHDL license, use `counter.vhd` and `tcounter.vhd` instead. Or, if you have a mixed license, feel free to use the Verilog testbench with the VHDL counter or vice versa.

Related Reading

User's Manual Chapters: [Design Libraries](#), [Verilog and SystemVerilog Simulation](#), and [VHDL Simulation](#).

Reference Manual commands: [vlib](#), [vmap](#), [vlog](#), [vcom](#), [vopt](#), [view](#), and [run](#).

Create the Working Design Library

Before you can simulate a design, you must first create a library and compile the source code into that library.

1. Create a new directory and copy the design files for this lesson into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons).

Verilog: Copy *counter.v* and *tcounter.v* files from
/<install_dir>/examples/tutorials/verilog/basicSimulation to the new directory.

VHDL: Copy *counter.vhd* and *tcounter.vhd* files from
/<install_dir>/examples/tutorials/vhdl/basicSimulation to the new directory.

2. Start QuestaSim *if necessary*.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

Upon opening QuestaSim for the first time, you will see the Welcome to QuestaSim dialog. Click **Close**.

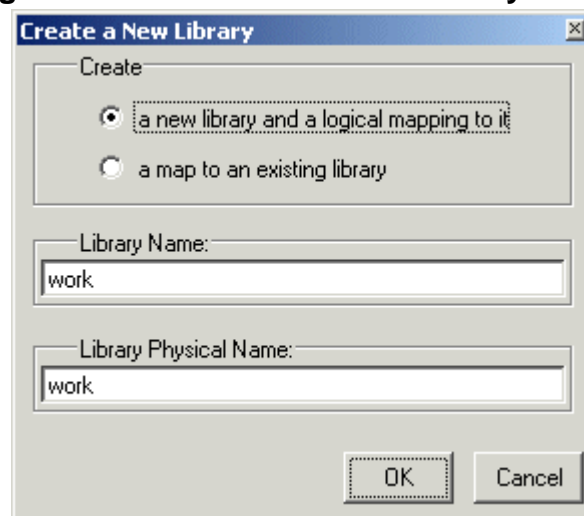
- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Create the working library.

- a. Select **File > New > Library**.

This opens a dialog where you specify physical and logical names for the library (Figure 3-2). You can create a new library or map to an existing library. We'll be doing the former.

Figure 3-2. The Create a New Library Dialog



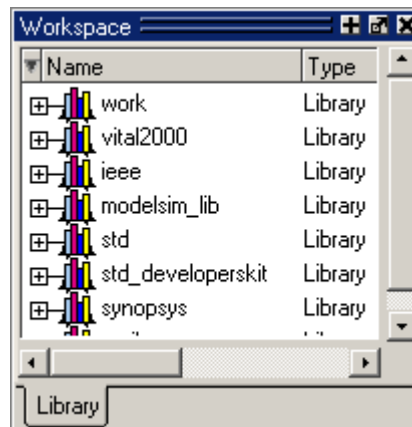
- b. Type **work** in the Library Name field (if it isn't already entered automatically).

- c. Click **OK**.

QuestaSim creates a directory called *work* and writes a specially-formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a QuestaSim library. Do not edit the folder contents from your operating system; all changes should be made from within QuestaSim.

QuestaSim also adds the library to the list in the Workspace (Figure 3-3) and records the library mapping for future reference in the QuestaSim initialization file (*modelsim.ini*).

Figure 3-3. work Library in the Workspace



When you pressed OK in step 3c above, the following was printed to the Transcript:

```
vlib work
vmap work work
```

These two lines are the command-line equivalents of the menu selections you made. Many command-line equivalents will echo their menu-driven functions in this fashion.

Compile the Design

With the working library created, you are ready to compile your source files.

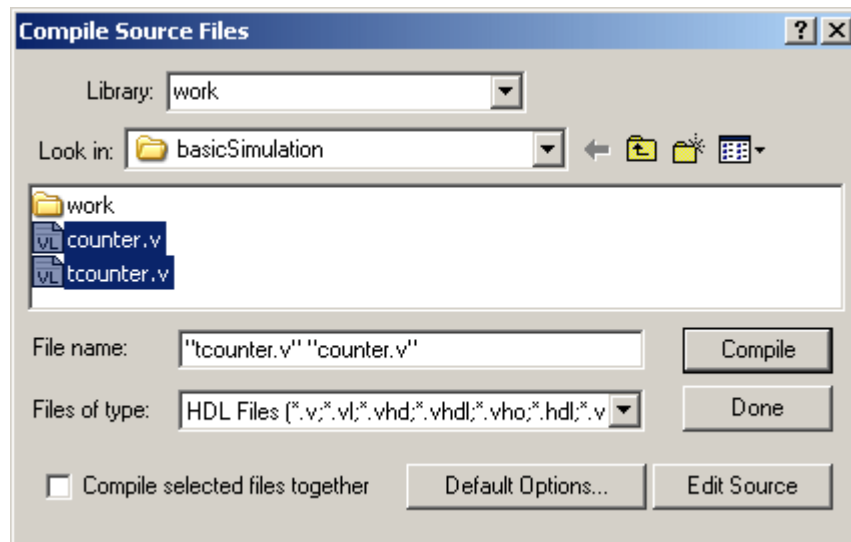
You can compile by using the menus and dialogs of the graphic interface, as in the Verilog example below, or by entering a command at the QuestaSim> prompt.

1. Compile *counter.v* and *tcounter.v*.
 - a. Select **Compile > Compile**. This opens the Compile Source Files dialog (Figure 3-4).

If the Compile menu option is not available, you probably have a project open. If so, close the project by making the Workspace pane active and selecting **File > Close** from the menus.

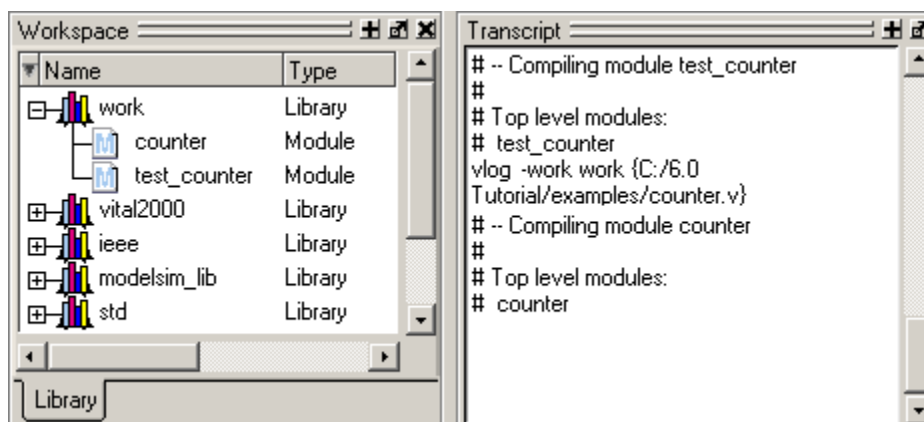
- b. Select both *counter.v* and *tcounter.v* modules from the Compile Source Files dialog and click **Compile**. The files are compiled into the *work* library.
- c. When compile is finished, click **Done**.

Figure 3-4. Compile Source Files Dialog



2. View the compiled design units.
 - a. On the Library tab, click the '+' icon next to the *work* library and you will see two design units (Figure 3-5). You can also see their types (Modules, Entities, etc.) and the path to the underlying source files (scroll to the right if necessary).

Figure 3-5. Verilog Modules Compiled into work Library



Load the Design

1. Load the *test_counter* module into the simulator.
 - a. Enter the following command at the QuestaSim> prompt in the Transcript window:

```
vsim -voptargs="+acc" test_counter
```

The `-voptargs="+acc"` argument for the `vsim` command provides visibility into the design for debugging purposes.

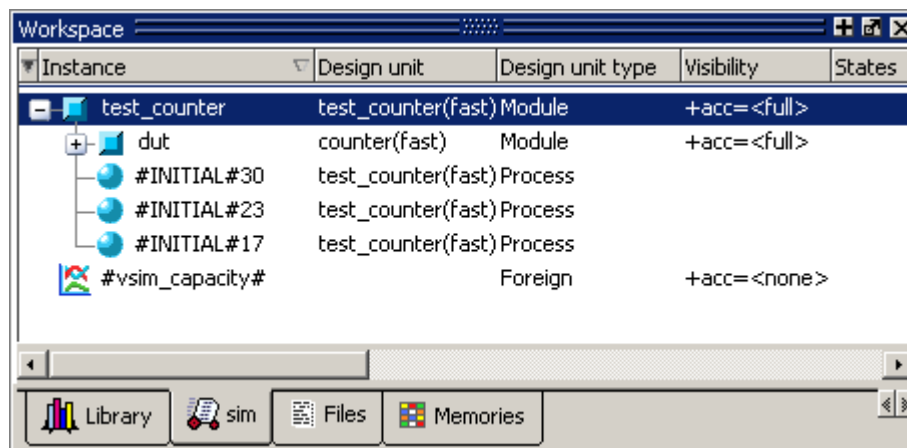
Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

When the design is loaded, you will see a new tab in the Workspace named *sim* that displays the hierarchical structure of the design ([Figure 3-6](#)). You can navigate within the hierarchy by clicking on any line with a '+' (expand) or '-' (contract) icon. You will also see a tab named *Files* that displays all files included in the design.

Figure 3-6. Workspace sim Tab Displays Design Hierarchy

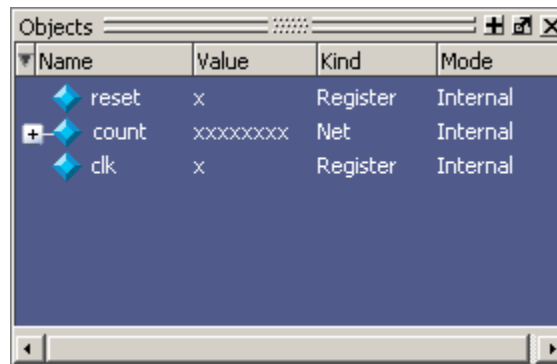


2. View design objects in the Objects pane.
 - a. Open the **View** menu and select **Objects**. The command line equivalent is:


```
view objects
```

The Objects pane ([Figure 3-7](#)) shows the names and current values of data objects in the current region (selected in the Workspace). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters.

Figure 3-7. Object Pane Displays Design Objects



You may open other windows and panes with the **View** menu or with the **view** command. See [Navigating the Interface](#).

Run the Simulation

Now you will open the Wave window, add signals to it, then run the simulation.

1. Open the Wave debugging window.
 - a. Enter **view wave** at the command line.

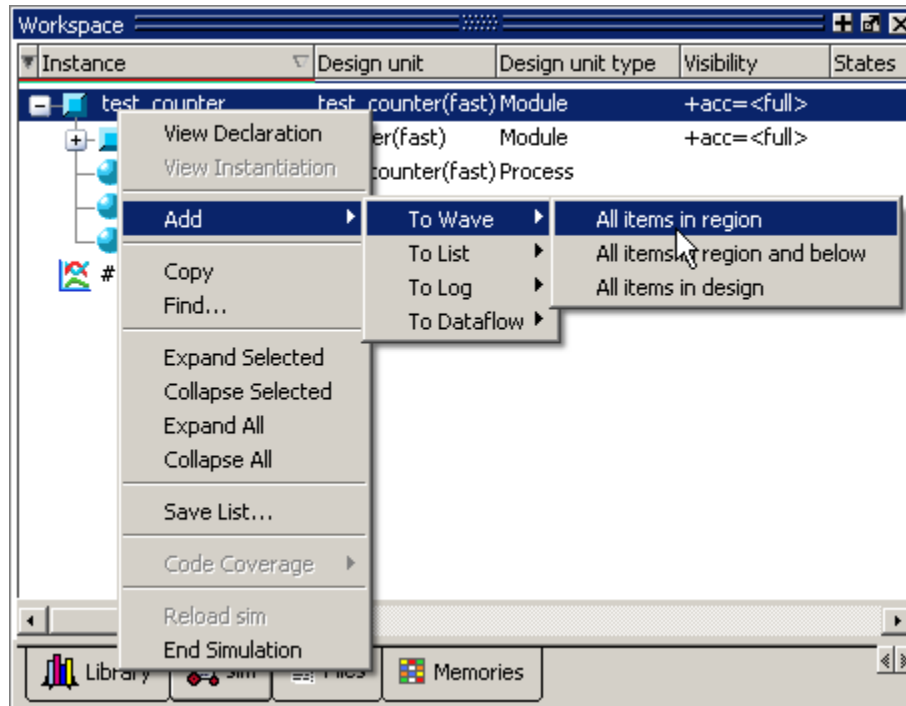
You can also use the **View > Wave** menu selection to open a Wave window.

The Wave window is one of several windows available for debugging. To see a list of the other debugging windows, select the **View** menu. You may need to move or resize the windows to your liking. Window panes within the Main window can be zoomed to occupy the entire Main window or undocked to stand alone. For details, see [Navigating the Interface](#).

2. Add signals to the Wave window.
 - a. In the Workspace pane, select the **sim** tab.
 - b. Right-click *test_counter* to open a popup context menu.
 - c. Select **Add > To Wave > All items in region** ([Figure 3-8](#)).

All signals in the design are added to the Wave window.

Figure 3-8. Using the Popup Menu to Add Signals to Wave Window



3. Run the simulation.

- a. Click the Run icon in the Main or Wave window toolbar.

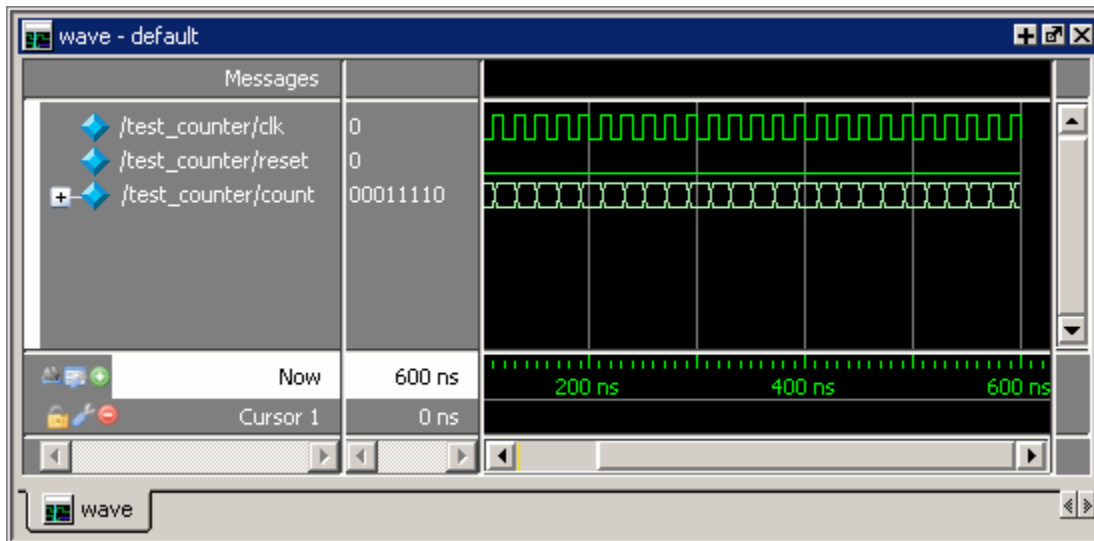
The simulation runs for 100 ns (the default simulation length) and waves are drawn in the Wave window.



- b. Enter **run 500** at the VSIM> prompt in the Main window.

The simulation advances another 500 ns for a total of 600 ns (Figure 3-9).

Figure 3-9. Waves Drawn in Wave Window



- c. Click the **Run -All** icon on the Main or Wave window toolbar.

The simulation continues running until you execute a break command or it hits a statement in your code (e.g., a Verilog \$stop statement) that halts the simulation.



- d. Click the Break icon.



The simulation stops running.

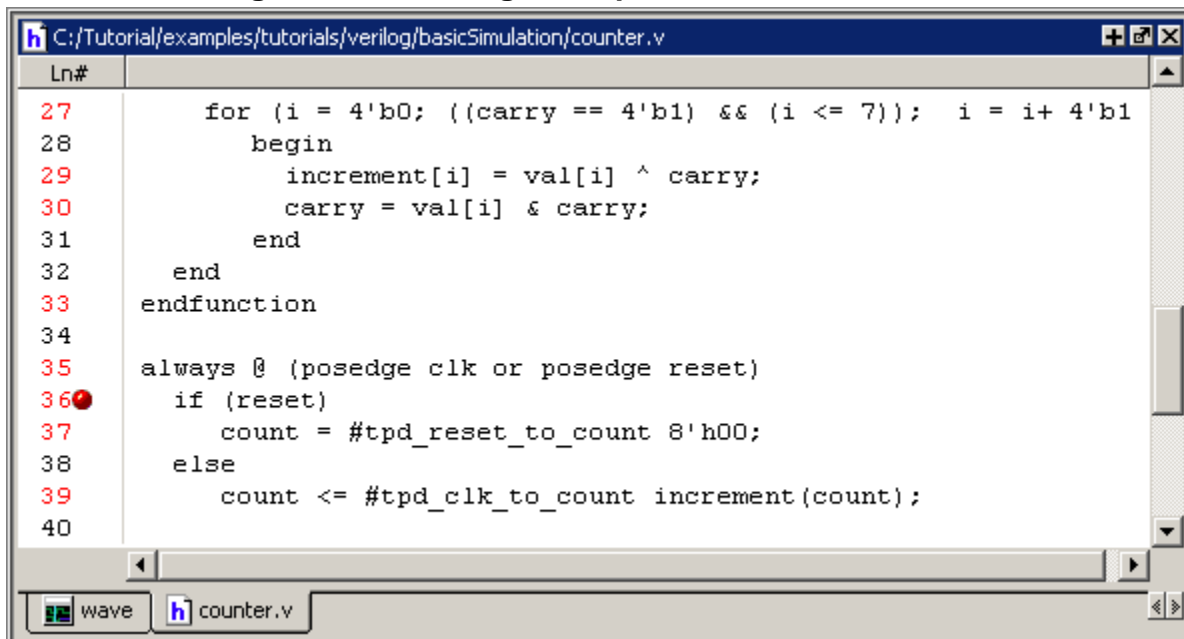
Set Breakpoints and Step through the Source

Next you will take a brief look at one interactive debugging feature of the QuestaSim environment. You will set a breakpoint in the Source window, run the simulation, and then step through the design under test. Breakpoints can be set only on lines with red line numbers.

1. Open *counter.v* in the Source window.
 - a. Select the **Files** tab in the Main window Workspace.
 - b. Click the + sign next to the *sim* filename to see the contents of *vsim.wlf* dataset.
 - c. Double-click *counter.v* (or *counter.vhd* if you are simulating the VHDL files) to open it in the Source window.
2. Set a breakpoint on line 36 of *counter.v* (or, line 39 of *counter.vhd* for VHDL).
 - a. Scroll to line 36 and click in the BP (breakpoint) column next to the line number.

A red ball appears in the line number column at line number 36 (Figure 3-10), indicating that a breakpoint has been set.

Figure 3-10. Setting Breakpoint in Source Window

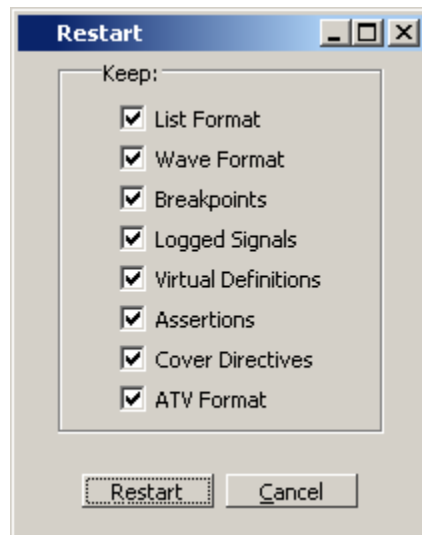


3. Disable, enable, and delete the breakpoint.
 - a. Click the red ball to disable the breakpoint. It will become a black ball.
 - b. Click the black ball again to re-enable the breakpoint. It will become a red ball.
 - c. Click the red ball with your right mouse button and select **Remove Breakpoint 36**.
 - d. Click in the line number column next to line number 36 again to re-create the breakpoint.
4. Restart the simulation.
 - a. Click the Restart icon to reload the design elements and reset the simulation time to zero.



The Restart dialog that appears gives you options on what to retain during the restart (Figure 3-11).

Figure 3-11. Setting Restart Functions

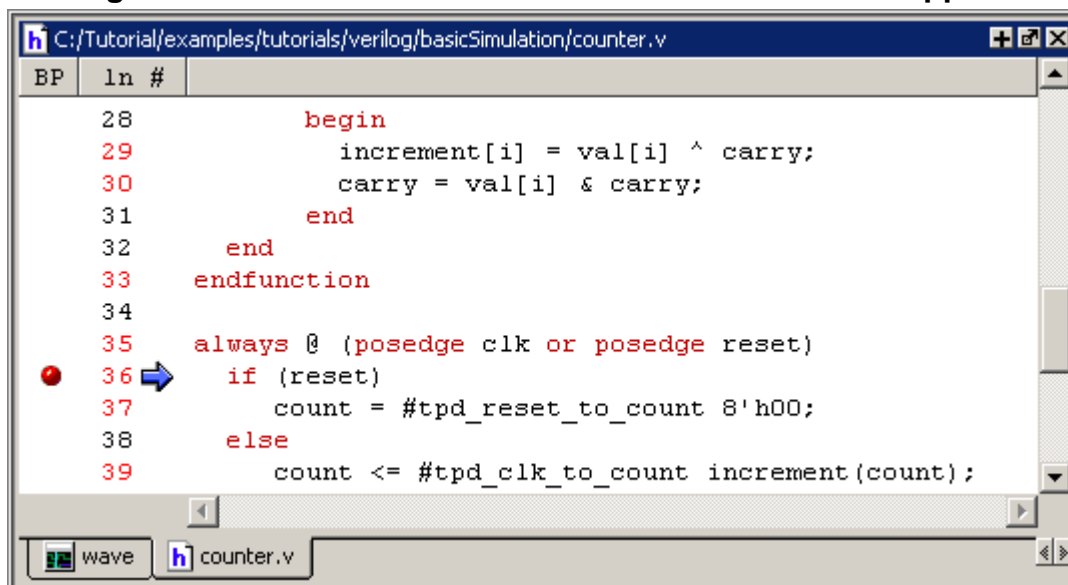


- b. Click the **Restart** button in the Restart dialog.
- c. Click the Run -All icon.

The simulation runs until the breakpoint is hit. When the simulation hits the breakpoint, it stops running, highlights the line with a blue arrow in the Source view (Figure 3-12), and issues a Break message in the Transcript pane.



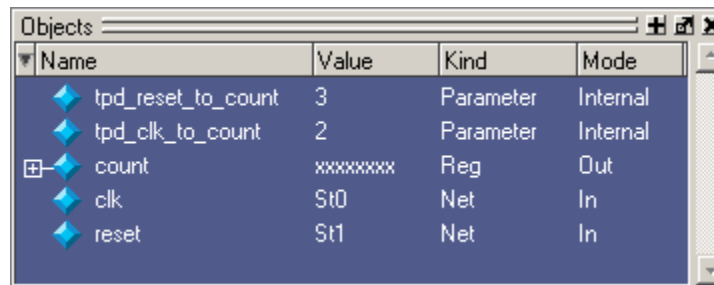
Figure 3-12. Blue Arrow Indicates Where Simulation Stopped.



When a breakpoint is reached, typically you want to know one or more signal values. You have several options for checking values:

- look at the values shown in the Objects window (Figure 3-13).

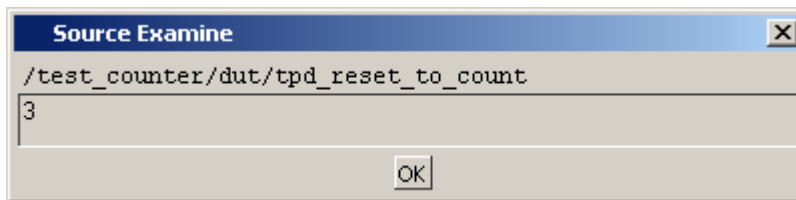
Figure 3-13. Values Shown in Objects Window



Name	Value	Kind	Mode
tpd_reset_to_count	3	Parameter	Internal
tpd_clk_to_count	2	Parameter	Internal
count	xxxxxxxx	Reg	Out
clk	St0	Net	In
reset	St1	Net	In

- set your mouse pointer over a variable in the Source window and a yellow box will appear with the variable name and the value of that variable at the time of the selected cursor in the Wave window
- highlight a signal, parameter, or variable in the Source window, right-click it, and select **Examine** from the pop-up menu to display the variable and its current value in a Source Examine window (Figure 3-14)

Figure 3-14. Parameter Name and Value in Source Examine Window



- use the **examine** command at the VSIM> prompt to output a variable value to the Main window Transcript (i.e., `examine count`)
5. Try out the step commands.
- a. Click the Step icon on the Main window toolbar.

This single-steps the debugger.

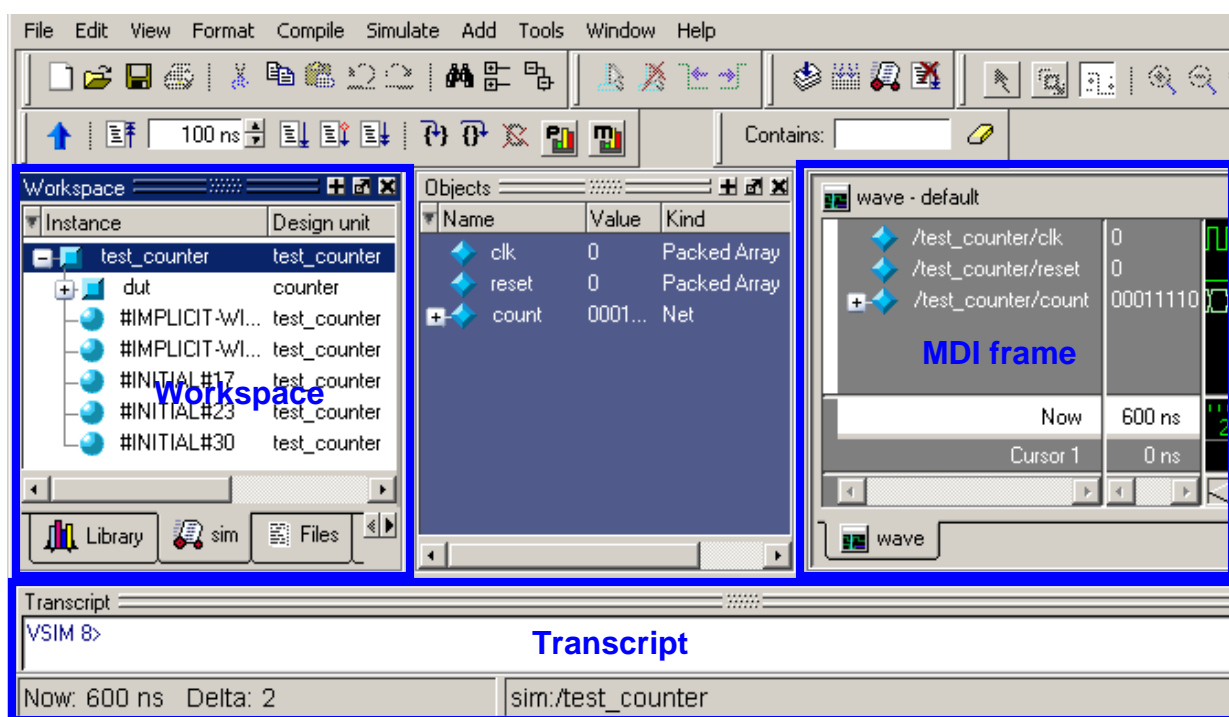


Experiment on your own. Set and clear breakpoints and use the Step, Step Over, and Continue Run commands until you feel comfortable with their operation.

Navigating the Interface

The Main window is composed of a number of "panes" and sub-windows that display various types of information about your design, simulation, or debugging session. You can also access other tools from the Main window that display in stand-alone windows (e.g., the Dataflow window).

Figure 3-15. The Main Window



The following table describes some of the key elements of the Main window.

Table 3-1. The Main Window

Window/pane	Description
Workspace	This pane comprises multiple tabs that contain various sorts of information about the current project or design. Once a design is loaded, additional tabs will appear. Refer to the section Workspace in the User's Manual for more information.
Transcript	The Transcript pane provides a command-line interface and serves as an activity log including status and error messages. Refer to the section Transcript Window in the User's Manual for more information.

Table 3-1. The Main Window

Window/pane	Description
MDI frame	The Multiple Document Interface (MDI) frame holds windows for which there can be multiple instances. These include Source editor windows, Wave windows, and Memory content windows. Refer to the section Multiple Document Interface (MDI) Frame in the User's Manual for more information.

Here are a few important points to keep in mind about the QuestaSim interface:

- Windows/panes can be resized, moved, zoomed, undocked, etc. and the changes are persistent.

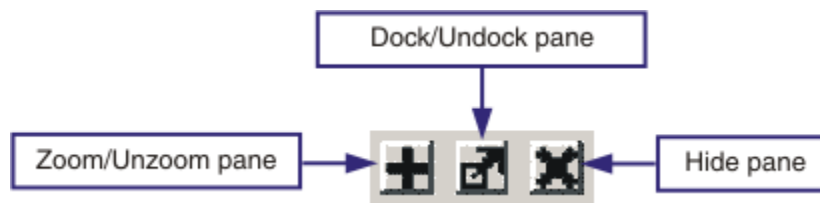
You have a number of options for re-sizing, re-positioning, undocking/redocking, and generally modifying the physical characteristics of windows and panes. When you exit QuestaSim, the current layout is saved so that it appears the same the next time you invoke the tool. Refer to the [Main Window](#) section in the User's Manual for more information.

- Menus are context sensitive.

The menu items that are available and how certain menu items behave depend on which pane or window is active. For example, if the *sim* tab in the Workspace is active and you choose Edit from the menu bar, the Clear command is disabled. However, if you click in the Transcript pane and choose Edit, the Clear command is enabled. The active pane is denoted by a blue title bar.

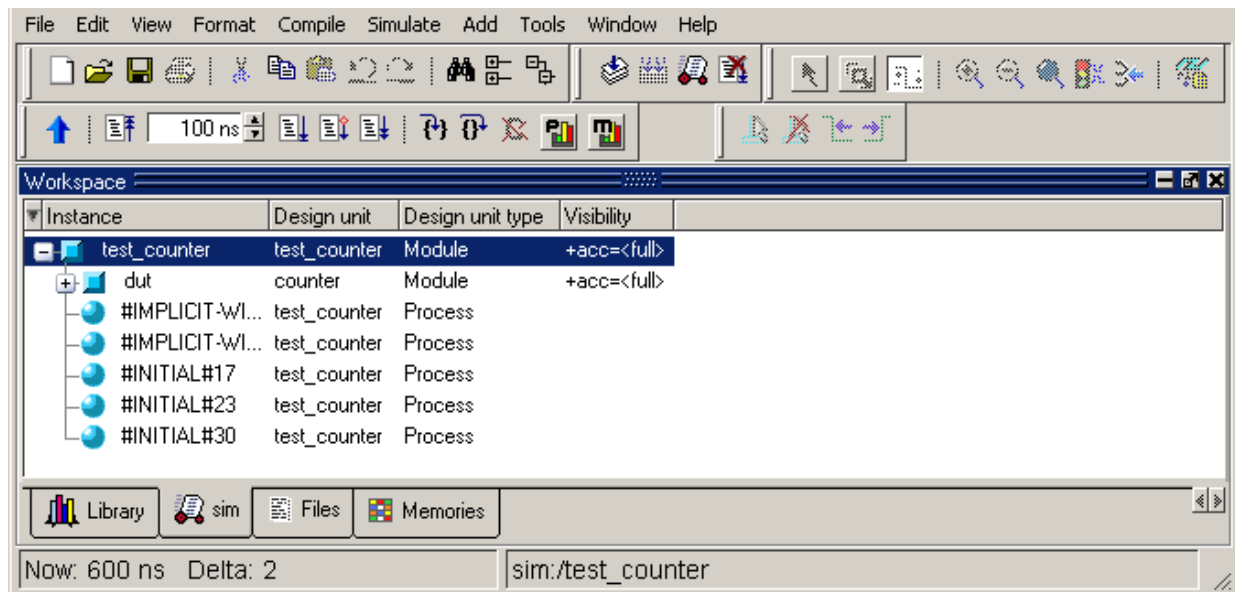
Let us try a few things.


1. Zoom and undock panes.
 - a. Click the Zoom/Unzoom icon in the upper right corner of the Workspace pane ([Figure 3-16](#)).

Figure 3-16. Window/Pane Control Icons

The pane fills the entire Main window ([Figure 3-17](#)).

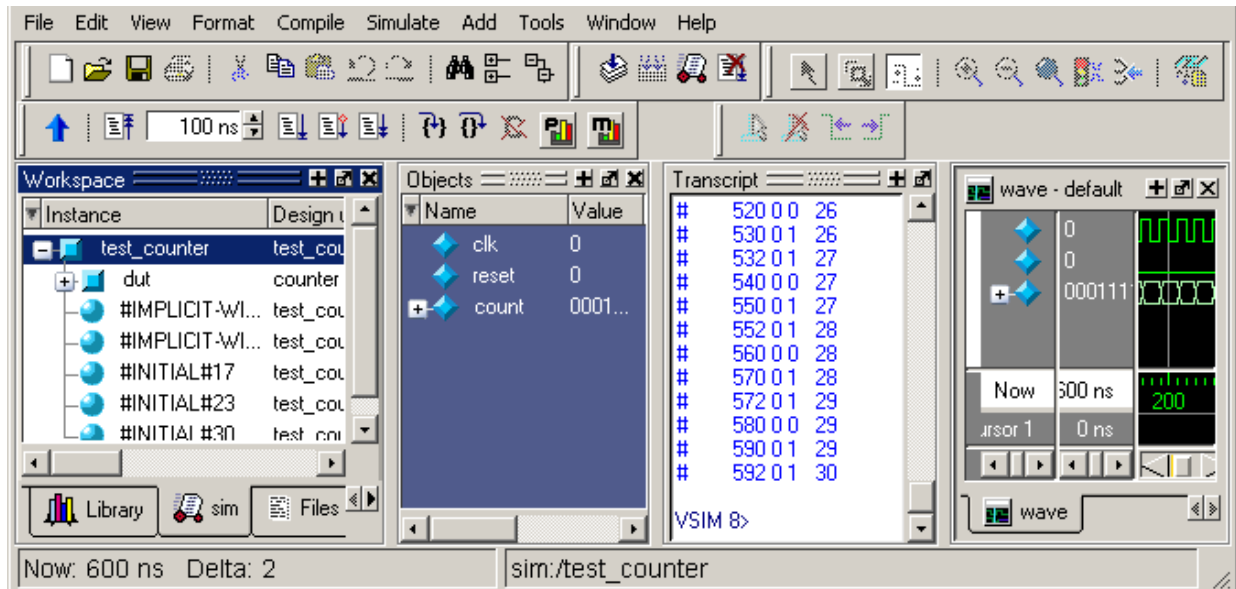
Figure 3-17. zooming in on Workspace Pane



- b. Click the **Zoom/Unzoom pane** icon in the Workspace.
 - c. Click the **Dock/Undock pane** icon in the upper right corner of the Transcript pane.
The Transcript becomes a stand-alone window.
 - d. Click the **Dock/Undock pane** icon on the Transcript.
 - e. Click the **Hide pane** icon in the Workspace.
 - f. Select **View > Workspace** from the menus to re-open the Workspace.
2. Move and resize panes.
 - a. Hover your mouse pointer in the center of the Transcript title bar, where the two parallel lines are interrupted by 3 lines of small dots. This is the handle for the pane. When the cursor is over the pane handle it becomes a four-headed arrow. 
 - b. Click and drag the Transcript up and to the right until you see a gray outline on the right-hand side of the MDI frame.

When you let go of the mouse button, the Transcript is moved and the MDI frame and Workspace panes shift to the left ([Figure 3-18](#)).

Figure 3-18. Panes Rearranged in Main Window



- c. Select **Layout > Reset**.

The layout returns to its original setting.



Tip: Moving panes can get confusing, and you may not always obtain the results you expect. Practice moving a pane around, watching the gray outline to see what happens when you drop it in various places. Your layout will be saved when you exit QuestaSim and will reappear in the last configuration when you next open QuestaSim. (It's a good idea to close all panes in the MDI frame at the end of each lesson in this tutorial so only files relevant to each lesson will be displayed.)

As you practice, notice that the MDI frame cannot be moved in the same manner as the panes. It does not have a handle in its header bar.

Selecting **Layout > Reset** is the easiest way to rectify an undesired layout.

- d. Hover your mouse pointer on the border between two panes so it becomes a double-headed arrow. \longleftrightarrow
 - e. Click-and-drag left and right or up and down to resize the pane.
 - f. Select **Layout > Reset**.
3. Observe context sensitivity of menu commands.
 - a. Click anywhere in the Workspace.
 - b. Select the Edit menu and notice that the **Clear** command is disabled.

- c. Click in the Transcript and select **Edit > Clear**.

This command applies to the Transcript pane but not the Workspace pane.

- d. Click on a design object in the sim tab of the Workspace and select **File > Open**.
- e. Notice that the Open dialog filters to show Log files (*.wlf).
- f. Now click on a filename in the Files tab of the Workspace and select **File > Open**.

Notice that the Open dialog filters to show HDL file types instead.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. Select **Simulate > End Simulation**.
2. Click **Yes** when prompted to confirm that you wish to quit simulating.

Introduction

In this lesson you will practice creating a project.

At a minimum, projects contain a work library and a session state that is stored in a *.mpf* file. A project may also consist of:

- HDL source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries

Design Files for this Lesson

The sample design for this lesson is a simple 8-bit, binary up-counter with an associated testbench. The pathnames are as follows:

Verilog – `<install_dir>/examples/tutorials/verilog/projects/counter.v` and `tcounter.v`

VHDL – `<install_dir>/examples/tutorials/vhdl/projects/counter.vhd` and `tcounter.vhd`

This lesson uses the Verilog files *tcounter.v* and *counter.v*. If you have a VHDL license, use *tcounter.vhd* and *counter.vhd* instead.

Related Reading

User's Manual Chapter: [Projects](#).

Create a New Project

1. Create a new directory and copy the design files for this lesson into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons).

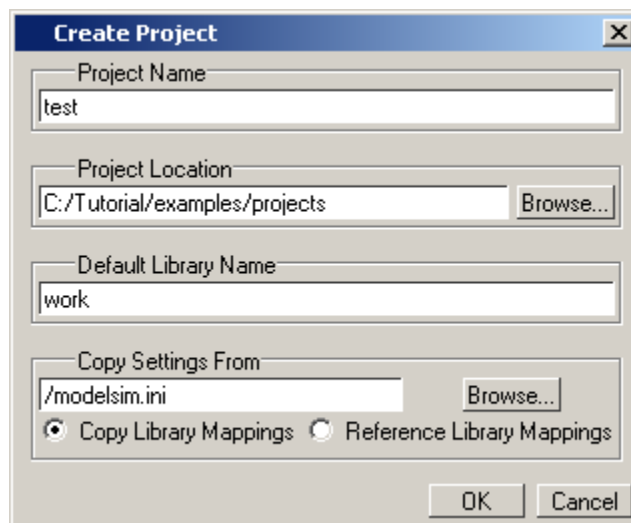
Verilog: Copy *counter.v* and *tcounter.v* files from
`/<install_dir>/examples/tutorials/verilog/projects` to the new directory.

VHDL: Copy *counter.vhd* and *tcounter.vhd* files from
`/<install_dir>/examples/tutorials/vhdl/projects` to the new directory.

2. If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.
 - a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.
 - b. Select **File > Change Directory** and change to the directory you created in step 1.
3. Create a new project.
 - a. Select **File > New > Project** (Main window) from the menu bar.

This opens the Create Project dialog where you can enter a Project Name, Project Location (i.e., directory), and Default Library Name (Figure 4-1). You can also reference library settings from a selected .ini file or copy them directly into the project. The default library is where compiled design units will reside.
 - b. Type **test** in the Project Name field.
 - c. Click the **Browse** button for the Project Location field to select a directory where the project file will be stored.
 - d. Leave the Default Library Name set to *work*.
 - e. Click **OK**.

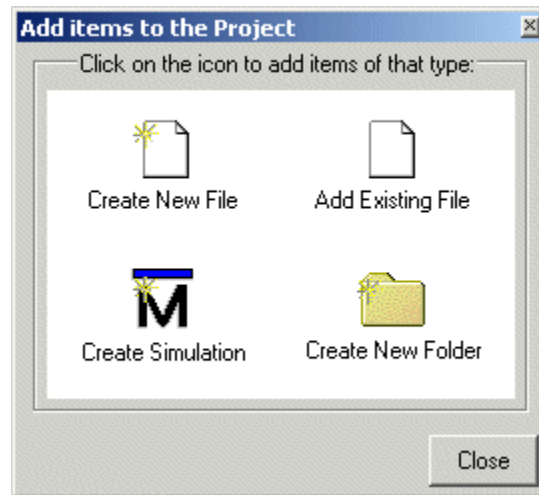
Figure 4-1. Create Project Dialog - Project Lab



Add Objects to the Project

Once you click OK to accept the new project settings, you will see a blank Project tab in the Workspace area of the Main window and the Add items to the Project dialog will appear (Figure 4-2). From this dialog you can create a new design file, add an existing file, add a folder for organization purposes, or create a simulation configuration (discussed below).

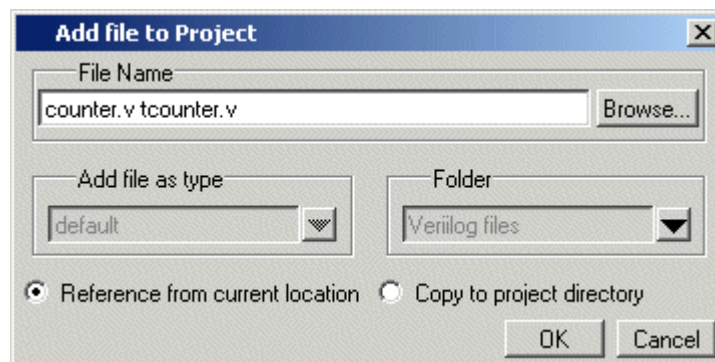
Figure 4-2. Adding New Items to a Project



1. Add two existing files.
 - a. Click **Add Existing File**.

This opens the Add file to Project dialog (Figure 4-3). This dialog lets you browse to find files, specify the file type, specify a folder to which the file will be added, and identify whether to leave the file in its current location or to copy it to the project directory.

Figure 4-3. Add file to Project Dialog



- b. Click the **Browse** button for the File Name field. This opens the “Select files to add to project” dialog and displays the contents of the current directory.
 - c. **Verilog:** Select *counter.v* and *tcounter.v* and click **Open**.
VHDL: Select *counter.vhd* and *tcounter.vhd* and click **Open**.

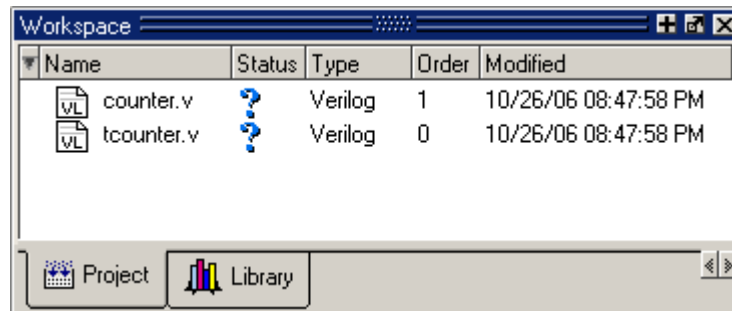
This closes the “Select files to add to project” dialog and displays the selected files in the “Add file to Project” dialog (Figure 4-3).

- d. Click **OK** to add the files to the project.

- e. Click **Close** to dismiss the Add items to the Project dialog.

You should now see two files listed in the Project tab of the Workspace pane (Figure 4-4). Question mark icons (?) in the Status column indicate that the file has not been compiled or that the source file has changed since the last successful compile. The other columns identify file type (e.g., Verilog or VHDL), compilation order, and modified date.

Figure 4-4. Newly Added Project Files Display a “?” for Status



Changing Compile Order (VHDL)

By default QuestaSim performs default binding of VHDL designs when you load the design with `vsim`. However, you can elect to perform default binding at compile time. (For details, refer to the section [Default Binding](#) in the User's Manual.) If you elect to do default binding at compile, then the compile order is important. Follow these steps to change compilation order within a project.

1. Change the compile order.
 - a. Select **Compile > Compile Order**.

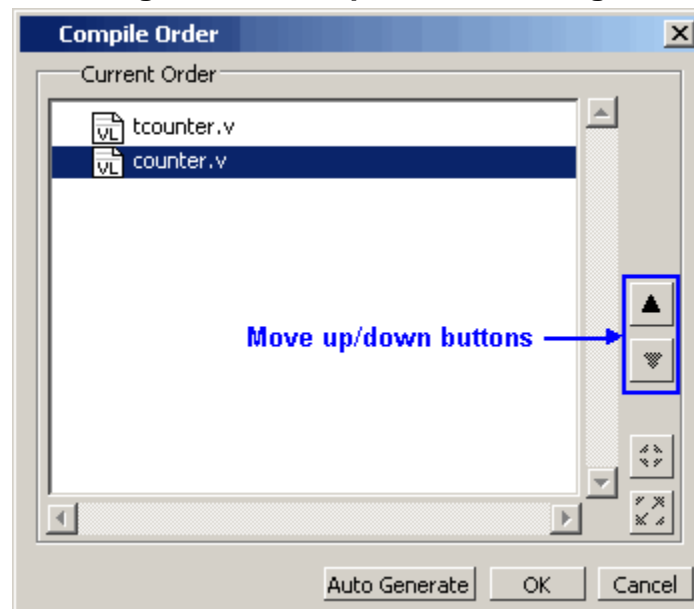
This opens the Compile Order dialog box.

- b. Click the **Auto Generate** button.

QuestaSim "determines" the compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling the rest of the files. It continues in this manner until all files compile successfully or until a file(s) can't be compiled for reasons other than dependency.

Alternatively, you can select a file and use the Move Up and Move Down buttons to put the files in the correct order (Figure 4-5).

Figure 4-5. Compile Order Dialog



- c. Click **OK** to close the Compile Order dialog.

Compile the Design

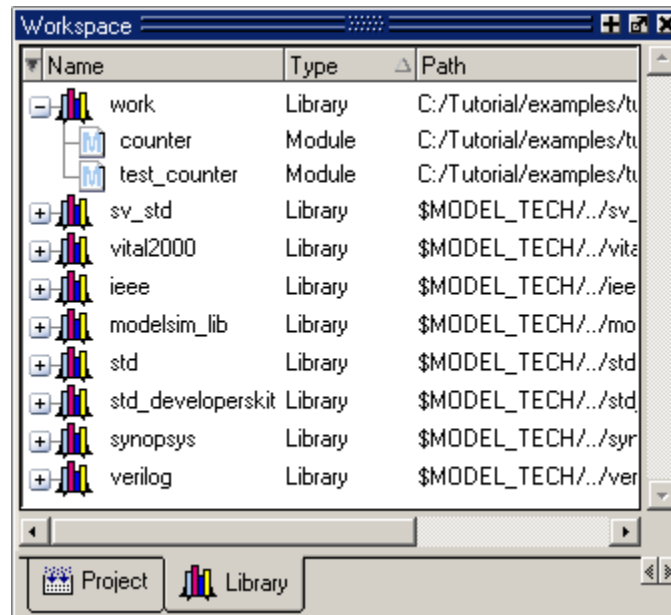
1. Compile the files.
 - a. Right-click either *counter.v* or *tcounter.v* in the Project tab and select **Compile > Compile All** from the pop-up menu.

QuestaSim compiles both files and changes the symbol in the Status column to a green check mark. A check mark means the compile succeeded. If compile fails, the symbol will be a red 'X', and you will see an error message in the Transcript pane.

2. View the design units.
 - a. Click the **Library** tab in the workspace ([Figure 4-6](#)).
 - b. Click the "+" icon next to the *work* library.

You should see two compiled design units, their types (modules in this case), and the path to the underlying source files.

Figure 4-6. Library Tab with Expanded Library



Load the Design

1. Load the *test_counter* design unit.
 - a. Enter the following command at the QuestaSim> prompt in the Transcript pane.

```
vsim -voptargs="+acc" test_counter
```

The `-voptargs="+acc"` argument for the `vsim` command provides visibility into the design for debugging purposes.

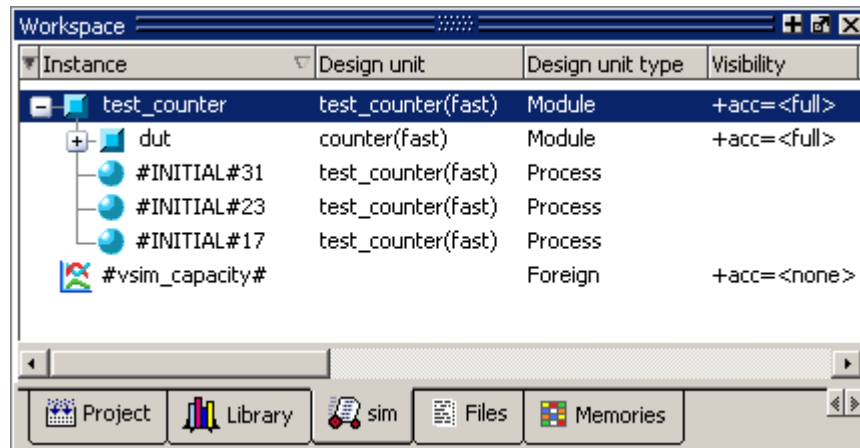
Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

You should see 3 new tabs in the Main window Workspace. The *sim* tab displays the structure of the *test_counter* design unit (Figure 4-7). The *Files* tab contains information about the underlying source files. The *Memories* tab lists all memories in the design.

Figure 4-7. Structure Tab for a Loaded Design



At this point you would typically run the simulation and analyze or debug your design like you did in the previous lesson. For now, you'll continue working with the project. However, first you need to end the simulation that started when you loaded *test_counter*.

2. End the simulation.
 - a. Select **Simulate > End Simulation**.
 - b. Click **Yes**.

Organizing Projects with Folders

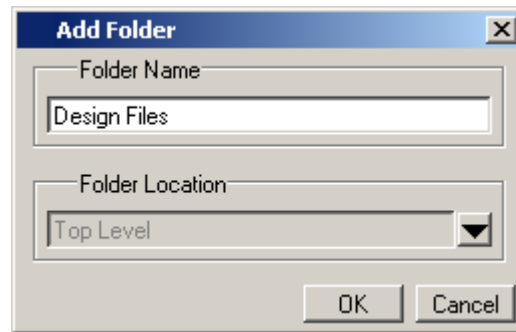
If you have a lot of files to add to a project, you may want to organize them in folders. You can create folders either before or after adding your files. If you create a folder before adding files, you can specify in which folder you want a file placed at the time you add the file (see Folder field in [Figure 4-3](#)). If you create a folder after adding files, you edit the file properties to move it to that folder.

Add Folders

As shown previously in [Figure 4-2](#), the Add items to the Project dialog has an option for adding folders. If you have already closed that dialog, you can use a menu command to add a folder.

1. Add a new folder.
 - a. Right-click inside the Projects tab of the Workspace and select **Add to Project > Folder**.
 - b. Type **Design Files** in the **Folder Name** field ([Figure 4-8](#)).

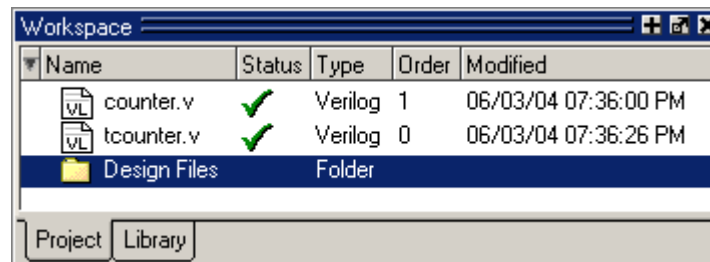
Figure 4-8. Adding New Folder to Project



- c. Click **OK**.

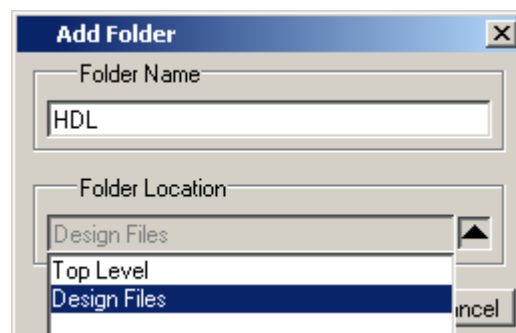
The new Design Files folder is displayed in the Project tab ([Figure 4-9](#)).

Figure 4-9. A Folder Within a Project



2. Add a sub-folder.
- Right-click anywhere in the Project tab and select **Add to Project > Folder**.
 - Type **HDL** in the **Folder Name** field ([Figure 4-10](#)).

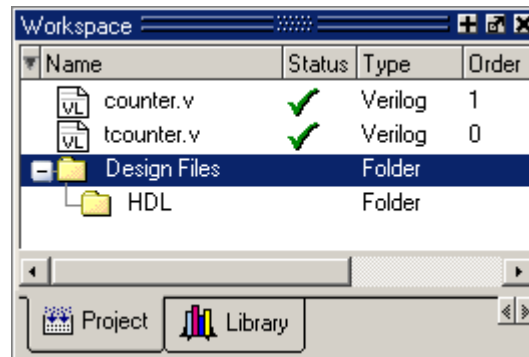
Figure 4-10. Creating Subfolder



- Click the **Folder Location** drop-down arrow and select *Design Files*.
- Click **OK**.

A '+' icon appears next to the *Design Files* folder in the Project tab ([Figure 4-11](#)).

Figure 4-11. A folder with a Sub-folder



- e. Click the '+' icon to see the *HDL* sub-folder.

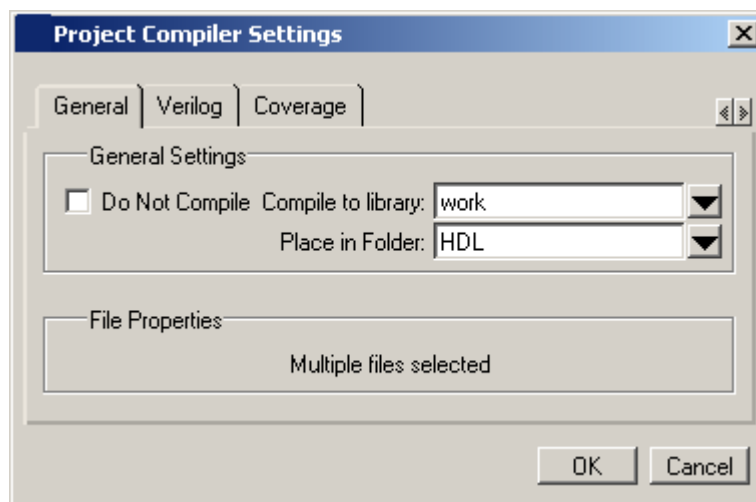
Moving Files to Folders

If you don't place files into a folder when you first add the files to the project, you can move them into a folder using the properties dialog.

1. Move *tcounter.v* and *counter.v* to the *HDL* folder.
 - a. Select both *counter.v* and *tcounter.v* in the Project tab of the Workspace.
 - b. Right-click either file and select **Properties**.

This opens the Project Compiler Settings dialog (Figure 4-12), which allows you to set a variety of options on your design files.

Figure 4-12. Changing File Location via the Project Compiler Settings Dialog



- c. Click the **Place In Folder** drop-down arrow and select *HDL*.
- d. Click **OK**.

The selected files are moved into the HDL folder. Click the '+' icon next to the HDL folder to see the files.

The files are now marked with a '?' in the Status column because you moved the files. The project no longer knows if the previous compilation is still valid.

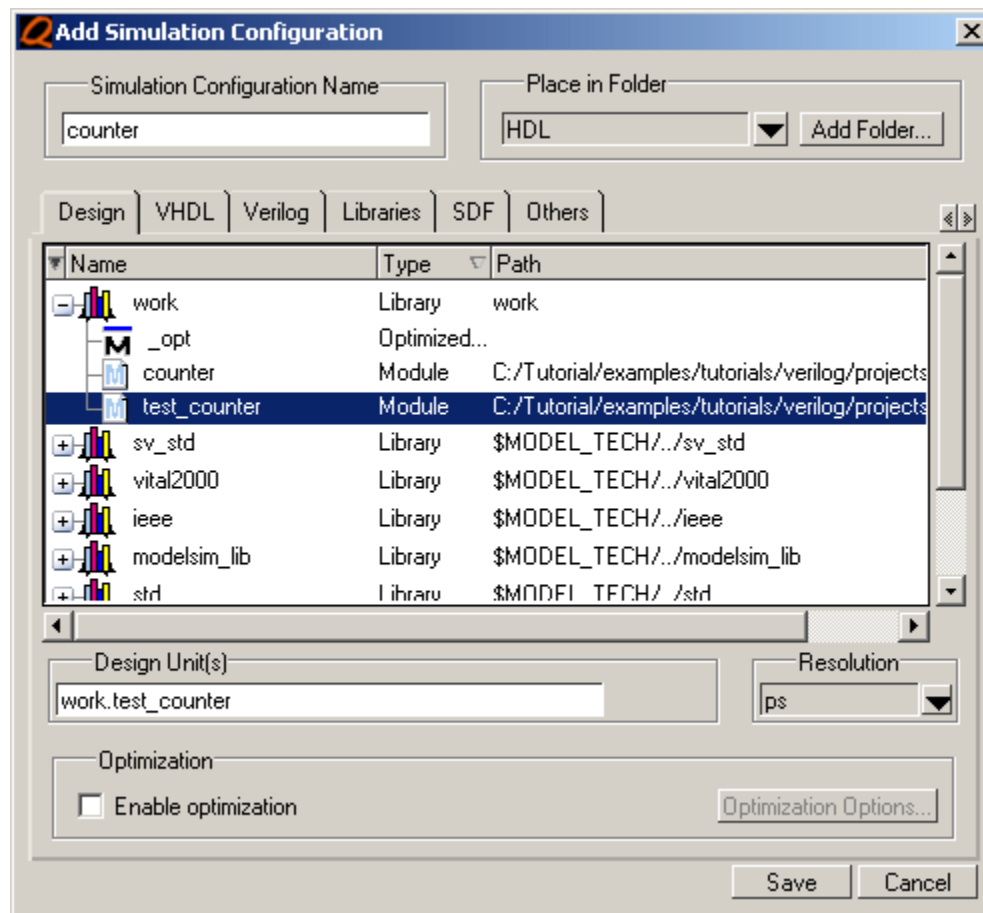
Simulation Configurations

A Simulation Configuration associates a design unit(s) and its simulation options. For example, let's say that every time you load *tcounter.v* you want to set the simulator resolution to picoseconds (ps) and enable event order hazard checking. Ordinarily, you would have to specify those options each time you load the design. With a Simulation Configuration, you specify options for a design and then save a "configuration" that associates the design and its options. The configuration is then listed in the Project tab and you can double-click it to load *tcounter.v* along with its options.

1. Create a new Simulation Configuration.
 - a. Right-click in the Projects tab and select **Add to Project > Simulation Configuration** from the popup menu.

This opens the Add Simulation Configuration dialog ([Figure 4-13](#)). The tabs in this dialog present a myriad of simulation options. You may want to explore the tabs to see what is available. You can consult the QuestaSim User's Manual to get a description of each option.

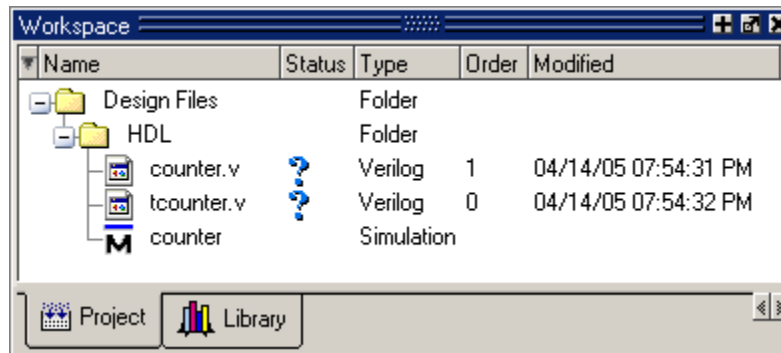
Figure 4-13. Simulation Configuration Dialog



- Type **counter** in the **Simulation Configuration Name** field.
- Select **HDL** from the **Place in Folder** drop-down.
- Click the '+' icon next to the *work* library and select *test_counter*.
- Click the **Resolution** drop-down and select *ps*.
- Uncheck the **Enable optimization** selection box.
- For Verilog, click the Verilog tab and check **Enable hazard checking (-hazards)**.
- Click **Save**.

The Project tab now shows a Simulation Configuration named *counter* in the HDL folder (Figure 4-14).

Figure 4-14. A Simulation Configuration in the Project Tab

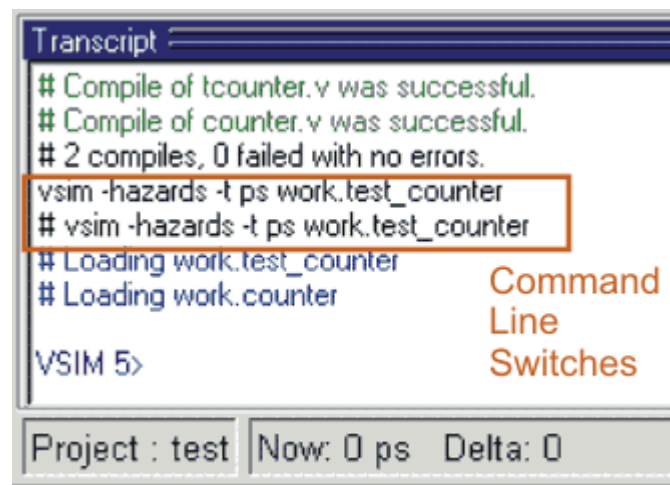


2. Load the Simulation Configuration.

a. Double-click the *counter* Simulation Configuration in the Project tab.

In the Transcript pane of the Main window, the **vsim** (the QuestaSim simulator) invocation shows the **-hazards** and **-t ps** switches (Figure 4-15). These are the command-line equivalents of the options you specified in the Simulate dialog.

Figure 4-15. Transcript Shows Options for Simulation Configurations



Lesson Wrap-Up

This concludes this lesson. Before continuing you need to end the current simulation and close the current project.

1. Select **Simulate > End Simulation**. Click Yes.
2. Select the Project tab in the Main window Workspace.
3. Right-click in this tab to open a popup menu and select **Close Project**.
4. Click **OK**.

If you do not close the project, it will open automatically the next time you start QuestaSim.

Chapter 5

Working With Multiple Libraries

Introduction

In this lesson you will practice working with multiple libraries. You might have multiple libraries to organize your design, to access IP from a third-party source, or to share common parts between simulations.

You will start the lesson by creating a resource library that contains the *counter* design unit. Next, you will create a project and compile the testbench into it. Finally, you will link to the library containing the counter and then run the simulation.

Design Files for this Lesson

The sample design for this lesson is a simple 8-bit, binary up-counter with an associated testbench. The pathnames are as follows:

Verilog – `<install_dir>/examples/tutorials/verilog/libraries/counter.v` and `tcounter.v`

VHDL – `<install_dir>/examples/tutorials/vhdl/libraries/counter.vhd` and `tcounter.vhd`

This lesson uses the Verilog files *tcounter.v* and *counter.v* in the examples. If you have a VHDL license, use *tcounter.vhd* and *counter.vhd* instead.

Related Reading

User's Manual Chapter: [Design Libraries](#).

Creating the Resource Library

Before creating the resource library, make sure the *modelsim.ini* in your install directory is “Read Only.” This will prevent permanent mapping of resource libraries to the master *modelsim.ini* file. See [Permanently Mapping VHDL Resource Libraries](#).

1. Create a directory for the resource library.

Create a new directory called *resource_library*. Copy *counter.v* from `<install_dir>/examples/tutorials/verilog/libraries` to the new directory.

2. Create a directory for the testbench.

Create a new directory called *testbench* that will hold the testbench and project files. Copy *tcounter.v* from `<install_dir>/examples/tutorials/verilog/libraries` to the new directory.

You are creating two directories in this lesson to mimic the situation where you receive a resource library from a third-party. As noted earlier, we will link to the resource library in the first directory later in the lesson.

3. Start QuestaSim and change to the *resource_library* directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

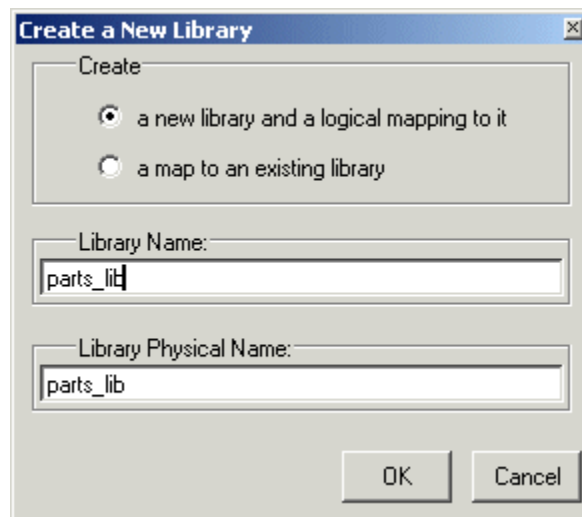
If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the *resource_library* directory you created in step 1.

4. Create the resource library.

- a. Select **File > New > Library**.
 - b. Type **parts_lib** in the Library Name field (Figure 5-1).

Figure 5-1. Creating New Resource Library



The Library Physical Name field is filled out automatically.

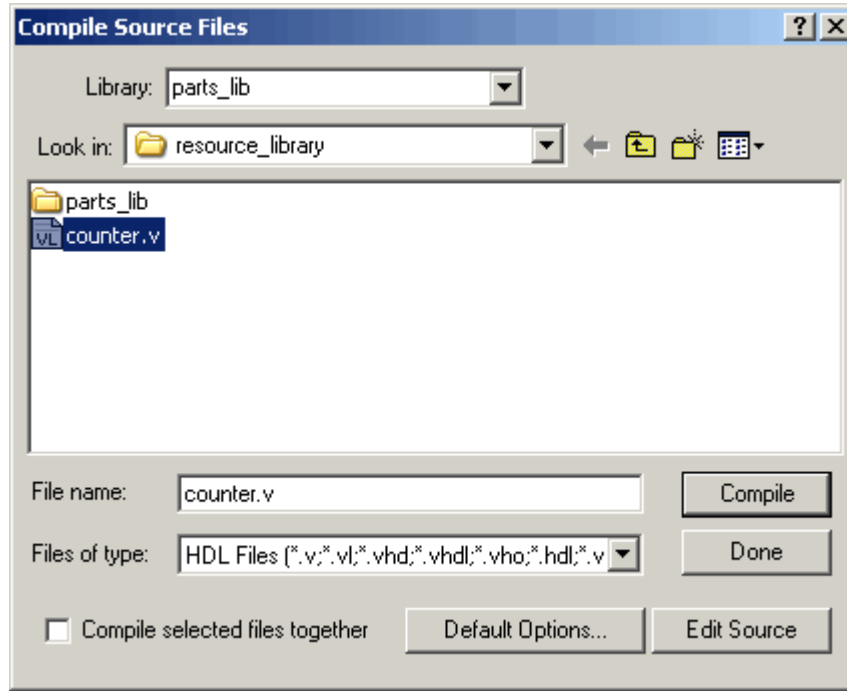
Once you click OK, QuestaSim creates a directory for the library, lists it in the Library tab of the Workspace, and modifies the *modelsim.ini* file to record this new library for the future.

5. Compile the counter into the resource library.

- a. Click the Compile icon on the Main window toolbar.
- b. Select the *parts_lib* library from the Library list (Figure 5-2).



Figure 5-2. Compiling into the Resource Library



- c. Double-click *counter.v* to compile it.
- d. Click **Done**.

You now have a resource library containing a compiled version of the *counter* design unit.

6. Change to the *testbench* directory.
 - a. Select **File > Change Directory** and change to the *testbench* directory you created in step 2.

Creating the Project

Now you will create a project that contains *tcounter.v*, the counter's testbench.

1. Create the project.
 - a. Select **File > New > Project**.
 - b. Type **counter** in the Project Name field.
 - c. Do not change the Project Location field or the Default Library Name field. (The default library name is *work*.)

- d. Make sure “Copy Library Mappings” is selected. The default *modelsim.ini* file will be used.
 - e. Click **OK**.
 2. Add the testbench to the project.
 - a. Click **Add Existing File** in the Add items to the Project dialog.
 - b. Click the **Browse** button and select *tcounter.v* in the “Select files to add to project” dialog.
 - c. Click **Open**.
 - d. Click **OK**.
 - e. Click **Close** to dismiss the “Add items to the Project” dialog.

The *tcounter.v* file is listed in the Project tab of the Main window.
 3. Compile the testbench.
 - a. Right-click *tcounter.v* and select **Compile > Compile Selected**.

Linking to the Resource Library

To wrap up this part of the lesson, you will link to the *parts_lib* library you created earlier. But first, try simulating the testbench without the link and see what happens.

QuestaSim responds differently for Verilog and VHDL in this situation.

Verilog

1. Simulate a Verilog design with a missing resource library.
 - a. Enter the following command at the QuestaSim> prompt in the Transcript pane.

```
vsim -voptargs="+acc" test_counter
```

The `-voptargs="+acc"` argument for the `vsim` command provides visibility into the design for debugging purposes.

Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

The Main window Transcript reports an error loading the design because the *counter* module is not defined.

VHDL

1. Simulate a VHDL design with a missing resource library.
 - a. Enter the following command at the QuestaSim> prompt in the Transcript pane.

```
vsim -voptargs="+acc" test_counter
```

The -voptargs="+acc" argument for the **vsim** command provides visibility into the design for debugging purposes.

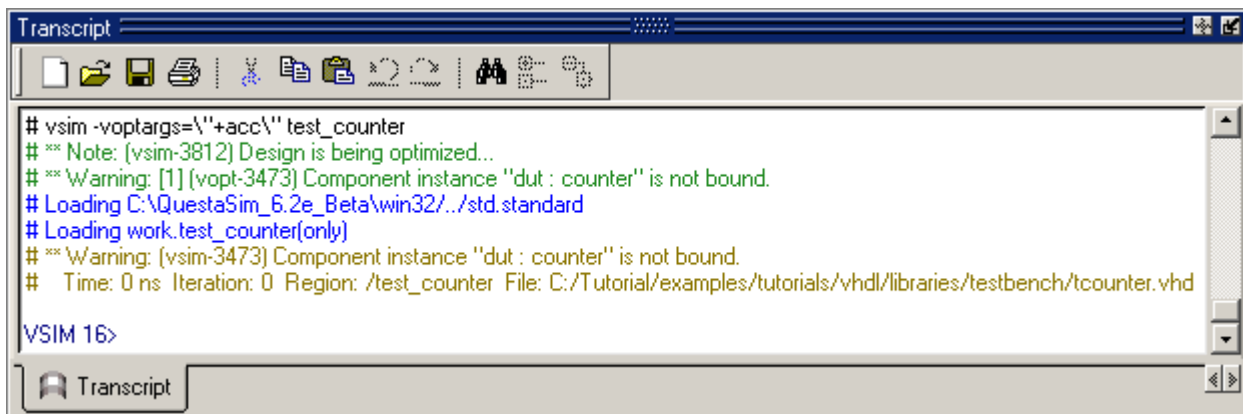
Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

The Main window Transcript reports a warning ([Figure 5-3](#)). When you see a message that contains text like "Warning: (vsim-3473)", you can view more detail by using the **error** command.

Figure 5-3. VHDL Simulation Warning Reported in Main Window



- b. Type **error 3473** at the VSIM> prompt.

The expanded error message tells you that a component ('dut' in this case) has not been explicitly bound and no default binding can be found.


- c. Type **quit -sim** to quit the simulation.

The process for linking to a resource library differs between Verilog and VHDL. If you are using Verilog, follow the steps in [Linking in Verilog](#). If you are using VHDL, follow the steps in [Linking in VHDL](#) one page later.

Linking in Verilog

Linking in Verilog requires that you specify a "search library" when you invoke the simulator.

1. Specify a search library during simulation.

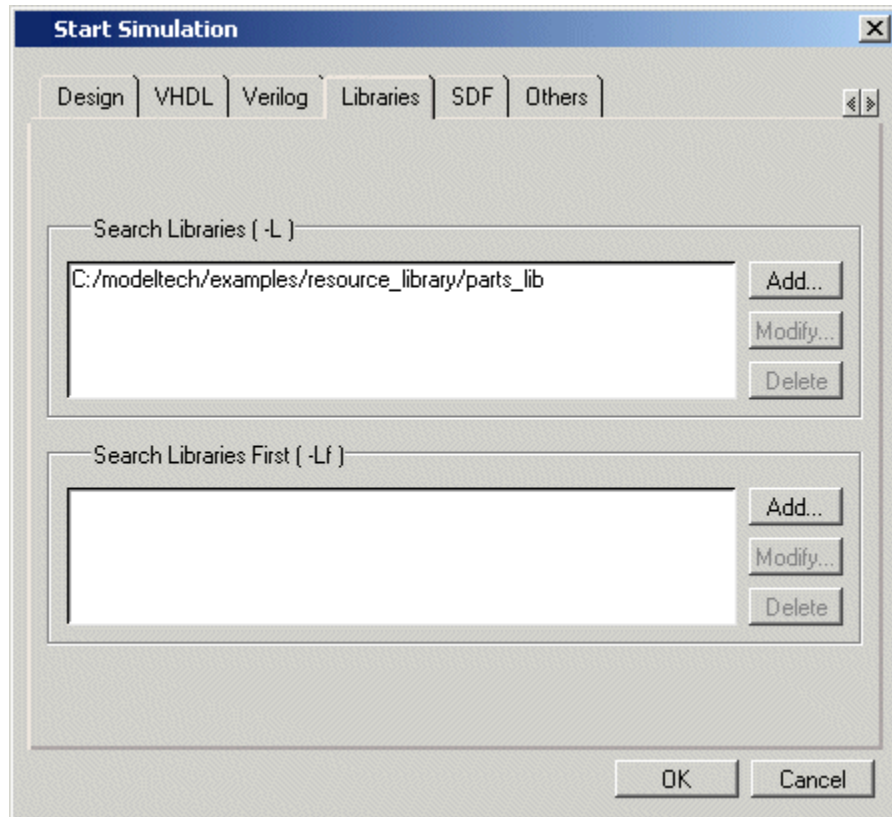
- a. Click the Simulate icon on the Main window toolbar. 
- b. Click the '+' icon next to the *work* library and select *test_counter*.
- c. Uncheck the Enable optimization selection box.
- d. Click the Libraries tab.
- e. Click the Add button next to the Search Libraries field and browse to *parts_lib* in the *resource_library* directory you created earlier in the lesson.
- f. Click OK.

The dialog should have *parts_lib* listed in the Search Libraries field ([Figure 5-4](#)).

- g. Click OK.

The design loads without errors.

Figure 5-4. Specifying a Search Library in the Simulate Dialog



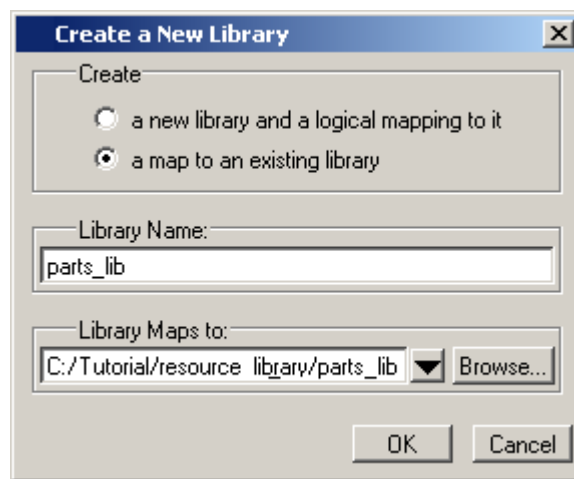
Linking in VHDL

To link to a resource library in VHDL, you have to create a logical mapping to the physical library and then add `LIBRARY` and `USE` statements to the source file.

1. Create a logical mapping to *parts_lib*.

- a. Select **File > New > Library**.
- b. In the Create a New Library dialog, select **a map to an existing library**.
- c. Type **parts_lib** in the Library Name field.
- d. Click Browse to open the Browse for Folder dialog and browse to *parts_lib* in the *resource_library* directory you created earlier in the lesson.
- e. Click OK to select the library and close the Select Library dialog.
- f. The Create a New Library dialog should look similar to the one shown in [Figure 5-5](#). Click **OK** to close the dialog.

Figure 5-5. Mapping to the parts_lib Library



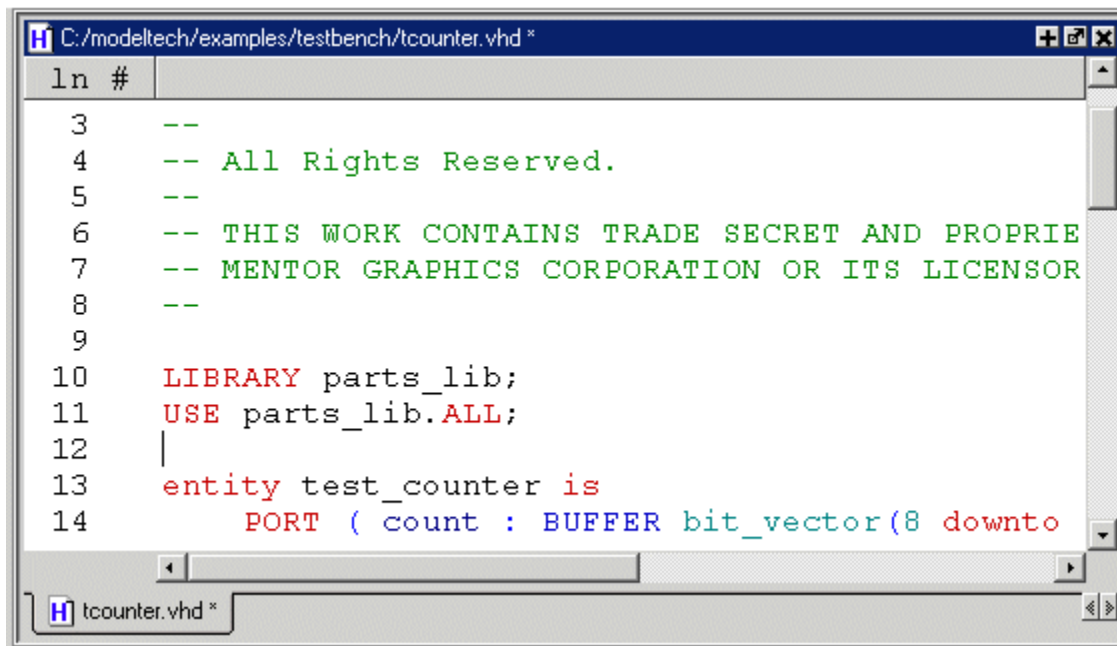
2. Add **LIBRARY** and **USE** statements to *tcounter.vhd*.
 - a. In the Library tab of the Main window, click the '+' icon next to the *work* library.
 - b. Right-click *test_counter* in the work library and select **Edit**.
 - c. This opens the file in the Source window.
 - d. Right-click in the Source window and uncheck **Read Only**.
 - e. Add these two lines to the top of the file:

```
LIBRARY parts_lib;  
USE parts_lib.ALL;
```

The testbench source code should now look similar to that shown in [Figure 5-6](#).

- f. Select **File > Save**.

Figure 5-6. Adding LIBRARY and USE Statements to the Testbench



3. Recompile and simulate.
 - a. In the Project tab of the Workspace, right-click *tcounter.vhd* and select **Compile > Compile Selected**.
 - b. Enter the following command at the QuestaSim> prompt in the Transcript pane.

```
vsim -voptargs="+acc" test_counter
```
 - c. The design loads without errors.

Permanently Mapping VHDL Resource Libraries

If you reference particular VHDL resource libraries in every VHDL project or simulation, you may want to permanently map the libraries. Doing this requires that you edit the master *modelsim.ini* file in the installation directory. Though you won't actually practice it in this tutorial, here are the steps for editing the file:

1. Locate the *modelsim.ini* file in the QuestaSim installation directory (*<install_dir>/questasim/modelsim.ini*).
2. IMPORTANT - Make a backup copy of the file.
3. Change the file attributes of *modelsim.ini* so it is no longer "read-only."
4. Open the file and enter your library mappings in the [Library] section. For example:

```
parts_lib = C:/libraries/parts_lib
```

5. Save the file.
6. Change the file attributes so the file is "read-only" again.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation and close the project.

1. Select **Simulate > End Simulation**. Click **Yes**.
2. Select the Project tab of the Main window Workspace.
3. Select **File > Close**. Click **OK**.

Chapter 6

Simulating Designs With SystemC

Introduction

QuestaSim treats SystemC as just another design language. With only a few exceptions in the current release, you can simulate and debug your SystemC designs the same way you do HDL designs.

Note



The functionality described in this lesson requires a systemc license feature in your QuestaSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Design Files for this Lesson

There are two sample designs for this lesson. The first is a very basic design, called "basic", containing only SystemC code. The second design is a ring buffer where the testbench and top-level chip are implemented in SystemC and the lower-level modules are written in HDL.

The pathnames to the files are as follows:

SystemC – *<install_dir>/examples/systemc/sc_basic*

SystemC/Verilog – *<install_dir>/examples/systemc/sc_vlog*

SystemC/VHDL – *<install_dir>/examples/systemc/sc_vhdl*

This lesson uses the SystemC/Verilog version of the ringbuf design in the examples. If you have a VHDL license, use the VHDL version instead. There is also a mixed version of the design, but the instructions here do not account for the slight differences in that version.

Related Reading

User's Manual Chapters: [SystemC Simulation](#), [Mixed-Language Simulation](#), and [C Debug](#).

Reference Manual command: [sccom](#).

Setting up the Environment

SystemC is a licensed feature. You need the *systemc* license feature in your QuestaSim license file to simulate SystemC designs. Please contact your Mentor Graphics sales representatives if you currently do not have such a feature.

The table below shows the supported operating systems for SystemC and the corresponding required versions of a C compiler.

Table 6-1. Supported Operating Systems for SystemC

Platform	Supported compiler versions
RedHat Linux 7.2 and 7.3 RedHat Linux Enterprise version 2.1	gcc 3.2.3, gcc 4.0.2
AMD64 / SUSE Linux Enterprise Server 9.0, 9.1, 10 or Red Hat Enterprise Linux 3, 4	gcc 4.0.2 VCO is linux (32-bit binary) VCO is linux_x86_64 (64-bit binary)
Solaris 8, 9, 10	gcc 3.3
Windows 2000 and XP	Minimalist GNU for Windows (MinGW) gcc 3.3.1

See *SystemC simulation* in the *QuestaSim User's Manual* for further details.

Preparing an OSCI SystemC design

For an OpenSystemC Initiative (OSCI) compliant SystemC design to run on QuestaSim, you must first:

- Replace **sc_main()** with an SC_MODULE, potentially adding a process to contain any testbench code
- Replace **sc_start()** by using the [run](#) command in the GUI
- Remove calls to **sc_initialize()**
- Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macro

In order to maintain portability between OSCI and QuestaSim simulations, we recommend that you preserve the original code by using **#ifdef** to add the QuestaSim-specific information. When the design is analyzed, **sccom** recognizes the MTI_SYSTEMC preprocessing directive and handles the code appropriately.

For more information on these modifications, refer to [Modifying SystemC Source Code](#) in the User's Manual.

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory, then copy all files from `<install_dir>/examples/systemc/sc_basic` into the new directory.

2. Start QuestaSim and change to the exercise directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

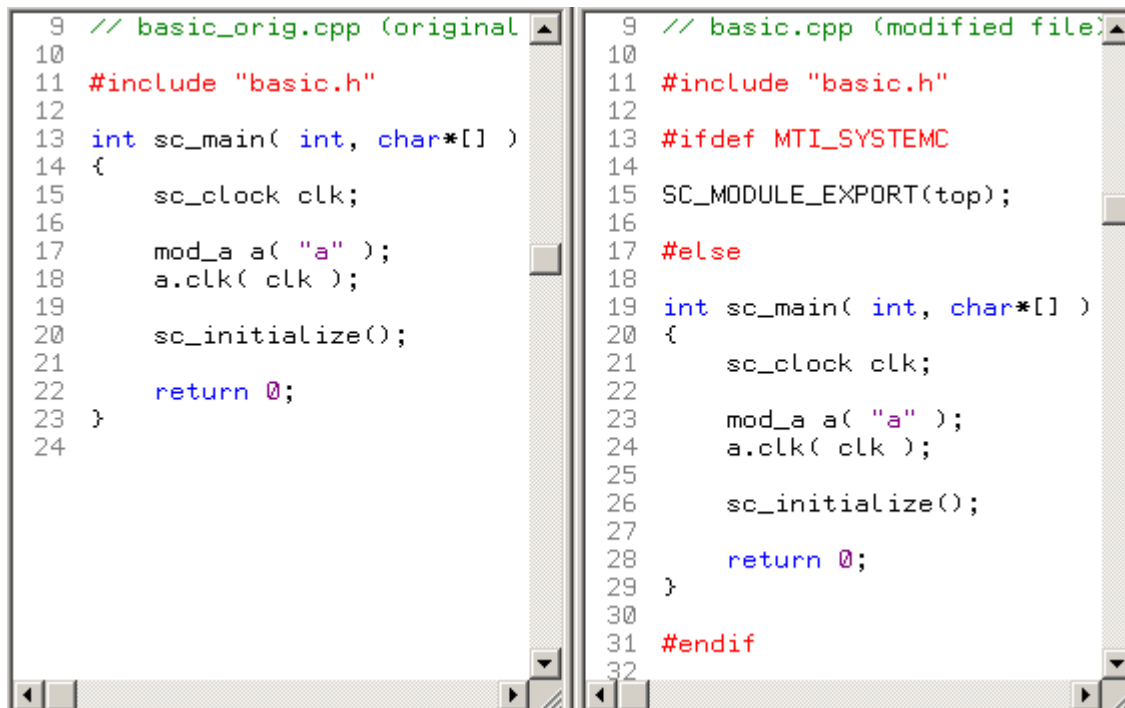
3. Use a text editor to view and edit the *basic_orig.cpp* file. To use QuestaSim's editor, from the Main Menu select **File > Open**. Change the files of type to C/C++ files then double-click *basic_orig.cpp*.

- a. Using the **#ifdef MTI_SYSTEMC** preprocessor directive, add the **SC_MODULE_EXPORT(top);** to the design as shown in [Figure 6-1](#). (The left side of [Figure 6-1](#) is the original code; the right side is the modified code.) Close the preprocessing directive with **#else**.

The original code in the *.cpp* file follows directly after **#else**. End that section of the file with **#endif**.

- b. Save the file as *basic.cpp*.

Figure 6-1. SystemC Code Before and After Modifications



A correctly modified copy of the *basic.cpp* is also available in the *sc_basic/gold* directory.

4. Edit the *basic_orig.h* header file as shown in [Figure 6-2](#).
 - a. Add a QuestaSim specific SC_MODULE (top) as shown in lines 52 through 65 of [Figure 6-2](#).

The declarations that were in *sc_main* are placed here in the header file, in SC_MODULE (top). This creates a top level module above *mod_a*, which allows the tool's automatic name binding feature to properly associate the primitive channels with their names.

Figure 6-2. Editing the SystemC Header File.

```

9  // basic.h (modified header file)
10
11  #ifndef INCLUDED_BASIC
12  #define INCLUDED_BASIC
13
14  #include "systemc.h"
15
16  SC_MODULE( mod_a )
17  {
18      sc_in_clk clk;
19
20      void main_action_method()
21      {
22          cout << simcontext()->delta_count()
23              << " main_action_method called" << endl;
24      }
25
26      void main_action_thread()
27      {
28          while( true ) {
29              cout << simcontext()->delta_count()
30                  << " main_action_thread called" << endl;
31              wait();
32          }
33      }
34
35      void main_action_ctypead()
36      {
37          while( true ) {
38              cout << simcontext()->delta_count()
39                  << " main_action_ctypead called" << endl;
40              wait();
41          }
42      }
43
44      SC_CTOR( mod_a )
45      {
46          SC_METHOD( main_action_method );
47          SC_THREAD( main_action_thread );
48          SC_CTYPEAD( main_action_ctypead, clk.pos() );
49      }
50  };
51
52  #ifdef MTI_SYSTEMC
53  SC_MODULE(top)
54  {
55      sc_clock clk;
56      mod_a a;
57
58      SC_CTOR(top)
59      : clk("clk", 200, 0.5, 0.0, false),
60        a("a")
61      {
62          a.clk( clk );
63      }
64  };
65  #endif

```

- b. Save the file as *basic.h*.

A correctly modified copy of the *basic.h* is also available in the *sc_basic/gold* directory.

You have now made all the edits that are required for preparing the design for compilation.

Compiling a SystemC-only Design

With the edits complete, you are ready to compile the design. Designs that contain only SystemC code are compiled with [sccom](#).

1. Set the working library.
 - a. Type **vlib work** in the QuestaSim Transcript window to create the working library.
2. Compile and link all SystemC files.
 - a. Type **sccom -g basic.cpp** at the QuestaSim> prompt.
The **-g** argument compiles the design for debug.
 - b. Type **sccom -link** at the QuestaSim> prompt to perform the final link on the SystemC objects.

You have successfully compiled and linked the design. The successful compilation verifies that all the necessary file modifications have been entered correctly.

In the next exercise you will compile and load a design that includes both SystemC and HDL code.

Mixed SystemC and HDL Example

In this next example, you have a SystemC testbench that instantiates an HDL module. In order for the SystemC testbench to interface properly with the HDL module, you must create a stub module, a foreign module declaration. You will use the [scgenmod](#) utility to create the foreign module declaration. Finally, you will link the created C object files using **sccom -link**.

1. Create a new exercise directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory, then copy all files from *<install_dir>/examples/systemc/sc_vlog* into the new directory.

If you have a VHDL license, copy the files in *<install_dir>/examples/systemc/sc_vhdl* instead.

2. Start QuestaSim and change to the exercise directory

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a command shell prompt.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Set the working library.

- a. Type **vlib work** in the QuestaSim Transcript window to create the working library.

4. Compile the design.

- a. **Verilog:**

Type **vlog *.v** in the QuestaSim Transcript window to compile all Verilog source files.

VHDL:

Type **vcom -93 *.vhd** in the QuestaSim Transcript window to compile all VHDL source files.

5. Create the foreign module declaration (SystemC stub) for the Verilog module *ringbuf*.

- a. **Verilog:**

Type **scgenmod -map "scalar=bool" ringbuf > ringbuf.h** at the QuestaSim> prompt.

The **-map "scalar=bool"** argument is used to generate boolean scalar port types inside the foreign module declaration. See [scgenmod](#) for more information.

VHDL:

Type **scgenmod ringbuf > ringbuf.h** at the QuestaSim> prompt.

The output is redirected to the file *ringbuf.h* ([Figure 6-3](#)).

Figure 6-3. The ringbuf.h File.

```
1  #ifndef _SCGENMOD_ringbuf_
2  #define _SCGENMOD_ringbuf_
3
4  #include "systemc.h"
5
6  class ringbuf : public sc_foreign_module
7  {
8  public:
9      sc_in<bool> clock;
10     sc_in<bool> reset;
11     sc_in<bool> txda;
12     sc_out<bool> rxda;
13     sc_out<bool> txc;
14     sc_out<bool> outstrobe;
15
16
17     ringbuf(sc_module_name nm, const char* hdl_name,
18           int num_generics, const char** generic_list)
19       : sc_foreign_module(nm),
20         clock("clock"),
21         reset("reset"),
22         txda("txda"),
23         rxda("rxda"),
24         txc("txc"),
25         outstrobe("outstrobe")
26     {
27         elaborate_foreign_module(hdl_name, num_generics, generic_list);
28     }
29     ~ringbuf()
30     {}
31 };
32
33
34 #endif
35
```

The *test_ringbuf.h* file is included in *test_ringbuf.cpp*, as shown in [Figure 6-4](#).

Figure 6-4. The test_ringbuf.cpp File

```
8
9  // test_ringbuf.cpp
10
11  #include "test_ringbuf.h"
12  #include <iostream>
13
14
15  SC_MODULE_EXPORT(test_ringbuf);
16
```

6. Compile and link all SystemC files, including the generated *ringbuf.h*.
 - a. Type **sccom -g test_ringbuf.cpp** at the QuestaSim> prompt.

The *test_ringbuf.cpp* file contains an include statement for *test_ringbuf.h* and a required `SC_MODULE_EXPORT(top)` statement, which informs QuestaSim that the top-level module is SystemC.

- b. Type **sccom -link** at the QuestaSim> prompt to perform the final link on the SystemC objects.
7. Load the design.
 - a. Enter the following command at the QuestaSim> prompt in the Transcript pane.

vsim -voptargs="+acc" test_ringbuf

The -voptargs="+acc" argument for the **vsim** command provides visibility into the design for debugging purposes.

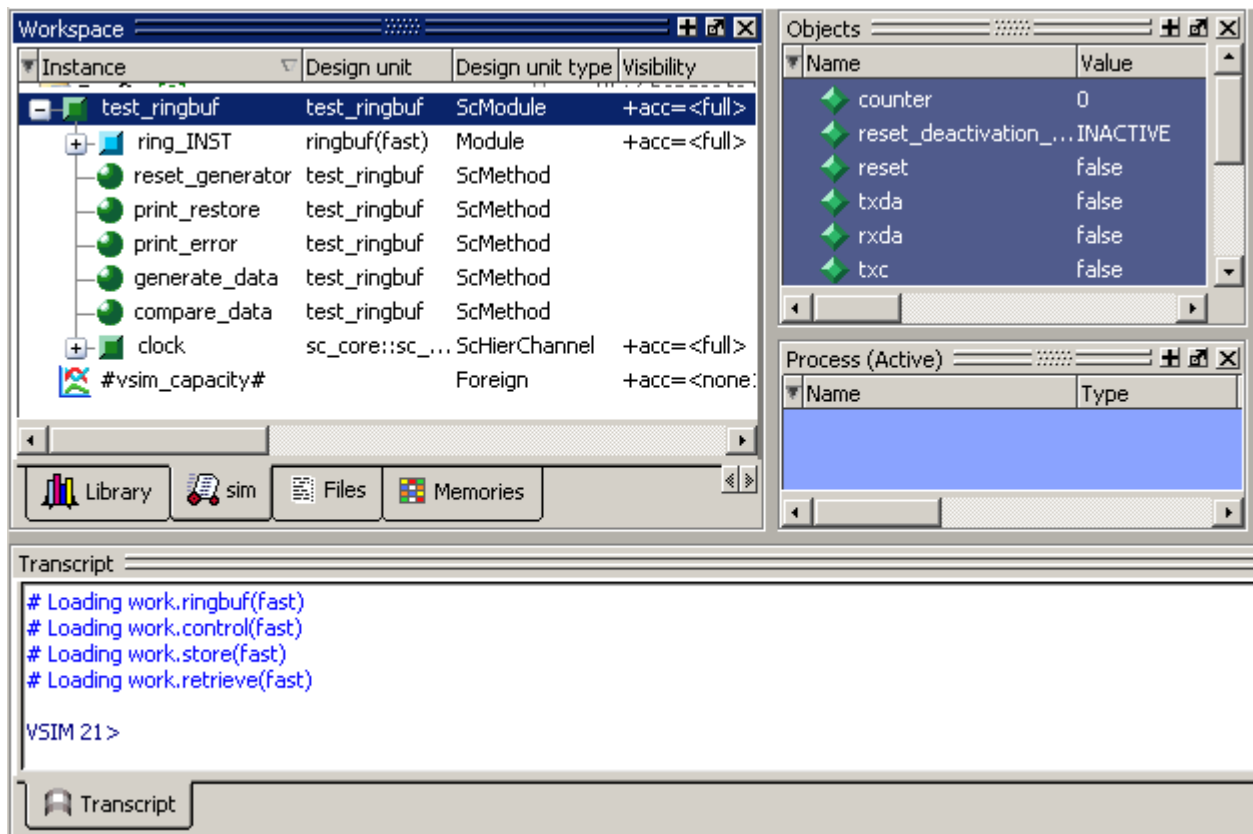
Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

8. If necessary, you may close the Locals, Profile, and Watch panes of the Main window. Make sure the Objects pane is open and the Process pane is open in “Active” mode, as shown in [Figure 6-5](#). To open or close these windows, use the **View** menu.

Figure 6-5. The test_ringbuf Design

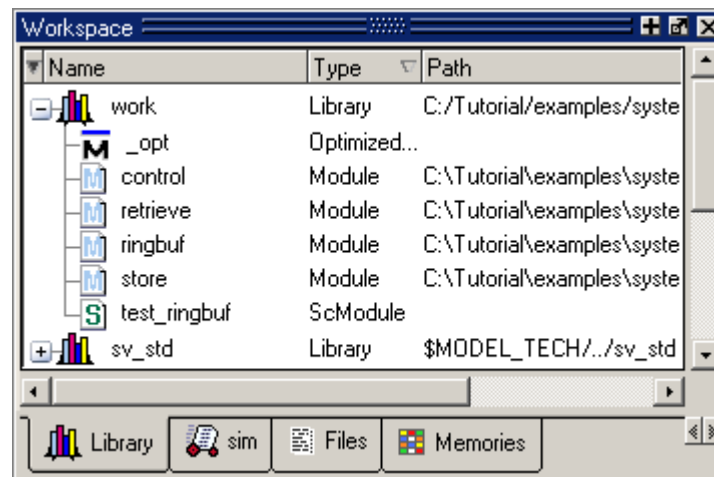


Viewing SystemC Objects in the GUI

SystemC objects are denoted in the QuestaSim GUI with a green 'S' in the Library tab and a green square, circle, or diamond icon elsewhere.

1. View Workspace and objects.
 - a. Click on the Library tab in the Workspace pane of the Main window.
- SystemC objects have a green 'S' next to their names ([Figure 6-6](#)).

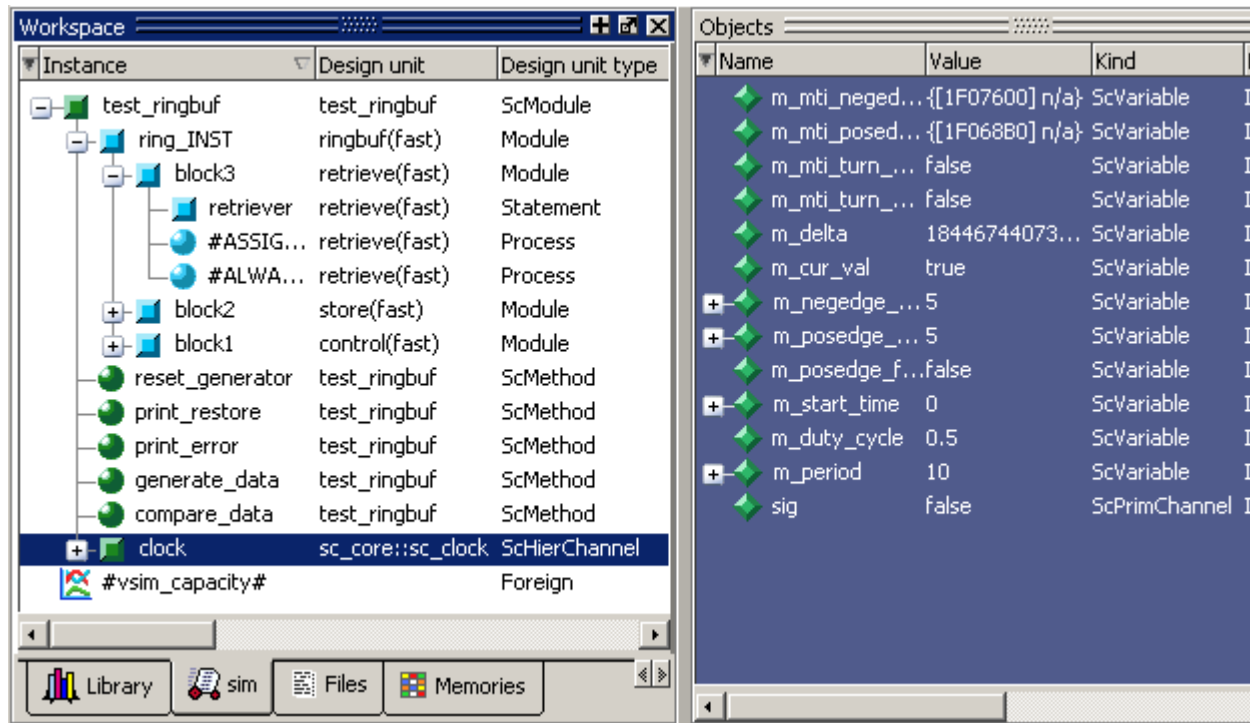
Figure 6-6. SystemC Objects in the work Library



2. Observe window linkages.
 - a. Click on the sim tab in the Workspace pane of the Main window.
 - b. Select the *clock* instance in the sim tab ([Figure 6-7](#)).

The Objects window updates to show the associated SystemC or HDL objects.

Figure 6-7. SystemC Objects in the sim Tab of the Workspace



3. Add objects to the Wave window.
 - a. Right-click *test_ringbuf* in the sim tab of the Workspace and select **Add > To Wave > All items in region**.

Setting Breakpoints and Stepping in the Source Window

As with HDL files, you can set breakpoints and step through SystemC files in the Source window. In the case of SystemC, QuestaSim uses C Debug, an interface to the open-source **gdb** debugger. Refer to the [C Debug](#) chapter in the User's Manual for complete details.

1. Before we set a breakpoint, we must disable the Auto Lib Step Out feature, which is on by default. With Auto Lib Step Out, if you try to step into a standard C++ or SystemC header file (modeltech/include/systemc), QuestaSim will automatically do a step-out.
 - a. Select **Tools > C Debug > Allow lib step** from the Main menus.
2. Set a breakpoint.
 - a. Double-click *test_ringbuf* in the **sim** tab of the Workspace to open the source file.
 - b. In the Source window:
 - Verilog:** scroll to the area around line 150 of *test_ringbuf.h*.
 - VHDL:** scroll to the area around line 155 of *test_ringbuf.h*.

- c. Click in the line number column next to the red line number of the line containing (shown in [Figure 6-8](#)) :

Verilog: `bool var_dataerror_newval = actual.read()...`

VHDL: `sc_logic var_dataerror_newval = acutal.read ...`

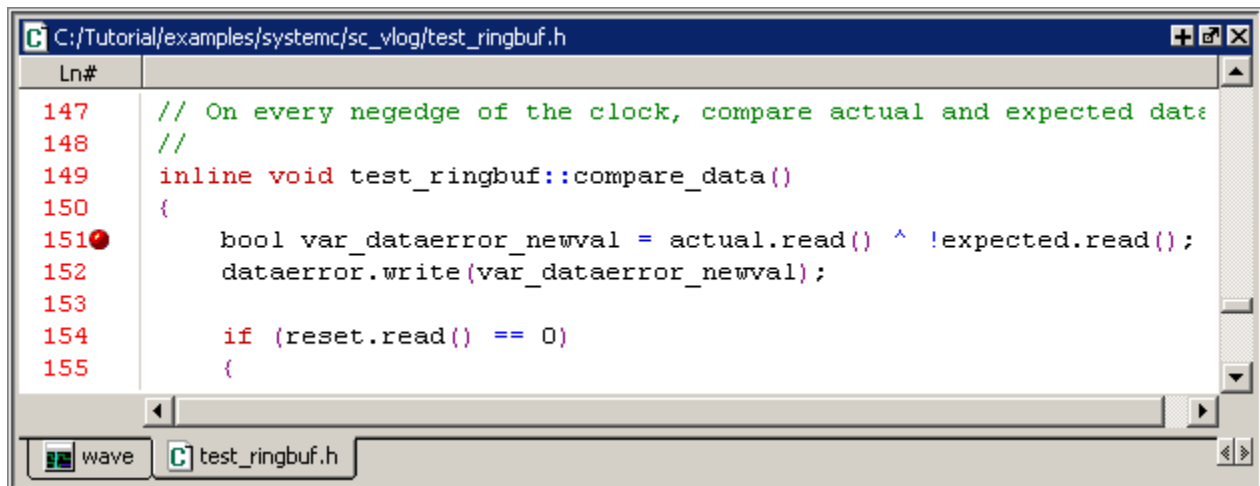
Note



QuestaSim recognizes that the file contains SystemC code and automatically launches C Debug. There will be a slight delay while C Debug opens before the breakpoint appears.

Once the debugger is running, QuestaSim places a solid red ball next to the line number ([Figure 6-8](#)).

Figure 6-8. Active Breakpoint in a SystemC File

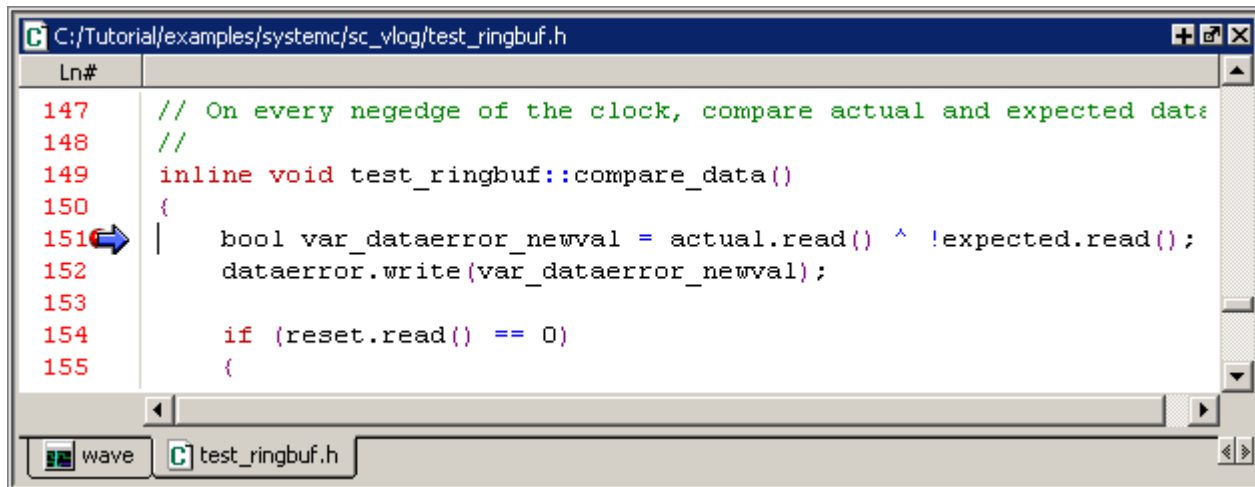


3. Run and step through the code.
 - a. Type **run 500** at the VSIM> prompt.

When the simulation hits the breakpoint, it stops running, highlights the line with a blue arrow in the Source window ([Figure 6-9](#)), and issues a message like this in the Transcript:

```
# C breakpoint c.1  
# test_ringbuf::compare_data (this=0x1f13bc8) at  
test_ringbuf.h:<151>
```

Figure 6-9. Simulation Stopped at Breakpoint

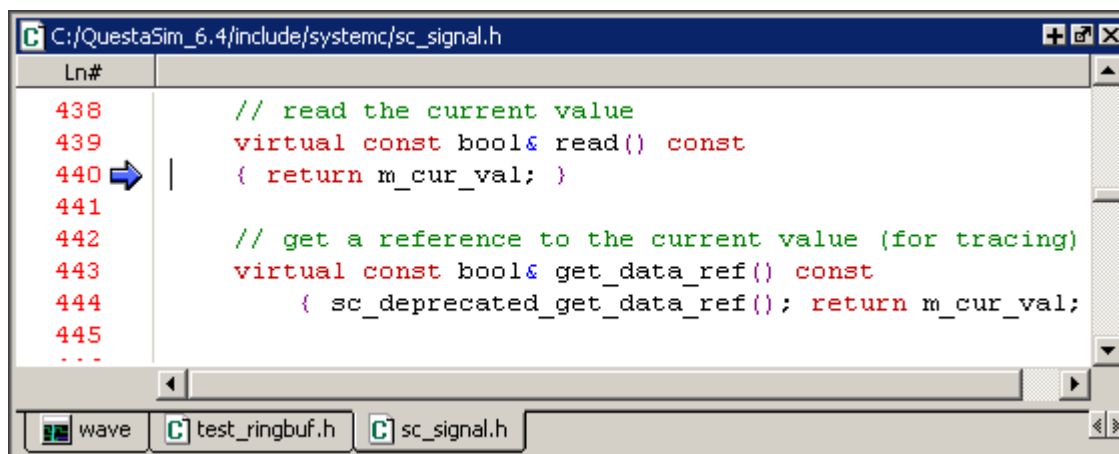


- b. Click the Step icon on the toolbar.



This steps the simulation to the next statement. Because the next statement is a function call, Questasim steps into the function, which is in a separate file — *sc_signal.h* (Figure 6-10).

Figure 6-10. Stepping into a Separate File



- c. Click the Continue Run icon in the toolbar.



The breakpoint in *test_ringbuf.h* is hit again.

Examining SystemC Objects and Variables

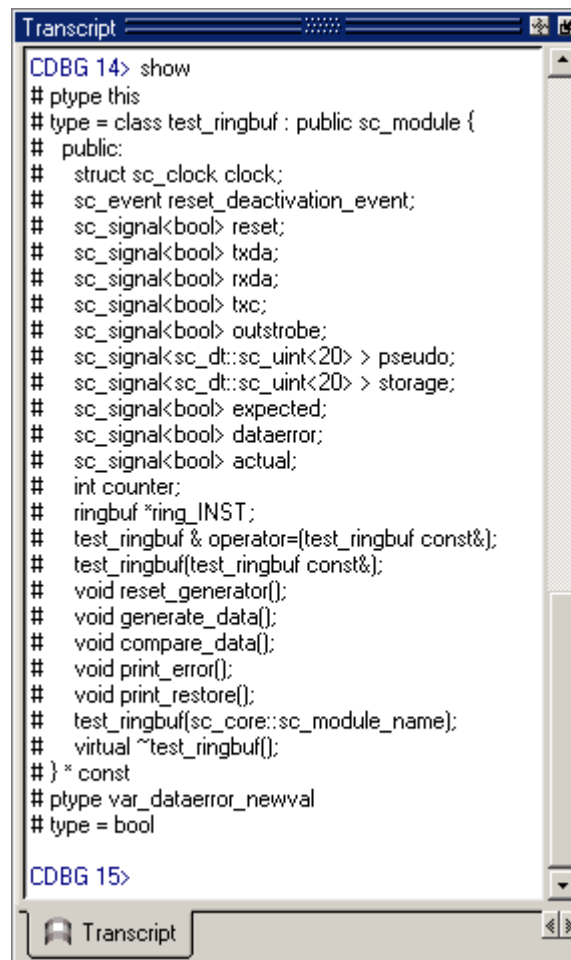
To examine the value of a SystemC object or variable, you can use the **examine** command or view the value in the Objects window.

1. View the value and type of an *sc_signal*.

- a. Enter the **show** command at the **CDBG >** prompt to display a list of all design objects, including their types, in the Transcript.

In this list, you'll see that the type for *dataerror* is "boolean" (sc_logic for VHDL) and *counter* is "int" (Figure 6-11).

Figure 6-11. Output of show Command



```
CDBG 14> show
# ptype this
# type = class test_ringbuf : public sc_module {
#   public:
#     struct sc_clock clock;
#     sc_event reset_deactivation_event;
#     sc_signal<bool> reset;
#     sc_signal<bool> txda;
#     sc_signal<bool> rxda;
#     sc_signal<bool> txc;
#     sc_signal<bool> outstrobe;
#     sc_signal<sc_dt::sc_uint<20>> pseudo;
#     sc_signal<sc_dt::sc_uint<20>> storage;
#     sc_signal<bool> expected;
#     sc_signal<bool> dataerror;
#     sc_signal<bool> actual;
#     int counter;
#     ringbuf *ring_INST;
#     test_ringbuf & operator=(test_ringbuf const&);
#     test_ringbuf(test_ringbuf const&);
#     void reset_generator();
#     void generate_data();
#     void compare_data();
#     void print_error();
#     void print_restore();
#     test_ringbuf(sc_core::sc_module_name);
#     virtual ~test_ringbuf();
# } * const
# ptype var_dataerror_newval
# type = bool

CDBG 15>
```

- b. Enter the **examine dataerror** command at the CDBG > prompt.
The value returned is "true".
2. View the value of a SystemC variable.
 - a. Enter the **examine counter** command at the CDBG > prompt to view the value of this variable.
The value returned is "-1".

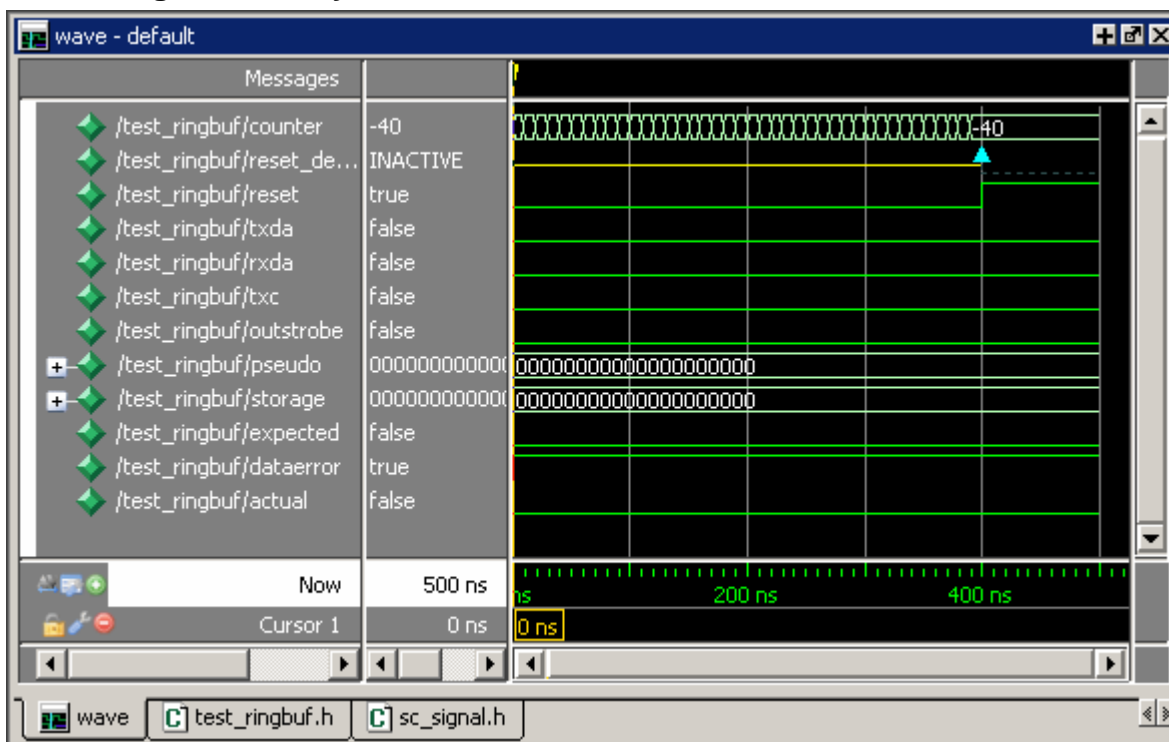
Removing a Breakpoint

1. Return to the Source window for `test_ringbuf.h` and right-click the red ball in the line number column. Select **Remove Breakpoint** from the popup menu.
2. Click the Continue Run button again.

The simulation runs for 500 ns and waves are drawn in the Wave window (Figure 6-12).

If you are using the VHDL version, you might see warnings in the Main window transcript. These warnings are related to VHDL value conversion routines and can be ignored.

Figure 6-12. SystemC Primitive Channels in the Wave Window



Lesson Wrap-up

This concludes the lesson. Before continuing we need to quit the C debugger and end the current simulation.

1. Select **Tools > C Debug > Quit C Debug**.
2. Select **Simulate > End Simulation**. Click **Yes** when prompted to confirm that you wish to quit simulating.

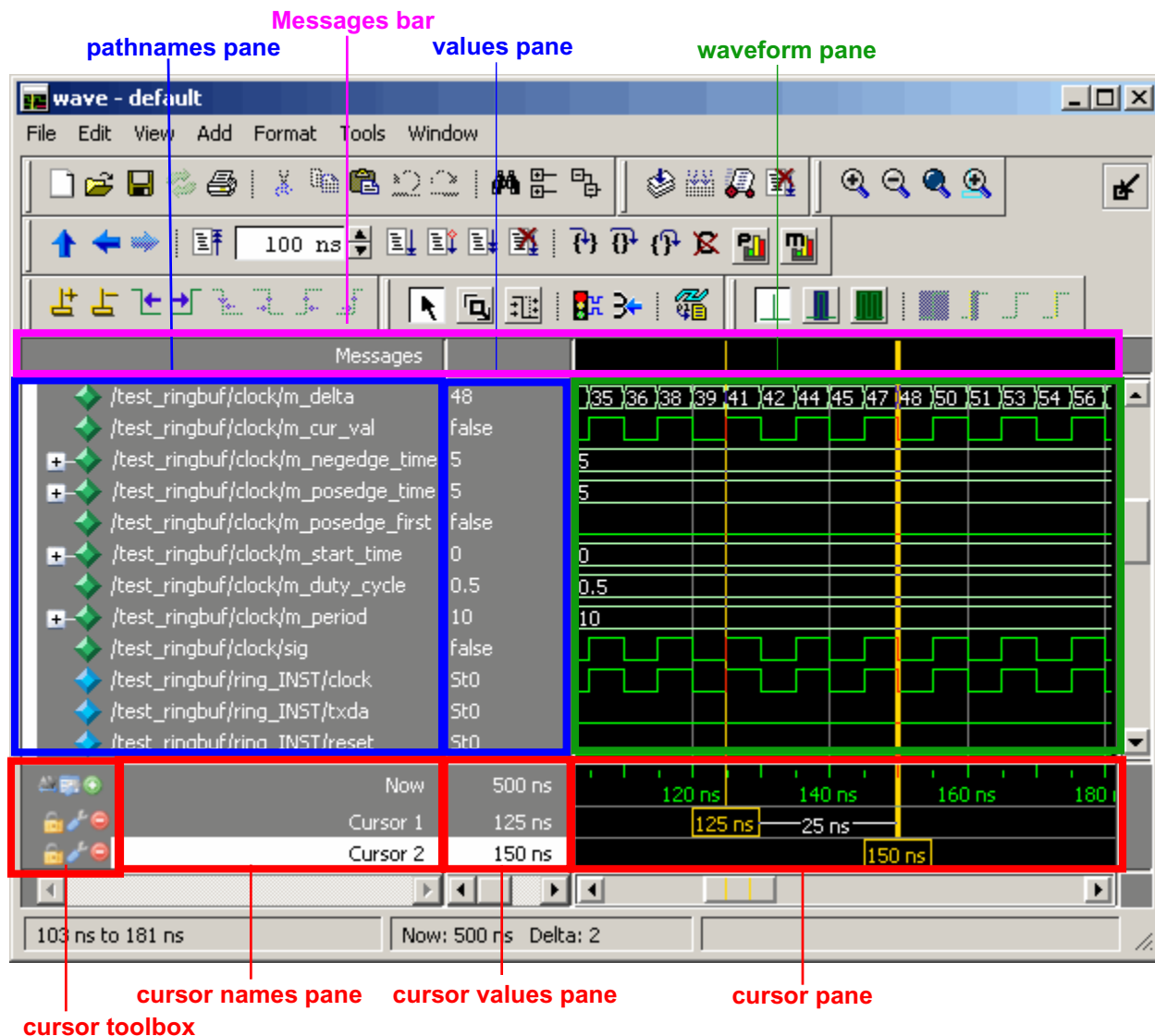
Chapter 7

Analyzing Waveforms

Introduction

The Wave window allows you to view the results of your simulation as HDL waveforms and their values. The Wave window is divided into a number of panes (Figure 7-1). You can resize the pathnames pane, the values pane, and the waveform pane by clicking and dragging the bar between any two panes.

Figure 7-1. Panes of the Wave Window



Related Reading

User's Manual sections: [Wave Window](#) and [Recording Simulation Results With Datasets](#)

Loading a Design

For the examples in this lesson, we have used the design simulated in [Basic Simulation](#).

1. If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

2. Load the design.

- a. Select **File > Change Directory** and open the directory you created in the “Basic Simulation” lesson.

The *work* library should already exist.

- b. Enter the following command at the QuestaSim> prompt in the Transcript pane.

```
vsim -voptargs="+acc" test_counter
```

The `-voptargs="+acc"` argument for the **vsim** command provides visibility into the design for debugging purposes.

Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

QuestaSim loads the design and adds *sim* and *Files* tabs to the Workspace.

Add Objects to the Wave Window

QuestaSim offers several methods for adding objects to the Wave window. In this exercise, you will try different methods.

1. Add objects from the Objects pane.
 - a. Select an item in the Objects pane of the Main window, right-click, and then select **Add > To Wave > All items in region**.

QuestaSim adds several signals to the Wave window.

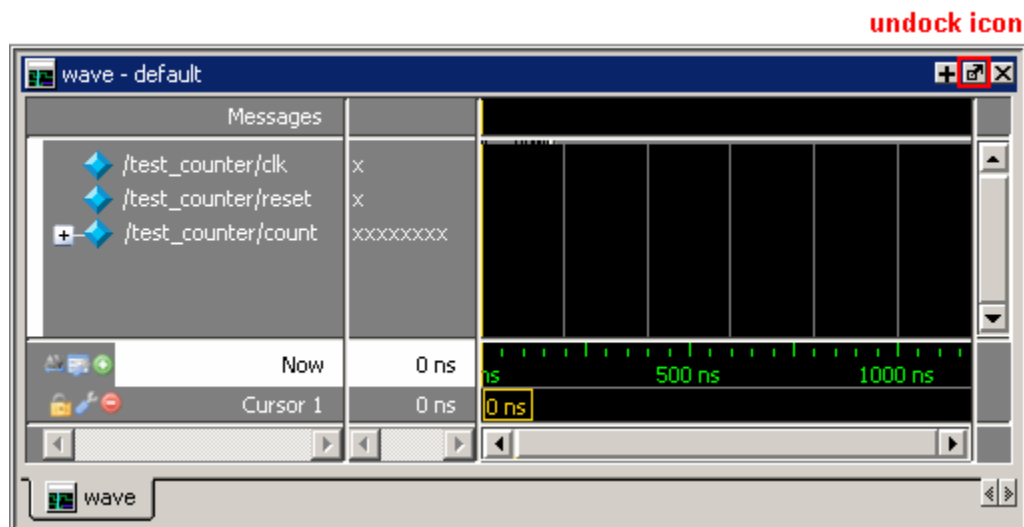
2. Undock the Wave window.

By default QuestaSim opens Wave windows as a tab in the MDI frame of the Main window. You can change the default via the Preferences dialog (**Tools > Edit Preferences**). Refer to the section [Simulator GUI Preferences](#) in the User's Manual for more information.

- a. Click the undock button on the Wave pane ([Figure 7-2](#)).

The Wave pane becomes a standalone, un-docked window. You may need to resize the window.

Figure 7-2. Undocking the Wave Window



3. Add objects using drag-and-drop.

You can drag an object to the Wave window from many other windows and panes (e.g., Workspace, Objects, and Locals).

- a. In the Wave window, select **Edit > Select All** and then **Edit > Delete**.
- b. Drag an instance from the *sim* tab of the Main window to the Wave window.

QuestaSim adds the objects for that instance to the Wave window.

- c. Drag a signal from the Objects pane to the Wave window.
- d. In the Wave window, select **Edit > Select All** and then **Edit > Delete**.

4. Add objects using a command.

- a. Type **add wave *** at the **VSIM>** prompt.

QuestaSim adds all objects from the current region.

- b. Run the simulation for awhile so you can see waveforms.

Zooming the Waveform Display

Zooming lets you change the display range in the waveform pane. There are numerous methods for zooming the display.

1. Zoom the display using various techniques.

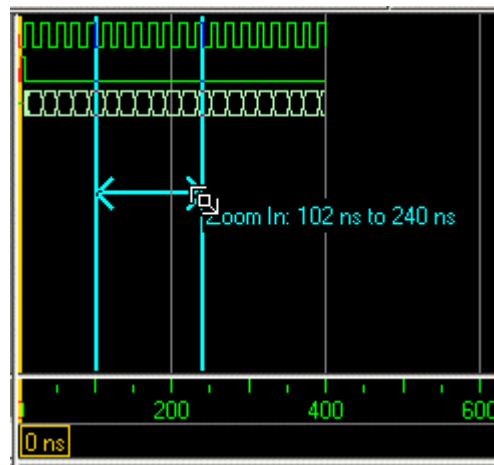
- a. Click the Zoom Mode icon on the Wave window toolbar.



- b. In the waveform pane, click and drag down and to the right.

You should see blue vertical lines and numbers defining an area to zoom in (Figure 7-3).

Figure 7-3. Zooming in with the Mouse Pointer



- c. Select **View > Zoom > Zoom Last**.

The waveform pane returns to the previous display range.

- d. Click the Zoom In 2x icon a few times.



- e. In the waveform pane, click and drag up and to the right.

You should see a blue line and numbers defining an area to zoom out.

- f. Select **View > Zoom > Zoom Full**.

Using Cursors in the Wave Window

Cursors mark simulation time in the Wave window. When QuestaSim first draws the Wave window, it places one cursor at time zero. Clicking anywhere in the waveform pane brings that cursor to the mouse location.

You can also add additional cursors; name, lock, and delete cursors; use cursors to measure time intervals; and use cursors to find transitions.

First, dock the Wave window in the Main window by clicking the dock icon.



Working with a Single Cursor

1. Position the cursor by clicking and dragging.

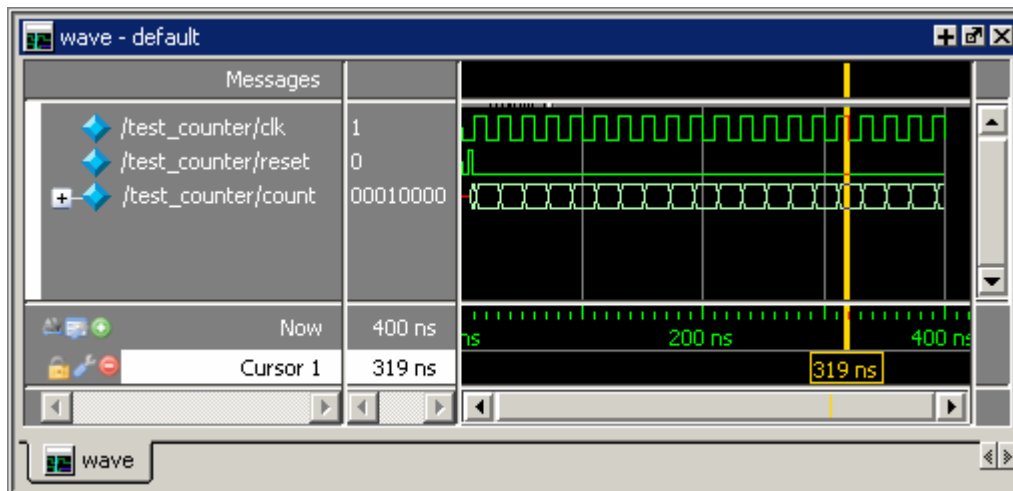
- a. Click the Select Mode icon on the Wave window toolbar.



- b. Click anywhere in the waveform pane.

A cursor is inserted at the time where you clicked ([Figure 7-4](#)).

Figure 7-4. Working with a Single Cursor in the Wave Window



- c. Drag the cursor and observe the value pane.

The signal values change as you move the cursor. This is perhaps the easiest way to examine the value of a signal at a particular time.

- d. In the waveform pane, drag the cursor to the right of a transition with the mouse positioned over a waveform.

The cursor "snaps" to the nearest transition to the left. Cursors "snap" to a waveform edge if you click or drag a cursor to within ten pixels of a waveform edge. You can set the snap distance in the Window Preferences dialog (select **Tools > Window Preferences**).

- e. In the cursor pane, drag the cursor to the right of a transition ([Figure 7-4](#)).

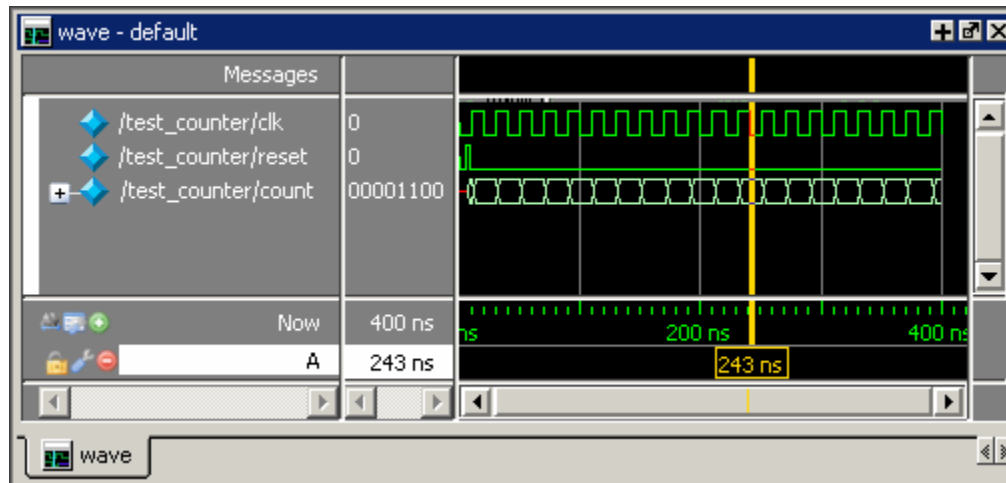
The cursor doesn't snap to a transition if you drag in the cursor pane.

2. Rename the cursor.

- a. Right-click "Cursor 1" in the cursor name pane, and select and delete the text.
 - b. Type **A** and press Enter.

The cursor name changes to "A" (Figure 7-5).

Figure 7-5. Renaming a Cursor



3. Jump the cursor to the next or previous transition.

- a. Click signal *count* in the pathname pane.

- b. Click the Find Next Transition icon on the Wave window toolbar.



The cursor jumps to the next transition on the currently selected signal.

- c. Click the Find Previous Transition icon on the Wave window toolbar.



The cursor jumps to the previous transition on the currently selected signal.

Working with Multiple Cursors

1. Add a second cursor.

- a. Click the Add Cursor icon on the Wave window toolbar.

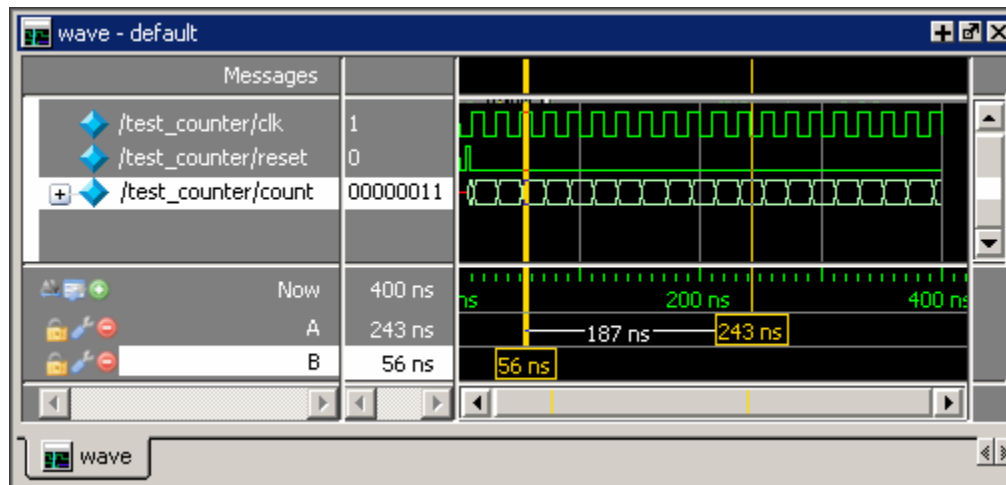


- b. Right-click the name of the new cursor and delete the text.

- c. Type **B** and press Enter.

- d. Drag cursor *B* and watch the interval measurement change dynamically (Figure 7-6).

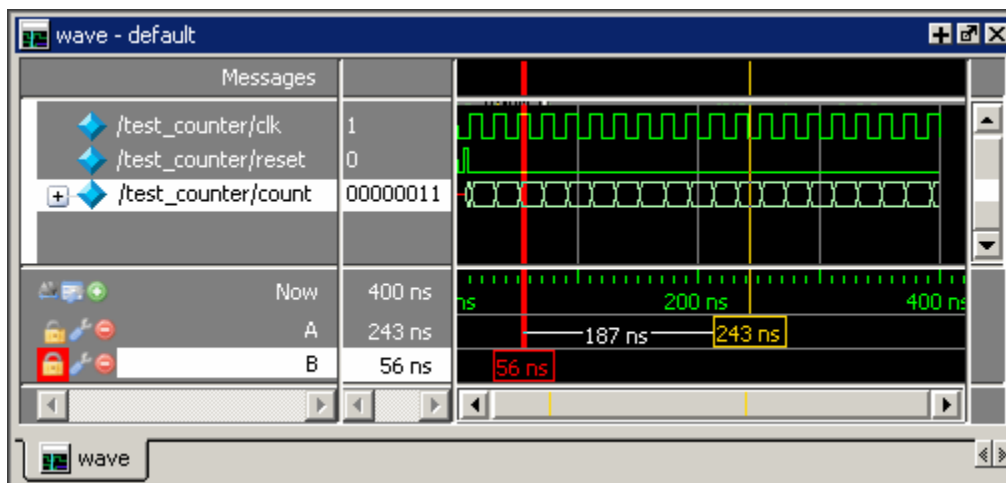
Figure 7-6. Interval Measurement Between Two Cursors



2. Lock cursor *B*.
 - a. Right-click cursor *B* in the cursor pane and select **Lock B**.

The cursor color changes to red and you can no longer drag the cursor (Figure 7-7).

Figure 7-7. A Locked Cursor in the Wave Window



3. Delete cursor *B*.
 - a. Right-click cursor *B* and select **Delete B**.

Saving and Reusing the Window Format

If you close the Wave window, any configurations you made to the window (e.g., signals added, cursors set, etc.) are discarded. However, you can use the Save Format command to capture the

current Wave window display and signal preferences to a *.do* file. You open the *.do* file later to recreate the Wave window as it appeared when the file was created.

Format files are design-specific; use them only with the design you were simulating when they were created.

1. Save a format file.
 - a. In the Wave window, select **File > Save**.
 - b. In the Pathname field of the Save Format dialog, leave the file name set to *wave.do* and click **OK**.
 - c. Close the Wave window.
2. Load a format file.
 - a. In the Main window, select **View > Wave**.
 - b. Undock the window.

All signals and cursor(s) that you had set are gone.
 - c. In the Wave window, select **File > Load**.
 - d. In the Open Format dialog, select *wave.do* and click **Open**.

QuestaSim restores the window to its previous state.
 - e. Close the Wave window when you are finished by selecting **File > Close Window**.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. Select **Simulate > End Simulation**. Click Yes.

Chapter 8

Creating Stimulus With Waveform Editor

Introduction

The Waveform Editor creates stimulus for your design via interactive manipulation of waveforms. You can then run the simulation with these edited waveforms or export them to a stimulus file for later use.

In this lesson you will do the following:

- Load the *counter* design unit without a testbench
- Create waves via a wizard
- Edit waves interactively in the Wave window
- Export the waves to an HDL testbench and extended VCD file
- Run the simulation
- Re-simulate using the exported testbench and VCD file

Related Reading

User's Manual Sections: [Generating Stimulus with Waveform Editor](#) and [Wave Window](#).

Load a Design Unit

For the examples in this lesson, we will use part of the design simulated in [Basic Simulation](#).

Note



You can also use the Waveform Editor prior to loading a design. Refer to the section [Using Waveform Editor Prior to Loading a Design](#) in the User Manual for more information.

1. If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.
 - a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.
If the Welcome to QuestaSim dialog appears, click **Close**.
2. Open a Wave window.
 - a. Select **View > Wave** from the Main window menus.

3. Load the *counter* design unit.
 - a. Select **File > Change Directory** and open the directory you created in Lesson 2.

The *work* library should already exist.

- b. Enter the following command at the QuestaSim> prompt in the Transcript pane.

```
vsim -voptargs="+acc" counter
```

The `-voptargs="+acc"` argument for the `vsim` command provides visibility into the design for debugging purposes.

Note

By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

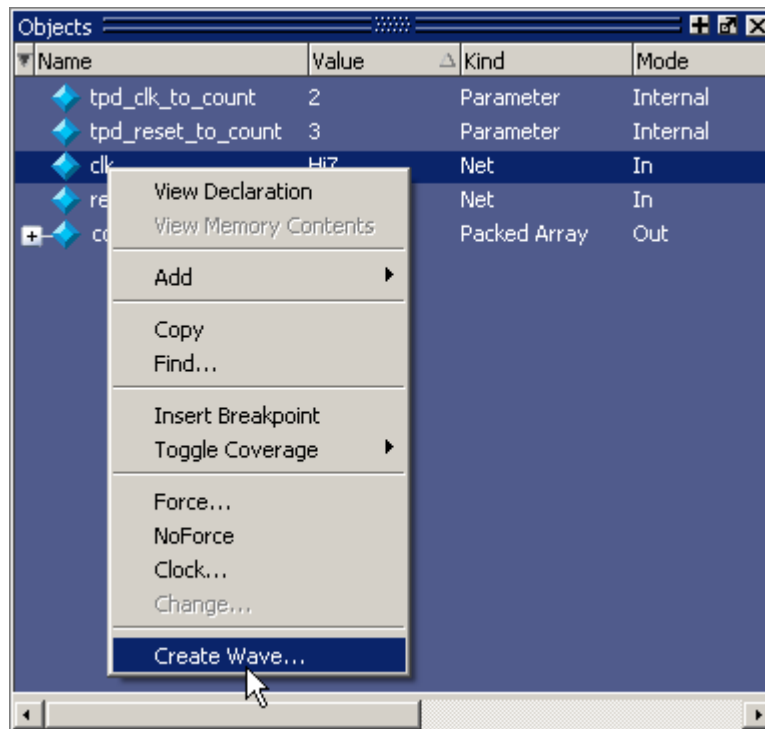
QuestaSim loads the *counter* design unit and adds *sim*, *Files*, and *Memories* tabs to the Workspace.

Create Graphical Stimulus with a Wizard

Waveform Editor includes a Create Pattern Wizard that walks you through the process of creating editable waveforms.

1. Use the Create Pattern Wizard to create a clock pattern.
 - a. In the Objects pane, right click signal *clk* and select **Create Wave** ([Figure 8-1](#)).

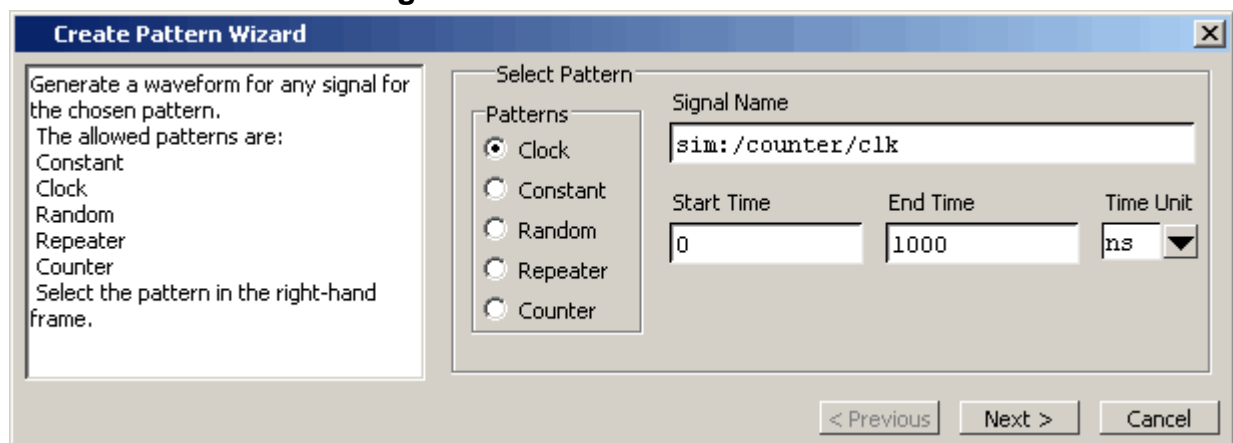
Figure 8-1. Initiating the Create Pattern Wizard from the Objects Pane



This opens the Create Pattern Wizard dialog where you specify the type of pattern (Clock, Repeater, etc.) and a start and end time.

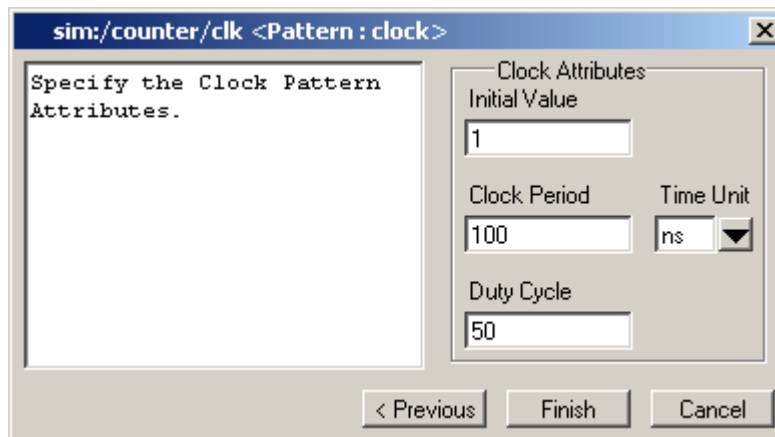
- b. The default pattern is Clock, which is what we need, so click **Next** (Figure 8-2).

Figure 8-2. Create Pattern Wizard



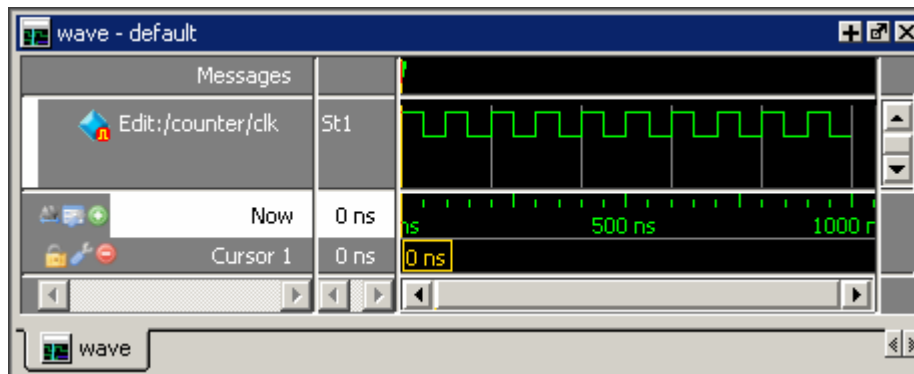
- c. In the second dialog of the wizard, enter **1** for Initial Value. Leave everything else as is and click **Finish** (Figure 8-3).

Figure 8-3. Specifying Clock Pattern Attributes



A generated waveform appears in the Wave window (Figure 8-4). Notice the small red dot on the waveform icon and the prefix "Edit:". These items denote an editable wave. (You may want to undock the Wave window.)

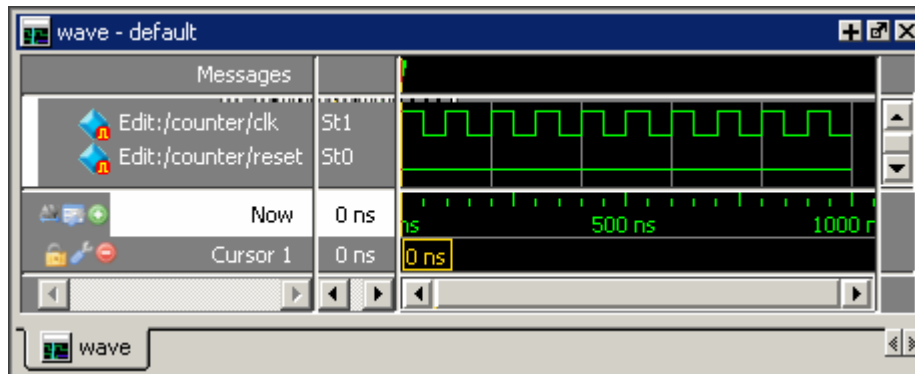
Figure 8-4. The *clk* Waveform



2. Create a second wave using the wizard.
 - a. Right-click signal *reset* in the Objects pane and select **Create Wave** from the popup menu.
 - b. Select **Constant** for the pattern type and click **Next**.
 - c. Enter **0** for the Value and click **Finish**.

A second generated waveform appears in the Wave window (Figure 8-5).

Figure 8-5. The *reset* Waveform



Edit Waveforms in the Wave Window

Waveform Editor gives you numerous commands for interactively editing waveforms (e.g., invert, mirror, stretch edge, cut, paste, etc.). You can access these commands via the menus, toolbar buttons, or via keyboard and mouse shortcuts. You will try out several commands in this part of the exercise.

1. Insert a pulse on signal *reset*.

- a. Click the Edit Mode icon in the toolbar.



- b. In the Wave window, click the *reset* signal so it is selected.

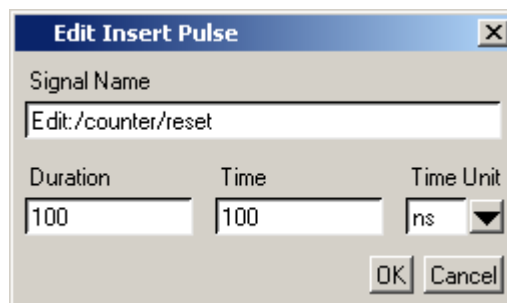
- c. Click the Insert Pulse icon in the toolbar.



Or, in the waveform pane of the Wave window, right-click on the *reset* signal waveform and select **Wave > Insert Pulse**.

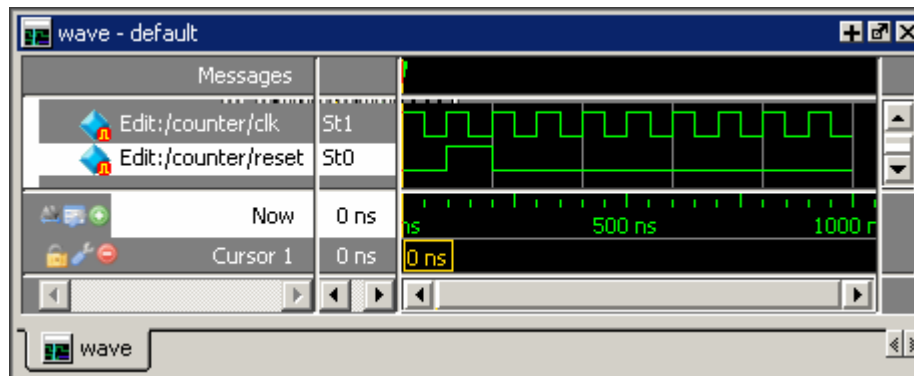
- d. In the Edit Insert Pulse dialog, enter **100** in the Duration field and **100** in the Time field (Figure 8-6), and click OK.

Figure 8-6. Edit Insert Pulse Dialog



Signal *reset* now goes high from 100 ns to 200 ns (Figure 8-7).

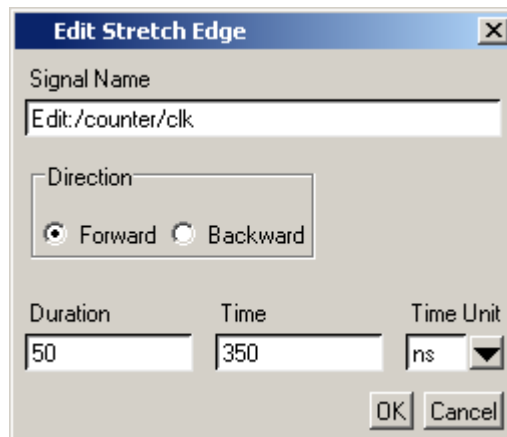
Figure 8-7. Signal *reset* with an Inserted Pulse



2. Stretch an edge on signal *clk*.
 - a. Click the signal *clk* waveform just to the right of the transition at 350 ns. The cursor should snap to the transition at 350 ns.
 - b. Right-click that same transition and select **Wave > Stretch Edge** from the popup menu.

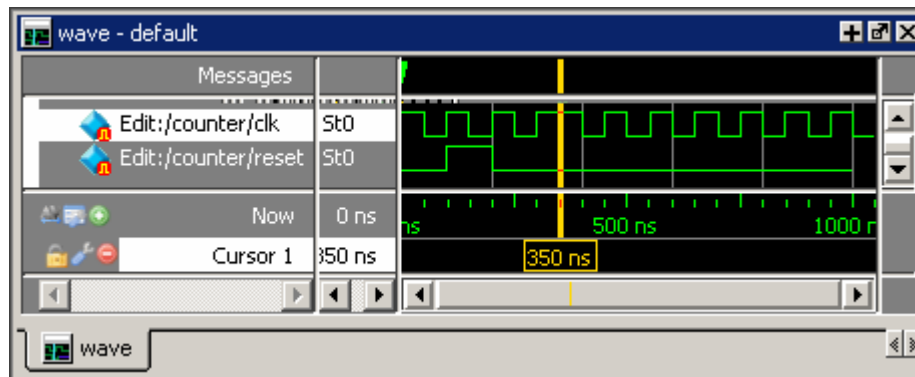
If the command is dimmed out, the cursor probably isn't on the edge at 350 ns.
 - c. In the Edit Stretch Edge dialog, enter 50 for Duration, make sure the Time field shows 350, and then click OK (Figure 8-8).

Figure 8-8. Edit Stretch Edge Dialog



The wave edge stretches so it is high from 300 to 400 ns (Figure 8-9).


Figure 8-9. Stretching an Edge on the *clk* Signal



Note the difference between stretching and moving an edge — the Stretch command moves an edge by moving other edges on the waveform (either increasing waveform duration or deleting edges at the beginning of simulation time); the Move command moves an edge but does not move other edges on the waveform. You should see in the Wave window that the waveform for signal *clk* now extends to 1050 ns.

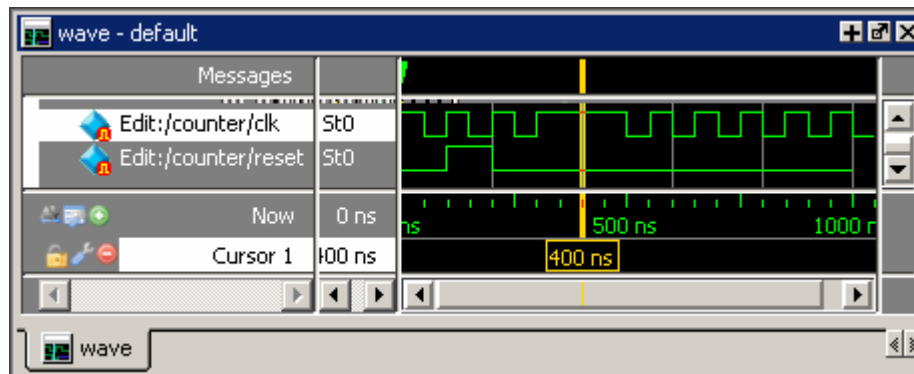
3. Delete an edge.
 - a. Click signal *clk* just to the right of the transition at 400 ns.


The cursor should "snap" to 400 ns.

- b. Click the Delete Edge icon. 

This opens the Edit Delete Edge dialog. The Time is already set to 400 ns. Click **OK**. The edge is deleted and *clk* now stays high until 500 ns ([Figure 8-10](#)).

Figure 8-10. Deleting an Edge on the *clk* Signal



4. Undo and redo an edit.
 - a. Click the Undo icon. 

The Edit Undo dialog opens, allowing you to select the Undo Count - the number of past actions to undo. Click **OK** with the Undo Count set to 1 and the deleted edge at 400 ns reappears.

- b. Click the Redo icon. 

The edge is deleted again. You can undo and redo any number of editing operations *except* extending all waves and changing drive types. Those two edits cannot be undone.

Save and Reuse the Wave Commands

You can save the commands that QuestaSim used to create the waveforms. You can load this "format" file at a later time to re-create the waves. In this exercise, we will save the commands, quit and reload the simulation, and then open the format file.

1. Save the wave commands to a format file.
 - a. Select **File > Close** in the menu bar and you will be prompted to save the wave commands.
 - b. Click **Yes**.
 - c. Type *waveedit.do* in the File name field of the Save Commands dialog that opens and then click Save.

This saves a DO file named *waveedit.do* to the current directory and closes the Wave window.

2. Quit and then reload the simulation.
 - a. In the Main window, select **Simulate > End Simulation**, and click Yes to confirm you want to quit simulating.
 - b. To reload the simulation, enter the following command at the QuestaSim> prompt.

```
vsim -voptargs="+acc" counter
```

3. Open the format file.
 - a. Select **View > Wave** to open the Wave window.
 - b. Select **File > Load** from the menu bar.
 - c. Double-click *waveedit.do* to open the file.

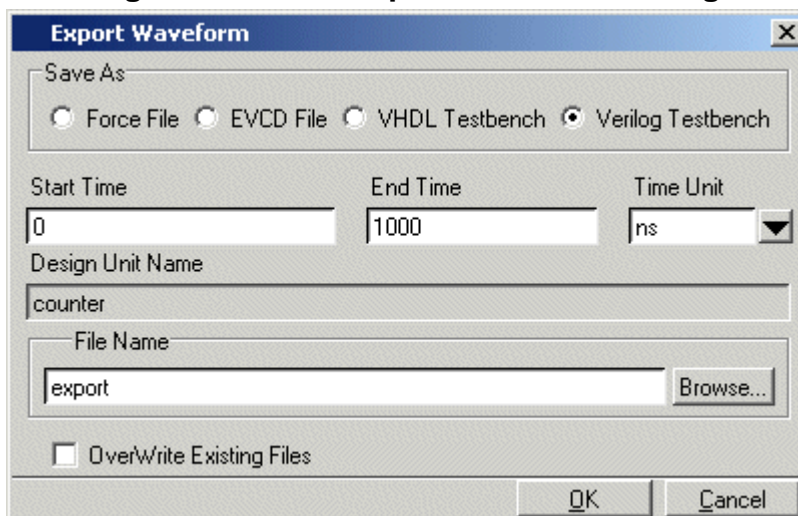
The waves you created earlier in the lesson reappear. If waves do not appear, you probably did not load the *counter* design unit.

Exporting the Created Waveforms

At this point you can run the simulation or you can export the created waveforms to one of four stimulus file formats. You will run the simulation in a minute but first let us export the created waveforms so we can use them later in the lesson.

1. Export the created waveforms in an HDL testbench format.
 - a. Select **File > Export > Waveform**.
 - b. Select **Verilog Testbench** (or **VHDL Testbench** if you are using the VHDL sample files).
 - c. Enter **1000** for End Time if necessary.
 - d. Enter **export** in the File Name field and click **OK** (Figure 8-11).

Figure 8-11. The Export Waveform Dialog



QuestaSim creates a file named *export.v* (or *export.vhd*) in the current directory. Later in the lesson we will compile and simulate the file.

2. Export the created waveforms in an extended VCD format.
 - a. Select **File > Export > Waveform**.
 - b. Select **EVCD File**.
 - c. Enter **1000** for End Time if necessary and click OK.

QuestaSim creates an extended VCD file named *export.vcd*. We will import this file later in the lesson.

Run the Simulation

Once you have finished editing the waveforms, you can run the simulation straight away.


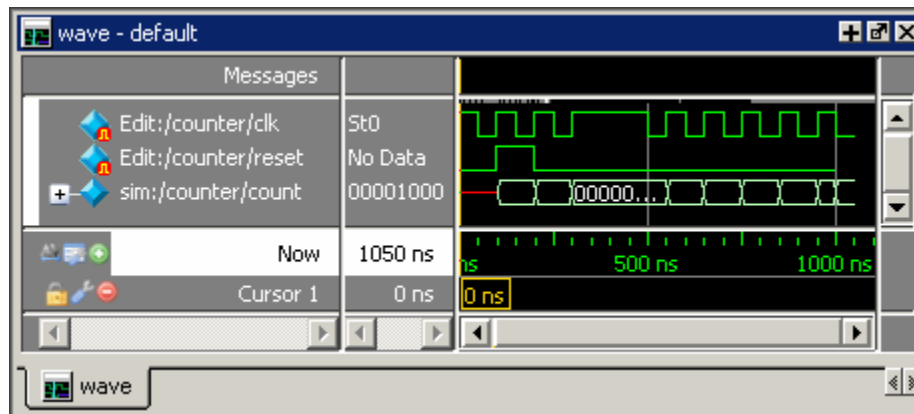
1. Add a design signal.
 - a. In the Objects pane, right-click *count* and select **Add > To Wave > Selected items**.
The signal is added to the Wave window.
2. Run the simulation.
 - a. Click the Run -All icon. 
The simulation runs for 1000 ns and the waveform is drawn for *sim:/counter/count* (Figure 8-12).

Figure 8-12. The counter Waveform Reacts to Stimulus Patterns



Look at the signal transitions for *count* from 300 ns to 500 ns. The transitions occur when *clk* goes high, and you can see that *count* follows the pattern you created when you edited *clk* by stretching and deleting edges.

3. Quit the simulation.
 - a. In the Main window, select **Simulate > End Simulation**, and click Yes to confirm you want to quit simulating.

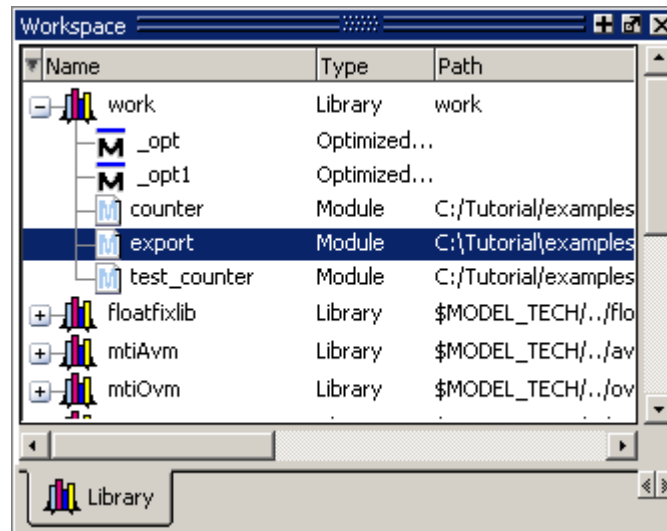
Simulating with the Testbench File

Earlier in the lesson you exported the created waveforms to a testbench file. In this exercise you will compile and load the testbench and then run the simulation.

1. Compile and load the testbench.
 - a. At the QuestaSim prompt, enter **vlog export.v** (or **vcom export.vhd** if you are working with VHDL files).

You should see a design unit named *export* appear in the Library tab (Figure 8-13).

Figure 8-13. The *export* Testbench Compiled into the work Library



- b. Enter the following command at the QuestaSim> prompt.

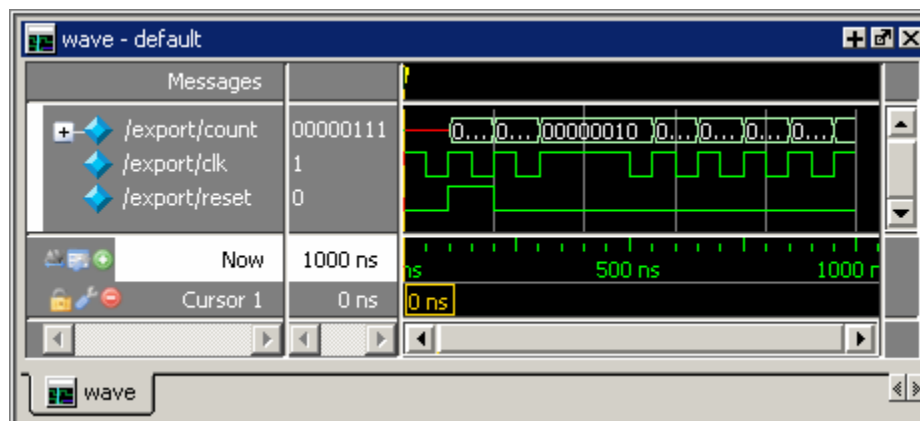
```
vsim -voptargs="+acc" export
```

2. Add waves and run the design.

- a. At the VSIM> prompt, type **add wave ***.
b. Next type **run 1000**.

The waveforms in the Wave window match those you saw in the last exercise (Figure 8-14).

Figure 8-14. Waves from Newly Created Testbench



3. Quit the simulation.

- a. In the Main window, select **Simulate > End Simulation**, and click Yes to confirm you want to quit simulating.

Importing an EVCD File

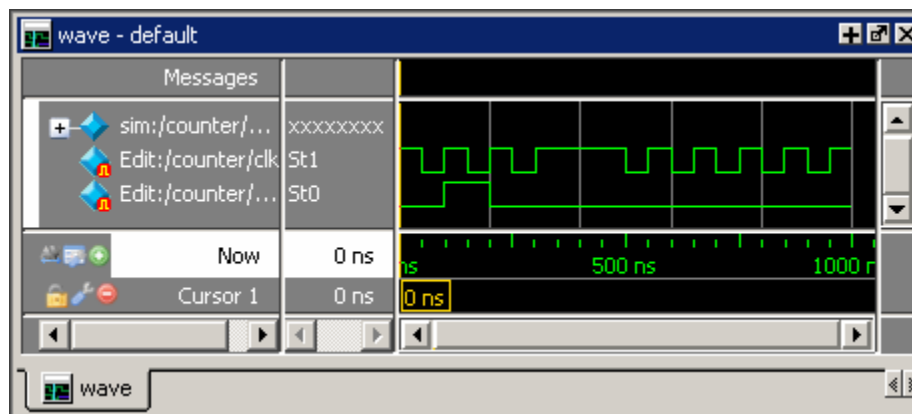
Earlier in the lesson you exported the created waveforms to an extended VCD file. In this exercise you will use that file to stimulate the *counter* design unit.


1. Load the *counter* design unit and add waves.
 - a. Enter the following command at the QuestaSim> prompt.

```
vsim -voptargs="+acc" counter
```
 - b. In the Objects pane, right-click *count* and select **Add > To Wave > Selected items**.
2. Import the VCD file.
 - a. Make sure the Wave window is active, then select **File > Import > EVCD** from the menu bar.
 - b. Double-click *export.vcd*.

The created waveforms draw in the Wave window (Figure 8-15).

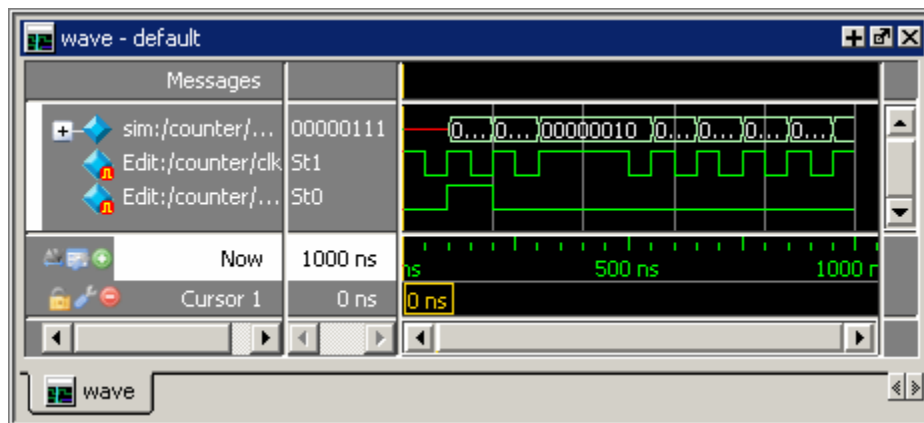
Figure 8-15. EVCD File Loaded in Wave Window



- c. Click the Run -All icon. 

The simulation runs for 1000 ns and the waveform is drawn for *sim:/counter/count* (Figure 8-16).

Figure 8-16. Simulation results with EVCD File



When you import an EVCD file, signal mapping happens automatically if signal names and widths match. If they do not, you have to manually map the signals. Refer to the section [Signal Mapping and Importing EVCD Files](#) in the User's Manual for more information.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. In the Main window, select **Simulate > End Simulation**. Click Yes.

Chapter 9

Debugging With The Dataflow Window

Introduction

The Dataflow window allows you to explore the "physical" connectivity of your design; to trace events that propagate through the design; and to identify the cause of unexpected outputs. The window displays processes; signals, nets, and registers; and interconnect.

Note



The functionality described in this lesson requires a dataflow license feature in your QuestaSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Design Files for this Lesson

The sample design for this lesson is a testbench that verifies a cache module and how it works with primary memory. A processor design unit provides read and write requests.

The pathnames to the files are as follows:

Verilog – *<install_dir>/examples/tutorials/verilog/dataflow*

VHDL – *<install_dir>/examples/tutorials/vhdl/dataflow*

This lesson uses the Verilog version in the examples. If you have a VHDL license, use the VHDL version instead. When necessary, we distinguish between the Verilog and VHDL versions of the design.

Related Reading

User's Manual Sections: [Debugging with the Dataflow Window](#) and [Dataflow Window](#).

Compile and Load the Design

In this exercise you will use a DO file to compile and load the design.

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory and copy all files from *<install_dir>/examples/tutorials/verilog/dataflow* to the new directory.

If you have a VHDL license, copy the files in
<install_dir>/examples/tutorials/vhdl/dataflow instead.

2. Start QuestaSim and change to the exercise directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Execute the lesson DO file.

- a. Type **do run.do** at the QuestaSim> prompt.

The DO file does the following:

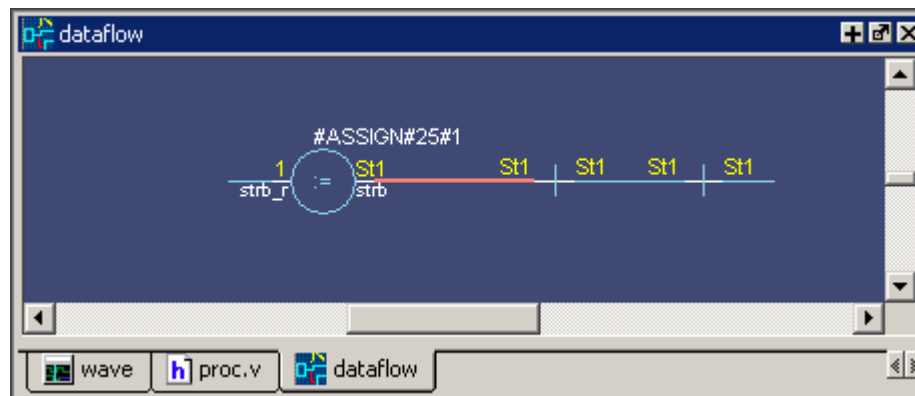
- Creates the working library
- Compiles the design files
- Opens the Dataflow window
- Loads the design into the simulator
- Adds signals to the Wave window
- Logs all signals in the design
- Runs the simulation

Exploring Connectivity

A primary use of the Dataflow window is exploring the "physical" connectivity of your design. You do this by expanding the view from process to process. This allows you to see the drivers/receivers of a particular signal, net, or register.

1. Add a signal to the Dataflow window.
 - a. Make sure instance *p* is selected in the **sim** tab of the Workspace pane.
 - b. Drag signal *strb* from the Objects pane to the Dataflow window ([Figure 9-1](#)).

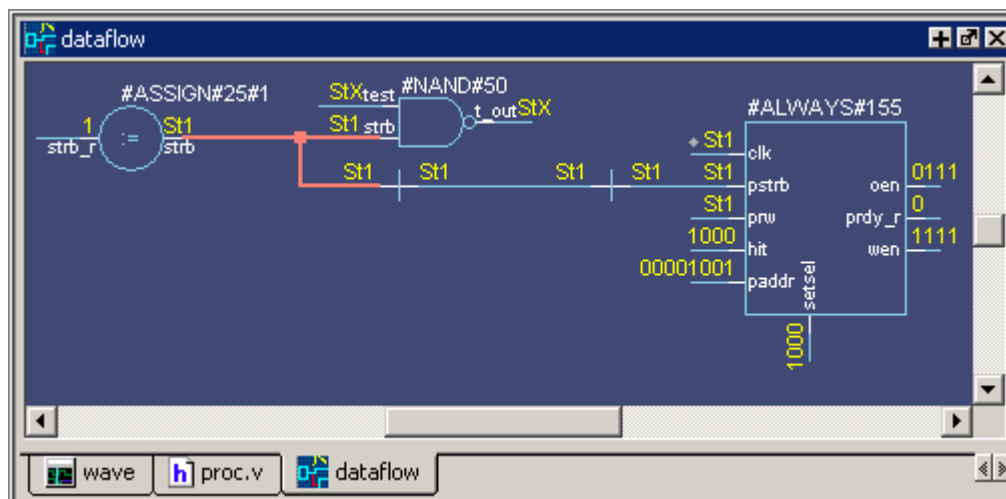
Figure 9-1. A Signal in the Dataflow Window



2. Explore the design.
 - a. Double-click the net highlighted in red.

The view expands to display the processes that are connected to *strb* (Figure 9-2).

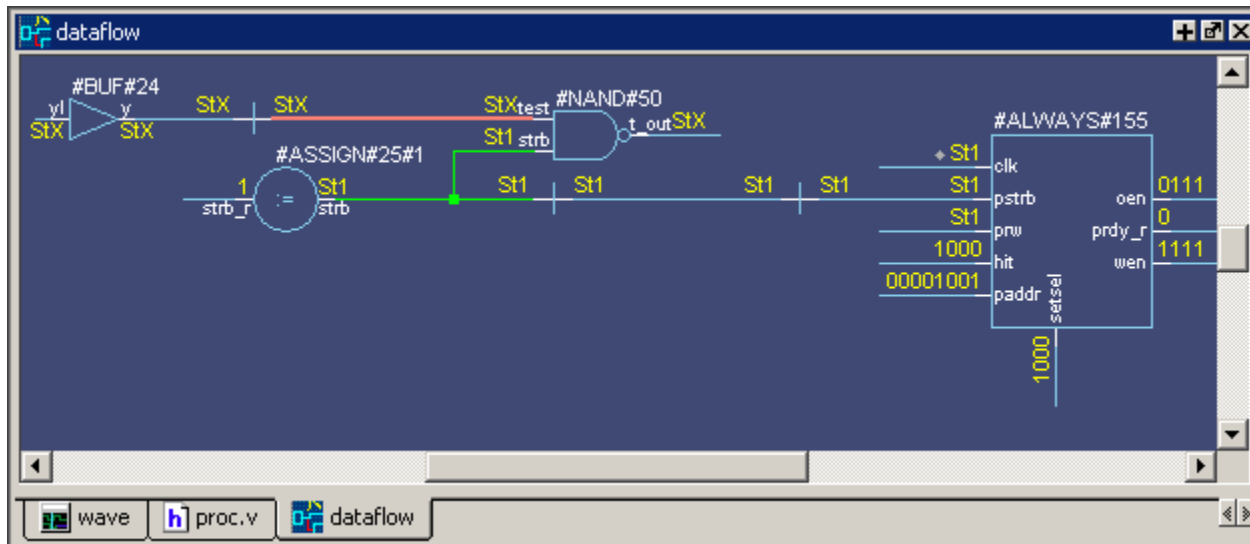
Figure 9-2. Expanding the View to Display Connected Processes



Select signal *test* on process *#NAND#50* (labeled *line_71* in the VHDL version) and click the **Expand net to all drivers** icon.



Figure 9-3. The test Net Expanded to Show All Drivers



Notice that after the display expands, the signal line for *strb* is highlighted in green. This highlighting indicates the path you have traversed in the design.

Select signal *oen* on process #ALWAYS#155(labeled *line_84* in the VHDL version), and click the **Expand net to all readers** icon.



Continue exploring if you wish.

When you are done, click the **Erase All** icon.

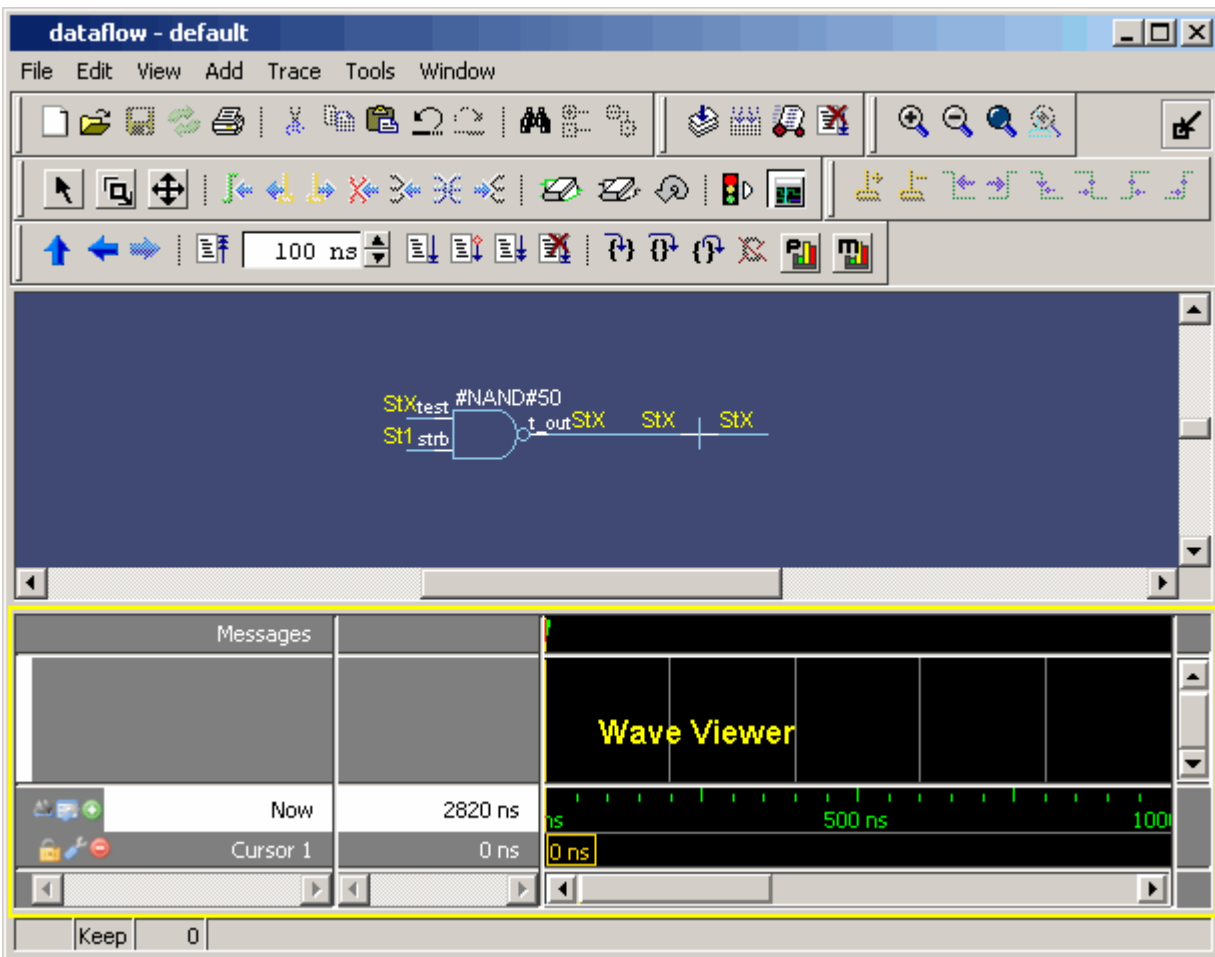


Tracing Events

Another useful debugging feature is tracing events that contribute to an unexpected output value. Using the Dataflow window's embedded wave viewer, you can trace backward from a transition to see which process or signal caused the unexpected output.

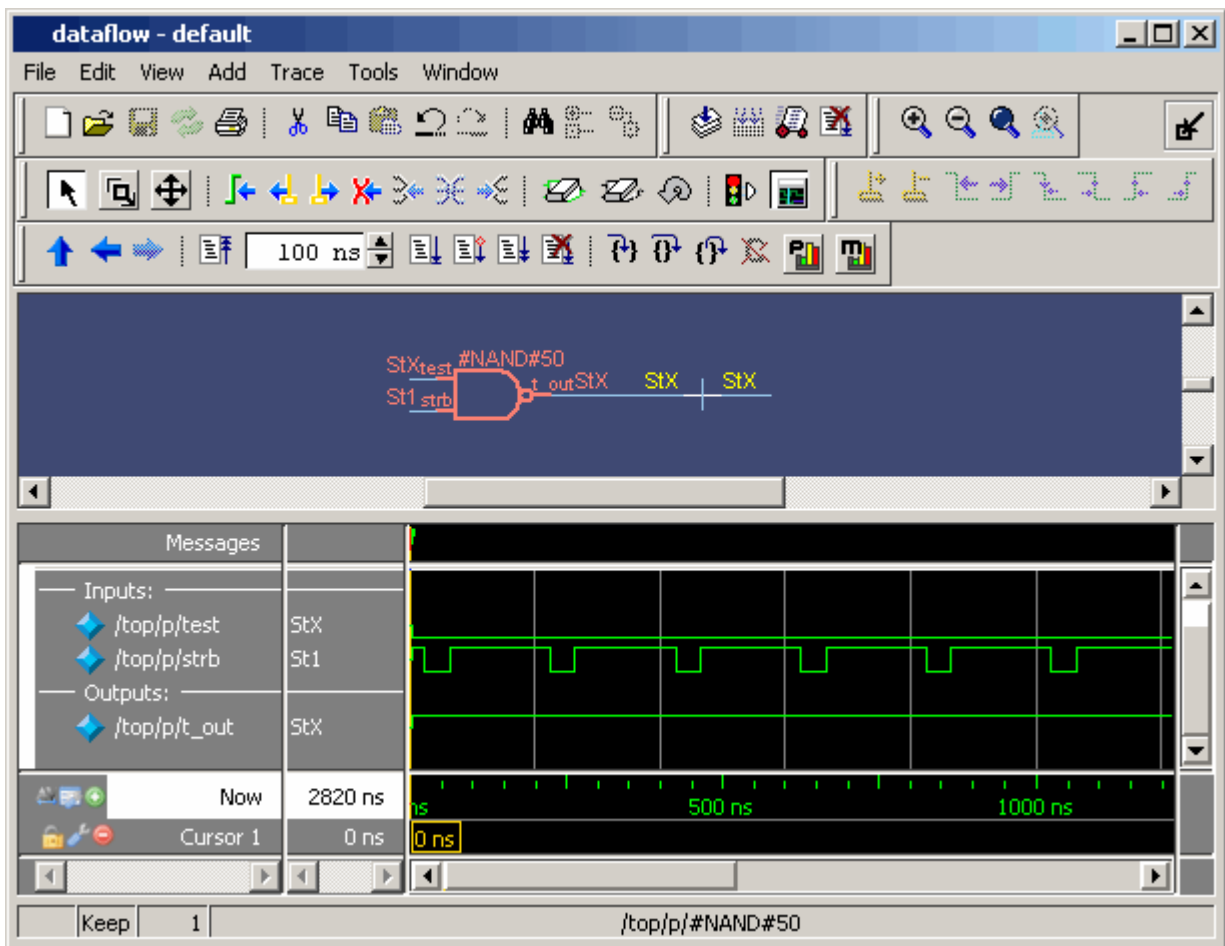
1. Add an object to the Dataflow window.
 - a. Make sure instance *p* is selected in the sim tab of the Main window.
 - b. Drag signal *t_out* from the Objects pane into the Dataflow window.
 - c. Undock the Dataflow window.
 - d. Select **View > Show Wave** in the Dataflow window to open the Wave Viewer (Figure 9-4). You may need to increase the size of the Dataflow window and scroll the panes to see everything.

Figure 9-4. The embedded wave viewer pane



2. Trace the inputs of the nand gate.
 - a. Select process `#NAND#50` (labeled `line_71` in the VHDL version) in the dataflow pane. All input and output signals of the process are displayed in the wave viewer (Figure 9-5).

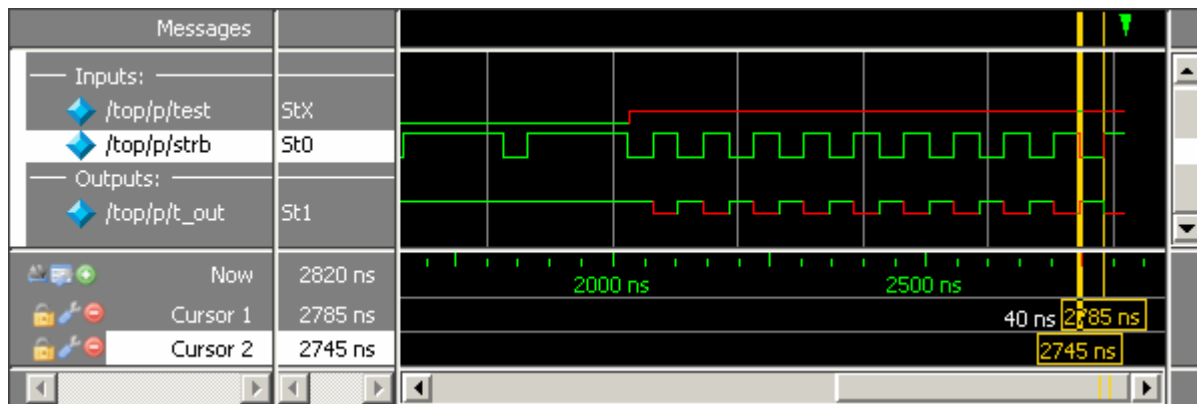
Figure 9-5. Signals Added to the Wave Viewer Automatically



- In the wave view, scroll to the last transition of signal t_out .
- Click just to the right of the last transition of signal t_out . The cursor should snap to time 2785 ns.
- Click on the t_out signal in the dataflow diagram to highlight it.
- Select **Trace > Trace next event** to trace the first contributing event.

QuestaSim adds a cursor marking the last event, the transition of the strobe to 0 at 2745 ns, which caused the output of 1 on t_out (Figure 9-6).

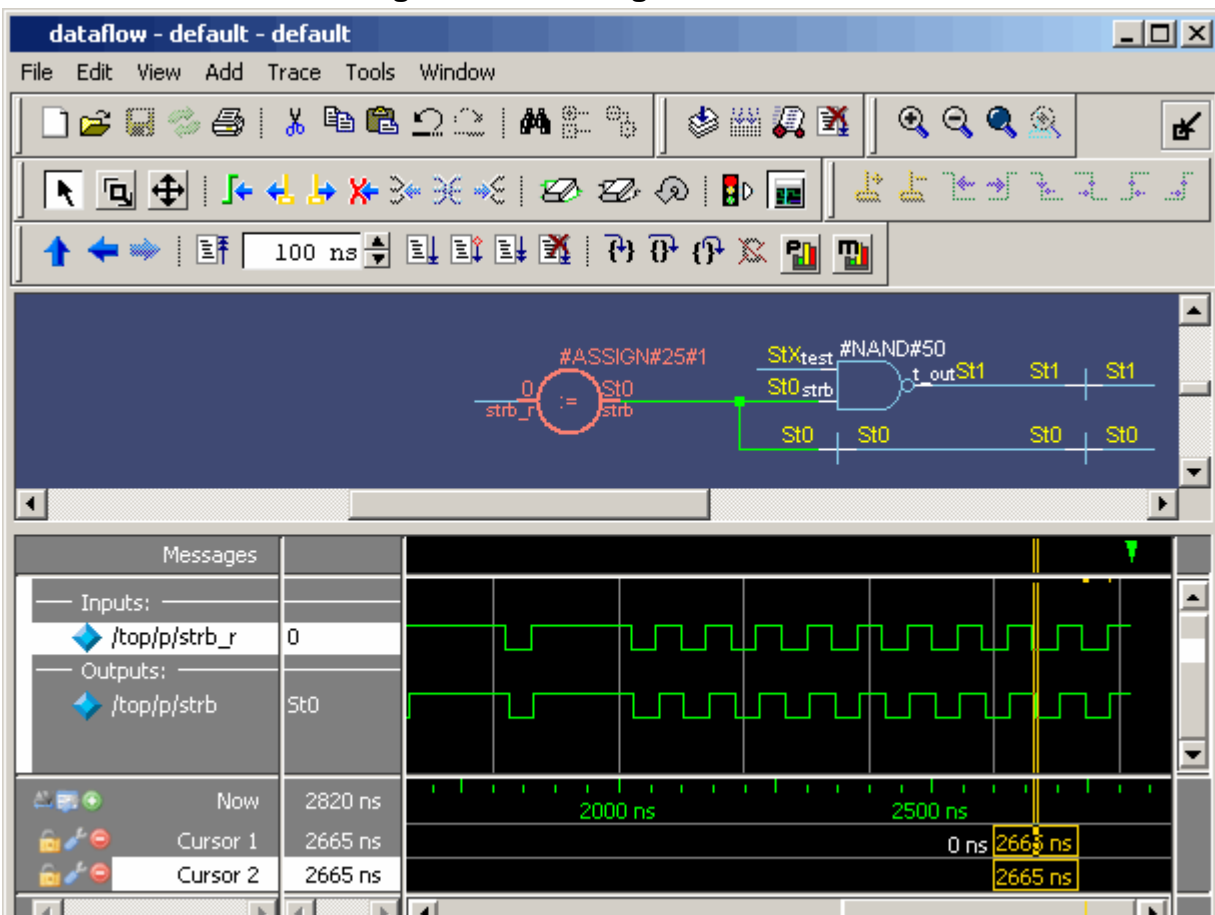
Figure 9-6. Cursor in Wave Viewer Marks Last Event



- f. Select **Trace > Trace next event** two more times.
- g. Select **Trace > Trace event set**.

The dataflow pane sprouts to the preceding process and shows the input driver of the *strb* signal (Figure 9-7). Notice, also, that the wave viewer now shows the input and output signals of the newly selected process.

Figure 9-7. Tracing the Event Set



You can continue tracing events through the design in this manner: select **Trace next event** until you get to a transition of interest in the wave viewer, and then select **Trace event set** to update the dataflow pane.

3. Select **File > Close Window** to close the Dataflow window.

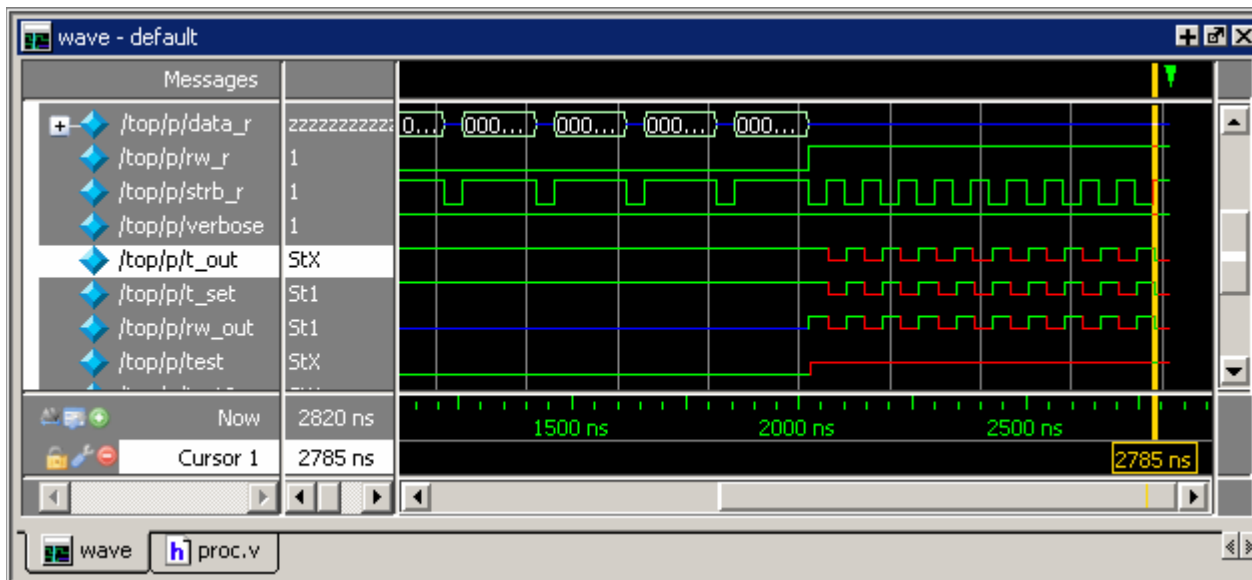
Tracing an X (Unknown)

The Dataflow window lets you easily track an unknown value (X) as it propagates through the design. The Dataflow window is linked to the Wave window, so you can view signals in the Wave window and then use the Dataflow window to track the source of a problem. As you traverse your design in the Dataflow window, appropriate signals are added automatically to the Wave window.

1. View t_out in the Wave and Dataflow windows.
 - a. Scroll in the Wave window until you can see `/top/p/t_out`.

t_out goes to an unknown state, StX, at 2065 ns and continues transitioning between 1 and unknown for the rest of the run (Figure 9-8). The red color of the waveform indicates an unknown value.

Figure 9-8. A Signal with Unknown Values



- b. Double-click the t_out waveform at the last transition of signal t_out at 2785 ns.

This automatically opens a **dataflow** tab in the MDI frame and displays t_out , its associated process, and its waveform. You may need to increase the size of the Dataflow window and scroll the panes to see everything.

- c. Undock the Dataflow window.

- d. Move the cursor in the Wave window.

As previously mentioned the Wave and Dataflow windows are designed to work together. As you move the cursor in the Wave, the value of t_out changes in the flow diagram portion of the Dataflow window.

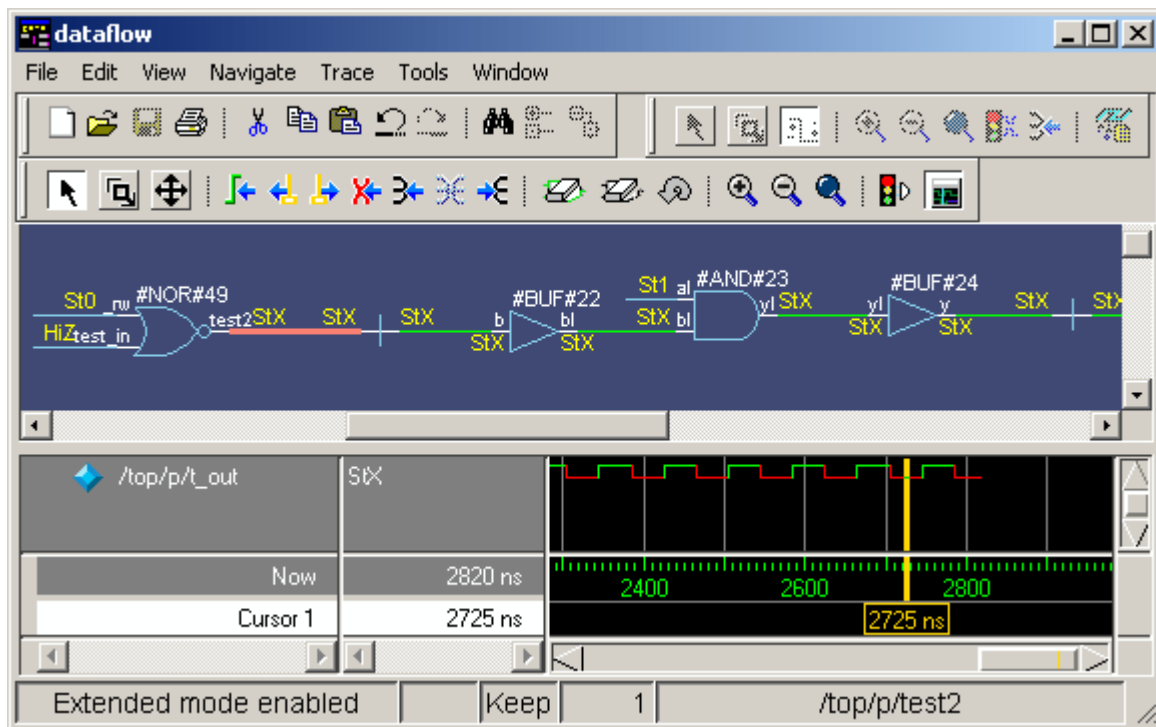
- e. Move the cursor in the Wave Viewer of the Dataflow window to a time when t_out is unknown (e.g., 2725 ns). If the Wave Viewer is not showing, click the Show Wave icon or select **View > Show Wave**.

2. Trace the unknown.

- a. In the Dataflow window, make sure t_out is selected and then select **Trace > ChaseX**.

The design expands to show the source of the unknown (Figure 9-9). In this case there is a HiZ (U in the VHDL version) on input signal $test_in$ and a 0 on input signal $_rw$ (bar_rw in the VHDL version). This causes the $test2$ output signal to resolve to an unknown state (StX). The unknown state propagates through the design to t_out .

Figure 9-9. ChaseX Identifies Cause of Unknown on t_out



Scroll to the bottom of the Wave window, and you will see that all of the signals contributing to the unknown value have been added.

3. Clear the Dataflow window before continuing.
 - a. Click the **Erase All** icon to clear the Dataflow view.

- b. Click the Show Wave icon to close the Wave view of the Dataflow window.

Displaying Hierarchy in the Dataflow Window

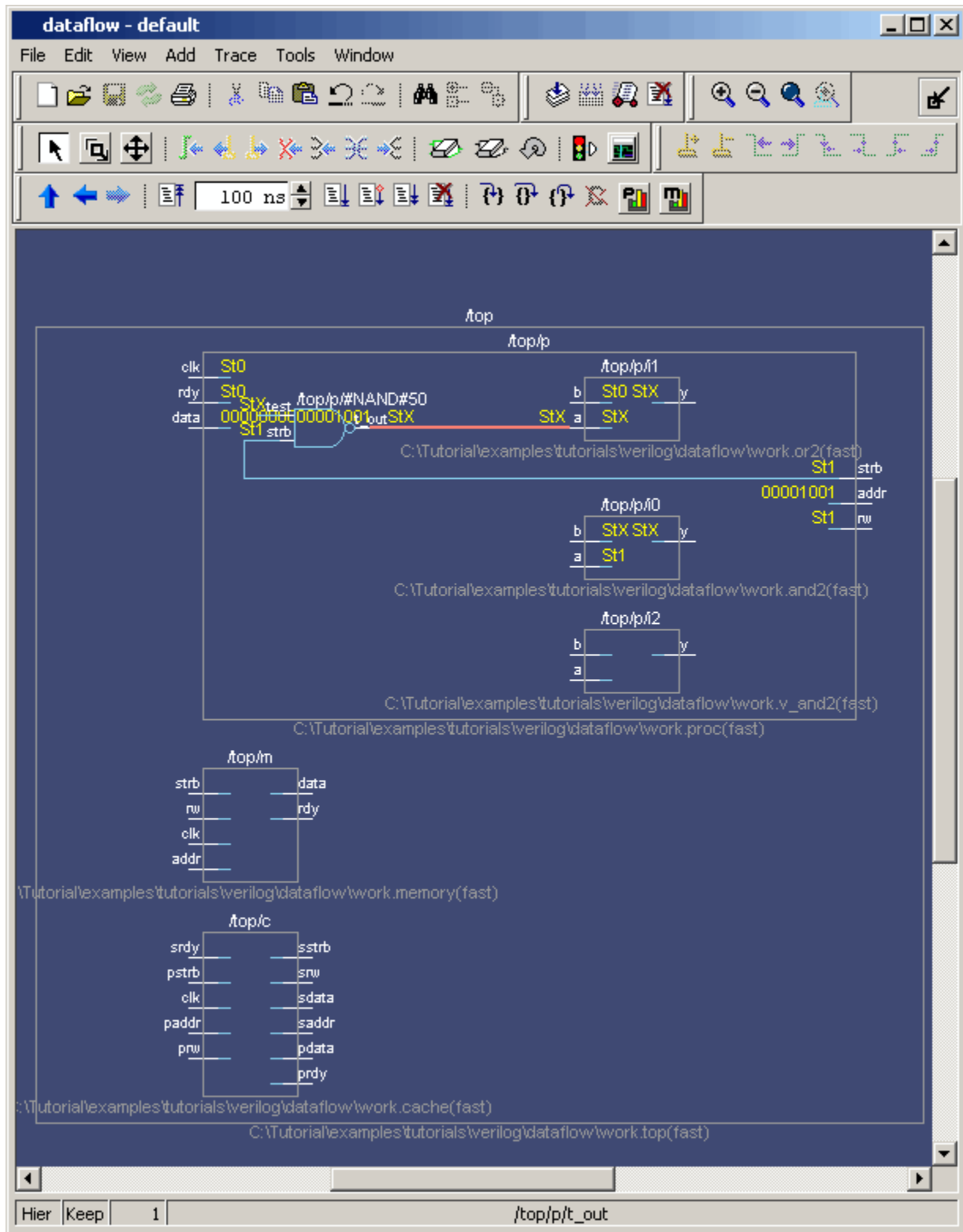
You can display connectivity in the Dataflow window using hierarchical instances. You enable this by modifying the options prior to adding objects to the window.

1. Change options to display hierarchy.
 - a. With the Dataflow window undocked, select **Tools > Options** from the Dataflow window menu bar.

With the Dataflow window docked, and the dataflow tab selected in the MDI frame, select **Dataflow > Dataflow Preferences > Options** from the Main window menus.
 - b. Check **Show Hierarchy** and then click **OK**.
2. Add signal *t_out* to the Dataflow window.
 - a. Type **add dataflow /top/p/t_out** at the VSIM> prompt.

The Dataflow window will display *t_out* and all hierarchical instances ([Figure 9-10](#)).

Figure 9-10. Displaying Hierarchy in the Dataflow Window



Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. Type **quit -sim** at the VSIM> prompt.

Chapter 10

Viewing And Initializing Memories

Introduction

In this lesson you will learn how to view and initialize memories in QuestaSim. QuestaSim defines and lists as memories any of the following:

- reg, wire, and std_logic arrays
- Integer arrays
- Single dimensional arrays of VHDL enumerated types other than std_logic

Design Files for this Lesson

The QuestaSim installation comes with Verilog and VHDL versions of the example design. The files are located in the following directories:

Verilog – *<install_dir>/examples/tutorials/verilog/memory*

VHDL – *<install_dir>/examples/tutorials/vhdl/memory*

This lesson uses the Verilog version for the exercises. If you have a VHDL license, use the VHDL version instead.

Related Reading

User's Manual Section: [Memory Panes](#).

Reference Manul commands: [mem display](#), [mem load](#), [mem save](#), and [radix](#).

Compile and Load the Design

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory and copy all files from *<install_dir>/examples/tutorials/verilog/memory* to the new directory.

If you have a VHDL license, copy the files in *<install_dir>/examples/tutorials/vhdl/memory* instead.

2. Start QuestaSim and change to the exercise directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.
3. Create the working library and compile the design.

- a. Type **vlib work** at the QuestaSim> prompt.

- b. **Verilog:**

Type **vlog sp_syn_ram.v dp_syn_ram.v ram_tb.v** at the QuestaSim> prompt.

VHDL:

Type **vcom -93 sp_syn_ram.vhd dp_syn_ram.vhd ram_tb.vhd** at the QuestaSim> prompt.

4. Load the design.

- a. Enter the following command at the QuestaSim> prompt in the Transcript window.

vsim -voptargs="+acc" ram_tb

The **-voptargs="+acc"** argument for the **vsim** command provides visibility into the design for debugging purposes.

Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

View a Memory and its Contents

The Memories tab of the Main window lists all memories in the design when the design is loaded; with the range, depth, and width of each memory displayed.

VHDL: The radix for enumerated types is Symbolic. To change the radix to binary for the purposes of this lesson, type the following command at the VSIM> prompt:

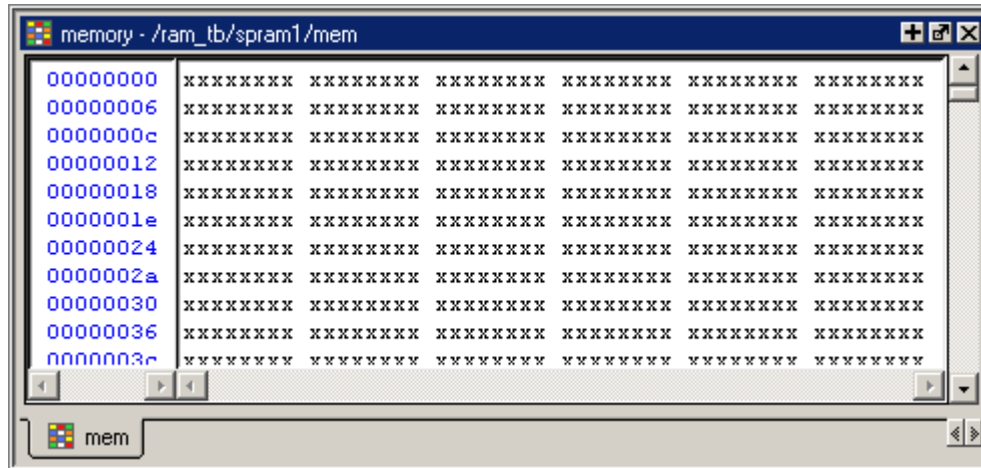
radix bin

1. Open a Memory instance to show its contents.
 - a. Double-click the `/ram_tb/spram1/mem` instance in the memories list to view its contents in the MDI frame.

A **mem** tab is created in the MDI frame to display the memory contents. The data are all **X** (**0** in VHDL) since you have not yet simulated the design. The first column

(blue hex characters) lists the addresses (Figure 10-1), and the remaining columns show the data values.

Figure 10-1. The mem Tab in the MDI Frame Shows Addresses and Data




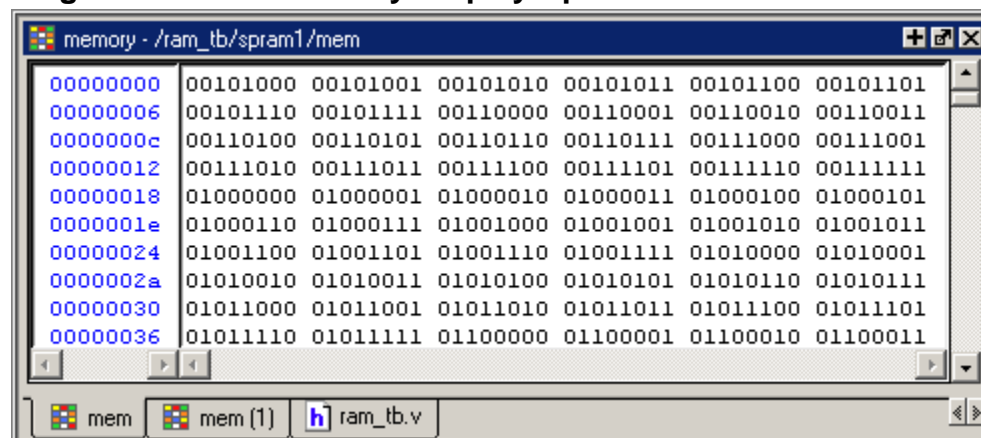
- b. Double-click instance `/ram_tb/spram2/mem` in the Memories tab of the Workspace. This creates a new tab in the MDI frame called **mem(1)** that contains the addresses and data for the `spram2` instance. Each time you double-click a new memory instance in the Workspace, a new tab is created for that instance in the MDI frame.
2. Simulate the design.
 - a. Click the **run -all** icon in the Main window. 
 - b. Click the **mem** tab of the MDI frame to bring the `/ram_tb/spram1/mem` to the foreground. The data fields now show values (Figure 10-2).

Figure 10-2. The Memory Display Updates with the Simulation



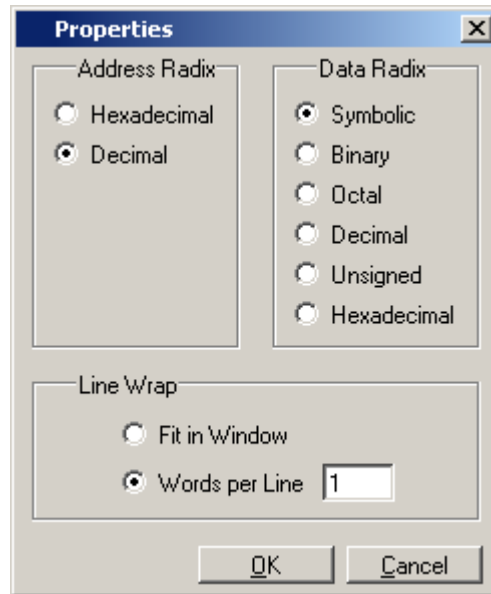
VHDL:

In the Transcript pane, you will see NUMERIC_STD warnings that can be ignored and

an assertion failure that is functioning to stop the simulation. The simulation itself has not failed.

3. Change the address radix and the number of words per line for instance */ram_tb/spram1/mem*.
 - a. Right-click anywhere in the Memory Contents pane and select **Properties**.
 - b. The Properties dialog box opens ([Figure 10-3](#)).

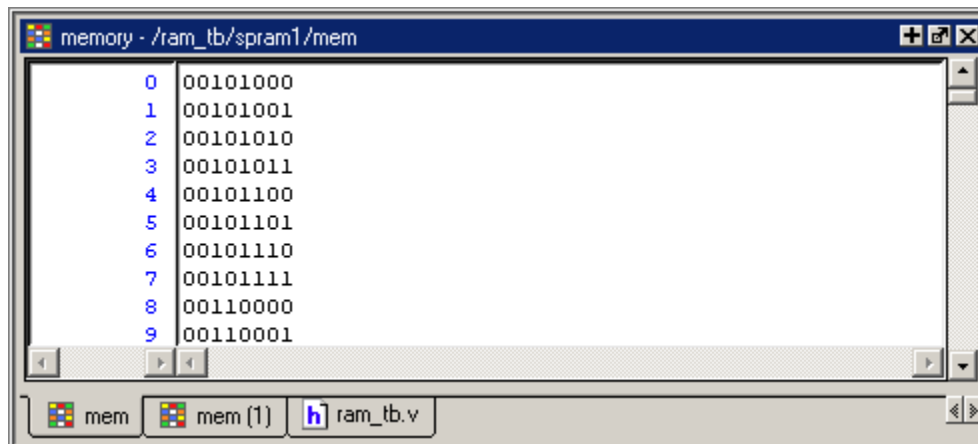
Figure 10-3. Changing the Address Radix



- c. For the **Address Radix**, select **Decimal**. This changes the radix for the addresses only.
 - d. Select **Words per line** and type **1** in the field.
 - e. Click OK.

You can see the results of the settings in [Figure 10-4](#). If the figure doesn't match what you have in your QuestaSim session, check to make sure you set the Address Radix rather than the Data Radix. Data Radix should still be set to Symbolic, the default.

Figure 10-4. New Address Radix and Line Length



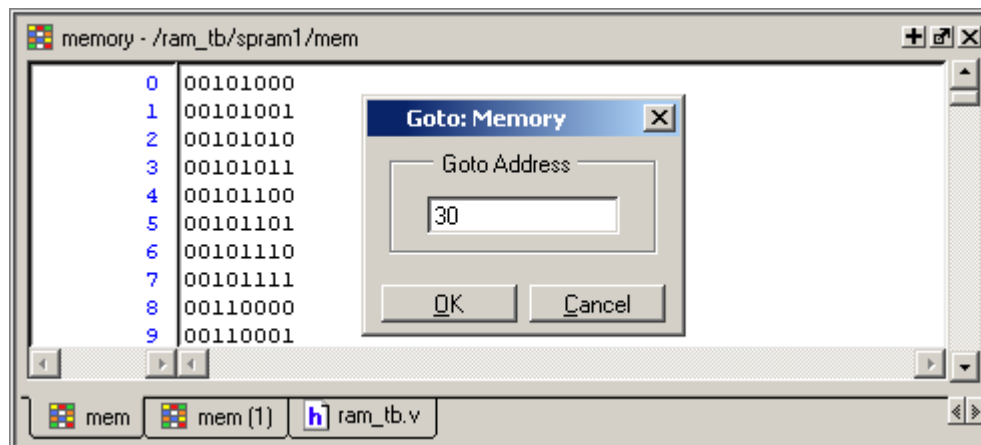
Navigate Within the Memory

You can navigate to specific memory address locations, or to locations containing particular data patterns. First, you will go to a specific address.

1. Use Goto to find a specific address.
 - a. Right-click anywhere in address column and select **Goto** (Figure 10-5).

The Goto dialog box opens in the data pane.

Figure 10-5. Goto Dialog



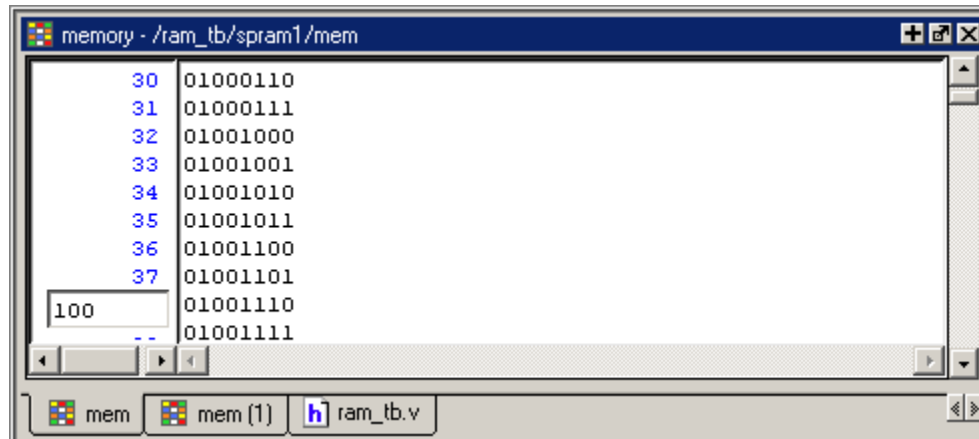
- b. Type **30** in the Goto Address field.
 - c. Click OK.

The requested address appears in the top line of the window.

2. Edit the address location directly.

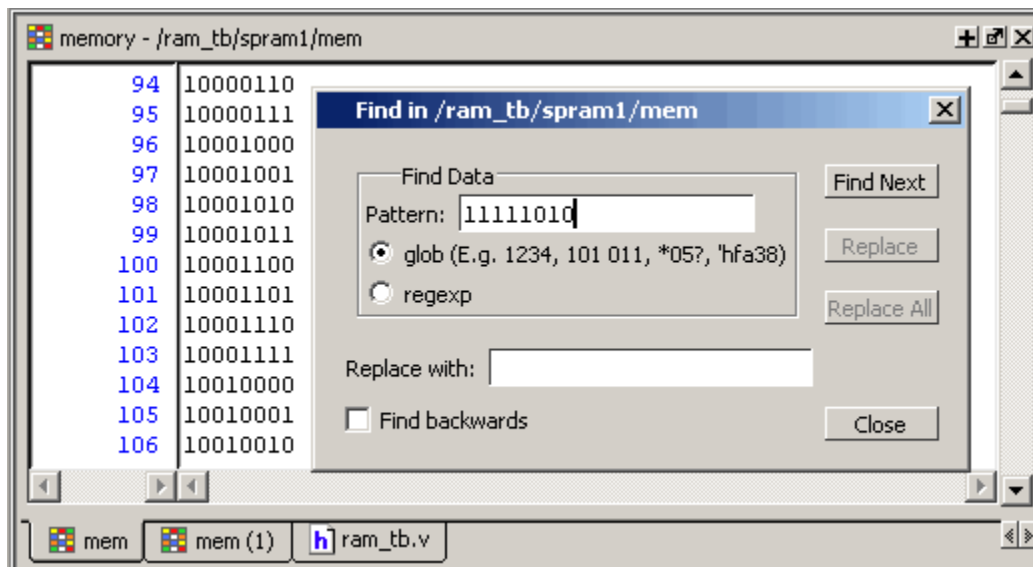
- a. To quickly move to a particular address, do the following:
 - i. Double click address 38 in the address column.
 - ii. Enter address 100 (Figure 10-6).

Figure 10-6. Editing the Address Directly



- iii. Press <Enter> on your keyboard.
 - The pane scrolls to that address.
 3. Now, let's find a particular data entry.
 - a. Right-click anywhere in the data column and select **Find**.
- The Find in dialog box opens (Figure 10-7).

Figure 10-7. Searching for a Specific Data Value



- b. Type **11111010** in the **Find data:** field and click **Find Next**.

The data scrolls to the first occurrence of that address. Click **Find Next** a few more times to search through the list.

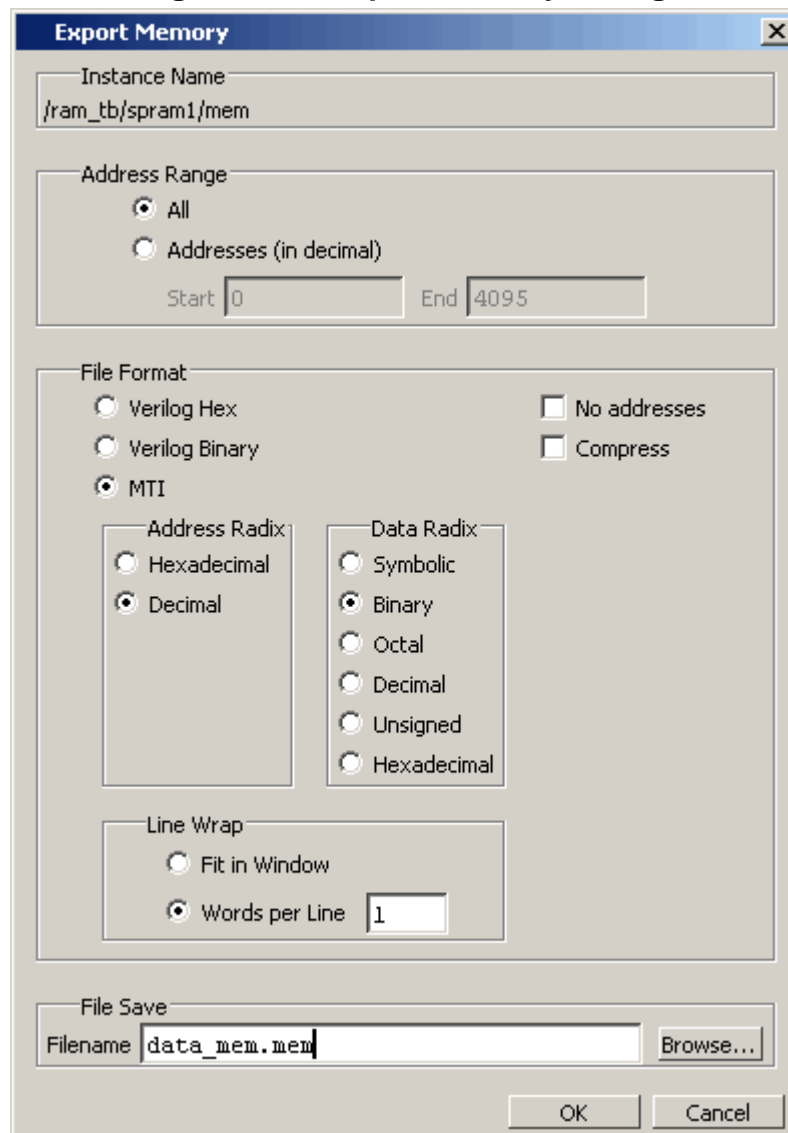
- c. Click **Close** to close the dialog box.

Export Memory Data to a File

You can save memory data to a file that can be loaded at some later point in simulation.

1. Export a memory pattern from the */ram_tb/spram1/mem* instance to a file.
 - a. Make sure */ram_tb/spram1/mem* is open and selected in the MDI frame.
 - b. Select **File > Export > Memory Data** to bring up the Export Memory dialog box (Figure 10-8).

Figure 10-8. Export Memory Dialog



- c. For the Address Radix, select **Decimal**.
- d. For the Data Radix, select **Binary**.
- e. For the Line Wrap, set to 1 word per line.
- f. Type **data_mem.mem** into the Filename field.
- g. Click OK.

You can view the exported file in any editor.

Memory pattern files can be exported as relocatable files, simply by leaving out the address information. Relocatable memory files can be loaded anywhere in a memory because no addresses are specified.

2. Export a relocatable memory pattern file from the */ram_tb/spram2/mem* instance.
 - a. Select the **mem(1)** tab in the MDI pane to see the data for the */ram_tb/spram2/mem* instance.
 - b. Right-click on the memory contents to open a popup menu and select **Properties**.
 - c. In the Properties dialog, set the Address Radix to **Decimal**; the Data Radix to **Binary**; and the Line Wrap to 1 **Words per Line**. Click OK to accept the changes and close the dialog.
 - d. Select **File > Export > Memory Data** to bring up the Export Memory dialog box.
 - e. For the Address Range, specify a Start address of **0** and End address of **250**.
 - f. For the File Format, select **MTI** and click **No addresses** to create a memory pattern that you can use to relocate somewhere else in the memory, or in another memory.
 - g. For Address Radix select **Decimal**, and for Data Radix select **Binary**.
 - h. For the Line Wrap, set 1 **Words per Line**.
 - i. Enter the file name as **reloc.mem**, then click OK to save the memory contents and close the dialog. You will use this file for initialization in the next section.

Initialize a Memory

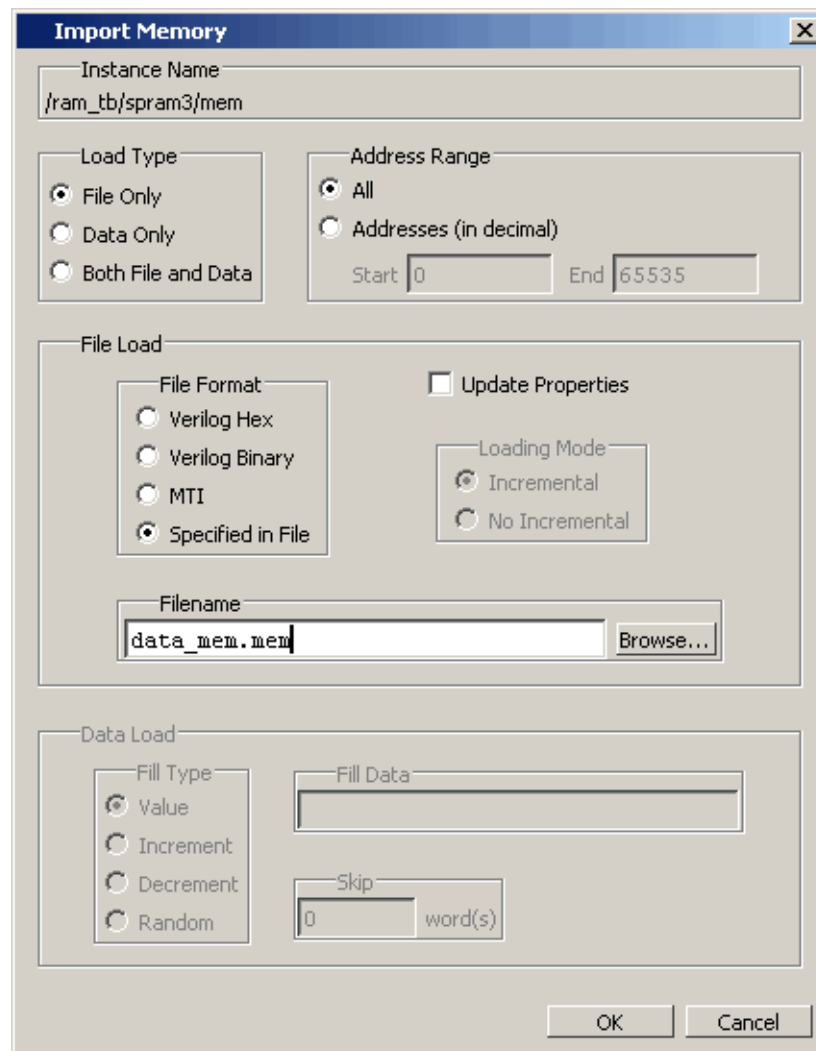
In QuestaSim, it is possible to initialize a memory using one of three methods: from an exported memory file, from a fill pattern, or from both.

First, let's initialize a memory from a file only. You will use one you exported previously, *data_mem.mem*.

1. View instance */ram_tb/spram3/mem*.
 - a. Double-click the */ram_tb/spram3/mem* instance in the Memories tab.

This will open a new tab – **mem(2)** – in the MDI frame to display the contents of */ram_tb/spram3/mem*. Scan these contents so you can identify changes once the initialization is complete.
 - b. Right-click and select **Properties** to bring up the Properties dialog.
 - c. Change the Address Radix to **Decimal**, Data Radix to **Binary**, **Line Wrap to 1 Words per Line**, and click OK.
2. Initialize *spram3* from a file.
 - a. Right-click anywhere in the data column and select **Import** to bring up the Import Memory dialog box ([Figure 10-9](#)).

Figure 10-9. Import Memory Dialog



The dialog box is titled "Import Memory" and contains the following sections:

- Instance Name:** A text field containing the path `/ram_tb/spram3/mem`.
- Load Type:** Three radio buttons:
☒ File Only
☐ Data Only
☐ Both File and Data
- Address Range:** Two radio buttons:
☒ All
☐ Addresses (in decimal)
Below these are two text fields: "Start" with value `0` and "End" with value `65535`.
- File Load:** A group box containing:
 - File Format:** Four radio buttons:
☐ Verilog Hex
☐ Verilog Binary
☐ MTI
☒ Specified in File
 - ☐ Update Properties
 - Loading Mode:** Two radio buttons:
☒ Incremental
☐ No Incremental
 - Filename:** A text field containing `data_mem.mem` and a "Browse..." button.
- Data Load:** A group box containing:
 - Fill Type:** Four radio buttons:
☒ Value
☐ Increment
☐ Decrement
☐ Random
 - Fill Data:** An empty text field.
 - Skip:** A text field containing `0` followed by the label "word(s)".

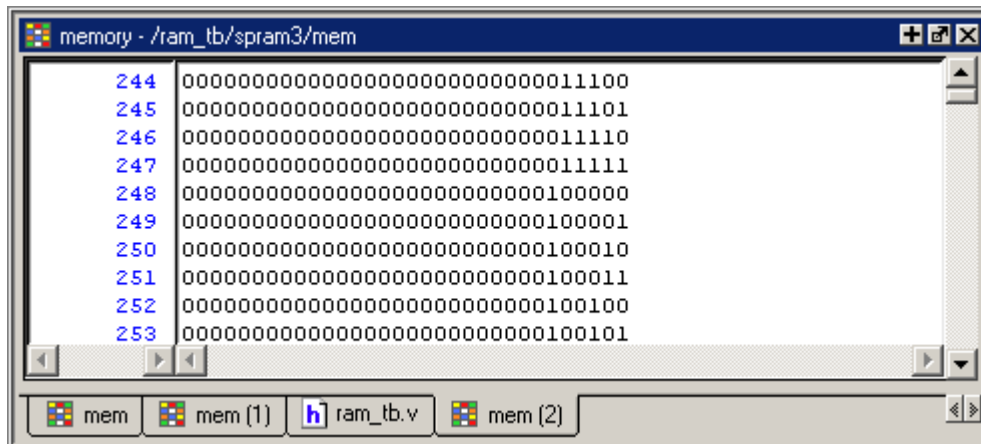
At the bottom right are "OK" and "Cancel" buttons.

The default Load Type is File Only.

- b. Type `data_mem.mem` in the Filename field.
- c. Click **OK**.

The addresses in instance `/ram_tb/spram3/mem` are updated with the data from `data_mem.mem` (Figure 10-10).

Figure 10-10. Initialized Memory from File and Fill Pattern



In this next step, you will experiment with importing from both a file and a fill pattern. You will initialize *spram3* with the 250 addresses of data you exported previously into the relocatable file *reloc.mem*. You will also initialize 50 additional address entries with a fill pattern.

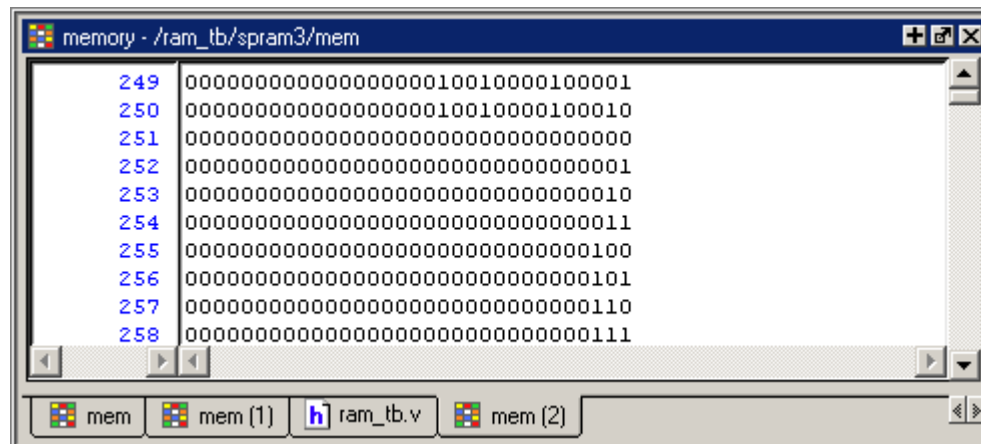
3. Import the `/ram_tb/spram3/mem` instance with a relocatable memory pattern (`reloc.mem`) and a fill pattern.
 - a. Right-click in the data column of the **mem(2)** tab and select **Import** to bring up the Import Memory dialog box.
 - b. For Load Type, select **Both File and Data**.
 - c. For Address Range, select **Addresses** and enter **0** as the Start address and **300** as the End address.

This means that you will be loading the file from 0 to 300. However, the *reloc.mem* file contains only 251 addresses of data. Addresses 251 to 300 will be loaded with the fill data you specify next.

- d. For File Load, select the MTI File Format and enter **reloc.mem** in the Filename field.
- e. For Data Load, select a Fill Type of **Increment**.
- f. In the Fill Data field, set the seed value of **0** for the incrementing data.
- g. Click **OK**.
- h. View the data near address 250 by double-clicking on any address in the Address column and entering **250**.

You can see the specified range of addresses overwritten with the new data. Also, you can see the incrementing data beginning at address 251 ([Figure 10-11](#)).

Figure 10-11. Data Increments Starting at Address 251



Now, before you leave this section, go ahead and clear the memory instances already being viewed.

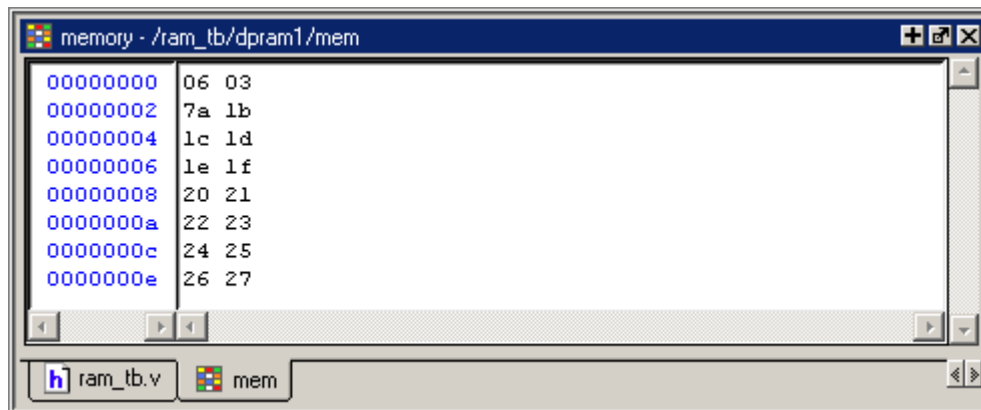
4. Right-click somewhere in the **mem(2)** pane and select **Close All**.

Interactive Debugging Commands

The memory panes can also be used interactively for a variety of debugging purposes. The features described in this section are useful for this purpose.

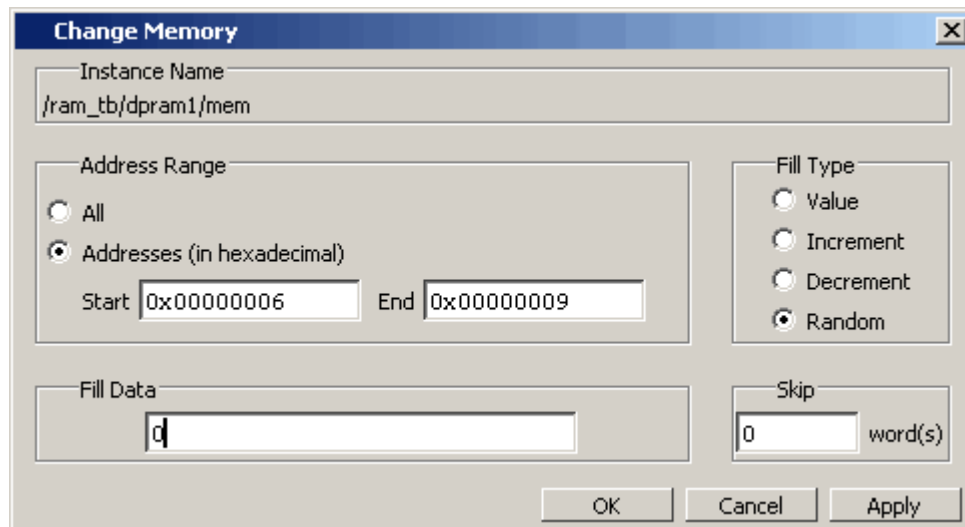
1. Open a memory instance and change its display characteristics.
 - a. Double-click instance */ram_tb/dpram1/mem* in the Memories tab.
 - b. Right-click in the memory contents pane and select **Properties**.
 - c. Change the Address and Data Radix to **Hexadecimal**.
 - d. Select **Words per line** and enter **2**.
 - e. Click **OK**. The result should be as in [Figure 10-12](#).

Figure 10-12. Original Memory Content



2. Initialize a range of memory addresses from a fill pattern.
 - a. Right-click in the data column of `/ram_tb/dpram1/mem` contents pane and select **Change** to open the Change Memory dialog (Figure 10-13).

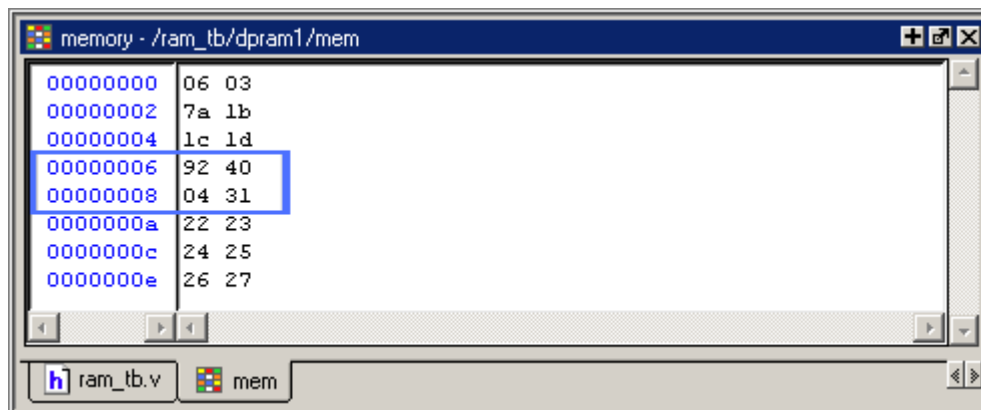
Figure 10-13. Changing Memory Content for a Range of Addresses



- b. Select **Addresses** and enter the start address as **0x00000006** and the end address as **0x00000009**. The "0x" hex notation is optional.
 - c. Select **Random** as the **Fill Type**.
 - d. Enter **0** as the **Fill Data**, setting the seed for the Random pattern.
 - e. Click **OK**.

The data in the specified range are replaced with a generated random fill pattern (Figure 10-14).

Figure 10-14. Random Content Generated for a Range of Addresses

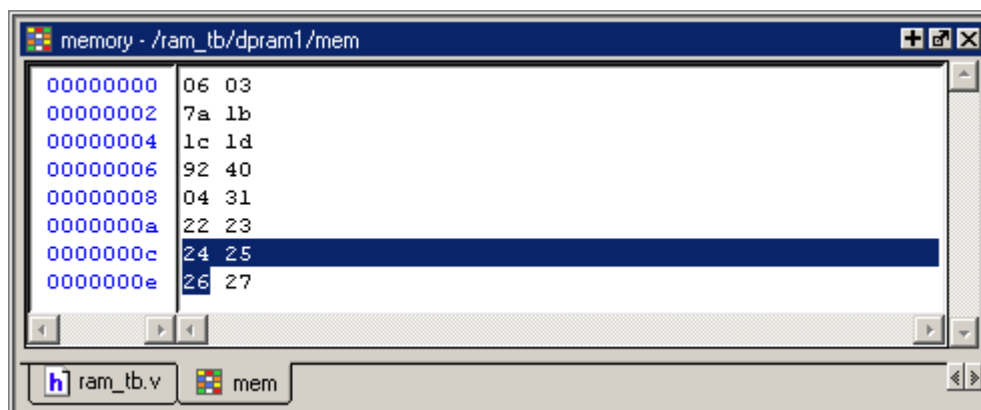


3. Change contents by highlighting.

You can also change data by highlighting them in the Address Data pane.

- a. Highlight the data for the addresses **0x0000000c:0x0000000e**, as shown in [Figure 10-15](#).

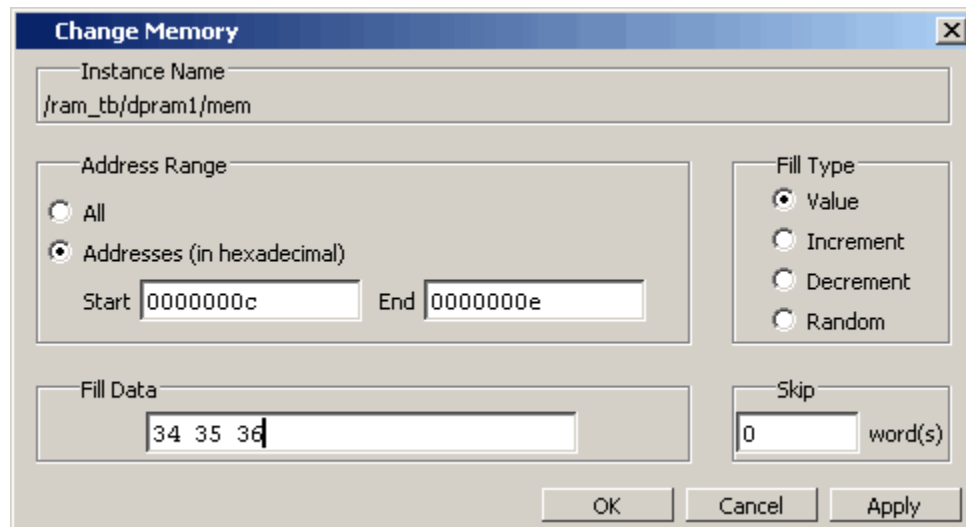
Figure 10-15. Changing Memory Contents by Highlighting



- b. Right-click the highlighted data and select **Change**.

This brings up the Change memory dialog box ([Figure 10-16](#)). Note that the Addresses field is already populated with the range you highlighted.

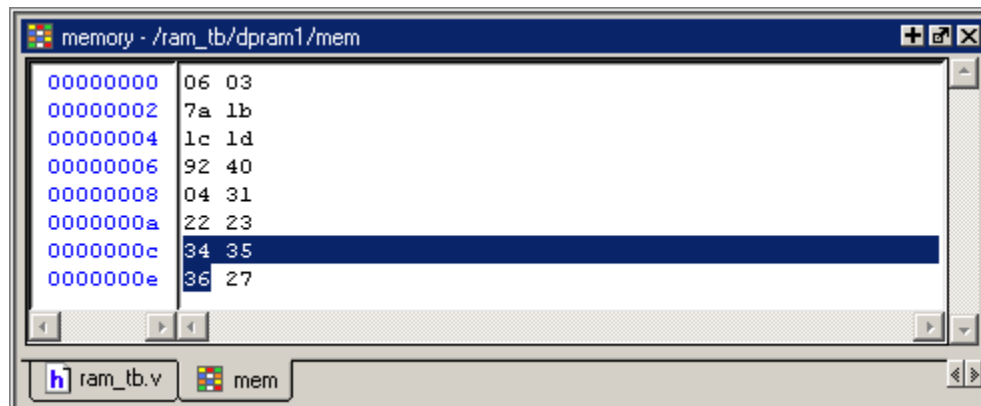
Figure 10-16. Entering Data to Change



- c. Select **Value** as the Fill Type.
- d. Enter the data values into the Fill Data field as follows: **34 35 36**
- e. Click **OK**.

The data in the address locations change to the values you entered ([Figure 10-17](#)).

Figure 10-17. Changed Memory Contents for the Specified Addresses



4. Edit data in place.

To edit only one value at a time, do the following:

- a. Double click any value in the Data column.
- b. Enter the desired value and press <Enter> on your keyboard.

If you needed to cancel the edit function, press the <Esc> key on your keyboard.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. Select **Simulate > End Simulation**. Click Yes.

Chapter 11

Analyzing Performance With The Profiler

Introduction

The Profiler identifies the percentage of simulation time spent in each section of your code as well as the amount of memory allocated to each function and instance. With this information, you can identify bottlenecks and reduce simulation time by optimizing your code. Users have reported up to 75% reductions in simulation time after using the Profiler.

This lesson introduces the Profiler and shows you how to use the main Profiler commands to identify performance bottlenecks.

Note



The functionality described in this tutorial requires a profile license feature in your QuestaSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Design Files for this Lesson

The example design for this lesson consists of a finite state machine which controls a behavioral memory. The testbench *test_sm* provides stimulus.

The QuestaSim installation comes with Verilog and VHDL versions of this design. The files are located in the following directories:

Verilog – `<install_dir>/examples/tutorials/verilog/profiler`

VHDL – `<install_dir>/examples/tutorials/vhdl/profiler_sm_seq`

This lesson uses the Verilog version for the exercises. If you have a VHDL license, use the VHDL version instead.

Related Reading

User's Manual Chapters: [Profiling Performance and Memory Use](#) and [Tcl and Macros \(DO Files\)](#).

Compile and Load the Design

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory and copy all files from `<install_dir>/examples/tutorials/verilog/profiler` to the new directory.

If you have a VHDL license, copy the files in `<install_dir>/examples/tutorials/vhdl/profiler_sm_seq` instead.

2. Start QuestaSim and change to the exercise directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Create the work library.

- a. Type **vlib work** at the QuestaSim> prompt.

4. Compile the design files.

- a. Verilog: Type **vlog test_sm.v sm_seq.v sm.v beh_sram.v** at the QuestaSim> prompt.

VHDL: Type **vcom -93 sm.vhd sm_seq.vhd sm_sram.vhd test_sm.vhd** at the QuestaSim> prompt.

5. Load the top-level design unit.

- a. Enter **vsim -voptargs="+acc" test_sm** at the QuestaSim> prompt of the Transcript pane.

The **-voptargs="+acc"** argument for the **vsim** command provides visibility into the design for debugging purposes.

Note

By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

Run the Simulation

You will now run the simulation and view the profiling data.

1. Enable the statistical sampling profiler.

- a. Select **Tools > Profile > Performance** or click the **Performance Profiling** icon in the toolbar.



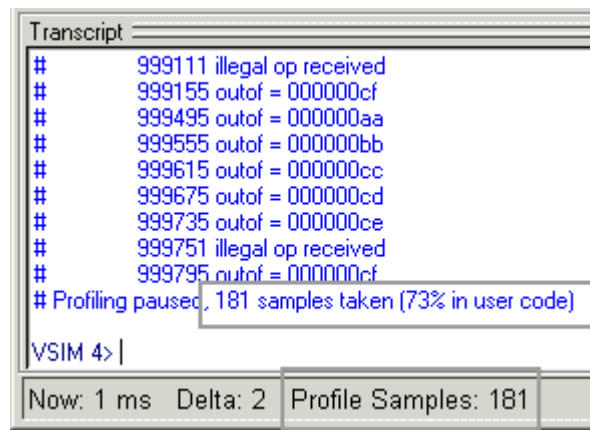
This must be done prior to running the simulation. QuestaSim is now ready to collect performance data when the simulation is run.

2. Run the simulation.

- a. Type **run 1 ms** at the VSIM> prompt.

Notice that the number of samples taken is displayed both in the Transcript and the Main window status bar (Figure 11-1). (Your results may not match those in the figure.) Also, QuestaSim reports the percentage of samples that were taken in your design code (versus in internal simulator code).

Figure 11-1. Sampling Reported in the Transcript



3. Display the statistical performance data in the Profile pane.

- a. Select **View > Profiling > Profile**.

The Profile pane (you may need to increase its size) displays four tab-selectable views of the data—Ranked, Design Units, Call Tree, and Structural (Figure 11-2). (Your results may not match those in the figure.)

Figure 11-2. The Profile Window

Name	Under(row)	In(row)	Under(%)	In(%)
Tcl_WaitForEvent	72	72	53.3%	53.3%
test_sm.v:105	85	17	63.0%	12.6%
sm.v:73	17	5	12.6%	3.7%
TclpHasSockets	4	3	3.0%	2.2%
Tcl_GetTime	3	3	2.2%	2.2%
test_sm.v:92	3	3	2.2%	2.2%
Tcl_OpenTcpServer	2	2	1.5%	1.5%
Tcl_DoOneEvent	79	0	58.5%	0.0%
Tcl_DeleteTimerHandler	3	0	2.2%	0.0%
Tcl_Flush	2	0	1.5%	0.0%

The table below gives a description of the columns in each tab. For more details on each pane, refer to the section [Viewing Profiler Results](#) in the User's Manual.

Table 11-1. Columns in the Profile Window

Column	Description
Count	(Design Unit view only) quantity of design objects analyzed
Under(row)	the raw number of Profiler samples collected during the execution of a function, including all support routines under that function; or, the number of samples collected for an instance, including all instances beneath it in the structural hierarchy
In(row)	the raw number of Profiler samples collected during a function or instance
Under(%)	the ratio (as a percentage) of the samples collected during the execution of a function and all support routines under that function to the total number of samples collected; or, the ratio of the samples collected during an instance, including all instances beneath it in the structural hierarchy, to the total number of samples collected
In(%)	the ratio (as a percentage) of the total samples collected during a function or instance
%Parent	(not in the Ranked view) the ratio (as a percentage) of the samples collected during the execution of a function or instance to the samples collected in the parent function or instance

Data in the Ranked view is sorted by default from highest to lowest percentage in the In(%) column. In the Design Unit, Call Tree, and Structural views, data is sorted (by default) according to the Under(%) column. You can click the heading of any column to sort data by that column.

The "Tcl_*" entries are functions that are part of the internal simulation code. They are not directly related to your HDL code.

- b. Click the Design Unit tab to view the profile data organized by design unit.

Figure 11-3. Design Unit Performance Profile

Name	Count	Under(raw)	In(raw)	Under(%)	In(%)	%Parent
sm_seq	1	1	1	0.7%	0.7%	...
sm	1	17	17	12.6%	12.6%	...
sm.v:73		17	5	12.6%	3.7%	100%
Tcl_DoOneEvent		11	0	8.1%	0.0%	65%
Tcl_WaitForEvent		11	11	8.1%	8.1%	100%
beh_sram	1	1	1	0.7%	0.7%	...
test_sm	1	90	90	66.7%	66.7%	...
test_sm.v:105		85	17	63.0%	12.6%	94%
Tcl_DoOneEvent		68	0	50.4%	0.0%	80%
test_sm.v:92		3	3	2.2%	2.2%	3%

- c. Click the **Call Tree** tab to view the profile data in a hierarchical, function-call tree display.

The results differ between the Verilog and VHDL versions of the design. In Verilog, line 105 (*test_sm.v:105*) is taking the majority of simulation time. In VHDL, *test_sm.vhd:203* and *sm.vhd:93* are taking the majority of the time.

Note

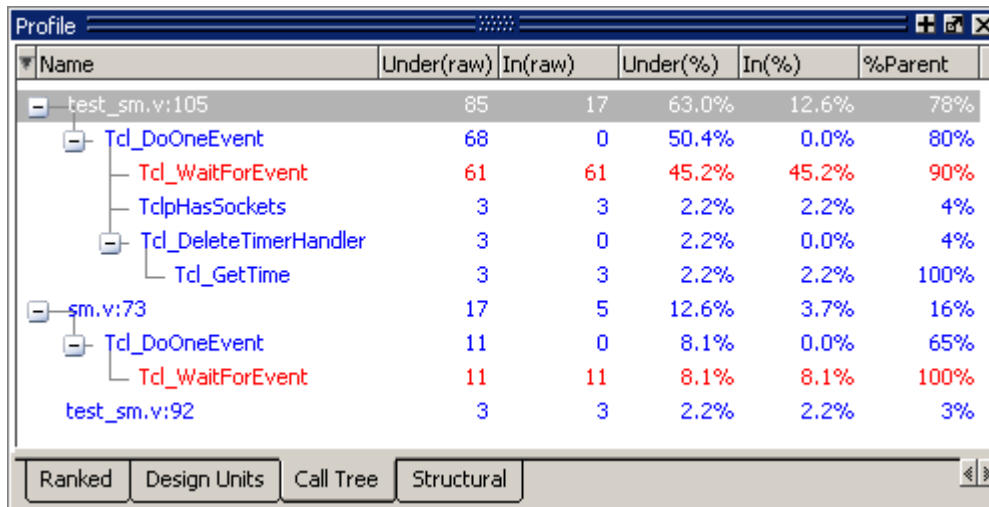


Your results may look slightly different as a result of the computer you're using and different system calls that occur during the simulation. Also, the line number reported may be one or two lines off in the actual source file. This happens due to how the stacktrace is decoded on different platforms.

- d. **Verilog:** Right-click *test_sm.v:105* and select **Expand All** from popup menu. This expands the hierarchy of *test_sm.v:105* and displays the functions that call it (Figure 11-4).

VHDL: Right-click *test_sm.vhd:203* and select **Expand All** from popup menu. This expands the hierarchy of *test_sm.vhd:203* and displays the functions that call it.

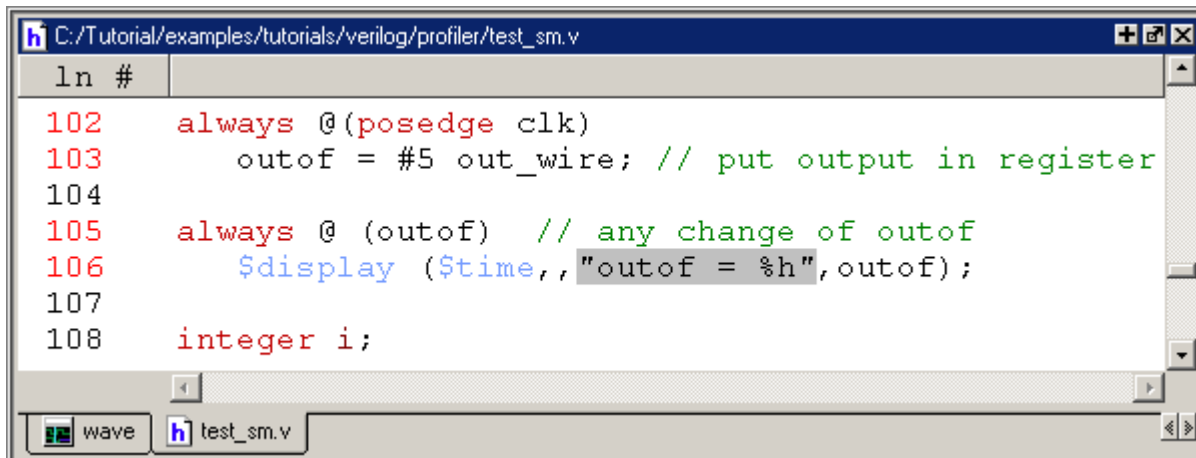
Figure 11-4. Expand the Hierarchical Function Call Tree



Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent
test_sm.v:105	85	17	63.0%	12.6%	78%
Tcd_DoOneEvent	68	0	50.4%	0.0%	80%
Tcd_WaitForEvent	61	61	45.2%	45.2%	90%
TcdHasSockets	3	3	2.2%	2.2%	4%
Tcd_DeleteTimerHandler	3	0	2.2%	0.0%	4%
Tcd_GetTime	3	3	2.2%	2.2%	100%
sm.v:73	17	5	12.6%	3.7%	16%
Tcd_DoOneEvent	11	0	8.1%	0.0%	65%
Tcd_WaitForEvent	11	11	8.1%	8.1%	100%
test_sm.v:92	3	3	2.2%	2.2%	3%

4. View the source code of a line that is using a lot of simulation time.
 - a. **Verilog:** Double-click *test_sm.v:105*. The Source window opens in the MDI frame with line 105 displayed (Figure 11-5).
- VHDL:** Double-click *test_sm.vhd:203*. The Source window opens in the MDI frame with line 203 displayed.

Figure 11-5. The Source Window Showing a Line from the Profile Data



```

C:/Tutorial/examples/tutorials/verilog/profiler/test_sm.v
ln #
102  always @(posedge clk)
103      outof = #5 out_wire; // put output in register
104
105  always @ (outof) // any change of outof
106      $display ($time, "outof = %h", outof);
107
108  integer i;
  
```

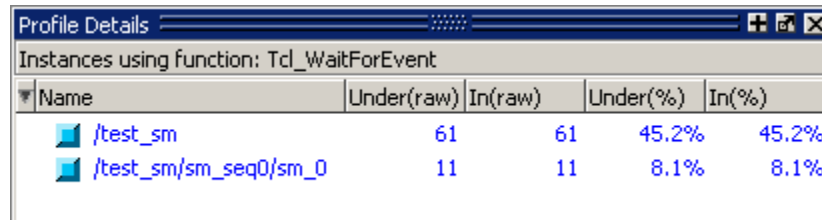
View Profile Details

The Profile Details pane increases visibility into simulation performance. Right-clicking any function in the Ranked or Call Tree views in the Profile pane opens a popup menu that includes a **Function Usage** selection. When you select **Function Usage**, the Profile Details pane opens and displays all instances that use the selected function.

1. View the Profile Details of a function in the Call Tree view.
 - a. Right-click the *Tcl_WaitForEvent* function and select **Function Usage** from the popup menu.

The Profile Details pane displays all instances using function *Tcl_WaitForEvent* (Figure 11-6). The statistical performance data show how much simulation time is used by *Tcl_Close* in each instance.

Figure 11-6. Profile Details of the Function *Tcl_Close*

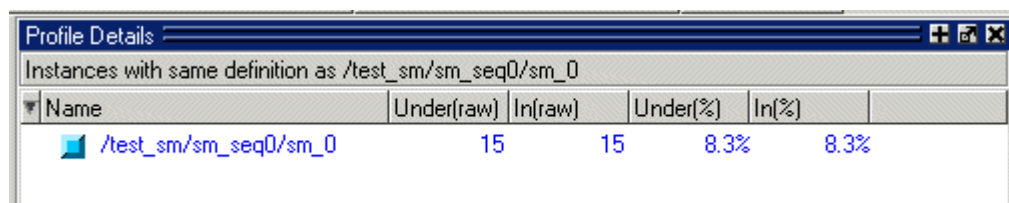


Profile Details				
Instances using function: Tcl_WaitForEvent				
Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm	61	61	45.2%	45.2%
/test_sm/sm_seq0/sm_0	11	11	8.1%	8.1%

When you right-click a selected function or instance in the Structural pane, the popup menu displays either a Function Usage selection or an Instance Usage selection, depending on the object selected.

1. View the Profile Details of an instance in the Structural view.
 - a. Select the **Structural** tab to change to the Structural view.
 - b. Right-click *test_sm* and select **Expand All** from the popup menu.
 - c. **Verilog:** Right-click the *sm_0* instance and select **Instance Usage** from the popup menu. The Profile Details shows all instances with the same definition as */test_sm/sm_seq0/sm_0* (Figure 11-7).

Figure 11-7. Profile Details of Function *sm_0*



Profile Details				
Instances with same definition as /test_sm/sm_seq0/sm_0				
Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm/sm_seq0/sm_0	15	15	8.3%	8.3%

VHDL: Right-click the *dut* instance and select **Instance Usage** from the popup menu. The Profile Details shows all instances with the same definition as */test_sm/dut*.

Filtering and Saving the Data

As a last step, you will filter out lines that take less than 3% of the simulation time using the Profiler toolbar, and then save the report data to a text file.

1. Filter lines that take less than 3% of the simulation time.
 - a. Click the Call Tree tab of the Profile pane.
 - b. Change the **Under(%)** field to 3 (Figure 11-8).

Figure 11-8. The Profiler Toolbar

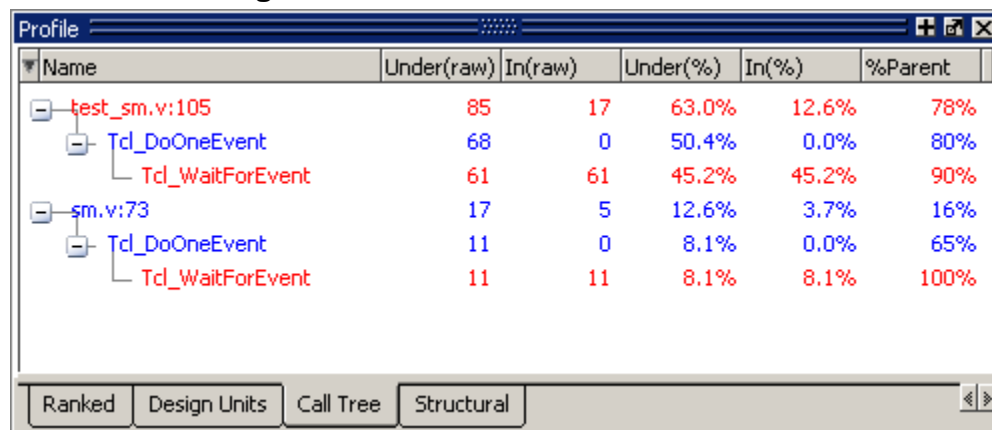


If you do not see these toolbar buttons, right-click in a blank area of the toolbar and select Profile.

- c. Click the **Refresh Profile Data** button. 

QuestaSim filters the list to show only those lines that take 3% or more of the simulation time (Figure 11-9).

Figure 11-9. The Filtered Profile Data



Name	Under(row)	In(row)	Under(%)	In(%)	%Parent
test_sm.v:105	85	17	63.0%	12.6%	78%
Tcl_DoOneEvent	68	0	50.4%	0.0%	80%
Tcl_WaitForEvent	61	61	45.2%	45.2%	90%
sm.v:73	17	5	12.6%	3.7%	16%
Tcl_DoOneEvent	11	0	8.1%	0.0%	65%
Tcl_WaitForEvent	11	11	8.1%	8.1%	100%


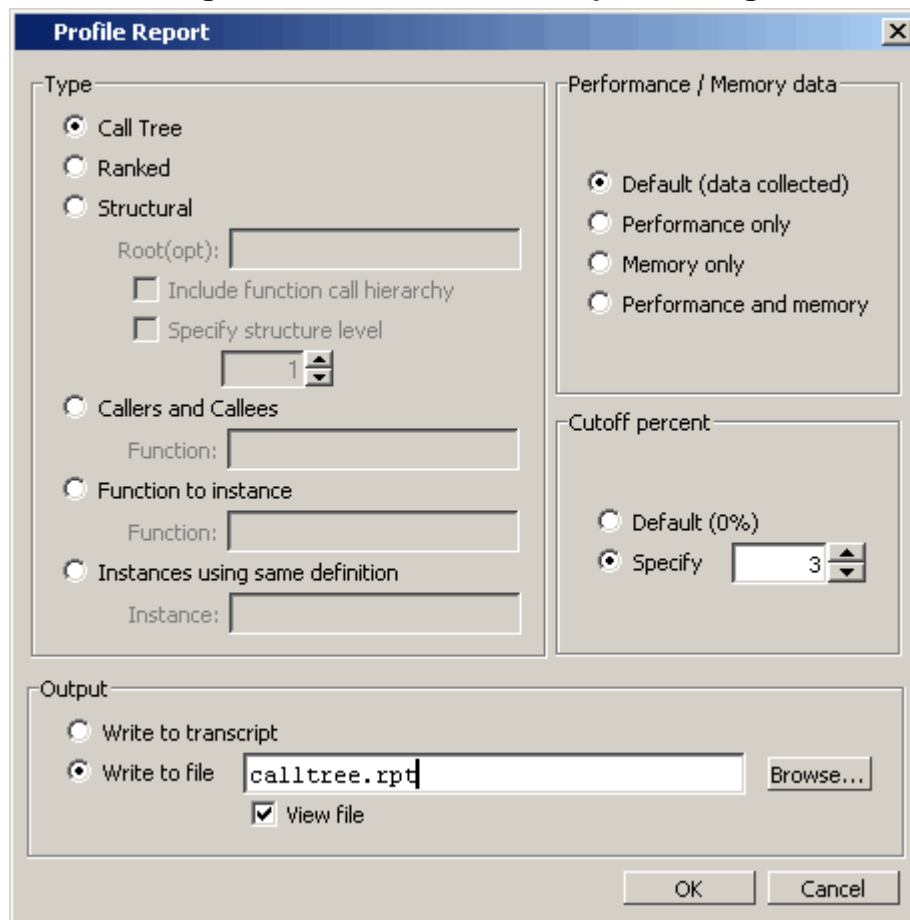
2. Save the report.
 - a. Click the save icon in the Profiler toolbar. 
 - b. In the Profile Report dialog (Figure 11-10), select the **Call Tree** Type.

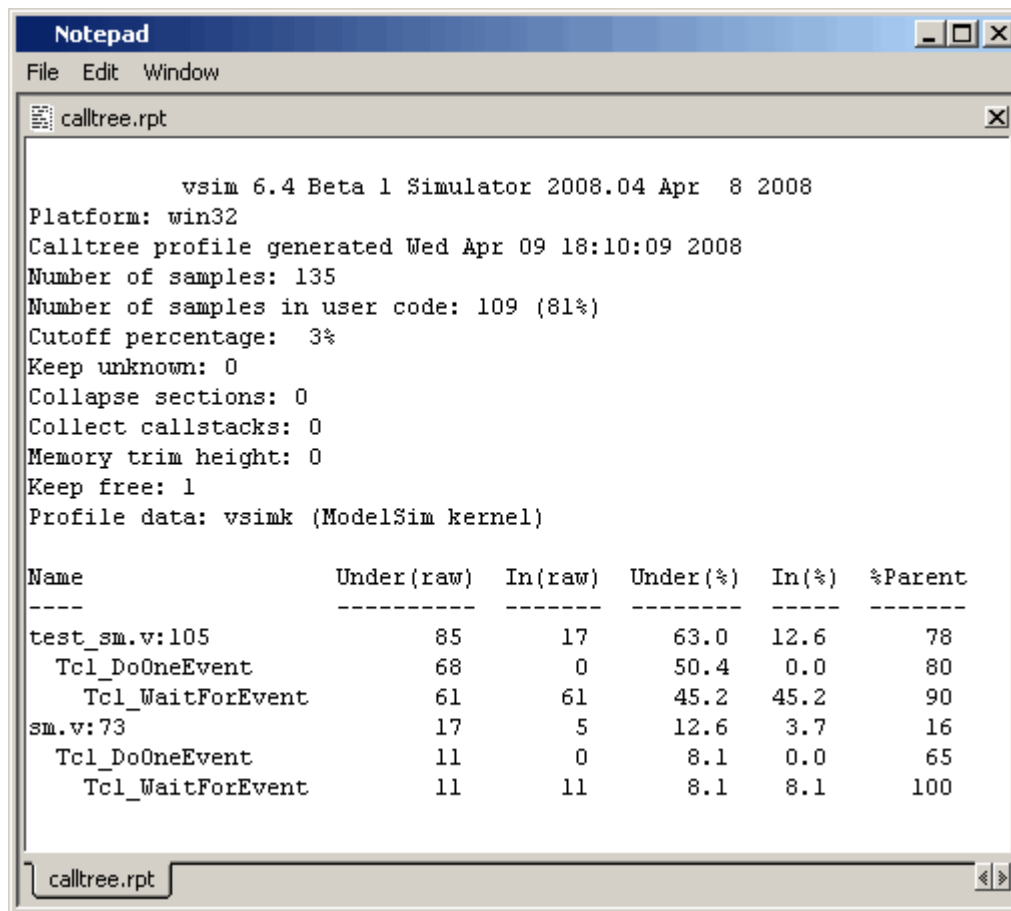
Figure 11-10. The Profile Report Dialog



- c. In the Performance/Memory data section select **Default (data collected)**.
- d. Specify the Cutoff percent as 3%.
- e. Select **Write to file** and type **calltree.rpt** in the file **name** field.
- f. **View file** is selected by default when you select **Write to file**. Leave it selected.
- g. Click **OK**.

The *calltree.rpt* report file will open automatically in Notepad ([Figure 11-11](#)).

Figure 11-11. The *calltree.rpt* Report



```
vsim 6.4 Beta 1 Simulator 2008.04 Apr 8 2008
Platform: win32
Calltree profile generated Wed Apr 09 18:10:09 2008
Number of samples: 135
Number of samples in user code: 109 (81%)
Cutoff percentage: 3%
Keep unknown: 0
Collapse sections: 0
Collect callstacks: 0
Memory trim height: 0
Keep free: 1
Profile data: vsimk (ModelSim kernel)
```

Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent
test_sm.v:105	85	17	63.0	12.6	78
Tcl_DoOneEvent	68	0	50.4	0.0	80
Tcl_WaitForEvent	61	61	45.2	45.2	90
sm.v:73	17	5	12.6	3.7	16
Tcl_DoOneEvent	11	0	8.1	0.0	65
Tcl_WaitForEvent	11	11	8.1	8.1	100

You can also output this report from the command line using the **profile report** command. See the *QuestaSim Command Reference* for details.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

Select **Simulate > End Simulation**. Click Yes.

Chapter 12

Simulating With Code Coverage

Introduction

QuestaSim Code Coverage gives you graphical and report file feedback on which executable statements, branches, conditions, and expressions in your source code have been executed. It also measures bits of logic that have been toggled during execution.

Note



The functionality described in this lesson requires a coverage license feature in your QuestaSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Design Files for this Lesson

The sample design for this lesson consists of a finite state machine which controls a behavioral memory. The testbench *test_sm* provides stimulus.

The QuestaSim installation comes with Verilog and VHDL versions of this design. The files are located in the following directories:

Verilog – `<install_dir>/examples/tutorials/verilog/coverage`

VHDL – `<install_dir>/examples/tutorials/vhdl/coverage`

This lesson uses the Verilog version in the examples. If you have a VHDL license, use the VHDL version instead. When necessary, we distinguish between the Verilog and VHDL versions of the design.

Related Reading

User's Manual Chapter: [Code Coverage](#).

Compile the Design

Enabling Code Coverage is a two step process. First, you identify which coverage statistics you want and compile the design files. Second, you load the design and tell QuestaSim to produce those statistics.

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory and copy all files from `<install_dir>/questasim/examples/tutorials/verilog/coverage` to the new directory.

If you have a VHDL license, copy the files in `<install_dir>/questasim/examples/tutorials/vhdl/coverage` instead.

2. Start QuestaSim and change to the exercise directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Create the working library.

- a. Type **vlib work** at the QuestaSim> prompt.

4. Compile the design files.

- a. For Verilog – Type **vlog -cover bcsxf sm.v sm_seq.v beh_sram.v test_sm.v** at the QuestaSim> prompt.

For VHDL – Type **vcom -cover bcsxf sm.vhd sm_seq.vhd sm_sram.vhd test_sm.vhd** at the QuestaSim> prompt.

The **-cover bcsxf** argument instructs QuestaSim to collect branch, condition, statement, extended toggle, and finite state machine coverage statistics. Refer to the section [Enabling Code Coverage](#) in the User's Manual for more information on the available coverage types.

Load and Run the Design

1. Load the design.

- a. Enter **vsim -voptargs="+acc" -coverage test_sm** at the QuestaSim> prompt of the Transcript pane.

The **-voptargs="+acc"** argument for the **vsim** command provides visibility into the design for debugging purposes.

Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

2. Run the simulation

- a. Type **run 1 ms** at the VSIM> prompt.

When you load a design with Code Coverage enabled, QuestaSim adds several columns to the Files and sim tabs in the Workspace (Figure 12-1). Your results may not match those shown in the figure.

Figure 12-1. Code Coverage Columns in the Main Window Workspace

Name	Specified path	Full path	Type	Stmt Count	Stmt Hits	Stmt %	Stmt Graph
sim	vsim.wlf	C:/Tutorial...					
VL sm.v	sm.v	C:/Tutorial... verilog		22	19	86.364	
VL sm_seq.v	sm_seq.v	C:/Tutorial... verilog		16	15	93.750	
VL beh_sram.v	beh_sram.v	C:/Tutorial... verilog		6	5	83.333	
VL test_sm.v	test_sm.v	C:/Tutorial... verilog		77	70	90.909	

By default, QuestaSim also displays three Code Coverage panes in the Main window:

- **Missed Coverage**

Select **View > Coverage > Missed Coverage** to open or close this pane. Displays the selected file's un-executed statements, branches, conditions, expressions and signals that have not toggled (Figure 12-2). It also includes missed states and transitions in finite state machines.








Figure 12-2. Missed Coverage Pane

Missed Statements			
VL test_sm.v			
X	31	#5	
X	31	into = {4'b0001, 28'b0};	
X	32	@ (posedge clk)	
X	33	#5	
X	33	into = data;	
X	134	#100	
X	134	\$stop;	

- **Instance Coverage**

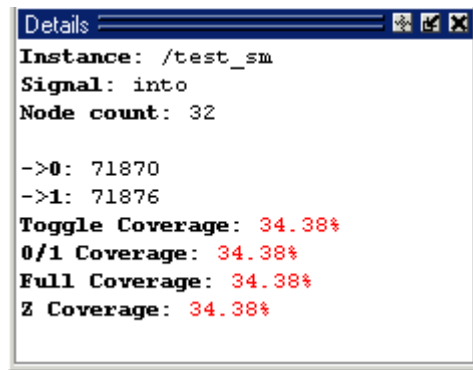
Select **View > Coverage > Instance Coverage** to open or close this pane. Displays statement, branch, condition, expression and toggle coverage statistics for each instance in a flat, non-hierarchical view (Figure 12-3).

Figure 12-3. Instance Coverage Pane

Stmt %	Stmt graph	Branch count	Branch hits	Branch misses	Branch %	Branch graph	Con
1	90% 	8	7	1	87.5%		
3	90% 	20	17	3	85%		
1	95.5% 	14	13	1	92.9%		
13	84.3% 						

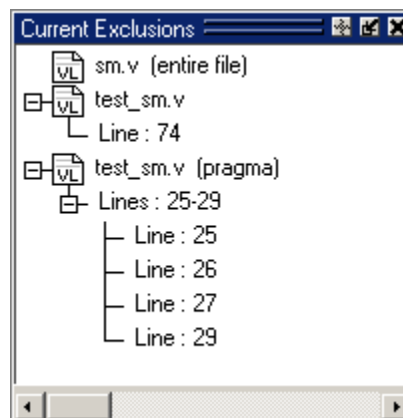
- Details**

Select **View > Coverage > Details** to open or close this pane. Shows coverage details for the item selected in the Missed Coverage pane. Details can include truth tables for conditions and expressions, or toggle details ([Figure 12-4](#)).

Figure 12-4. Details Pane

- Current Exclusions**

Select **View > Coverage > Current Exclusions** to open or close this pane. Lists all files and lines that are excluded from coverage statistics ([Figure 12-5](#)). See [Excluding Lines and Files from Coverage Statistics](#) for more information.

Figure 12-5. Current Exclusions Pane

All coverage panes can be re-sized, rearranged, and undocked to make the data more easily viewable. To resize a pane, click-and-drag on the top or bottom border. To move a pane, click-and-drag on the double-line to the right of the pane name. To undock a pane you can select it then drag it out of the Main window, or you can click the Dock/Undock Pane button in the header bar (top right). To redock the pane, click the Dock/Undock Pane button again.

We will look at these panes more closely in the next exercise. For complete details on each pane, Refer to the section [Code Coverage Panes](#) in the User's Manual.

Coverage Statistics in the Main window

Let's take a look at the data in these various panes.

1. View statistics in the Workspace pane.
 - a. Select the **sim** tab in the Workspace and scroll to the right.
Coverage statistics are shown for each object in the design.
 - b. Select the **Files** tab in the Workspace and scroll to the right.
Each file in the design shows summary statistics for statements, branches, conditions, expressions, and states.
 - c. Click the right-mouse button on any column name and select an object from the list ([Figure 12-6](#)).

Figure 12-6. Right-click a Column Heading to Show Column List

% Goal instance	% Goal total	✓ Branch %
✓ Branch count	✓ Branch graph	✓ Branch hits
✓ Branch misses	✓ Condition %	✓ Condition graph
✓ Condition hits	✓ Condition misses	✓ Condition rows
✓ Design unit	✓ Design unit type	✓ Expression %
✓ Expression graph	✓ Expression hits	✓ Expression misses
✓ Expression rows	✓ FEC Condition %	✓ FEC Condition graph
✓ FEC Condition hits	✓ FEC Condition misses	✓ FEC Condition rows
✓ FEC Expression %	✓ FEC Expression graph	✓ FEC Expression hits
✓ FEC Expression misses	✓ FEC Expression rows	Instance coverage
✓ State %	✓ State graph	✓ State hits
✓ State misses	✓ States	✓ Stmt %
✓ Stmt count	✓ Stmt graph	✓ Stmt hits
✓ Stmt misses	✓ Toggle %	✓ Toggle hits
✓ Toggle misses	✓ Toggle nodes	✓ Toggled graph
Total coverage	✓ Transition %	✓ Transition graph
✓ Transition hits	✓ Transition misses	✓ Transitions
✓ Visibility		

All checked columns are displayed. Unchecked columns are hidden. The status of every column, whether displayed or hidden, is persistent between invocations of QuestaSim.

2. View statistics in the Missed Coverage pane (see [Figure 12-2](#) above).
 - a. Select different files from the Files tab of the Workspace. The Missed Coverage pane updates to show statistics for the selected file.
 - b. Select any entry in the Statement tab to display that line in the Source window.
3. View statistics in the Details pane.
 - a. Select the Toggle tab in the Missed Coverage pane.

If the Toggle tab isn't visible, you can do one of two things: 1) widen the pane by clicking-and-dragging on the pane border; 2) if your mouse has a middle button, click-and-drag the tabs with the middle mouse button.
 - b. Select any object in the Toggle tab to see details in the Details pane (see [Figure 12-4](#) above).
4. View instance coverage statistics.

The Instance Coverage pane displays coverage statistics for each instance in a flat, non-hierarchical view (see [Figure 12-3](#) above). Select any instance in the Instance Coverage pane to see its source code displayed in the Source window.

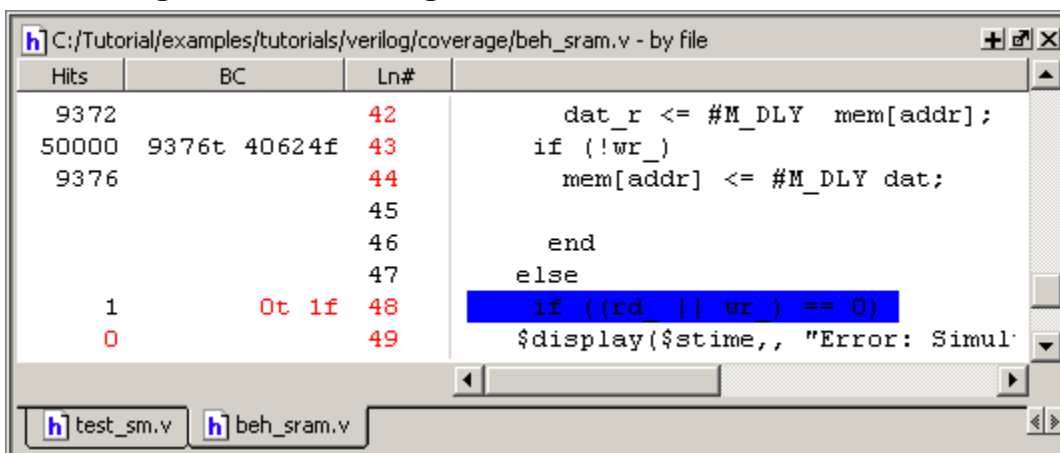
Coverage Statistics in the Source Window

In the previous section you saw that the Source window and the Main window coverage panes are linked. You can select objects in the Main window panes to view the underlying source code in the Source window. Furthermore, the Source window contains statistics of its own.

1. View coverage statistics for *beh_sram* in the Source window.
 - a. Select *beh_sram.v* in the **Files** tab of the Workspace.

In the Statement tab of the Missed Coverage pane, expand *beh_sram.v* if necessary and select line 48.
 - b. The Source window opens in the MDI frame with line 48 highlighted momentarily ([Figure 12-7](#)). Highlighting will disappear after a few seconds.

Figure 12-7. Coverage Statistics in the Source Window



- c. Switch to the Source window.

The table below describes the various icons.

Table 12-1. Coverage Icons in the Source Window

Icon	Description
green checkmark	indicates a statement that has been executed
red X	indicates that a statement in that line has not been executed (zero hits)
green E	indicates a line that has been excluded from code coverage statistics
red X _T or X _F	indicates that a true or false branch (respectively) of a conditional statement has not been executed

- d. Select **Tools > Code Coverage > Show coverage numbers**.

The icons are replaced by execution counts on every line. An ellipsis (...) is displayed whenever there are multiple statements on the line. Hover the mouse pointer over a statement to see the count for that statement.

Figure 12-8. Coverage Numbers Shown by Hovering the Mouse Pointer

Hits	BC	Ln#	Code
9372		42	dat_r <= #M_DLY mem[addr];
50000	9376t 40624f	43	if (!wr_)
9376		44	mem[addr] <= #M_DLY dat;
		45	
		46	end
		47	else
1	0t 1f	48	if ((rd_ wr_) == 0)
0		49	\$display(\$stime,, "Error: Simul'");

- e. Select **Tools > Code Coverage > Show coverage numbers** again to uncheck the selection and return to icon display.

Toggle Statistics in the Objects Pane

Toggle coverage counts each time a logic node transitions from one state to another. Earlier in the lesson you enabled six-state toggle coverage by using the **-cover x** argument with the **vlog** or **vcom** command. Refer to the section [Toggle Coverage](#) in the User's Manual for more information.

1. View toggle data in the Objects pane of the Main window.
 - a. Select *test_sm* in the **sim** tab of the Workspace.
 - b. If the Objects pane isn't open already, select **View > Objects**. Scroll to the right to see the various toggle coverage columns, or undock and expand the pane until all columns show ([Figure 12-9](#)).

Figure 12-9. Toggle Coverage in the Objects Pane

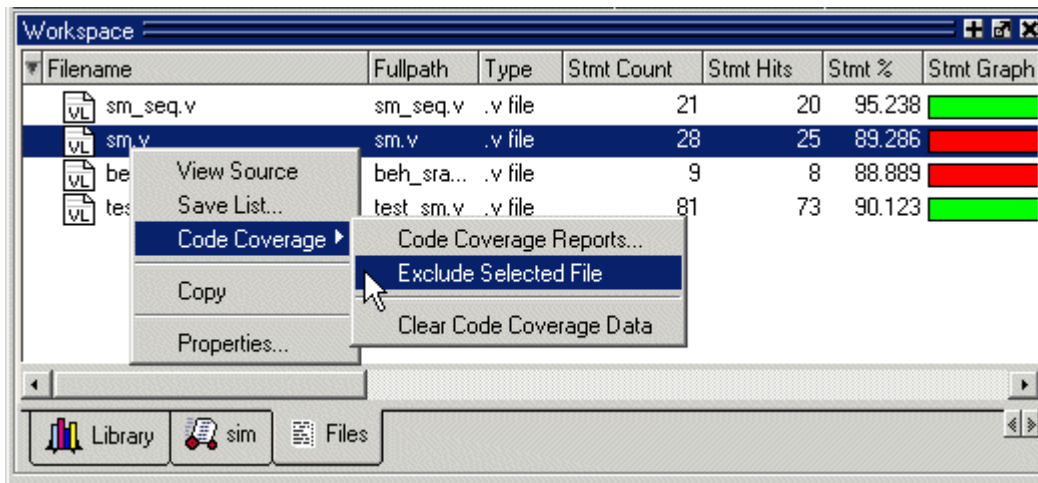
Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01
into	0000...	Packed Array	Internal	71870	71876	0	0	0	0	32	11	34.38%	34.3
outof	0000...	Packed Array	Internal	12493	12499	0	0	0	0	32	6	18.75%	21.8
rst	0	Packed Array	Internal	2	1	0	0	0	0	1	1	100%	10
clk	0	Packed Array	Internal	50000	50000	0	0	0	0	1	1	100%	10
out_wire	0000...	Net	Internal	12493	12499	0	0	0	0	32	6	18.75%	21.8
dat	0000...	Net	Internal	14055	17186	378058	381216	71862	68736	32	6	18.75%	21.8
addr	0000...	Net	Internal	15621	15624	0	0	0	0	10	4	40%	4
loop	xxxxxx...	Packed Array	Internal	0	0	0	0	0	0	32	0	0%	
i	x	Packed Array	Internal	0	0	0	0	0	0	32	0	0%	
rd_	St1	Net	Internal	9372	9372	0	0	0	0	1	1	100%	10
wr_	St0	Net	Internal	4689	4688	0	0	0	0	1	1	100%	10

Excluding Lines and Files from Coverage Statistics

Questasim allows you to exclude lines and files from code coverage statistics. You can set exclusions with the GUI, with a text file called an "exclusion filter file", or with "pragmas" in your source code. Pragmas are statements that instruct Questasim to not collect statistics for the bracketed code. Refer to the section [Excluding Objects from Coverage](#) in the User's Manual for more details on exclusion filter files and pragmas.

1. Display the Current Exclusions pane if necessary.
 - a. Select **View > Coverage > Current Exclusions**.
2. Exclude a line via the Missed Coverage pane.
 - a. Right click a line in the Missed Coverage pane and select **Exclude Selection**. (You can also exclude the selection for the current instance only by selecting **Exclude Selection For Instance <inst_name>**.) The line will appear in the Current Exclusions pane.
3. Exclude an entire file.
 - a. In the Files tab of the Workspace, locate *sm.v* (or *sm.vhd* if you are using the VHDL example).
 - b. Right-click the file name and select **Code Coverage > Exclude Selected File** ([Figure 12-10](#)).

Figure 12-10. Excluding a File Using Menus in the Workspace



The file is added to the Current Exclusions pane.

4. Cancel the exclusion of *sm.v*.
 - a. Right-click *sm.v* in the Current Exclusions pane and select **Cancel Selected Exclusions**.

Creating Code Coverage Reports

You can create textual or HTML reports on coverage statistics using menu selections in the GUI or by entering commands in the Transcript pane. You can also create a textual report of coverage exclusions using menu selections.

To create textual coverage reports via the GUI, do one of the following:

- Select **Tools > Coverage Report > Text** from the Main window menubar.
- Right-click any object in the **sim** or **Files** tab of the Workspace and select **Code Coverage > Code Coverage Reports** from the popup context menu.
- Right-click any object in the Instance Coverage pane and select **Code coverage reports** from the popup context menu. You may also select **Instance Coverage > Code coverage reports** from the Main window menu bar when the Instance Coverage pane is active.

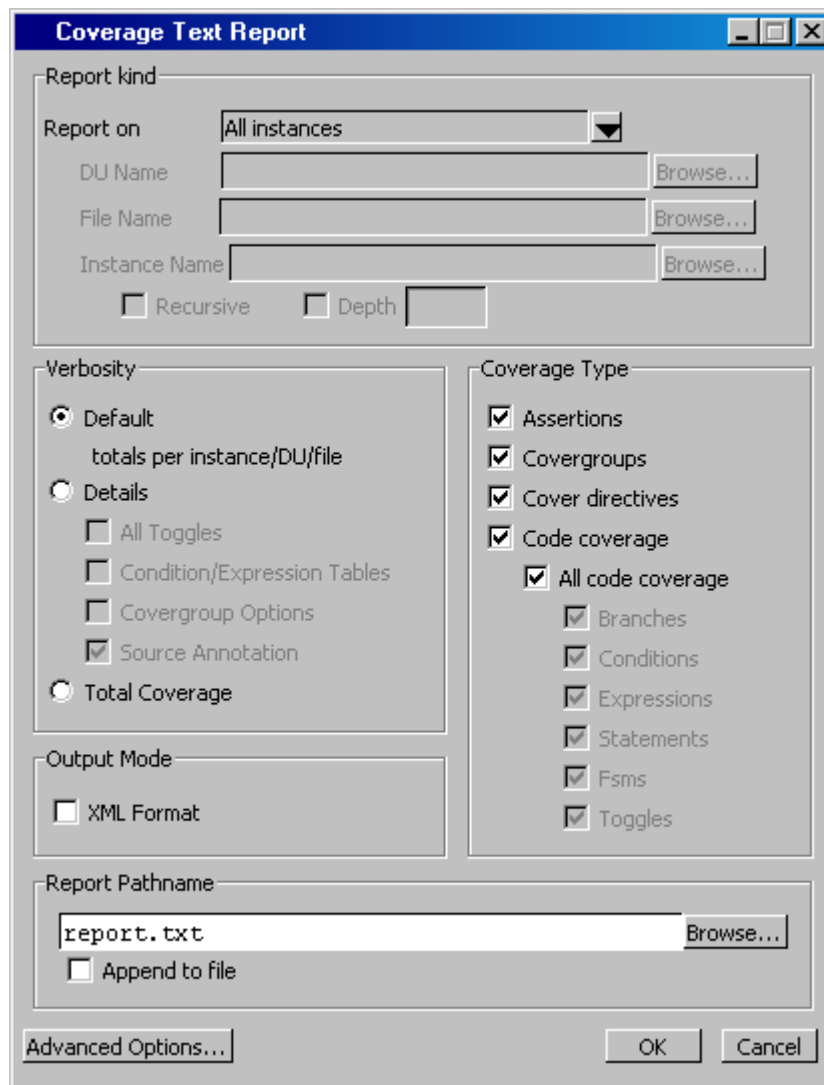
This will open the Coverage Text Report dialog (Figure 12-11) where you can elect to report on:

- all files,
- all instances,
- all design units,

- specified design unit(s),
- specified instance(s), or
- specified source file(s).

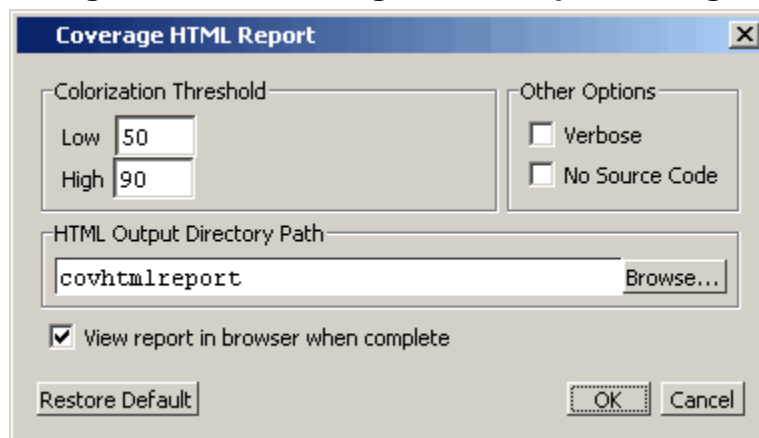
QuestaSim creates a file (named *report.txt* by default) in the current directory and immediately display the report in the Notepad text viewer/editor included with the product.

Figure 12-11. Coverage Text Report Dialog



To create a coverage report in HTML, select **Tools > Coverage Report > HTML** from the Main window menu bar. This opens the Coverage HTML Report dialog where you can designate an output directory path for the HTML report.

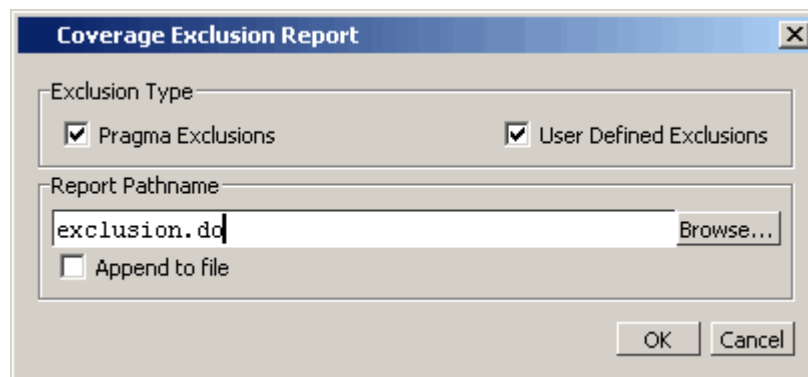
Figure 12-12. Coverage HTML Report Dialog



By default, the `coverage report` command will produce textual files unless the `-html` argument is used. You can display textual reports in the Notepad text viewer/editor included with the product by using the `notepad <filename>` command.

To create a coverage exclusions report, select **Tools > Coverage Report > Exclusions** from the Main window menubar. This opens the Coverage Exclusions Report dialog where you can elect to show only pragma exclusions, only user defined exclusions, or both.

Figure 12-13. Coverage Exclusions Report Dialog



Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. Type **quit -sim** at the `VSIM>` prompt.

Chapter 13

Debugging With PSL Assertions

Introduction

Using assertions in your HDL code increases visibility into your design and improves verification productivity. QuestaSim supports Property Specification Language (PSL) assertions for use in dynamic simulation verification. These assertions are simple statements of design intent that declare design or interface assumptions.

This lesson will familiarize you with the use of PSL assertions in QuestaSim. You will run a simulation with and without assertions enabled so you can see how much easier it is to debug with assertions. After running the simulation with assertions, you will use the QuestaSim debugging environment to locate a problem with the design.

Design Files for this Lesson

The sample design for this lesson uses a DRAM behavioral model and a self-checking testbench. The DRAM controller interfaces between the system processor and the DRAM and must be periodically refreshed in order to provide read, write, and refresh memory operations. Refresh operations have priority over other operations, but a refresh will not preempt an in-process operation.

The QuestaSim installation comes with Verilog and VHDL versions of this design. The files are located in the following directories:

Verilog – `<install_dir>/examples/psl/verilog/modeling/dram_controller`

VHDL – `<install_dir>/examples/psl/vhdl/modeling/dram_controller`

This lesson uses the Verilog version for the exercises. If you have a VHDL license, use the VHDL version instead.

You can embed assertions within your code or supply them in a separate file. This example design uses an external file.

Related Reading

User's Manual Chapter: [Verification with Assertions and Cover Directives](#).

Compile the Example Design

In this exercise you will use a DO file to compile the design.

1. Create a new directory and copy the lesson files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory and copy all files from `<install_dir>/examples/psl/verilog/modeling/dram_controller` to the new directory.

If you have a VHDL license, copy the files in `<install_dir>/examples/psl/vhdl/modeling/dram_controller` instead.

2. Start QuestaSim and change to the exercise directory you created.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. To start QuestaSim, type `vsim` at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Execute the lesson DO file.

- a. Type **do compile.do** at the command prompt.

The DO file does the following:

- Creates the working library
- Compiles the design files and assertions

Feel free to open the DO file and look at its contents.

Load and Run Without Assertions

1. Load the design without assertions.

- a. Type **vsim -voptargs="+acc" -nopsi tb** at the command prompt.

The **-voptargs="+acc"** argument for the `vsim` command provides visibility into the design for debugging purposes.

Note



By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

The **-nopsi** argument instructs the compiler to ignore PSL assertions.

2. Run the simulation.

- a. Type **run -all** at the command prompt or click the Run -All icon.

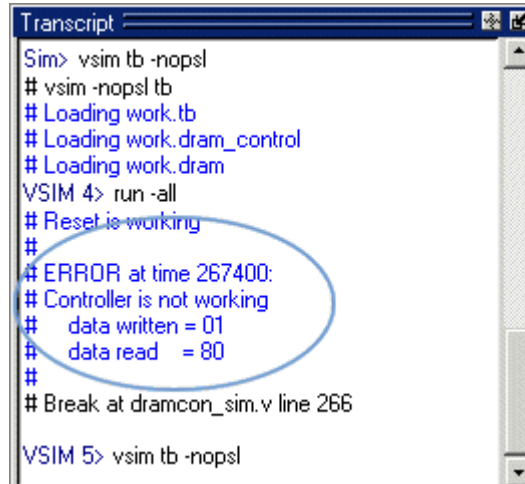


Verilog: The simulation reports an error at 267400 ns and stops on line 266 of the *dramcon_sim.v* module.

VHDL: The simulation reports an error at 246800 ns and stops on line 135 of the *dramcon_sim.vhd* entity.

The ERROR message indicates that the controller is not working because a value read from memory does not match the expected value (Figure 13-1).

Figure 13-1. Transcript After Running Simulation Without Assertions



```

Transcript
Sim> vsim tb -nopsi
# vsim -nopsi tb
# Loading work.tb
# Loading work.dram_control
# Loading work.dram
VSIM 4> run -all
# Reset is working
#
# ERROR at time 267400:
# Controller is not working
#   data written = 01
#   data read   = 80
#
# Break at dramcon_sim.v line 266
VSIM 5> vsim tb -nopsi
  
```

To debug the error, you might first examine the simulation waveforms and look for all writes to the memory location. You might also check the data on the bus and the actual memory contents at the location after each write. If that did not identify the problem, you might then check all refresh cycles to determine if a refresh corrupted the memory location.

Quite possibly, all of these debugging activities would be required, depending on one's skill (or luck) in determining the most likely cause of the error. Any way you look at it, it is a tedious exercise.

3. End the simulation.
 - a. Type **quit -sim** at the command prompt to end this simulation.

Using Assertions to Speed Debugging

To see how assertions can speed debugging, reload the design with assertion failure tracking enabled.

1. Reload the design.
 - a. Type **vsim -voptargs="+acc" -assertdebug tb** at the command prompt.

The **”+acc”** portion of the **-voptargs** argument preserves PSL assertion data, enabling pass count logging in the Transcript window and assertion viewing in the Wave window. If you do not specify **+acc**, the tool only transcribes assertion failure messages and reports only failure counts in the assertion browser.

+acc also enables the complete functionality of **vsim -assertdebug**. The **-assertdebug** option gives you another tool for debugging failed assertions, as we’ll see in a moment.

2. Execute the lesson DO file.
 - a. Type **do sim.do** at the command prompt.

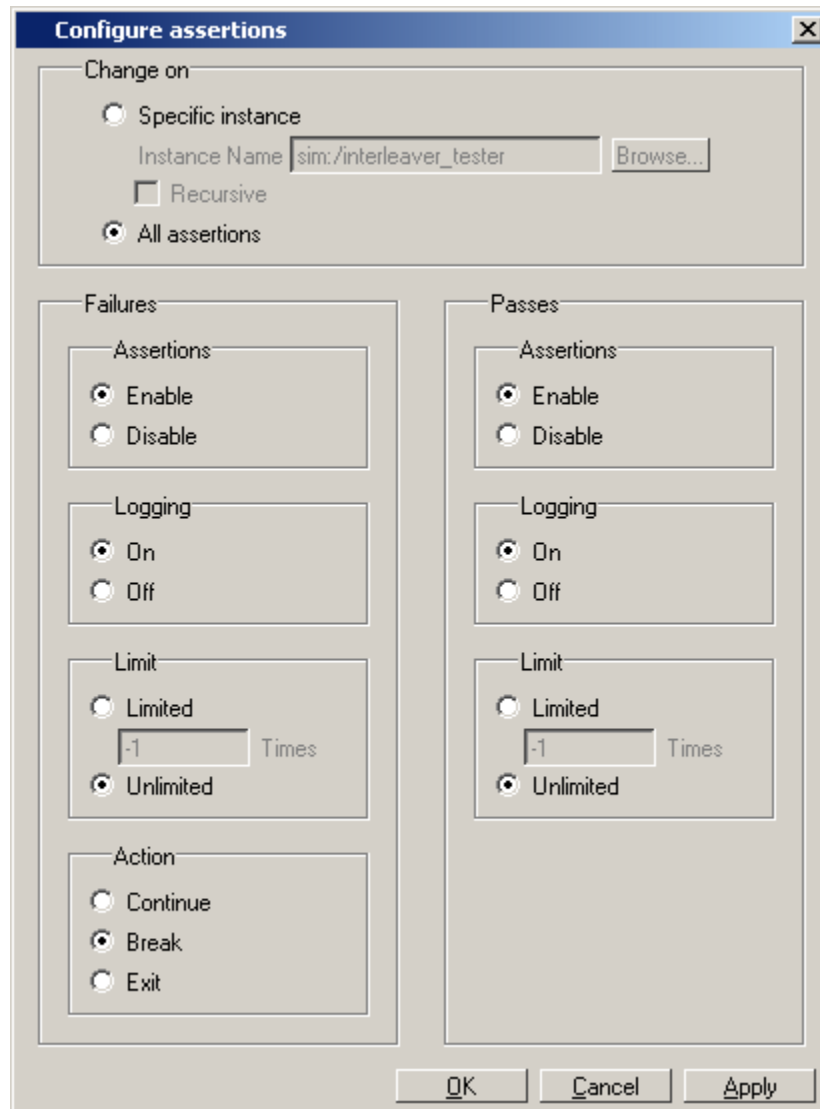
The DO file does the following:

- Opens the Assertions tab of the Analysis pane and displays all assertions
- Opens a Source window
- Adds signals to the Wave window

You may need to resize and move the panes to better view the data.

3. Set all assertions to Break on Failures.
 - a. Select the Analysis pane to make it active.
 - b. Select **Assertions > Configure** from the main menu to open the Configure assertions dialog ([Figure 13-2](#)).

Figure 13-2. Change Assertions Dialog



- c. In the Change on section, select **All assertions**.
- d. In the Failures Assertions section, select **Enable**.
- e. In the Failures Action section, select **Break**.
This causes the simulation to break (stop) on any failed assertion.
- f. In the Passes Logging section, select **On**.
- g. Click the OK button to accept your selections and close the dialog.

The command line equivalents for these actions are as follows:

```
assertion fail -action break -r *  
assertion pass -log on -r *
```

4. Add assertion signals to the Wave window
 - a. Select all assertions in the Assertions tab of the Analysis pane.
 - b. Right-click on the selected assertions to open a popup menu.
 - c. Select **Add Wave > Objects in Design**.

Scroll to the bottom of the Wave window and you will see the assertion signals (denoted by magenta triangles).

5. Run the simulation.
 - a. Type **run -all** at the command prompt.

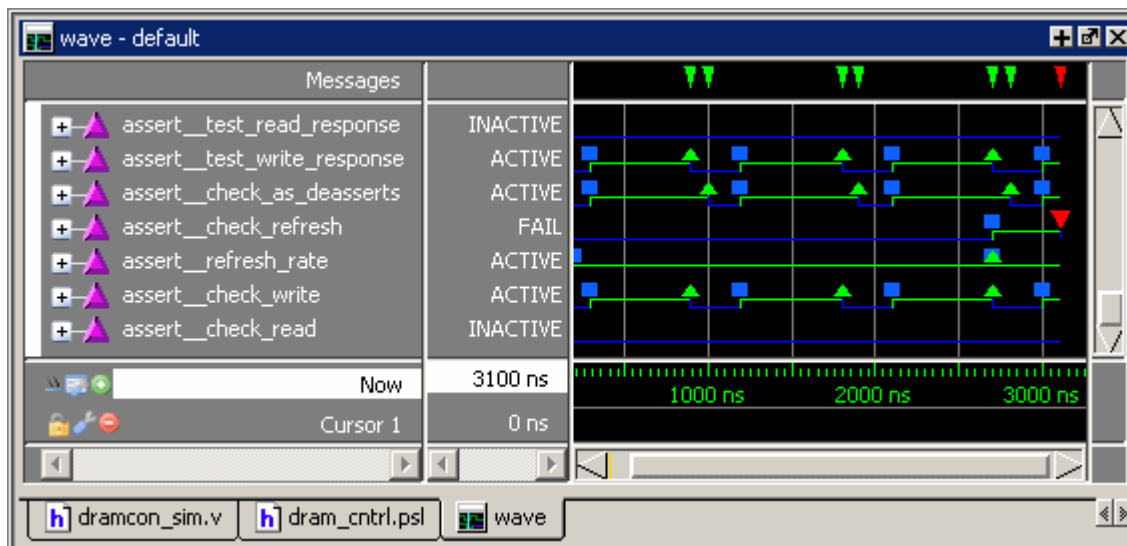
Verilog: The Main window transcript shows that the *assert_check_refresh* assertion in the *dram_cntrl.psl* file failed at 3100 ns. The simulation is stopped at that time. Note that with no assertions, the testbench did not report a failure until 267,400 ns, over 80x the simulation time required for a failure to be reported with assertions.

VHDL: The Main window transcript shows that the *assert_check_refresh* assertion in the *dram_cntrl.psl* file failed at 3800 ns. The simulation is stopped at that time. Note that with no assertions, the testbench did not report a failure until 246,800 ns, over 60x the simulation time required for a failure to be reported with assertions.

The blue arrow in the Source window shows where the simulation stopped - at the *check_refresh* assertion on line 24 of *dram_cntrl.psl*.

The Wave window displays a red triangle at the point of the simulation break and shows "FAIL" in the values column of the *assert_check_refresh* assert directive (Figure 13-3). Green triangles indicate assertion passes.

Figure 13-3. Assertion Failure Indicated in Wave Window



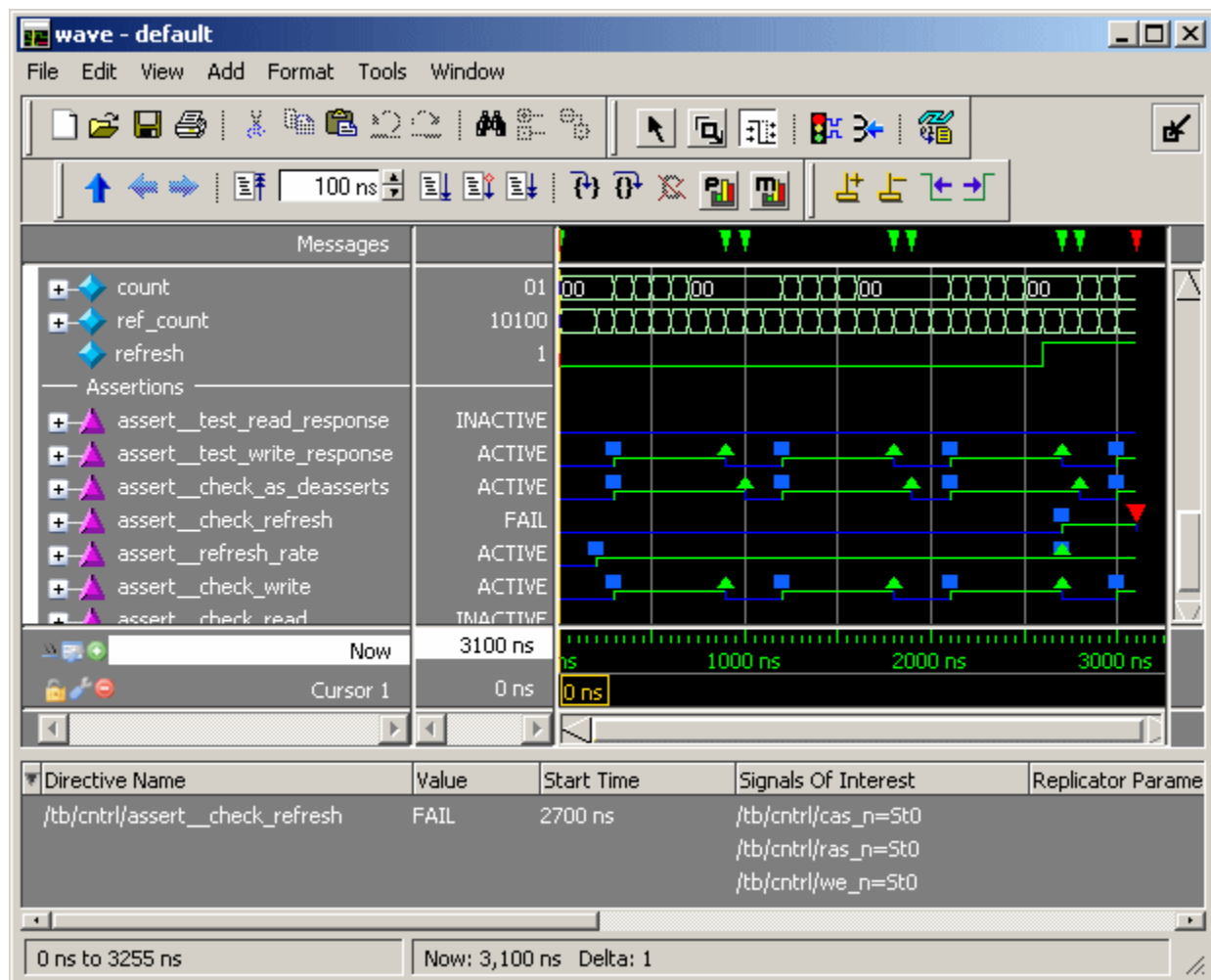
The blue sections of the assert directive waveforms indicate inactive directives; green indicates active directives.

6. View the assertion failure in the Assertion Debug pane of the Wave window.

Since you used the **-assertdebug** argument with the **vsim** command when you invoked the simulator, you can view the details of assertion failures in the Assertion Debug pane of the Wave window.

- a. Undock the Wave window.
- b. Select **View > Assertion Debug**. The Assertion Debug pane appears at the bottom of the Wave window, as shown in [Figure 13-4](#).
- c. Click the red triangle on the *assert_check_refresh* directive waveform (the red triangle indicates a failed assert directive).

Figure 13-4. The Assertion Debug Pane Shows Failed Assertion Details

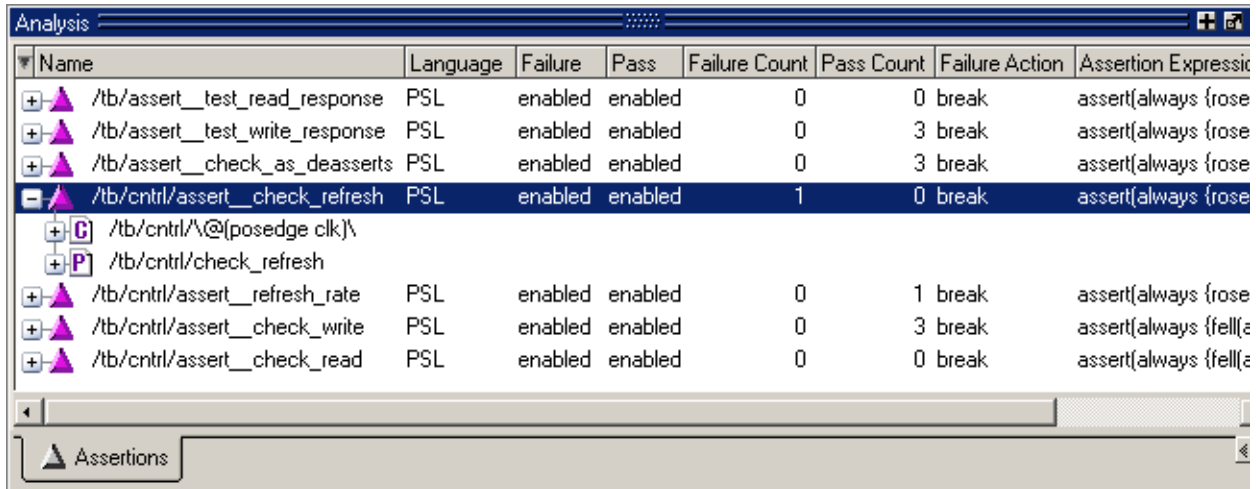


The Signals of Interest column displays the signals responsible for the assertion failure. You can analyze these signals further in the Dataflow window by right-clicking an assertion directive's waveform and selecting **Show Drivers** from the popup menu.

7. View assertion failure in the Assertions tab of the Analysis pane.

The Assertions tab indicates a failure of *assert_check_refresh* in the Failure Count column (Figure 13-5).

Figure 13-5. Assertion failure indicated in the Analysis pane



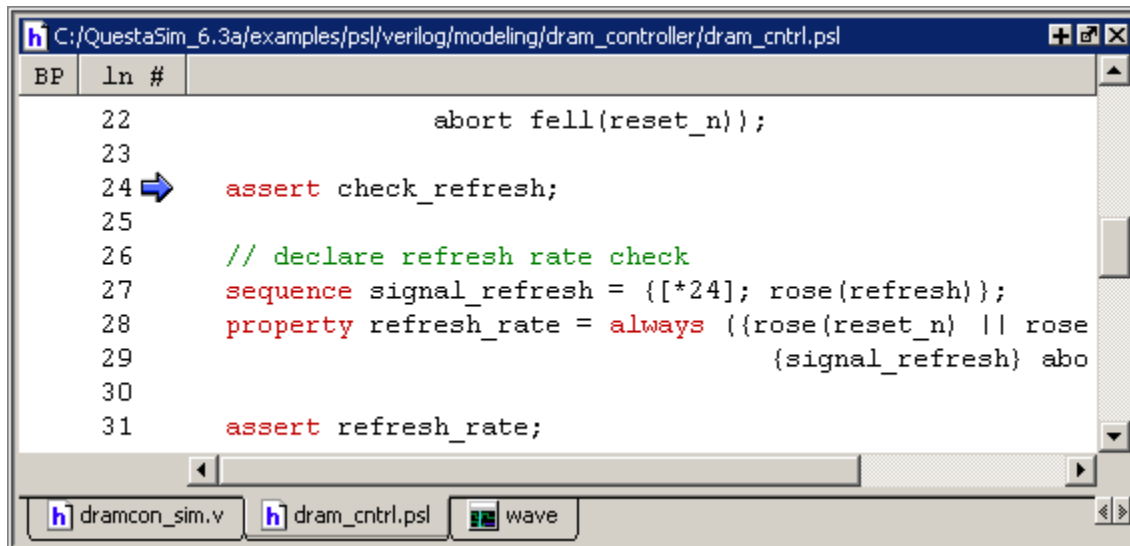
Name	Language	Failure	Pass	Failure Count	Pass Count	Failure Action	Assertion Expression
+ /tb/assert_test_read_response	PSL	enabled	enabled	0	0	break	assert(always {rose
+ /tb/assert_test_write_response	PSL	enabled	enabled	0	3	break	assert(always {rose
+ /tb/assert_check_as_deasserts	PSL	enabled	enabled	0	3	break	assert(always {rose
+ /tb/cntrl/assert_check_refresh	PSL	enabled	enabled	1	0	break	assert(always {rose
+ /tb/cntrl/\@(posedge clk)\							
+ /tb/cntrl/check_refresh							
+ /tb/cntrl/assert_refresh_rate	PSL	enabled	enabled	0	1	break	assert(always {rose
+ /tb/cntrl/assert_check_write	PSL	enabled	enabled	0	3	break	assert(always {fell
+ /tb/cntrl/assert_check_read	PSL	enabled	enabled	0	0	break	assert(always {fell

Debugging the Assertion Failure

1. View the source code of the failed assertion.

Verilog: The current line arrow points to the failed assertion on line 24 of the *dram_cntrl.psl* file (Figure 13-6). This assertion consists of checking the **check_refresh** property, which is defined on lines 20-22. The property states that when the refresh signal is active, then it will wait until the memory controller state goes to IDLE. The longest a read or write should take is 14 cycles. If the controller is already IDLE, then the wait is 0 cycles. Once the controller is in IDLE state, then the refresh sequence should start in the next cycle.

Figure 13-6. Source Code for Failed Assertion

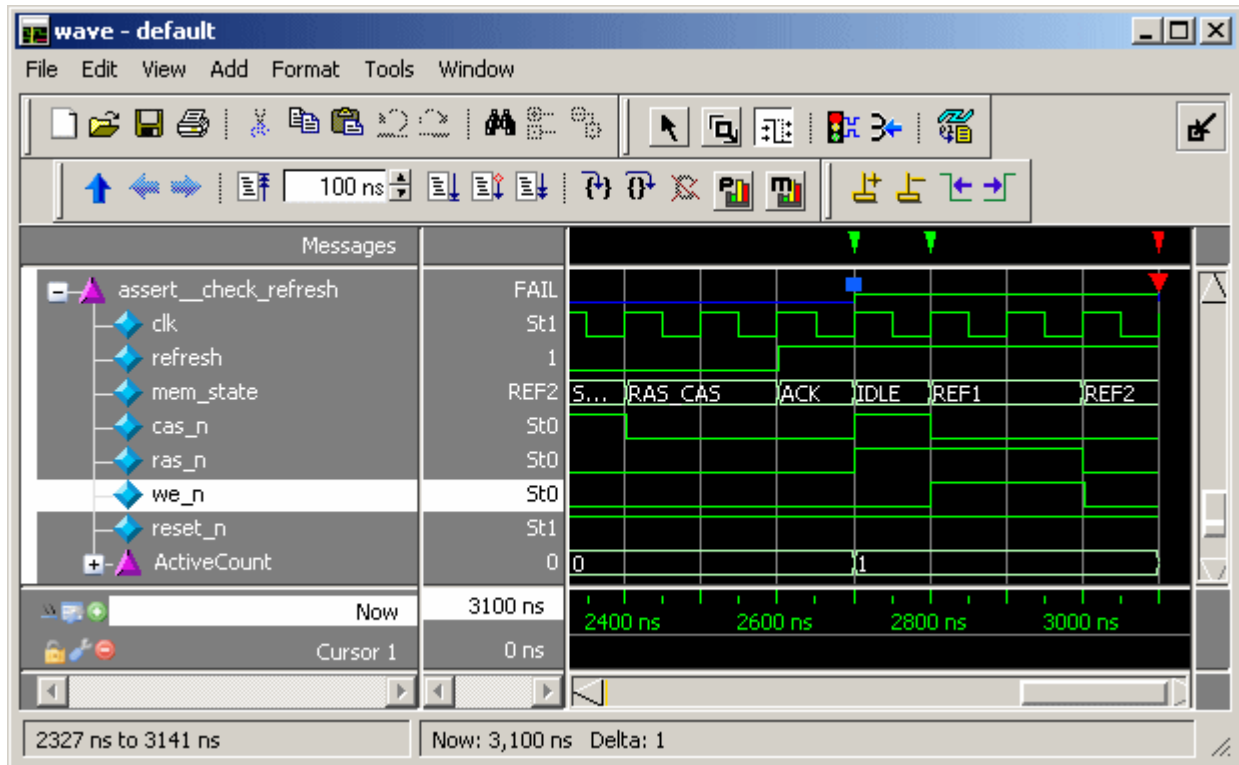


The *refresh_sequence* (second line of the property) is defined on line 18. The key part of the refresh protocol is that *we_n* must be held high (write enable not active) for the entire refresh cycle.

VHDL: The current line arrow points to the failed assertion on line 24 of the *dram_cntrl.psl* file. The *refresh_sequence* (second line of the property) is defined on line 20.

2. Check the Wave window to see if the write enable signal, *we_n*, was held high through both *REF1* and *REF2* states.
 - a. In the Wave window, expand *assert_check_refresh* to reveal all signals referenced by the assertion.
 - b. Zoom and scroll the Wave window so you can see *we_n* and *mem_state* (Figure 13-7).

Figure 13-7. Examining *we_n* With Respect to *mem_state*

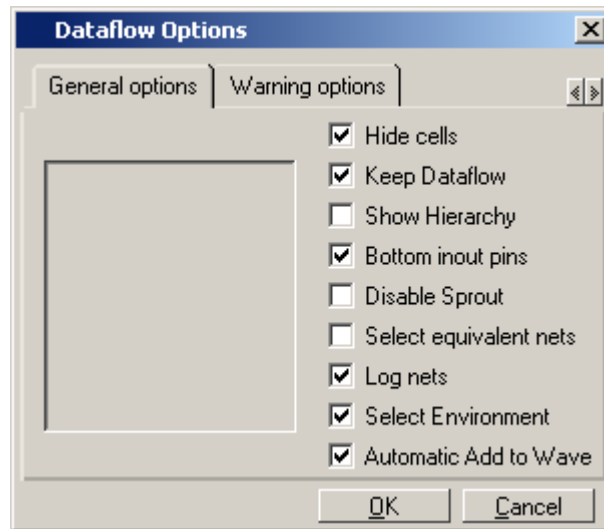


It is easy to see that *we_n* is high only during the *REF1* state. It is low during *REF2*.

Let's examine *we_n* further.

3. Examine *we_n* in the Dataflow and Source windows.
 - a. Open the Dataflow window by selecting **View > Dataflow** (Main window) then select the Dataflow window to make it active.
 - b. Select **Dataflow > Dataflow Preferences > Options** from the menus to open the Dataflow Options dialog. If the Dataflow window is undocked, select **Tools > Options** from the Dataflow window menus.
 - c. Uncheck the **Show Hierarchy** selection as shown in [Figure 13-8](#) and click **OK**.

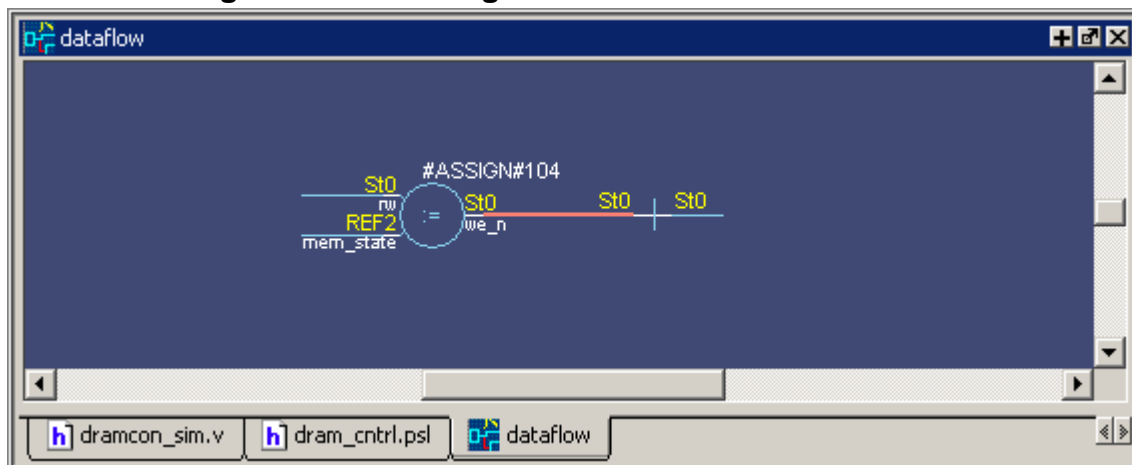
Figure 13-8. Dataflow Options Dialog



- d. Drag *we_n* from the Wave window to the Dataflow window.

Verilog: The Dataflow window shows that *we_n* is driven by the *#ASSIGN#104* process, with inputs *rw* and *mem_state* (Figure 13-9). The values shown in yellow are the values for each signal at the point at which the simulation stopped: 3100 ns. We see that *we_n* is St0 when *mem_state* is REF2. As noted above, *we_n* should be St1. This is the reason for the assertion failure.

Figure 13-9. Viewing *we_n* in the Dataflow Window

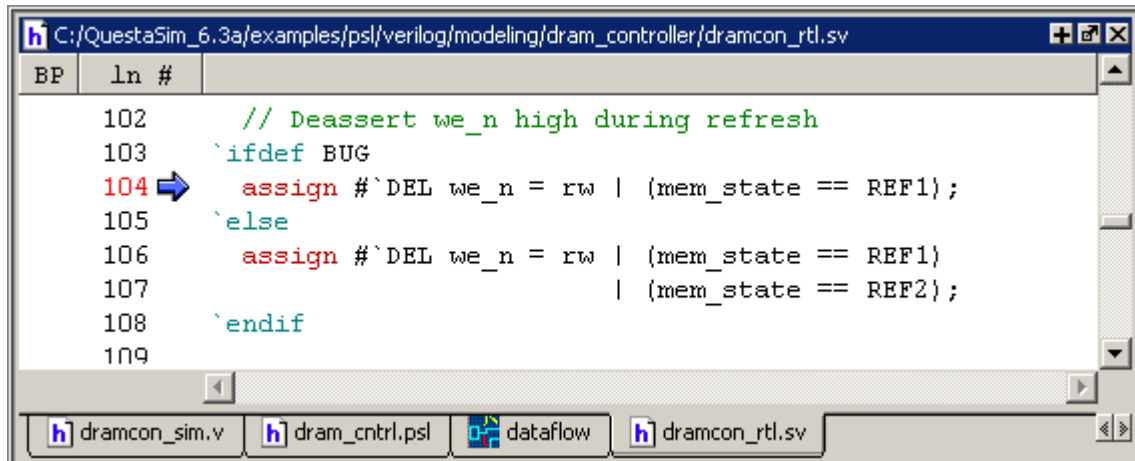


VHDL: The Dataflow window shows that *we_n* is driven by the process at line 61, which has inputs *rw* and *mem_state*. The values shown in yellow are the values for each signal at the point at which the simulation stopped: 3800 ns. We see that *we_n* is St0 when *mem_state* is REF2. As noted above, *we_n* should be St1. This is the reason for the assertion failure.

- e. Double-click the process that drives *we_n* in order to display its source code in the Source window.

Verilog: Looking at the Source window you will see that the current line arrow points to line 104 of the *dramcon_rtl.sv* file (Figure 13-10). In this line you can see that the logic assigning *we_n* is wrong - it does not account for the *REF2* state.

Figure 13-10. Finding the Bug in the Source Code



The code shows that the incorrect assignment is used for the example with the correct assignment immediately below (lines 106-107) that will hold *we_n* high through both states of the refresh cycle.

VHDL: Looking at the Source window you can see that the current line arrow points to line 61 of the *dramcon_rtl.vhd* file. In this line you can see that the logic assigning *we_n* is wrong - it does not account for the *REF2* state.

The code shows that the incorrect assignment is used for the example with the correct assignment immediately below (line 65) that will hold *we_n* high through both states of the refresh cycle.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation.

1. Select **Simulate > End Simulation**. Click Yes.

Chapter 14

SystemVerilog Assertions and Functional Coverage

Introduction

In this lesson you will:

- simulate the design with assertion failure tracking disabled in order to note how long the simulation runs before an error is reached
- rerun the simulation with assertion failure tracking enabled in order to see how quickly assertion failures can help you locate errors and speed debugging
- use cover directives and covergroups to cause testbench reactivity and enable functional coverage capabilities
- create a functional coverage report using the graphic interface.

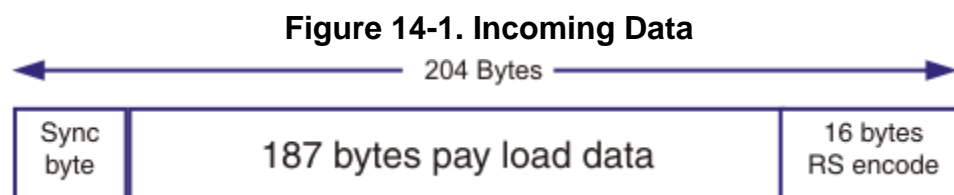
Design Files for this Lesson

This lesson uses an interleaver design with SystemVerilog assert and cover directives and SystemVerilog covergroups to gain a basic understanding of how functional verification information is gathered and displayed in QuestaSim.

The files for the interleaver design are located in
/<install_dir>/questasim/examples/tutorials/systemverilog/vlog_dut.

Understanding the Interleaver Design

An interleaver scrambles the byte order of incoming data in order to aid error detection and correction schemes such as Reed Solomon/Viterbi. In the design used for this lesson, the incoming data consists of a sync byte (0xb8, 0x47) followed by 203 bytes of packet data. The 203 bytes consist of 187 bytes of data to which a Reed Solomon encoder has previously appended 16 bytes of data.

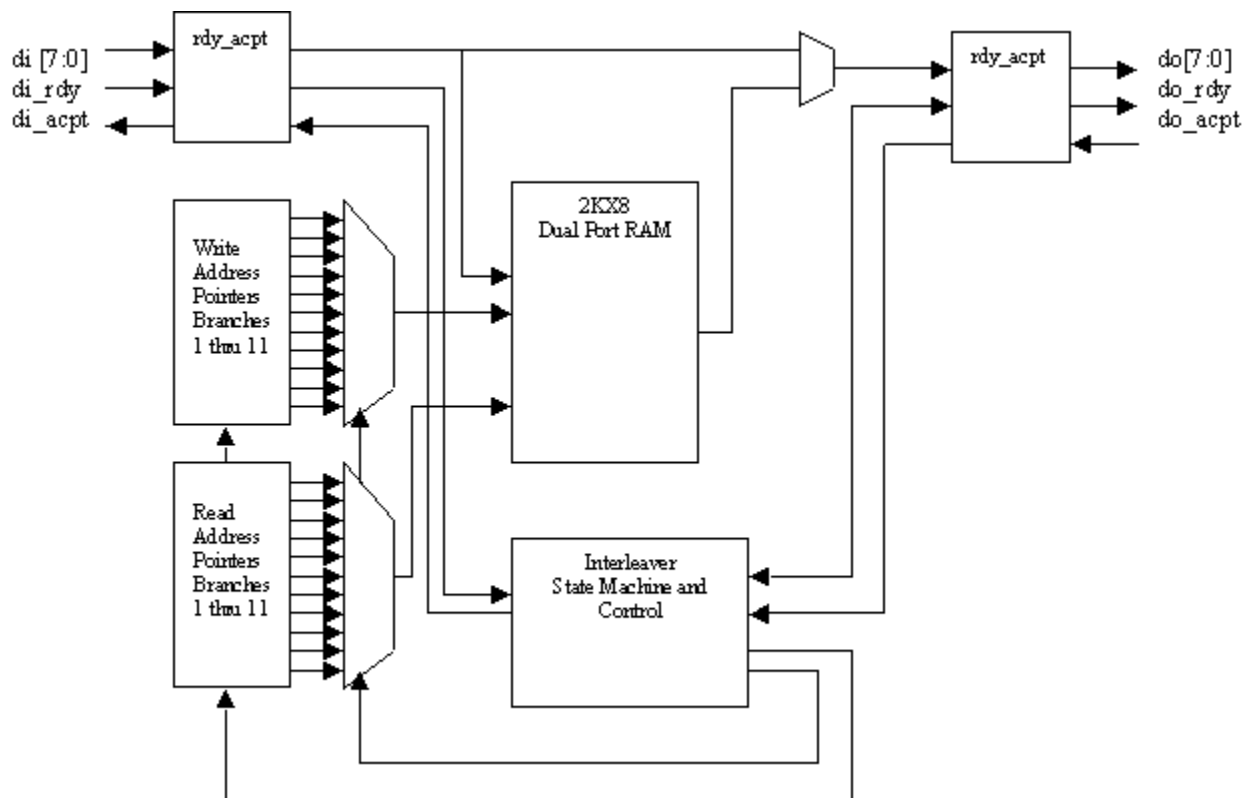


The interleaver has 12 levels numbered 0 to 11. Each level, except the first, can be conceptually thought of as a FIFO shift register. The depth of each register is 17 greater than the previous level. The first level (level 0) has a depth of zero (0); level 1 has a depth of 17; level 2, a depth of 34, and so on. Level 12 has a depth of 187. The sync byte of the packet is routed through level 0. When a byte is loaded into each level's FIFO shift register, the byte shifted out on the corresponding level is output by the interleaver.

The FIFO shift registers are implemented using a single 2KX8 RAM instead of actual registers. The RAM is divided into 11 different sections and each level has separate read and write address registers. A state machine controls which level is being written to and read, and determines which level's address registers are selected to drive the actual RAM address inputs.

A common block called *rdy_acpt* is used to receive and drive the interleaver data in (di) and data out (do) ports, respectively. The *rdy_acpt* block implements a simple handshake protocol. When the device upstream from the interleaver drives data to it, the data is driven and the ready signal (*di_rdy*) is asserted. The upstream block asserts the data along with its *rdy* signal and must leave them asserted until the downstream block asserts its accept (*di_acpt*) signal. In other words, the data isn't considered to have been transferred until both the *rdy* and *acpt* signals are asserted on the rising edge of the clock. Both sides of the *rdy_acpt* block follow this handshake protocol. The block diagram of the interleaver is shown in [Figure 14-2](#).

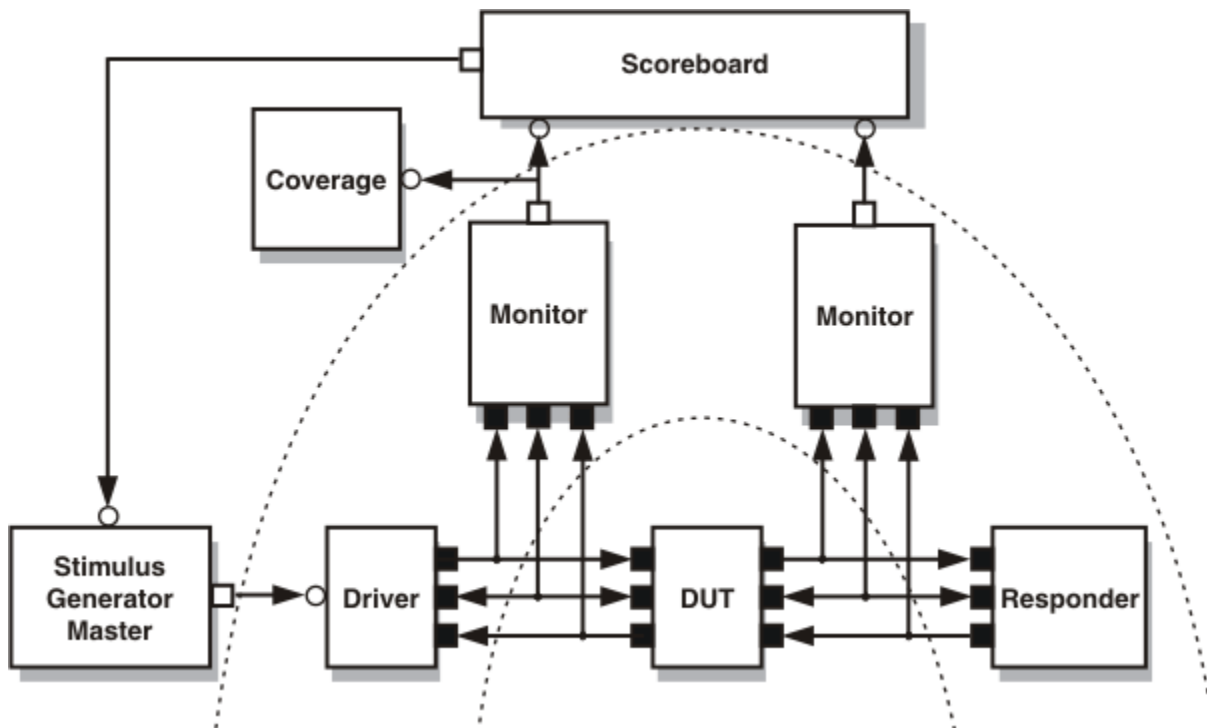
Figure 14-2. Block Diagram of the Interleaver



The Testbench

Figure 14-3 shows how the testbench components are connected. The stimulus generator creates random data packets and sends them to the driver. Even though the testbench is module based, the stimulus generator still creates packets that are transaction based (SV class). This is the big advantage offered by the QuestaSim Advanced Verification Methodology (AVM) - it allows you to take advantage of transaction level modeling (TLM) techniques without having to convert your test environment to a complete object oriented programming environment.

Figure 14-3. Block Diagram of the Testbench



The driver takes the TLM packets and converts them to pin-level signals. The driver also uses randomization to vary the timing of the packets delivered to the device.

The monitors take the pin level activity of the DUT inputs and outputs and convert that activity back to a transaction for use in the coverage collector and scoreboard.

The scoreboard contains a "golden" reference model of the interleaver that is then compared against the actual output the device. There is also a feedback loop from the scoreboard to the stimulus generator to tell the stimulus generator when testing is complete.

The coverage collector accumulates functional coverage information to help determine when testing is complete. It measures things like how many different delay values were used in the delivery of packets.

Finally the responder (which is actually part of the driver in this testbench) provides the handshaking *ready/accept* signals needed for packet delivery.

Related Reading

User's Manual Chapter: [Verification with Functional Coverage](#).

Run the Simulation without Assertions

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons).

Copy the files from `/<install_dir>/examples/tutorials/systemverilog/vlog_dut` to the new directory.

2. Start QuestaSim if necessary.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

Upon opening QuestaSim for the first time, you will see the Welcome to QuestaSim dialog. Click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Run the simulation with a *.do* file.

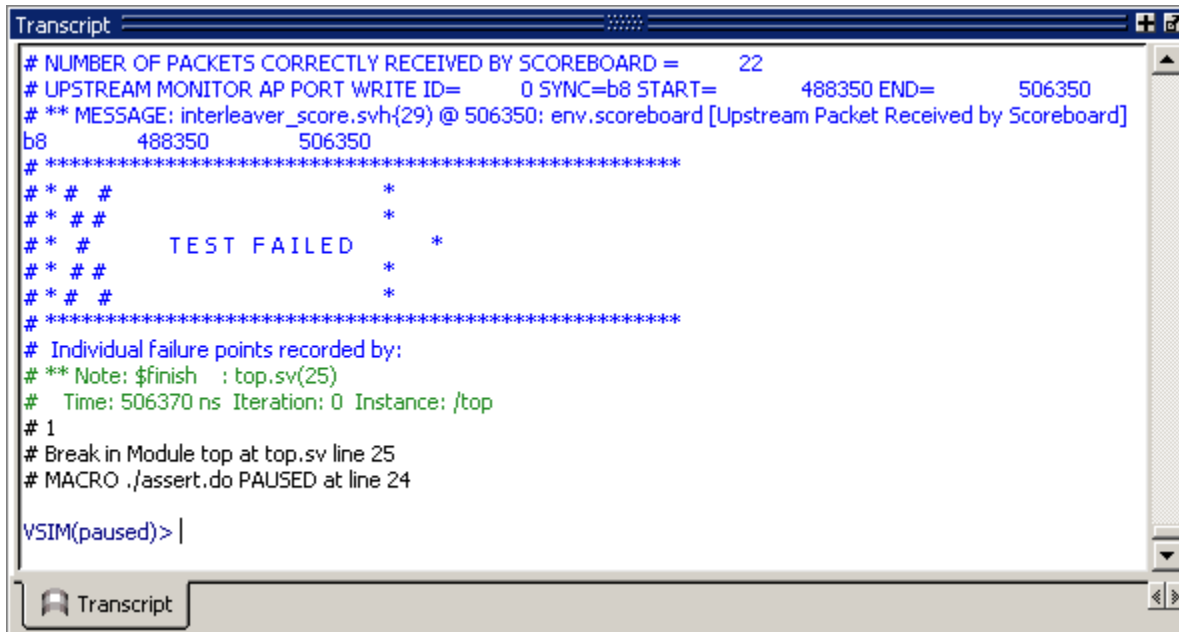
- a. Enter **do assert.do** at the QuestaSim> prompt.

The *assert.do* file will compile and load the design, run the simulation without assertions, then pause while you examine simulation results. (In Windows, you may see a “Finish Vsim” dialog that will ask, “Are you sure you want to finish?” Click **No**.)

In a moment, you will enter a **resume** command to rerun the simulation with assertions.

After the design loads, the first simulation runs until it reaches the \$finish in the *top.sv* module. At this point, a “Test Failed” message is displayed in the Transcript pane as shown in [Figure 14-4](#). The summary information shows that 22 packets were correctly received by the scoreboard. This is a typical message from a self-checking testbench.

Figure 14-4. First Simulation Stops at Error

The image shows a screenshot of a simulation transcript window titled "Transcript". The window contains the following text:

```
# NUMBER OF PACKETS CORRECTLY RECEIVED BY SCOREBOARD = 22
# UPSTREAM MONITOR AP PORT WRITE ID= 0 SYNC=b8 START= 488350 END= 506350
# ** MESSAGE: interleaver_score.svh(29) @ 506350: env.scoreboard [Upstream Packet Received by Scoreboard]
b8 488350 506350
# *****
# * # *
# * # *
# * # TEST FAILED *
# * # *
# * # *
# *****
# Individual failure points recorded by:
# ** Note: $finish : top.sv(25)
# Time: 506370 ns Iteration: 0 Instance: /top
# 1
# Break in Module top at top.sv line 25
# MACRO ./assert.do PAUSED at line 24

VSIM(paused)> |
```

The transcript window has a scrollbar on the right and a "Transcript" tab at the bottom left.

At this point, you would normally generate waveforms for debugging the test failure. But this information does not give a clear indication of the source of the problem. Where do you start? This can be a very difficult problem to find unless you have some debugging tools, such as assertions.

Run the Simulation with Assertions

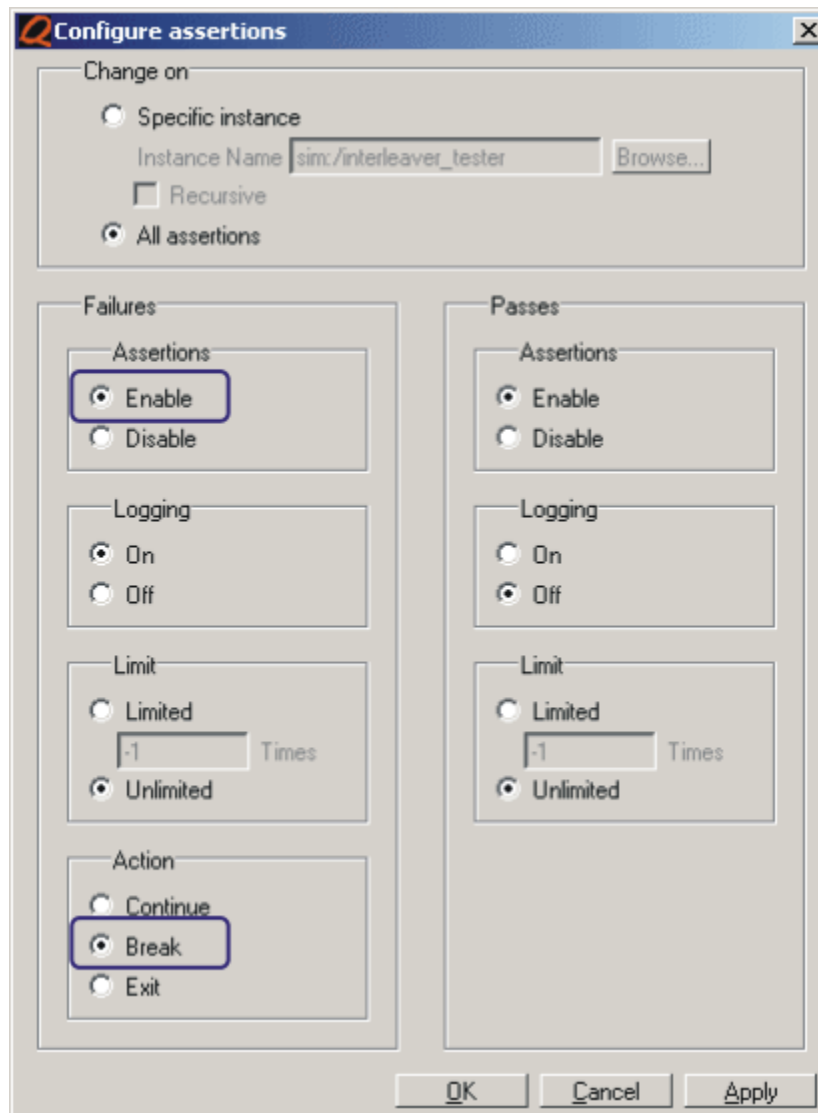
1. Rerun the simulation with assertions.
 - a. Enter the **resume** command at the VSIM(paused)> prompt.
2. After the design loads, configure all assertions to “Break on Failure.”
 - a. The Assertions tab of the Analysis window should open. If it does not, select **View > Coverage > Assertions** from the menus to open it.

Notice that the assertions are enabled for both Passes and Failures. This means that both counts and visual indications in the Wave window will be maintained for assertion Passes and Failures. It should be noted that this not the default behavior. To get this behavior the simulation must be invoked with the **vsim -assertdebug** switch, as we have done in this simulation. (This command is in the *assert.do* file)

- b. Make sure none of the assertions are selected (**Edit > Unselect All**).
 - c. Click the header bar of the Analysis window (with the Assertions tab open) to make it active. An “Assertions” menu selection will appear in the menu bar. (This assumes the Analysis window is docked in the Main window.)

- d. Select **Assertions > Configure**. This will open the Configure Assertions dialog box (Figure 14-5).
- e. Select **All assertions** in the Change On section.
- f. Select **Enable** failures.
- g. Set the Failures Action to **Break**.

Figure 14-5. Enabling Assertion Failure Tracking and Action



- h. Click **OK** to accept the changes and close the dialog.

The Failure column in the Assertions tab now shows assertion failure tracking "enabled," and the Failure Action column shows "break" for all assertion failures (Figure 14-6).

Figure 14-6. Assertions Set to Break on Failure

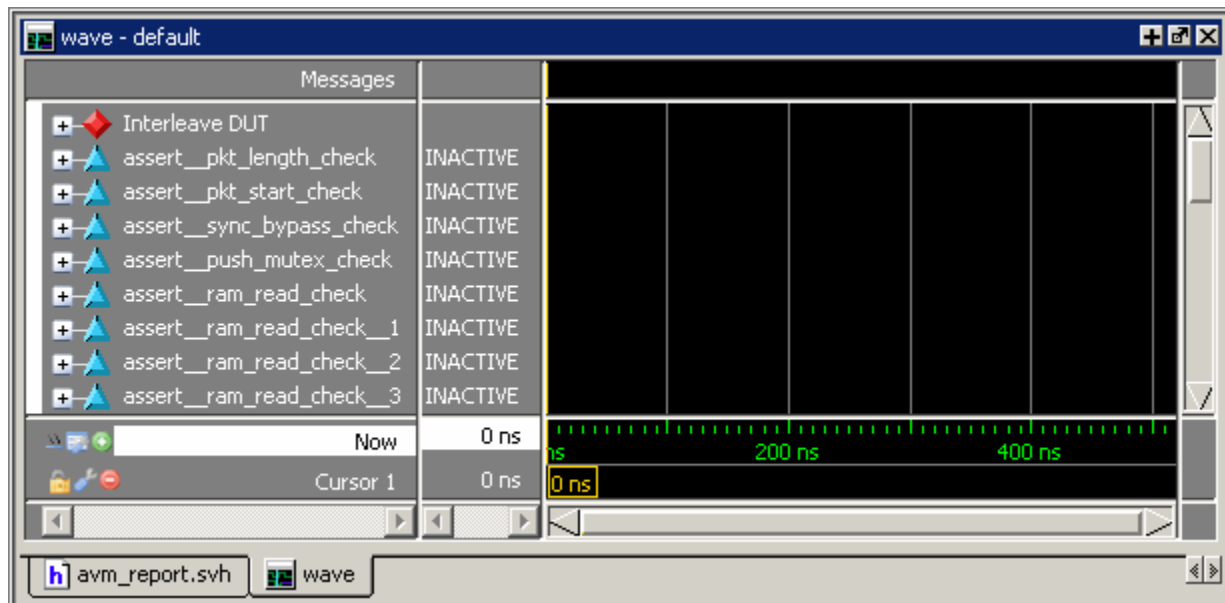
Failure	Pass	Failure Count	Pass Count	Active Count	ATV	Failure Action	Assertion Expression
enabled	enabled	0	0	0	off	break	assert(@(posedge pir
enabled	enabled	0	0	0	off	break	assert(@(posedge pir
enabled	enabled	0	0	0	off	break	assert(@(posedge pir
enabled	enabled	0	0	0	off	break	assert(@(posedge clk
enabled	enabled	0	0	0	off	break	assert(@(posedge clk
enabled	enabled	0	0	0	off	break	assert(@(posedge clk
enabled	enabled	0	0	0	off	break	assert(@(posedge clk
enabled	enabled	0	0	0	off	break	assert(@(posedge clk
enabled	enabled	0	0	0	off	break	assert(@(posedge clk

The Transcript pane shows the command line equivalent of the actions you have just performed:

```
assertion fail -action break -r *
```

3. Add all assertions to the Wave window
 - a. Select all assertions in the Assertions tab of the Analysis window and either drag and drop them into the Wave window, or use the right mouse button to open a context menu and select **Add Wave > Selected Objects**.

Figure 14-7. Assertions in Wave Window



Debugging with Assertions

Run the simulation and debug the assertion failure.

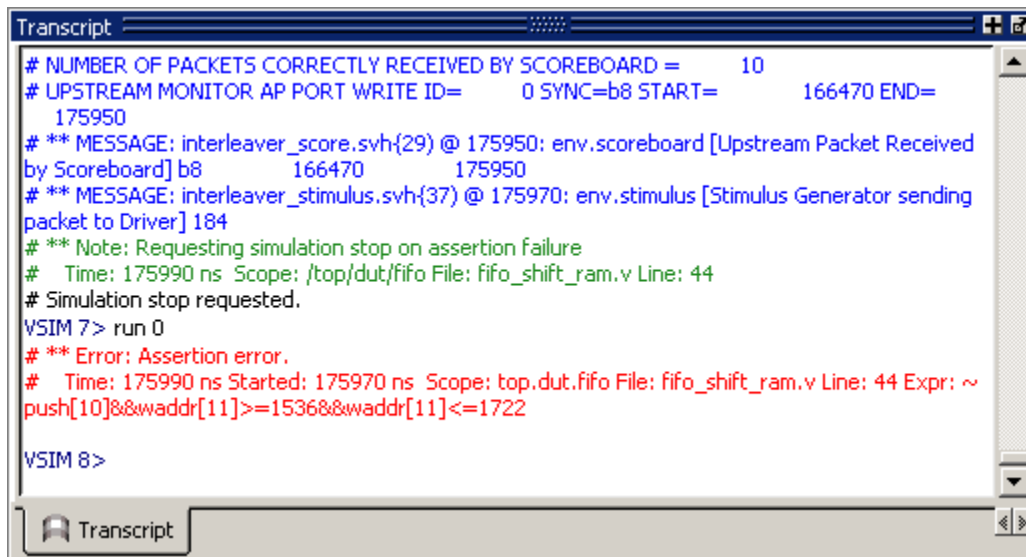
1. Run the simulation with assertion failure tracking enabled.
 - a. Enter **run -all** at the QuestaSim prompt.
 - b. When the simulator stops, enter **run 0**.

The **run 0** command is needed to print any assertion messages when the assertion failure action is set to Break. The reason this happens is due to scheduling. The "break" must occur in the active event queue. However, assertion messages are scheduled in the observed region. The observed region is later in the time step. The **run 0** command takes you to the end of the time step.

2. Verify the output of the Transcript pane (Figure 14-8).

Notice that the assertion failure message gives an indication of the failing expression. This feature is enabled when the **-assertdebug** switch is used with the **vsim** command at invocation. (This command is in the *assert.do* file.)

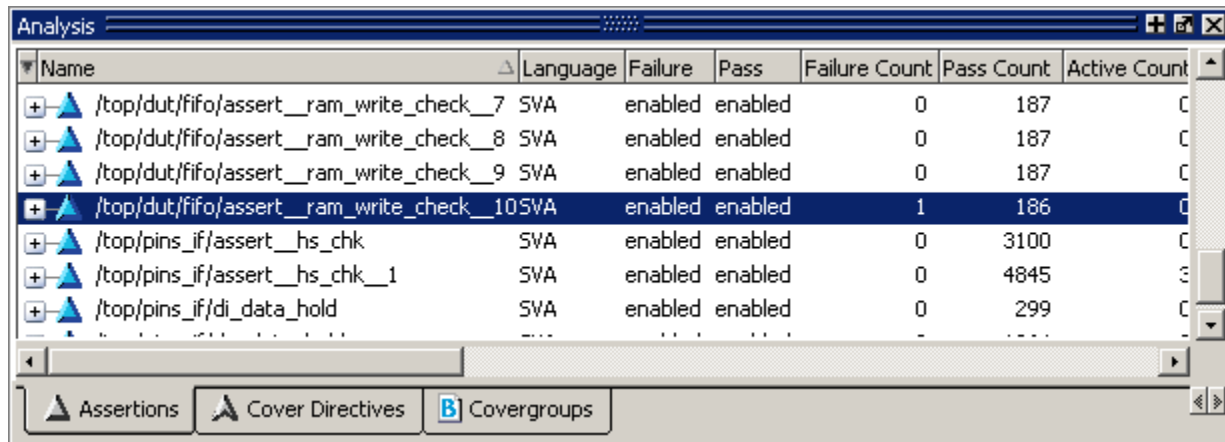
Figure 14-8. Assertion Failure Message in the Transcript



3. View the assertion failure in the Assertions tab of the Analysis pane.

The failed assertion is highlighted and '1' is displayed in the Failure Count column for that assertion (Figure 14-9).

Figure 14-9. Assertions Tab Shows Failure Count

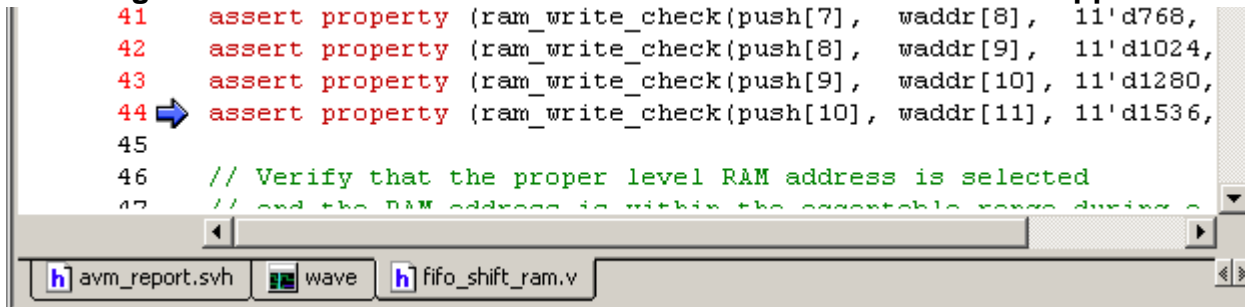


Name	Language	Failure	Pass	Failure Count	Pass Count	Active Count
/top/dut/fifo/assert_ram_write_check_7	SVA	enabled	enabled	0	187	0
/top/dut/fifo/assert_ram_write_check_8	SVA	enabled	enabled	0	187	0
/top/dut/fifo/assert_ram_write_check_9	SVA	enabled	enabled	0	187	0
/top/dut/fifo/assert_ram_write_check_10	SVA	enabled	enabled	1	186	0
/top/pins_if/assert_hs_chk	SVA	enabled	enabled	0	3100	0
/top/pins_if/assert_hs_chk_1	SVA	enabled	enabled	0	4845	0
/top/pins_if/di_data_hold	SVA	enabled	enabled	0	299	0

4. Examine the *fifo_shift_ram.v* source code view. The *fifo_shift_ram.v* tab should be open, as shown in (Figure 14-10).

The simulation breaks on line 44 of the *fifo_shift_ram.v* module because the assertion on that line has failed. A blue arrow in the Source window points to the assertion.

Figure 14-10. Source Pane Pointer Shows Where Simulation Stopped



The parameterized property definition starts on line 29.

- a. In the *fifo_shift_ram.v* source code view, scroll to the property definition that starts on line 29.

Example 14-1. Assertion Property Definition

```

29  property ram_write_check (we, waddr, lorange, hirange);
30  @({posedge clk}) we |-> ((addra == waddr && waddr >= lorange && waddr <= hirange) ##1
31  (!we && waddr >= lorange && waddr <= hirange));
32  endproperty

```

The property states that whenever *we* (push[10]) is asserted, in the same cycle:

- the ram address bus, *addra* should be equal to the write address bus for level 11 (*waddr[11]*)
- and, *waddr[11]* should be within the range of 1536 to 1722.

In the next cycle:

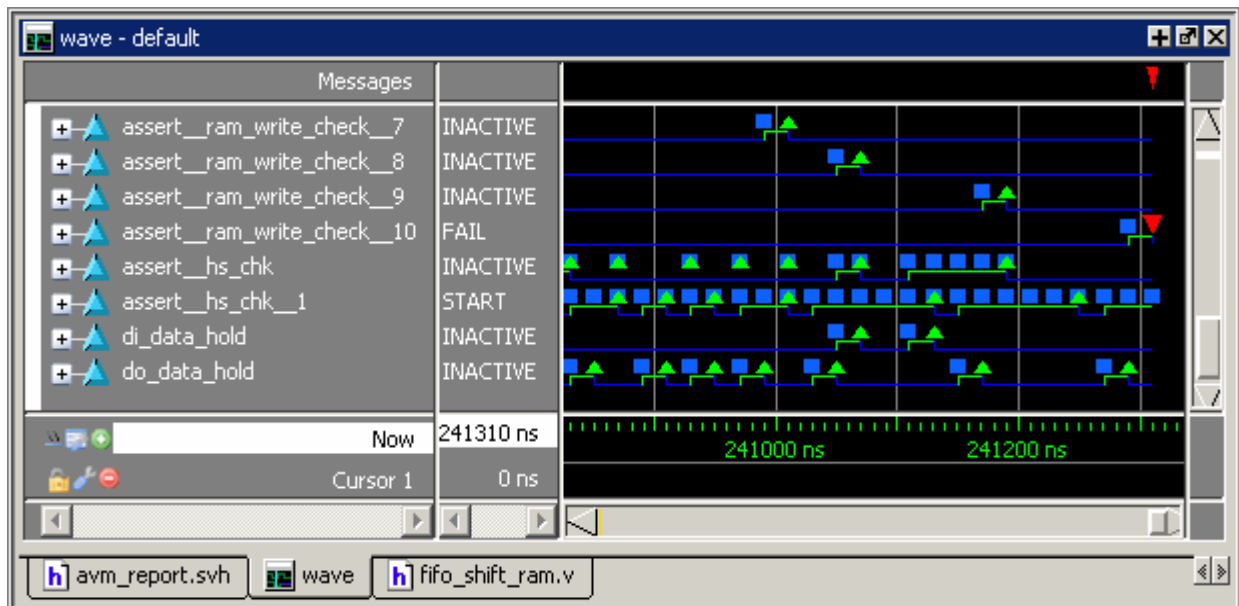
- we should be de-asserted,
- and, the next value of *waddr[11]* should still be within the range 1536 to 1722.

5. Click the **wave** tab to view the assertion failure in the Wave window.

a. Scroll to the *assert_ram_write_check_10* assertion.

The inverted red triangle indicates an assertion failure (Figure 14-11).

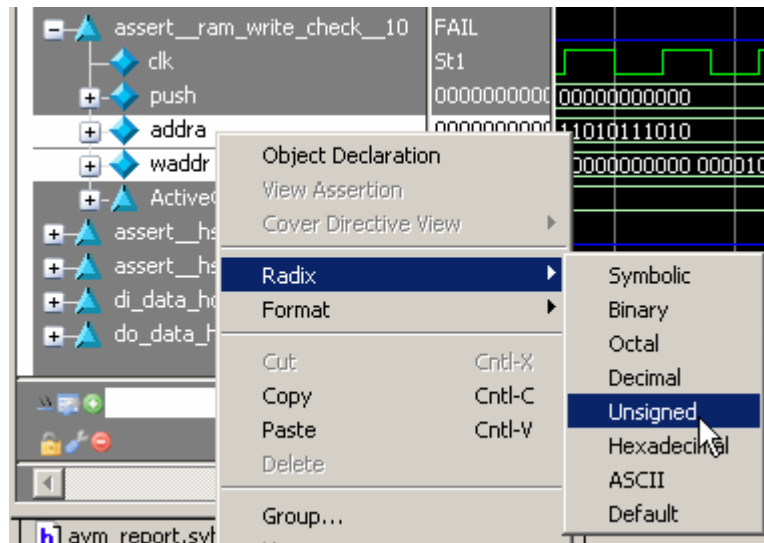
Figure 14-11. The Inverted Red Triangle Indicates an Assertion Failure



The green "midline" indicates where the assertion is active while the low blue line indicates where the assertion is inactive. Blue squares indicate where assertion threads start. Green triangles indicate assertion passes. Passes are only displayed when the **-assertdebug** switch for the **vsim** command is used at invocation (see the *assert.do* file).

- b. Expand the *assert_ram_write_check_10* assertion (click the + sign next to it) in the wave window and zoom in.
- c. Change the radix of *addra* and *waddr* to "Unsigned" by selecting both signals, right-clicking the selected signals to open a popup menu, then selecting **Radix > Unsigned** from the popup menu (Figure 14-12).

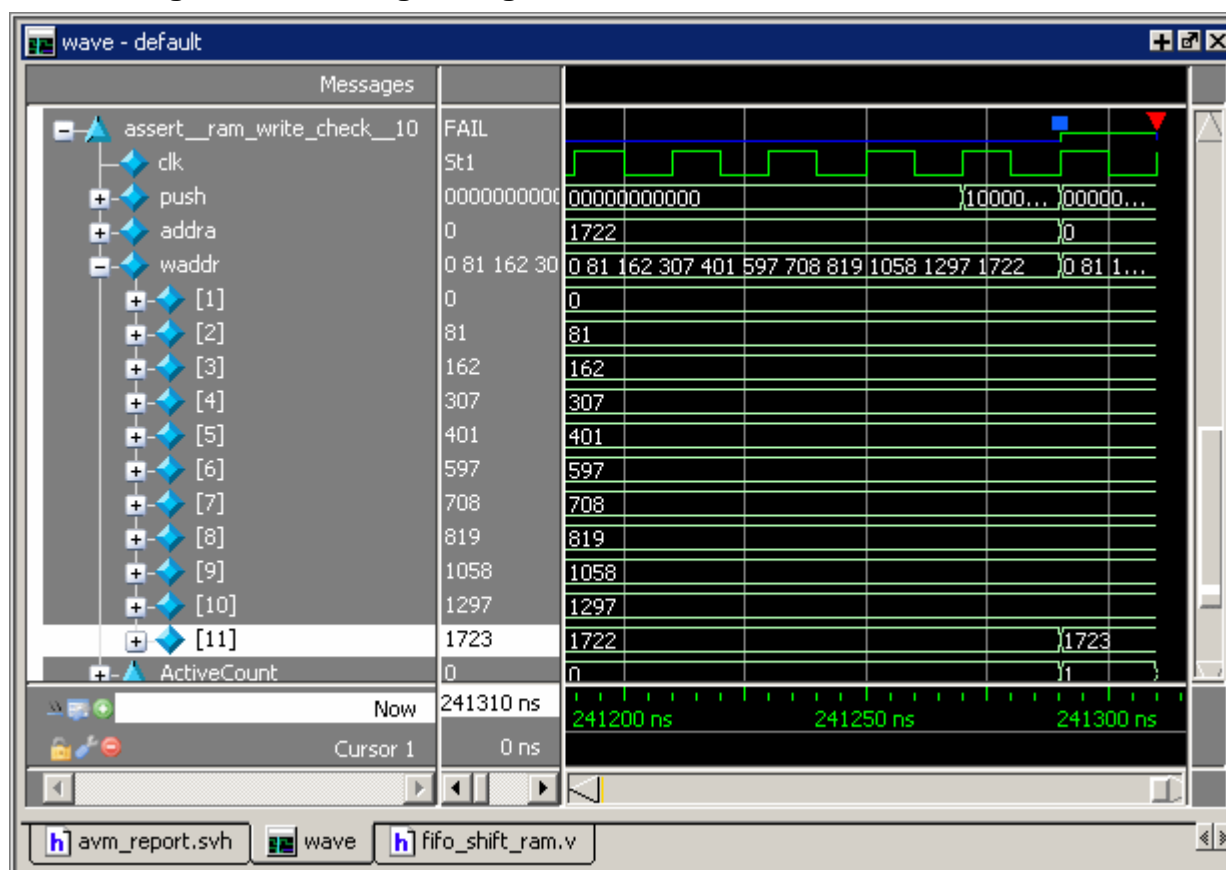
Figure 14-12. Setting the Radix



As you can see in [Figure 14-13](#), the value of `waddr[11]` has incremented to 1723 which is out of the allowable address range. Remember, in the Transcript message for the assertion violation, the failing expression indicated that `waddr[11]` was out of range.

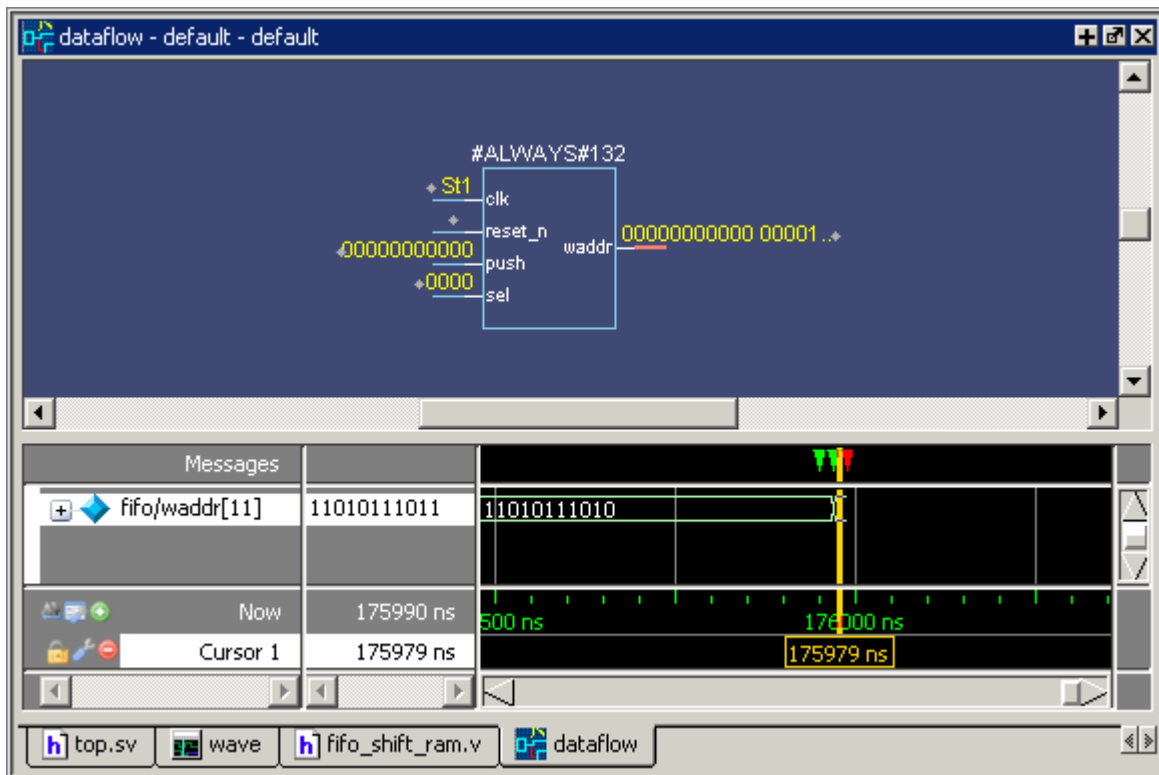
6. Examine the `waddr[11]` signal in the Dataflow window.
 - a. Expand the `waddr` signal by clicking the + sign next to it, then scroll to the `waddr[11]` signal ([Figure 14-13](#)).

Figure 14-13. Diagnosing Assertion Failure in the Wave Window



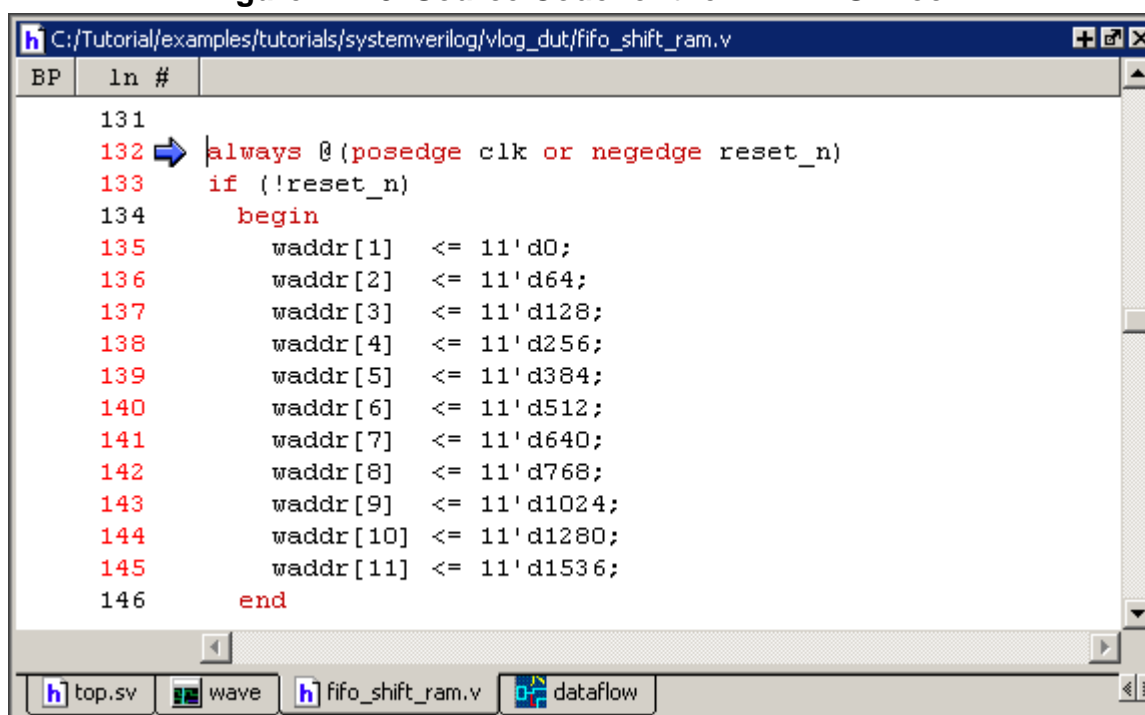
- b. Double-click the *waddr[11]* waveform in the Wave window to open it in the Dataflow window. The *waddr[11]* signal will be highlighted, as shown in [Figure 14-14](#), and the block shown is the ALWAYS procedure that created the *waddr* signal.

Figure 14-14. The *wadder11* Signal in the Dataflow Window



- c. Change the radix of `/top/dut/fifo/waddr[11]` in the Wave viewer portion of the Dataflow window by right-clicking it and selecting **Radix > Unsigned** from the popup menu. With the cursor at 241290 ns, as shown in Figure 14-14, we see that the value of `waddr[11]` is 1723.
- d. Select the symbol for the ALWAYS block in the Dataflow window. The `fifo_shift_ram.v` source code view will open automatically, with a blue arrow pointing to the code for the ALWAYS block (Figure 14-15).

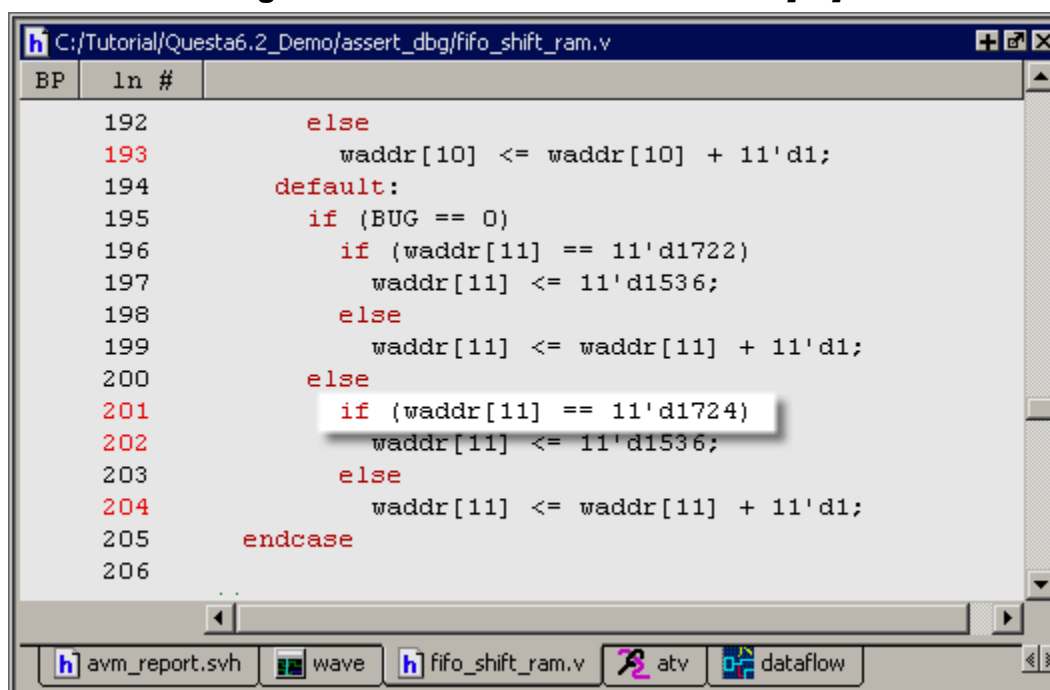
Figure 14-15. Source Code for the ALWAYS Block



```
C:/Tutorial/examples/tutorials/systemverilog/vlog_dut/fifo_shift_ram.v
BP  ln #
131
132  always @ (posedge clk or negedge reset_n)
133  if (!reset_n)
134  begin
135      waddr[1]  <= 11'd0;
136      waddr[2]  <= 11'd64;
137      waddr[3]  <= 11'd128;
138      waddr[4]  <= 11'd256;
139      waddr[5]  <= 11'd384;
140      waddr[6]  <= 11'd512;
141      waddr[7]  <= 11'd640;
142      waddr[8]  <= 11'd768;
143      waddr[9]  <= 11'd1024;
144      waddr[10] <= 11'd1280;
145      waddr[11] <= 11'd1536;
146  end
top.v  wave  fifo_shift_ram.v  dataflow
```

If you scroll down to the case covering *waddr[11]* you can see that the upper address range for resetting *waddr[11]* has been incorrectly specified as 11'd1724 (Figure 14-16). This is the cause of the error.

Figure 14-16. Source Code for *waddr[11]*



```
C:/Tutorial/Questa6.2_Demo/assert_dbg/fifo_shift_ram.v
BP  ln #
192  else
193      waddr[10] <= waddr[10] + 11'd1;
194  default:
195      if (BUG == 0)
196          if (waddr[11] == 11'd1722)
197              waddr[11] <= 11'd1536;
198          else
199              waddr[11] <= waddr[11] + 11'd1;
200      else
201          if (waddr[11] == 11'd1724)
202              waddr[11] <= 11'd1536;
203          else
204              waddr[11] <= waddr[11] + 11'd1;
205  endcase
206
avm_report.svh  wave  fifo_shift_ram.v  atv  dataflow
```

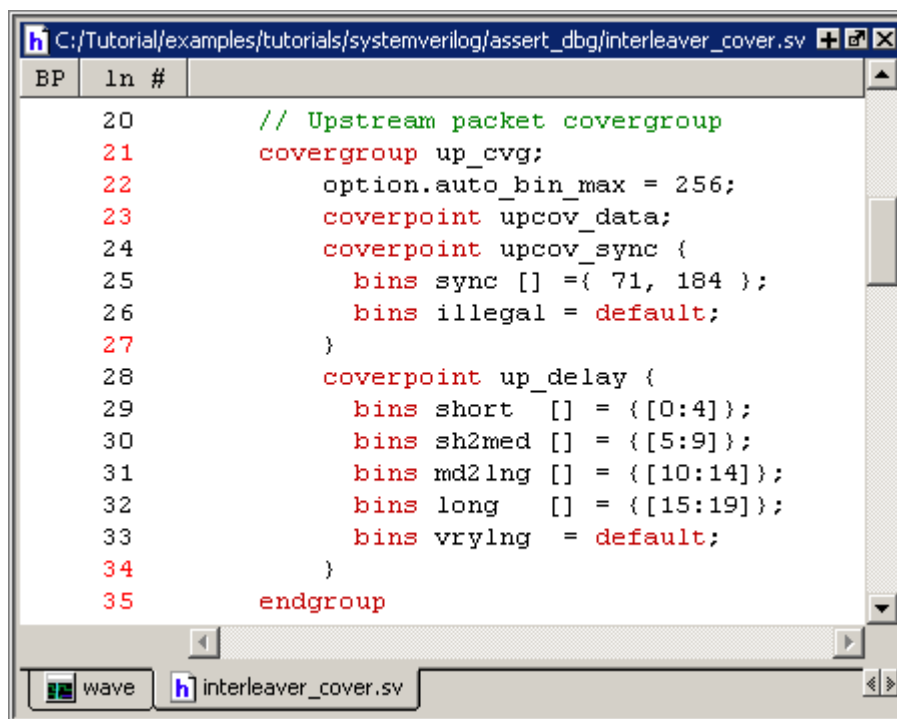
7. Quit the simulation.
 - a. Enter **quit -sim** at the **QuestaSim** prompt and close all tabs in the MDI Frame.

Exploring Functional Coverage

1. Load the interleaver once again.
 - a. Enter **do fcov.do** at the **QuestaSim**> prompt.

The interleaver uses a parameter (PKT_GEN_NUM), which is set to 80, to determine the number of valid packets that will be interleaved. After the scoreboard receives and verifies that 80 packets have been successfully interleaved it informs the test controller, which halts both the stimulus generator and driver. During the simulation, a coverage collector records several metrics for each packet sent to, and output by, the interleaver. Figure 14-17 shows the source code of the *up_cvg* covergroup.

Figure 14-17. Covergroup Code



The covergroup records the information stored in the upstream transaction captured by the monitor. The transaction includes the byte wide values of the packet payload data, the sync byte, and the individual data payload transfer times.

In order to have a bin created for each data value, `option.auto_bin_max = 256` is specified since the default number of auto bins created is defined by the

SystemVerilog LRM to be 64. The sync byte values are 71 and 184 which correspond to 8'h47 and 8'hb8 respectively.

All other sync byte values are stored in the bin named *illegal* which should remain empty. The packet payload data transfer delays times are recorded since the driver randomly drives data to the interleaver. The packet payload data transfer delay bin names are self descriptive and create a separate bin for each delay value except for the *vrylng* (very long) bin which records any data transfer delay of 20 or more cycles.

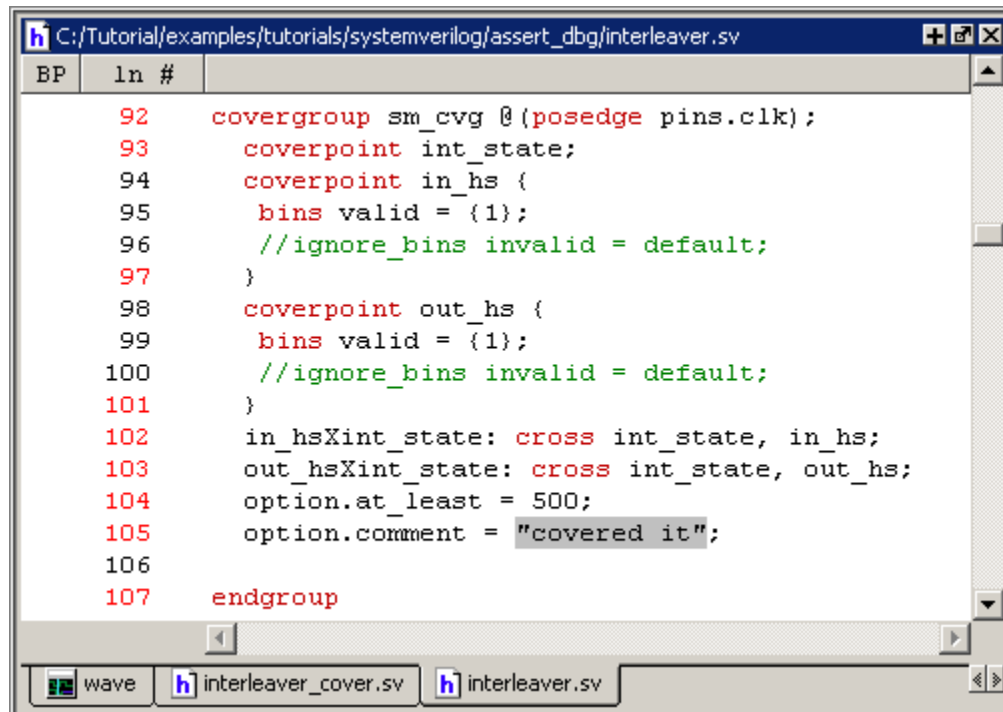
2. In the Covergroups tab of the Analysis window, expand the */top/dut* hierarchy (click the + sign next to */top/dut*) and you will find two additional covergroups – *sm_transitions_cvg* and *sm_cvg* – which monitor the interleaver state machine.

Figure 14-18. Covergroup Bins

Name	Coverage	Goal	% of Goal	Status
/top/dut/fifo				
TYPE ram_cvg	0.0%	100	0.0%	
/top/dut				
TYPE sm_transitions_cvg	0.0%	100	0.0%	
CVP sm_transitions_cvg::int_state	0.0%	100	0.0%	
bin idle_st	0	1	0.0%	
bin bypass_st	0	1	0.0%	
TYPE sm_cvg	0.0%	100	0.0%	
CVP sm_cvg::int_state	0.0%	100	0.0%	
CVP sm_cvg::in_hs	0.0%	100	0.0%	
CVP sm_cvg::out_hs	0.0%	100	0.0%	
CROSS sm_cvg::in_hsXint_state	0.0%	100	0.0%	
CROSS sm_cvg::out_hsXint_state	0.0%	100	0.0%	

The *sm_transitions_cvg* covergroup records the valid state machine transitions while *sm_cvg* records that the state machine correctly accepts incoming data and drives output data in the proper states. [Figure 14-19](#) shows the source code for the *sm_cvg* covergroup.

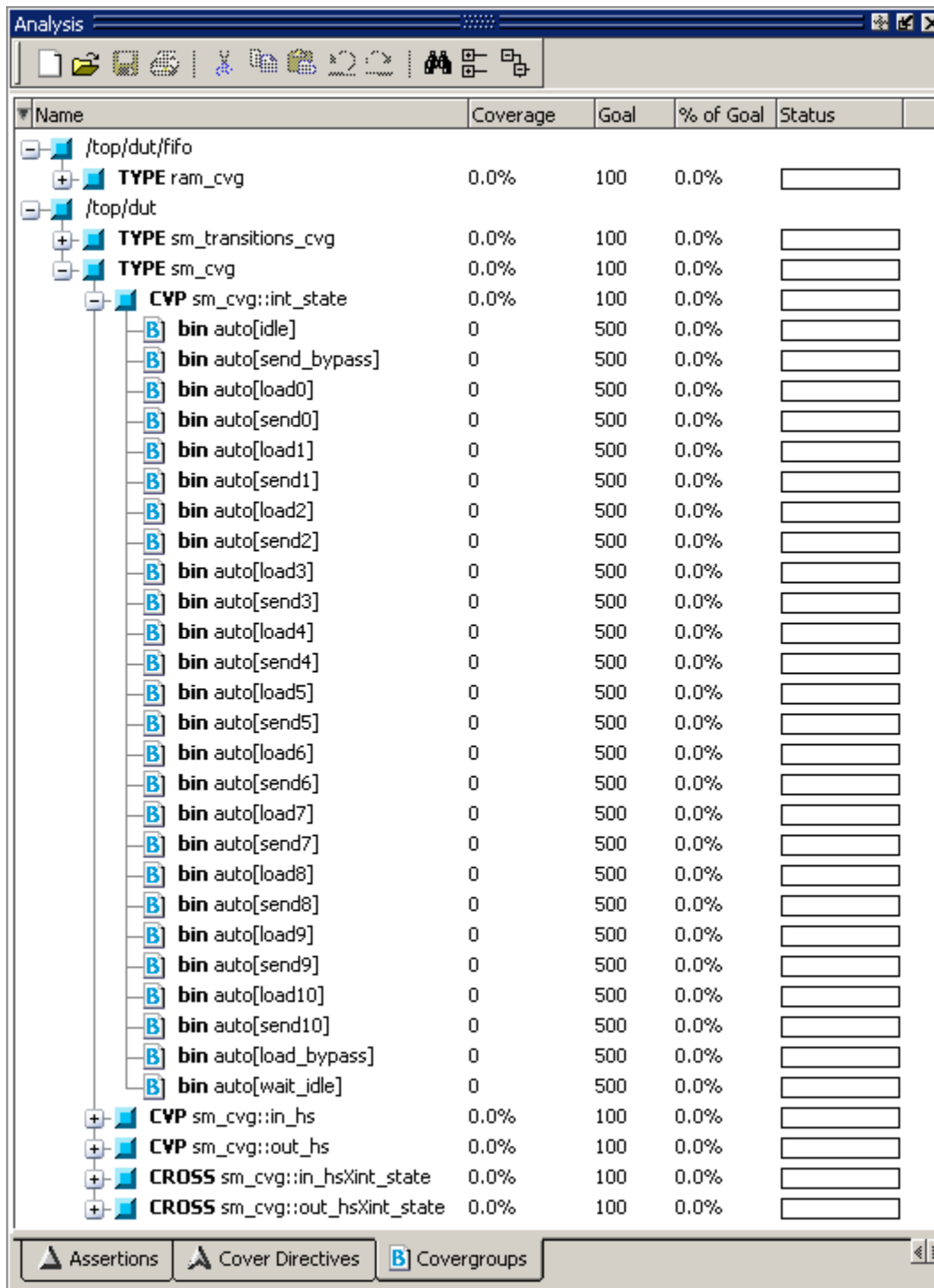
Figure 14-19. Covergroup *sm_svg*



The *in_hs* and *out_hs* signals are derived by ANDing *in_acpt* with *in_rdy*, and *out_acpt* with *out_rdy* respectively. The state machine asserts *in_acpt* when *idle*, *load_bypass*, or in any of the 10 load states, and asserts *out_rdy* when in the *send_bypass* or any of 10 send states.

During proper operation, the *in_hs* signal should only assert if the state machine is *idle*, *load_bypass* or in any of the other 10 load states. Likewise the *out_hs* should only assert if the state machine is in *send_bypass* or any of the 10 send states. By crossing *in_hs* with *int_state* and *out_hs* with *int_state*, this behavior can be verified. Figure 14-20 shows the *sm_svg* covergroup with the *int_state* coverpoint expanded to show all bins. Notice the bin values show the enumerated state names.

Figure 14-20. Bins for the *sm_cvg* Covergroup

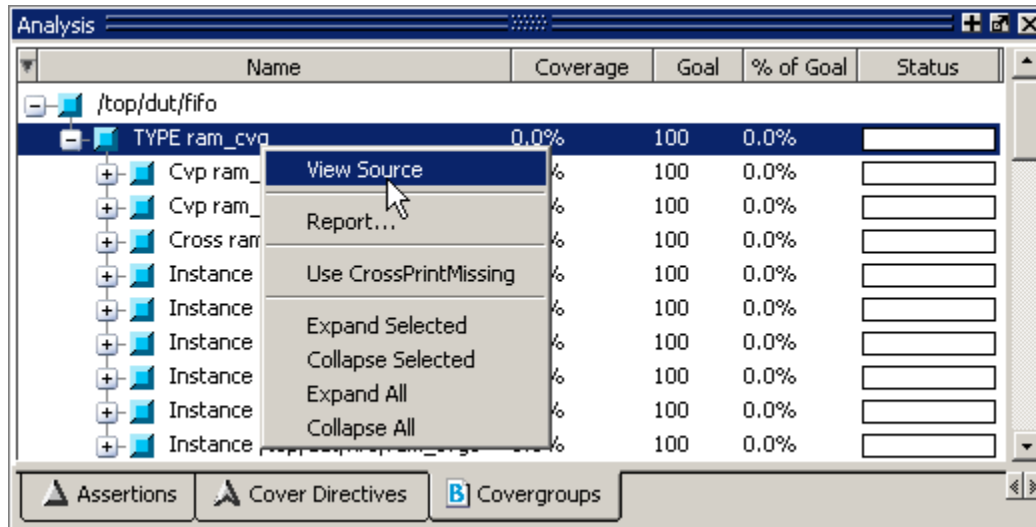


Name	Coverage	Goal	% of Goal	Status
/top/dut/fifo				
+ TYPE ram_cvg	0.0%	100	0.0%	
/top/dut				
+ TYPE sm_transitions_cvg	0.0%	100	0.0%	
- TYPE sm_cvg	0.0%	100	0.0%	
- CVP sm_cvg::int_state	0.0%	100	0.0%	
bin auto[idle]	0	500	0.0%	
bin auto[send_bypass]	0	500	0.0%	
bin auto[load0]	0	500	0.0%	
bin auto[send0]	0	500	0.0%	
bin auto[load1]	0	500	0.0%	
bin auto[send1]	0	500	0.0%	
bin auto[load2]	0	500	0.0%	
bin auto[send2]	0	500	0.0%	
bin auto[load3]	0	500	0.0%	
bin auto[send3]	0	500	0.0%	
bin auto[load4]	0	500	0.0%	
bin auto[send4]	0	500	0.0%	
bin auto[load5]	0	500	0.0%	
bin auto[send5]	0	500	0.0%	
bin auto[load6]	0	500	0.0%	
bin auto[send6]	0	500	0.0%	
bin auto[load7]	0	500	0.0%	
bin auto[send7]	0	500	0.0%	
bin auto[load8]	0	500	0.0%	
bin auto[send8]	0	500	0.0%	
bin auto[load9]	0	500	0.0%	
bin auto[send9]	0	500	0.0%	
bin auto[load10]	0	500	0.0%	
bin auto[send10]	0	500	0.0%	
bin auto[load_bypass]	0	500	0.0%	
bin auto[wait_idle]	0	500	0.0%	
+ CVP sm_cvg::in_hs	0.0%	100	0.0%	
+ CVP sm_cvg::out_hs	0.0%	100	0.0%	
+ CROSS sm_cvg::in_hsXint_state	0.0%	100	0.0%	
+ CROSS sm_cvg::out_hsXint_state	0.0%	100	0.0%	

- Expand the hierarchy (click the + sign) of */top/dut/fifo* and the *ram_cvg* covergroup. Notice that the TYPE *ram_cvg* covergroup contains several instances – designated by INST.

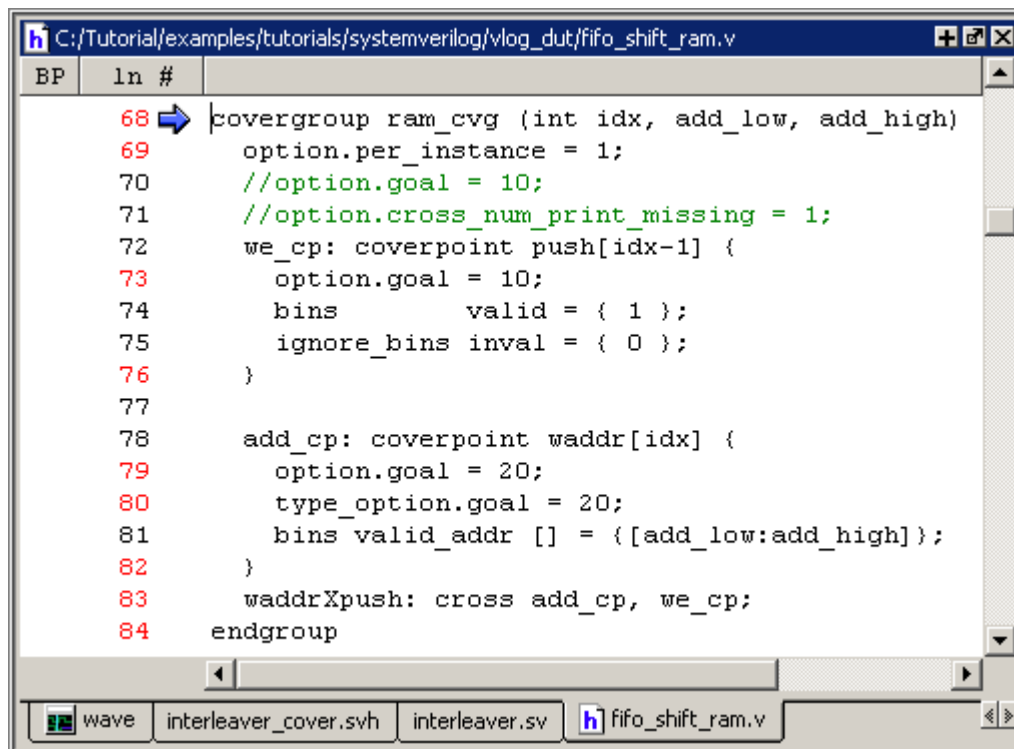
- a. View the source code for *TYPE ram_cvg* by right-clicking the covergroup name and selecting **View Source** from the popup menu (Figure 14-21).

Figure 14-21. Viewing the Source Code for a Covergroup



The *fifo_shift_ram.v* source view will open to show the source code (Figure 14-22).

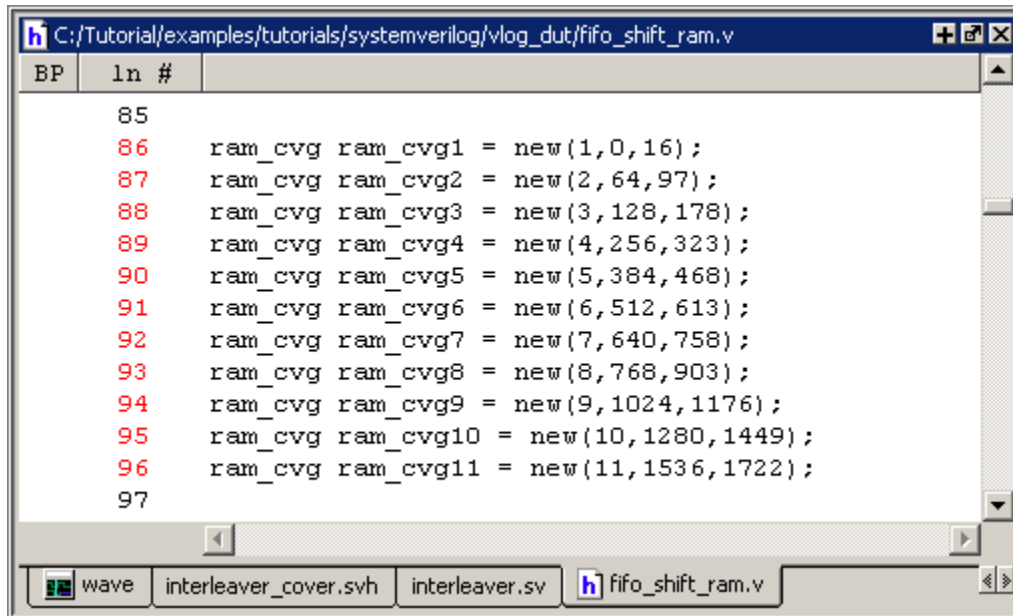
Figure 14-22. Source Code for *ram_cvg* Covergroup



Since the interleaver levels are implemented using a single RAM, with distinct RAM address ranges for each level, the covergroup verifies that only valid address locations are written and read.

Notice that there is only one covergroup but there are 11 covergroup instances that are constructed with different values passed into the constructor (Figure 14-23).

Figure 14-23. Covergroup Instances for *ram_cvg*

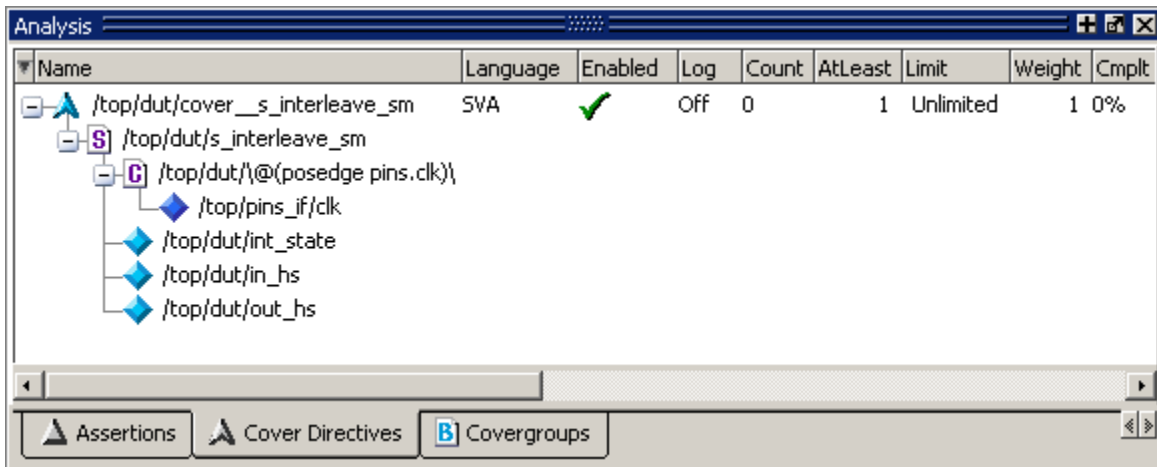


Since the covergroup contains the *option.per_instance = 1* statement (Figure 14-22), the simulator creates a separate covergroup for each instance which covers only the values passed to it in the constructor. The TYPE *ram_cvg* covergroup is the union of all the values of each individual covergroup instance.

4. Open the Cover Directives tab and view the source code for the cover directive.
 - a. If the Cover Directives tab is not open in the Analysis window, select **View > Coverage > Cover Directives**.

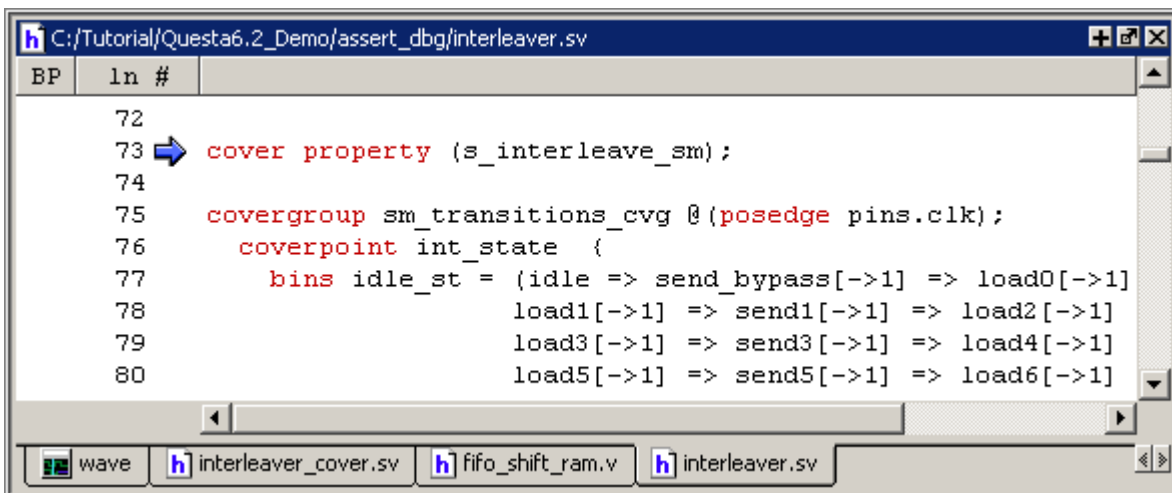
The Cover Directives tab contains a single cover directive (Figure 14-24).

Figure 14-24. Cover Directive for the Interleaver Design



- b. Right-click the cover directive and select **View Source** from the popup menu. Figure 14-25 shows that this cover directive also tracks the interleaver state machine transitions.

Figure 14-25. Source Code for the Cover Directive

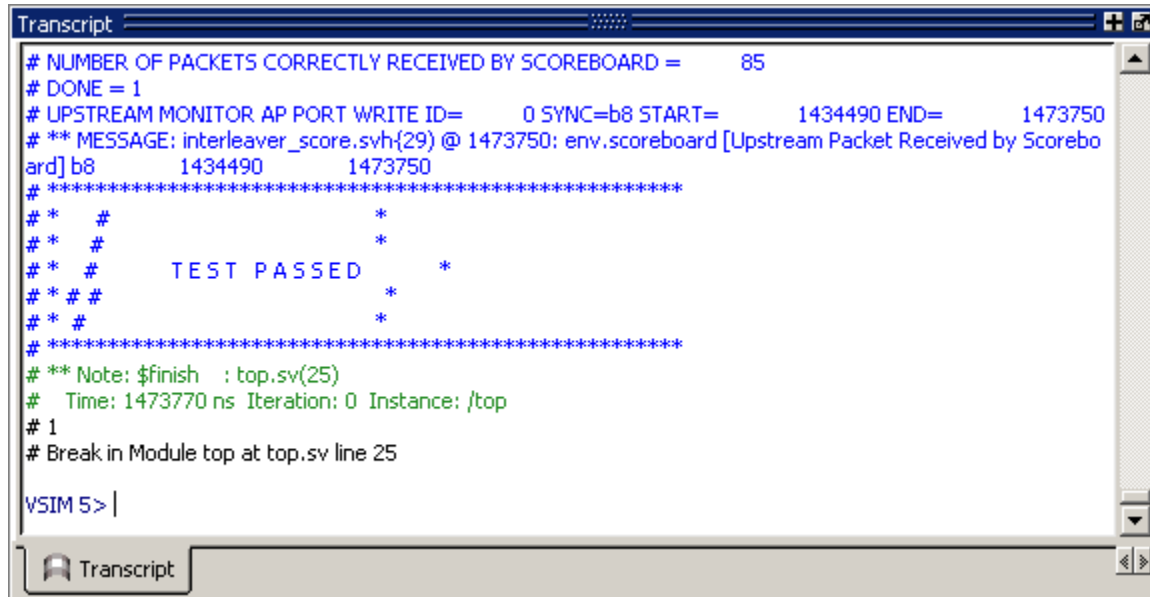


SystemVerilog provides multiple ways to cover important items in a design. The advantage of using a cover directive is that the QuestaSim Wave window provides the ability to see when a directive is hit. While covergroups provide no temporal aspect to determine the precise time an event is covered, covergroups are typically much better at covering data values. Both of SystemVerilog's coverage capabilities provide a powerful combination by using the cover directives temporal nature to determine when to sample data oriented values in a covergroup.

5. Run the simulation and view functional coverage information.
 - a. Enter **run -all** at the command prompt in the Transcript window. The design runs until at "TEST PASSED" message is reached. (In Windows, you may see a "Finish

Vsim” dialog that will ask, “Are you sure you want to finish?” Click **No.**) The Transcript window will display scoreboard information (Figure 14-26).

Figure 14-26. Scoreboard Information in the Transcript

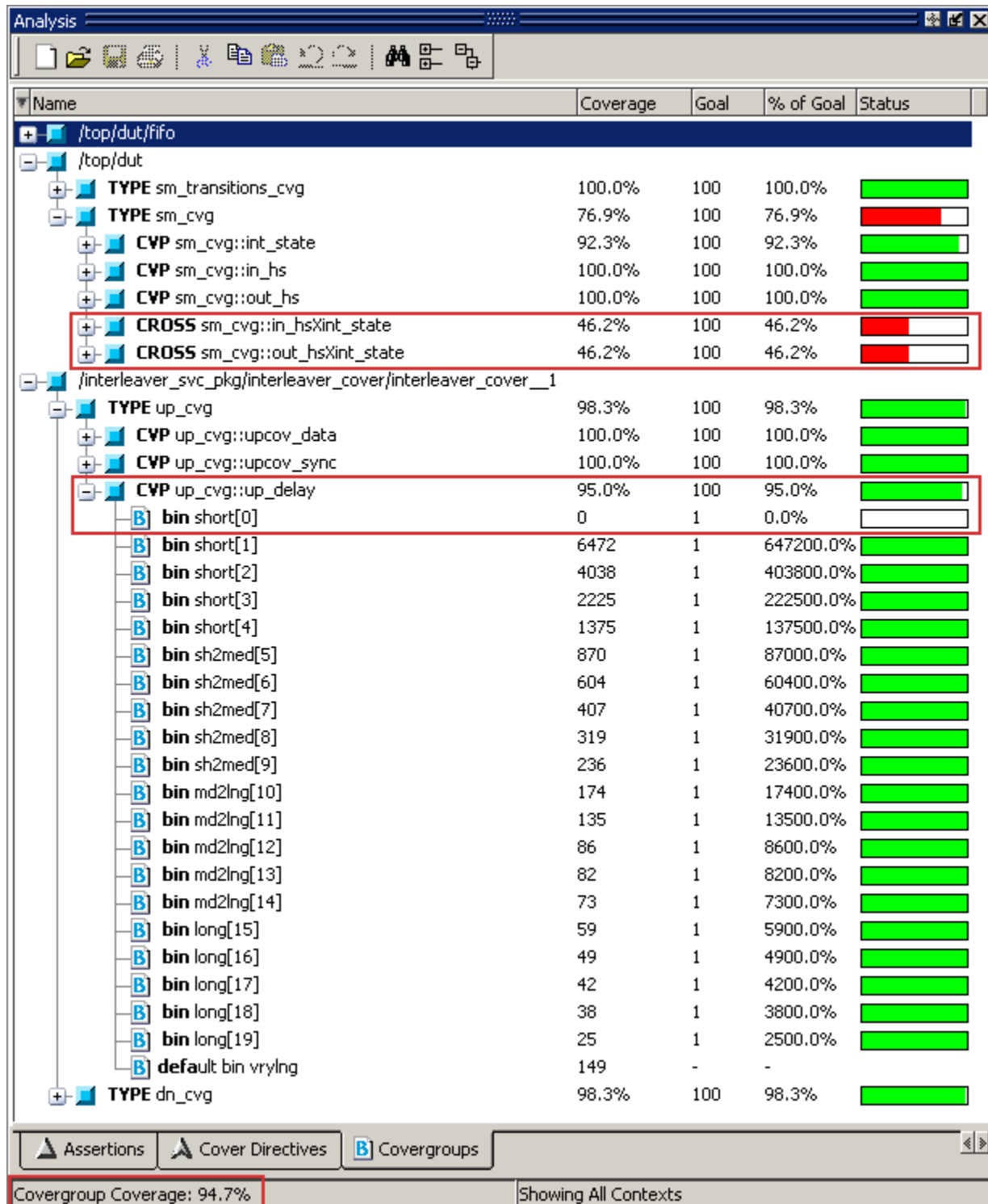


```
Transcript
# NUMBER OF PACKETS CORRECTLY RECEIVED BY SCOREBOARD =      85
# DONE = 1
# UPSTREAM MONITOR AP PORT WRITE ID=      0 SYNC=b8 START=      1434490 END=      1473750
# ** MESSAGE: interleaver_score.svh{29} @ 1473750: env.scoreboard [Upstream Packet Received by Scorebo
ard] b8      1434490      1473750
# *****
# * #
# * #
# * #      TEST PASSED      *
# * #
# * #
# * #
# *****
# ** Note: $finish : top.sv(25)
# Time: 1473770 ns Iteration: 0 Instance: /top
# 1
# Break in Module top at top.sv line 25
VSIM 5> |
```

- b. Expand the functional coverage information in the Covergroups tab of the Analysis window as shown in Figure 14-27. While our overall covergroup coverage is almost 95% (as shown in the status bar at the bottom of the window), there is one *short* bin in the *up_delay* covergroup that has no hits. Currently the driver inserts at least one cycle between words when driving packet payload data.

Also, the *sm_cvg* shows relatively low (76.9%) coverage due to low coverage in the *in_hsXint_state* and *out_hsXint_state* cross coverage bins. This is expected because the *in_hs* signal only asserts in either the idle state, the *load_bypass* state, or one of the 10 load states and the *out_hs* signal only asserts in the *send_bypass* or 10 other send states. So while the indicated coverage for these cross bins might appear to point to an area needing more testing, the absence of coverage is actually indicating that proper behavior took place.

Figure 14-27. Covergroup Coverage in the Analysis Window

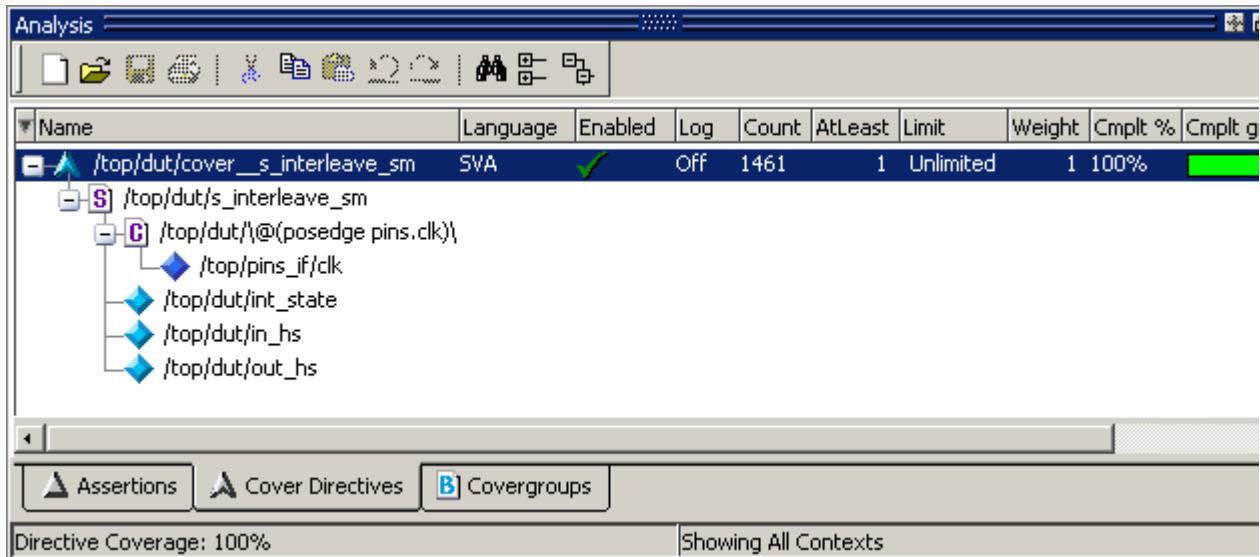


If you expand the *sm_transition_cvg* covergroup you will see that shows 1461 interleaver state transitions (86 when starting from the idle loop, and 1375 when starting from the bypass loop).

- c. Open the Cover Directives tab.

The cover directive counts the same state transitions and, therefore, also indicates a count of 1461 transitions (Figure 14-28).

Figure 14-28. Cover Directive Counts State Transitions



6. Add the cover directive to the Wave window twice.
 - a. Right-click the `/top/dut/cover__s_interleave_sm` cover directive and select **Add Wave > Selected Functional Coverage**.
 - b. Repeat.
7. Change the Cover Directive View of the second directive displayed in the Wave window from Temporal to Count Mode.
 - a. Right-click the second directive and select **Cover Directive View > Count Mode** (Figure 14-29).

Figure 14-29. Changing the Cover Directive View to Count View

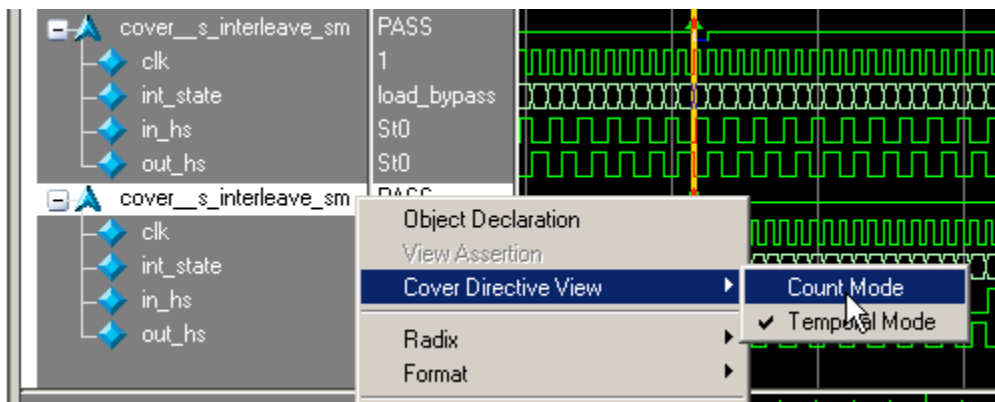


Figure 14-30 and Figure 14-31 are two screen shots of the cover directive. In both screen shots, the top view of the directive shows the temporal aspect of when the thread went active while the bottom view shows the actual count value. When you compare the two screen shots, which display different points in time, it is easy to see the random nature of the drives. In Figure 14-30 there is 780 ns between the start and end of the cover directive thread; in Figure 14-31 there is 920 ns.

Figure 14-30. First Temporal and Count Mode Views of Cover Directive

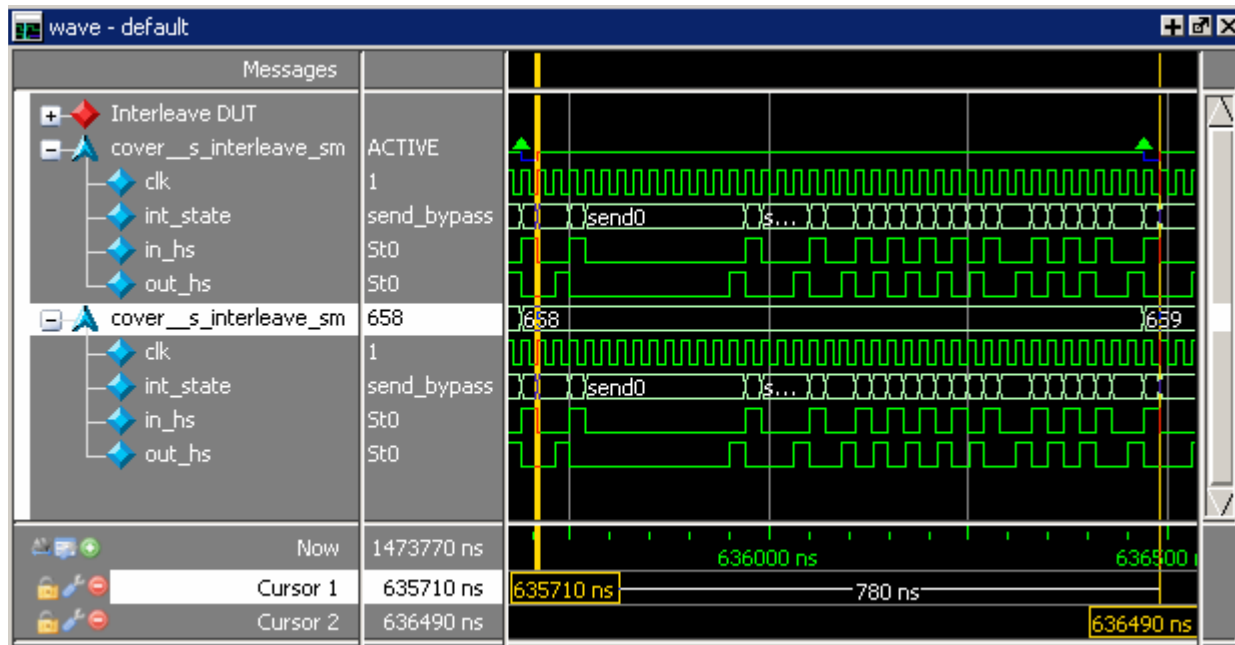
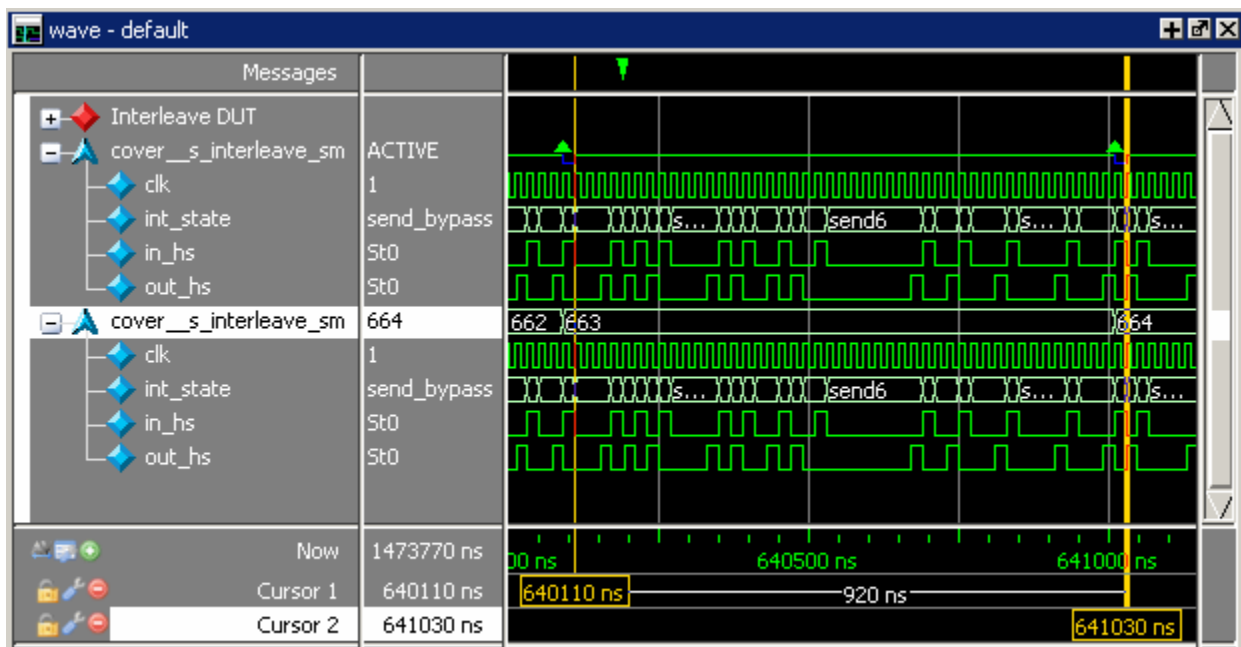


Figure 14-31. Second Temporal and Count Mode Views of Cover Directive

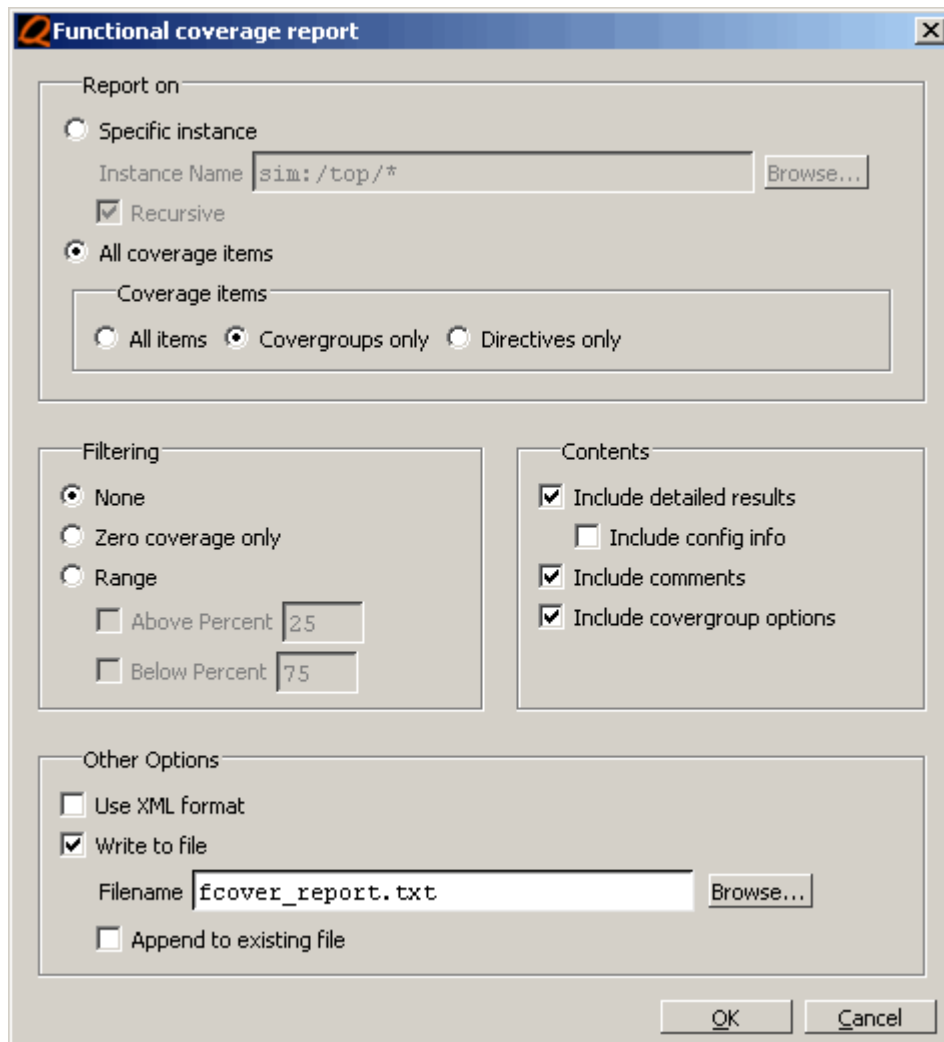


Creating Functional Coverage Reports

You can create functional coverage reports using dialogs accessible through the GUI or via commands entered at the command line prompt.

1. Create a functional coverage report using the GUI.
 - a. Right-click in the Analysis pane and select **Reports**. This opens the Functional coverage report dialog (Figure 14-32).

Figure 14-32. Functional Coverage Report Dialog



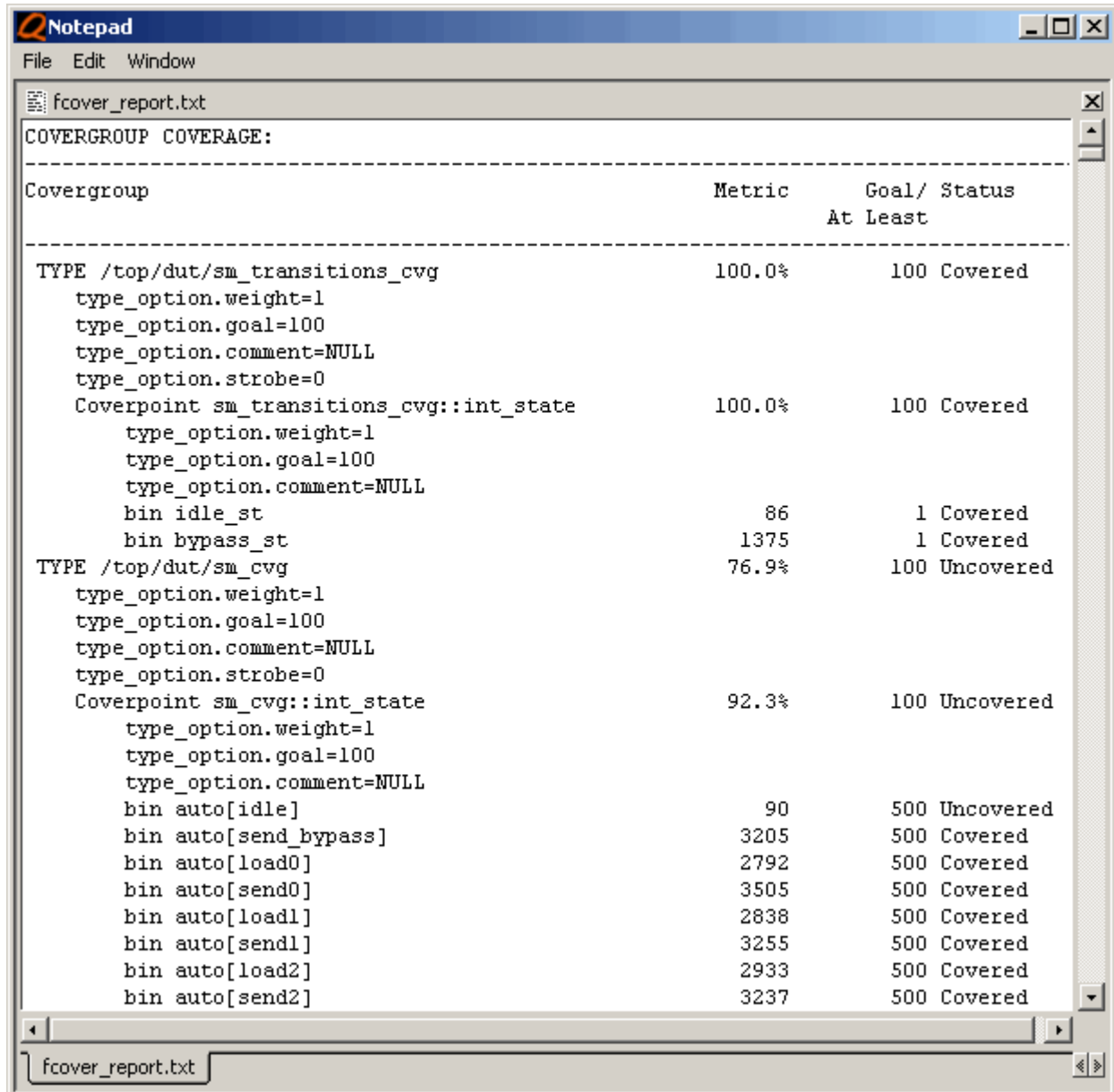
- b. With "All coverage items" selected, select **Covergroups only**.
 - c. Select **Include covergroup options**.
 - d. Select **OK** to write the report to the file *fcover_report.txt*.

The actions taken in the GUI are echoed in the transcript as follows:

```
fcover report -cvg -comments -option -file fcover_report.txt -r *
```

The report will appear automatically in QuestaSim Notepad as shown in (Figure 14-33).

Figure 14-33. The Functional Coverage Report



Covergroup	Metric	Goal/ Status At Least
TYPE /top/dut/sm_transitions_cvg	100.0%	100 Covered
type_option.weight=1		
type_option.goal=100		
type_option.comment=NULL		
type_option.strobe=0		
Coverpoint sm_transitions_cvg::int_state	100.0%	100 Covered
type_option.weight=1		
type_option.goal=100		
type_option.comment=NULL		
bin idle_st	86	1 Covered
bin bypass_st	1375	1 Covered
TYPE /top/dut/sm_cvg	76.9%	100 Uncovered
type_option.weight=1		
type_option.goal=100		
type_option.comment=NULL		
type_option.strobe=0		
Coverpoint sm_cvg::int_state	92.3%	100 Uncovered
type_option.weight=1		
type_option.goal=100		
type_option.comment=NULL		
bin auto[idle]	90	500 Uncovered
bin auto[send_bypass]	3205	500 Covered
bin auto[load0]	2792	500 Covered
bin auto[send0]	3505	500 Covered
bin auto[load1]	2838	500 Covered
bin auto[send1]	3255	500 Covered
bin auto[load2]	2933	500 Covered
bin auto[send2]	3237	500 Covered

You can also create textual, html, and exclusion coverage reports using the **Tools > Coverage Report** menu selection.

Lesson Wrap-Up

This concludes this lesson.

1. Select **File > Quit** to close QuestaSim.

Chapter 15

Using the SystemVerilog DPI

Introduction

This lesson is designed to walk you through the basics of using the SystemVerilog Direct Programming Interface (DPI) with QuestaSim. After completing this lesson, you should have a good understanding of the interface's intentions.

We will start with a small design that shows how simulation control flows back and forth across the boundary between Verilog simulation and code written in a foreign language. In this tutorial we will use code written in C, which is the foreign language most commonly used to interface with Verilog simulations.

The design mimics a traffic intersection. We will bring up the design in QuestaSim and monitor the waveform of a signal that represents a traffic light. We will run the simulation and watch how the light changes color as we call functions written in both Verilog and C, freely moving back and forth between the two languages.

This lesson is designed to work with the QuestaSim 6.1 release and newer.

Design Files for this Lesson

The QuestaSim installation comes with the design files you need, located in the following directory:

`<install_dir>/examples/tutorials/systemverilog/dpi_basic`

Start by creating a new directory for this exercise (in case other users will be working with these lessons) and copy all files from the above directory into it.

Related Reading

User's Manual Appendix: [Verilog Interfaces to C](#)

User's Manual Chapter: [Verification with Functional Coverage](#)

Examine the Source Files

Before getting started, take a look at the main design source files in order to get acquainted with the simulation flow and some of the basic requirements for DPI.

1. Open the code for module `test.sv` in a text editor. It should look like the code in [Figure 15-1](#).

Figure 15-1. Source Code for Module *test*.sv

```
1 module test ();
2
3 typedef enum {RED, GREEN, YELLOW} traffic_signal;
4
5 traffic_signal light;
6
7 function void sv_GreenLight ();
8 begin
9     light = GREEN;
10 end
11 endfunction
12
13 function void sv_YellowLight ();
14 begin
15     light = YELLOW;
16 end
17 endfunction
18
19 function void sv_RedLight ();
20 begin
21     light = RED;
22 end
23 endfunction
24
25 task sv_WaitForRed ();
26 begin
27     #10;
28 end
29 endtask
30
31 export "DPI-C" function sv_YellowLight;
32 export "DPI-C" function sv_RedLight;
33 export "DPI-C" task sv_WaitForRed;
34
35 import "DPI-C" context task c_CarWaiting ();
36
37 initial
38 begin
39     #10 sv_GreenLight;
40     #10 c_CarWaiting;
41     #10 sv_GreenLight;
42 end
43
44 endmodule
45
```

Line 1 – We have just one top-level module called *test* in which all the simulation activity will occur.

Line 3 – We declare a new data type called *traffic_signal*, which will contain the data values RED, GREEN, and YELLOW.

Line 5 – We declare an object of this new *traffic_signal* type and give it the name *light*.

Lines 7-11 – We define a Verilog function called *sv_GreenLight* which has no return value. It simply sets the light to a value of GREEN. Note also that we give the function name a prefix of *sv_* in order to distinguish between tasks/functions defined in SystemVerilog and functions defined in C.

Lines 13-17 – We define another function called *sv_YellowLight*, which changes the light to YELLOW.

Lines 19-23 – We define another function called *sv_RedLight*, which changes the light to RED.

Lines 25-29 – The Verilog task *sv_WaitForRed* simply delays for 10 time units (ns by default). Why do we define a task rather than a function? This will become apparent as we go through the actual simulation steps coming up.

Lines 31-33 – These lines do not look like typical Verilog code. They start with the keyword "export", followed by some additional information. These statements are export declarations – the basic mechanism for informing the Verilog compiler that something needs to be handled in a special way. In the case of DPI, special handling means that the specified task or function will be made visible to a foreign language and that its name must be placed in a special name space.

The syntax for these declarations is defined in the SystemVerilog LRM. There is a simple rule to remember regarding how they work:

When running a SystemVerilog simulation and using DPI in order to utilize foreign (C) code, the Verilog code should be thought of as the center of the universe (i.e. everything revolves around the Verilog code). When you wish to make something in Verilog visible to the foreign world, you need to export it to that world. Similarly, if there is something from that foreign world that you want your Verilog code to see and have access to, you need to import it to Verilog.

So in these lines, we export two of the functions and the task that we've just defined to the foreign world (*sv_YellowLight*, *sv_RedLight*, and *sv_WaitForRed*). But why don't we export the *sv_GreenLight* function? You'll see in a moment.

Line 35 – The import declaration is used to import code from the foreign (C) world into the Verilog world. The additional information needed with an import declaration includes:

- how you want this foreign code to be seen by Verilog (i.e. should it be considered a task or a function), and
- the name of the task or function.

In this case, we will import a task named *c_CarWaiting* from the C world (note the *c_* prefix so that we can keep track of where these tasks/functions originated). This is an important concept to remember. If you try to call a foreign task/function but forget to include an import declaration for it, you will get an error when you load simulation stating that you have an unresolved reference to that task/function.

Lines 37-42 – We use a little initial block that executes the simulation and walks us through the light changing scenario. The light starts out RED by default, since that is the first (left-most) value in the light's type definition (i.e. the *traffic_signal* type). When simulation starts, we wait for 10 time units and then change the light to GREEN via the *sv_GreenLight* function. All this occurs in the Verilog world, so there is no need to export the *sv_GreenLight* function. We won't be doing anything with it over in the foreign world.

Next, we wait for 10 time units again and then do something called *c_CarWaiting*. From our previous discussion of the import declaration, we know this is a C function that will be imported as a Verilog task. So when we call this task, we are actually stepping over into the foreign world and should be examining some C code. In fact, let's take a look at the other source file for this lesson to see what happens when this line executes during simulation.

2. Open the *foreign.c* source file in a text editor. It should look like the code in [Figure 15-2](#).

Figure 15-2. Source Code for the *foreign.c* File - DPI Lab

```
1 int c_CarWaiting()
2 {
3     printf("There's a car waiting on the other side. \n");
4     printf("Initiate change sequence ... \n");
5     sv_YellowLight();
6     sv_WaitForRed();
7     sv_RedLight();
8     return 0;
9 }
10
```

Line 1 – This is the function definition for *c_CarWaiting*. It is an *int* type function and returns a 0.

Lines 3-4 – The statement inside the function prints out a message indicating that a car is waiting on the other side of the intersection and that we should initiate a light change sequence.

Line 5 – We call the SystemVerilog function *sv_YellowLight*. Even though we are in the foreign (C) world now, executing C functions/statements until this function exits and returns control back over to Verilog, we can indeed call the Verilog world and execute tasks/functions from there. The reason the C code knows that *sv_YellowLight* exists is because we've exported it back in our Verilog code with the **export** declaration.

To follow along with the simulation, look at the *sv_YellowLight* function in lines 13 through 17 in the *test.sv* file ([Figure 15-3](#)). Here, we change the light to a value of YELLOW, then pass control back to *foreign.c* and go to the line following the *sv_YellowLight* function call.

Figure 15-3. The *sv_YellowLight* Function in the *test.sv* File

```

13 function void sv_YellowLight ();
14 begin
15     light = YELLOW;
16 end
17 endfunction

```

Line 6 – Now we call the *sv_WaitForRed* SystemVerilog task, defined on lines 25-29 of *test.sv* (Figure 15-4).

Figure 15-4. The *sv_WaitForRed* Task in the *test.sv* File

```

25 task sv_WaitForRed ();
26 begin
27     #10;
28 end
29 endtask

```

The task designates a wait for 10 time units. Since there is time delay associated with this procedure, it has to be a task. All the rules associated with tasks and functions in basic Verilog will also apply if you call them from the foreign world. Since we compile the two source files independently (one with a Verilog compiler and one with a C compiler), the rules of one language will not be known to the compiler for the other. We will not find out about issues like this in many cases until we simulate and hook everything together. Be aware of this when deciding how to import/export things.

An important thing to note here is that we made this call to the SystemVerilog *sv_WaitForRed()* task from the foreign (C) world. If we want to consume simulation time, C doesn't know anything about the SystemVerilog design or simulation time units. So we would need to make calls back over to Verilog in order to perform such operations. Again, just remember which world you are in as you move around in simulation.

Anyway, *sv_WaitForRed* just burns 10 time units of simulation and then returns control back over to C. So we go back over to *foreign.c* and proceed to the next line.

Line 7 – Here we call the *sv_RedLight* SystemVerilog function, which changes the light to RED. If you look up that function in *test.sv*, that is exactly what occurs (Figure 15-5).

Figure 15-5. The *sv_RedLight* Function in the *test.sv* File

```

19 function void sv_RedLight ();
20 begin
21     light = RED;
22 end
23 endfunction

```

This is the last statement in the *c_CarWaiting* function in *foreign.c*. So now this function exits and returns control back over to Verilog.

The simulator returns to line 40 in *test.sv*, which called this C function in the first place. There is nothing else to be done on this line. So we drop down to the next line of

execution in the simulation. We wait for 10 time units and then call the *sv_GreenLight* function (Figure 15-6). If you recall, this function just keeps execution in the Verilog world and changes the light back to GREEN. Then we're all done with simulation.

Figure 15-6. Function Calls in the *test.sv* File

```
37 initial
38 begin
39     #10 sv_GreenLight;
40     #10 c_CarWaiting;
41     #10 sv_GreenLight;
42 end
```

Compile and Load the Simulation

Create a new directory and copy into it all files from:

<install_dir>/questasim/examples/tutorials/systemverilog/dpi_basic

Change directory to this new directory and make sure your QuestaSim environment is set up properly.

UNIX and Linux: Use the **make** utility to compile and load the design into the simulator.

Windows: Double-click the *windows.bat* file.

Note



For Windows users, if you do not have the gcc-3.3.1-mingw32 compiler installed, download it from SupportNet (<http://supportnet.mentor.com/>) and unzip it into the QuestaSim install tree. In addition, make sure it is in your Path environment variable.

Explore the Makefile

A *Makefile* has been included with this lesson to help UNIX and Linux users compile and simulate the design (Figure 15-7), or you can run "make all" to kick off the whole thing all at once. There is also a clean target to help you clean up the directory should you want to start over and run again.

Figure 15-7. Makefile for Compiling and Running on UNIX or Linux Platforms

```
1 worklib:
2     vlib work
3
4 compile: test.sv
5     vlog test.sv -dpiheader dpi_types.h
6
7 foreign: foreign.c
```

```

8     gcc -I$(MTI_HOME)/include -shared -g -o foreign.so foreign.c
9
10    optimize:
11        vopt +acc test -o opt_test
12
13    foreign_windows: foreign.c
14        vsim -c opt_test -dpiexportobj exports
15        gcc -I$(MTI_HOME)/include -shared -g -o foreign.dll foreign.c
16        exports.obj -lmtipli -L$(MTI_HOME)/win32
17
17    sim:
18        vsim opt_test -sv_lib foreign
19
20    all:
21        worklib compile foreign optimize sim
22
23    all_windows:
24        worklib compile optimize foreign_windows sim
25
26    clean:
27        rm -rf work transcript vsim.wlf foreign.so foreign.dll exports.obj
28

```

The five targets in the *Makefile* are:

Line 1 – The **vlib** command creates the *work* library where everything will be compiled to.

Lines 4-5 – The **vlog** command invokes the vlog compiler on the *test.sv* source file.

Lines 7-8 – The **gcc** command invokes the gcc C compiler on the *foreign.c* source file and creates a shared object (*foreign.so*) that will be loaded during simulation. Note that this command assumes that you have the *MTI_HOME* environment variable set to the QuestaSim installation directory.

Lines 10-11 – The **vopt** command initiates optimization of the design. The **+acc** option provides full visibility into the design for debugging purposes. The **-o** option is required for naming the optimized design object (in this case, *opt_test*).

Lines 16-17 – The **vsim** command invokes the simulator using the *opt_test* optimized design object. The **-sv_lib** option specifies the shared object to be loaded during simulation. Without this option, the simulator will not be able to find any imported (C) functions you've defined.

Explore the *windows.bat* File

A *windows.bat* file has been included for Windows users ([Figure 15-8](#)).

Figure 15-8. The windows.bat File for Compiling and Running in Windows - DPI Lab

```
1 vlib work
2
3 vlog test.sv -dpiheader dpi_types.h
4
5 vopt +acc test -o opt_test
6
7 vsim -c test -dpiexportobj exports
8
9 gcc -I %MTI_HOME%\include -shared -g -o foreign.dll foreign.c
  exports.obj -lmtipli -L %MTI_HOME%\win32
10
11 vsim -i opt_test test -sv_lib foreign -do "add wave light; view source"
12
```

The *windows.bat* file compiles and runs the simulation as follows:

Line 1 – The **vlib** command creates the *work* library where everything will be compiled to.

Line 3 – The **vlog** command invokes the vlog compiler on the *test.sv* source file.

Line 5 – The **vopt** command initiates optimization of the design. The **+acc** option provides full visibility into the design for debugging purposes. The **-o** option is required for naming the optimized design object (in this case, *opt_test*).

Line 7 – The first **vsim** command creates an object called *exports* which is used by the gcc command.

Line 9 – The **gcc** command compiles and links together the *foreign.c* source file and the *exports.obj* file created with the previous command. The **-o** option creates an output library called *foreign.dll*.

Note

This command assumes that you have the MTI_HOME environment variable set to the QuestaSim installation directory.

Line 11 – The second **vsim** command invokes the simulator using the *opt_test* optimized design object. The **-sv_lib** option tells the simulator to look in the *foreign.dll* library for C design objects that can be used in the SystemVerilog simulation. The **-do "add wave light; view source"** option adds the *light* signal to the Wave window and opens the Source window for viewing.

Run the Simulation

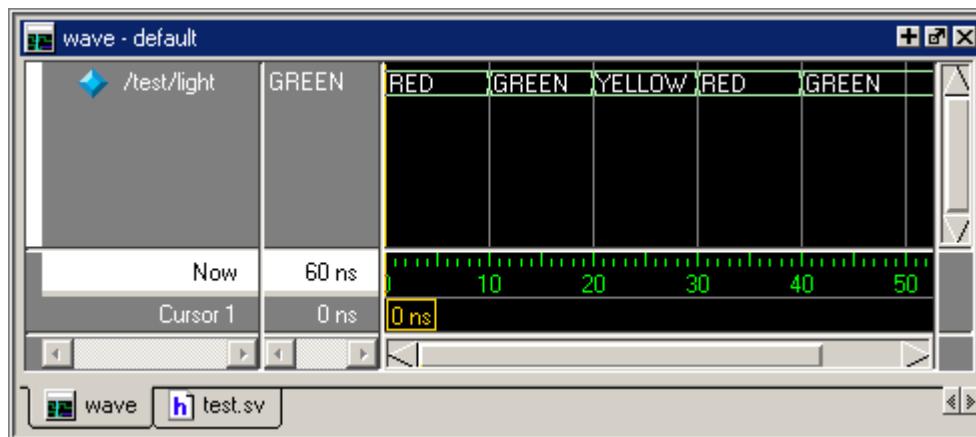
Once in simulation, you can step through the code or simply run the simulation in 10 ns increments to observe changes in the *light* signal's waveform. If you look in the Objects pane in the QuestaSim graphic interface ([Figure 15-9](#)), you should see the "light" object with its initial value of RED. If the Objects window is not open, select **View > Objects** from the Main menus to open it.


Figure 15-9. The *light* Signal in the Objects Pane

UNIX and Linux: Drag and drop that object into a Wave window.

Windows: The light object has already been placed in the Wave window.

1. Run the simulation for 10 ns.
 - a. Enter **run 10 ns** at the command line. You'll see *light* turn "GREEN" in the Objects and Wave windows.
 - b. Repeat several times and watch the Wave window as it changes values at the appropriate simulation times (Figure 15-10).

Figure 15-10. The *light* Signal in the Wave Window

2. Restart the simulation.
 - a. Click the Restart icon. 
 - b. In the Restart dialog, click the **Restart** button.
3. Run the simulation for 10 ns.
 - a. Enter **run 10 ns** at the command line.
4. View the *test.sv* code in the Source window.
 - a. Select the **test.sv** tab.

5. Step through the code.


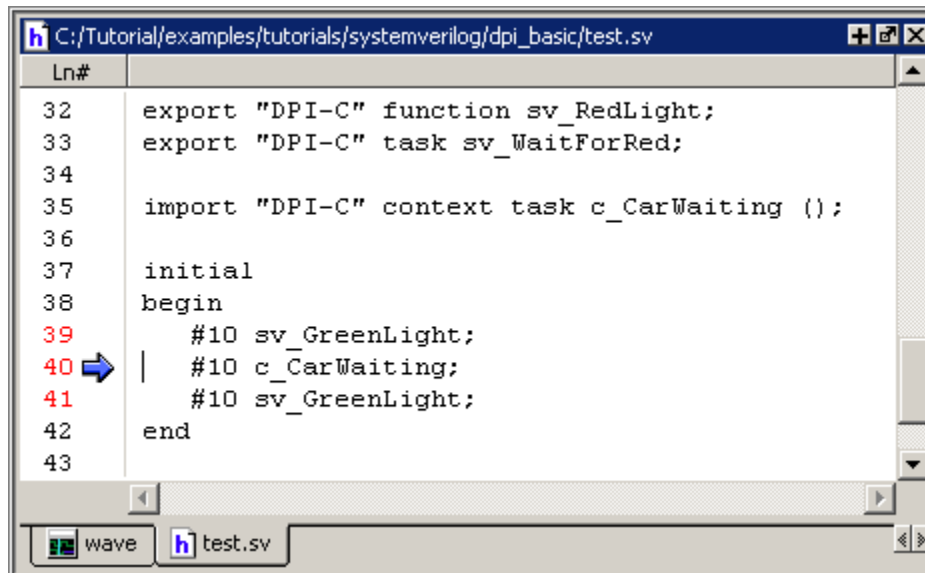
- a. Click the Step icon  and watch the blue arrow in the Source window move through the code for *test.sv* (Figure 15-11) and *foreign.c*. This allows you to keep track of where you are in the source files as you step through the simulation. Feel free to experiment and try adding your own functions, tasks, statements, etc.

Figure 15-11. Source Code for *test.sv*

Lesson Wrap-Up

This concludes this lesson on the basics of how DPI works in QuestaSim. You should feel comfortable with these elements before moving on to the next tutorial. This design only accomplishes some simple function calls to change the values of the signal light in order to stress how easy it is to step back and forth between Verilog and a foreign language like C. However, we have not done anything terribly interesting in regard to passing data from one language to the other. Is this possible? Most definitely. In fact, the next lesson will address this subject.

1. Select **Simulate > End Simulation**. Click Yes.

Chapter 16

Using SystemVerilog DPI for Data Passing

Introduction

This lesson is designed to build on your understanding of the Direct Programming Interface (DPI) for SystemVerilog. In the previous lesson, you were shown the basic elements of the interface and how to make simple function calls to/from Verilog and C. However, no data was passed across the boundary, which is a very important topic to understand. This lesson will focus on that aspect of the interface.

Although DPI allows Verilog to interface with any foreign language, we will concentrate on the C language in this lesson.

Mapping Verilog and C

Whenever we want to send the value of an object from Verilog to C, or vice versa, that value will have a dual personality. It may have been initialized as a bit or a reg over in the Verilog world, for example, and then passed over to C via an imported function call. The C world, however, does not have regs, bits, logic vectors, etc. How is this going to work?

What you need in this situation is a table that maps Verilog types to C types. Fortunately, much of the type definition that went into Verilog-2001 and SystemVerilog was done with the intention of matching C data types, so much of this mapping is pretty straightforward. However, some of the mapping is a little more complex and you will need to be aware of how an object in Verilog will map to its C counterpart.

Do you have to define this mapping? No. The SystemVerilog language defines it for you, and the simulator is set up to handle all of these dual personality issues itself. For example, in Verilog, an *int* is a 2-state, signed integer that is stored in 32 bits of memory (on the system; it's not an array or a vector). The fact that a Verilog *int* is a 2-state type is important in that it only allows 0 and 1 values to be assigned to its bits. In other words, no X or Z values are allowed (they are just converted to 0 if you try to assign them).

This is straightforward and it appears to behave just like a C *int* would, so the mapping is easy: a Verilog *int* will map to a C *int* as it crosses the boundary.

Design Files for This Lesson

The design files for this lesson are located in the following directory:

`<install_dir>/examples/tutorials/systemverilog/data_passing`

Start by creating a new directory for this exercise (in case other users will be working with these lessons) and copy all files from the above directory into it.

Related Reading

User's Manual Appendix: [Verilog Interfaces to C](#)

User's Manual Chapter: [Verification with Functional Coverage](#)

Examine the Source Files

Before getting started, let's look at the *foreign.c* file which contains the definitions for the two C functions we'll be using to read our data values coming over from the Verilog world and print messages to let us know what is going on.

1. Open the code for the *foreign.c* file in a text editor. It should look like the code in [Figure 16-1](#).

Figure 16-1. Source Code for the *foreign.c* File - Data Passing Lab

```
1 #include "dpi_types.h"
2
3 void print_int(int int_in)
4 {
5     printf("Just received a value of %d.\n", int_in);
6 }
7
8 void print_logic(svLogic logic_in)
9 {
10     switch (logic_in)
11     {
12         case sv_0: printf ("Just received a value of logic 0.\n");
13             break;
14         case sv_1: printf ("Just received a value of logic 1.\n");
15             break;
16         case sv_z: printf ("Just received a value of logic Z.\n");
17             break;
18         case sv_x: printf ("Just received a value of logic X.\n");
19             break;
20     }
21 }
22
```

Line 1 – We include a header file called *dpi_types.h* which will help us with type conversions – more to come on that a bit later.

Line 3 – This is the definition for a function called *print_int*, which simply takes an integer argument and prints its values.

Line 8 – This is the definition for a function called *print_logic* which takes an argument of type **svLogic** and then checks to see what value it is and prints a message accordingly.

2. Now let's look at the SystemVerilog source code. Open the *test.sv* source file in a text editor. It should look like the code in [Figure 16-2](#).

Figure 16-2. Source Code for the *test.sv* Module

```

1 module test ();
2
3 import "DPI-C" context function void print_int (input int int_in);
4 import "DPI-C" context function void print_logic (input logic logic_in );
5
6 int int_var;
7 bit bit_var;
8 logic logic_var;
9
10 initial
11 begin
12     print_int(int_var);
13     int_var = 1
14     print_int(int_var);
15     int_var = -12;
16     print_int(int_var);
17     print_int(bit_var);
18     bit_var = 1'b1;
19     print_int(bit_var);
20     bit_var = 1'bx;
21     print_int(bit_var);
22     logic_var = 1'b1;
23     print_int(logic_var);
24     logic_var = 1'bx;
25     print_int(logic_var);
26     print_logic(logic_var);
27     logic_var = 1'bz;
28     print_logic(logic_var);
29     logic_var = 1'b0;
30     print_logic(logic_var);
31 end
32
33 endmodule
34

```

Lines 3-4 – These lines don't look like typical Verilog code. They start with the **import** keyword and are followed by additional information. These statements are referred to as import declarations. An import declaration is a mechanism used to inform the Verilog compiler that something needs to be handled in a special way. In the case of DPI, the special handling means that the specified task or function will be made visible to SystemVerilog from a foreign language and that its name will need to be placed in a special name space.

The syntax for these declarations is defined in the SystemVerilog LRM. There is a simple rule to remember regarding how they work: When running a SystemVerilog simulation, and using DPI in order to utilize foreign (C) code, the Verilog code should be thought of as the center of the universe (i.e. everything revolves around the Verilog code).

If there is something from that foreign world that you want your Verilog code to see and have access to, you need to "import" it to Verilog. Similarly, when you wish to make

something in Verilog visible to the foreign world, you need to "export" it to that world (see the previous lesson). So in these lines, we import the two functions that we've just defined over in the foreign world (*print_int* & *print_logic*).

Lines 6-8 – Here, we declare three variables that will be used as arguments in the two functions. Note how they are defined as three different SystemVerilog types: *int*, *bit*, and *logic*.

Lines 10-31 – This initial block simply calls each function and sets values for each variable in a sequence that will be discussed when we run the design.

Compile and Load the Simulation

Create a new directory and copy into it all files from:

`<install_dir>/questasim/examples/tutorials/systemverilog/data_passing`

Change directory to this new directory and make sure your QuestaSim environment is set up properly.

UNIX and Linux: Use the **make** utility to compile and load the design into the simulator.

Windows: Double-click the *windows.bat* file.

Note



For Windows users, if you do not have the gcc-3.3.1-mingw32 compiler installed, download it from SupportNet (<http://supportnet.mentor.com/>) and unzip it into the QuestaSim install tree. In addition, make sure it is in your Path environment variable.

Explore the Makefile

A *Makefile* has been included with this lesson to help UNIX and Linux users compile and simulate the design (Figure 16-3), or you can run "make all" to kick off the whole thing all at once. There is also a clean target to help you clean up the directory should you want to start over and run again.

Figure 16-3. Makefile for Compiling and Running on UNIX and Linux Platforms

```
1 worklib:
2     vlib work
3
4 compile: test.sv
5     vlog test.sv -dpiheader dpi_types.h
6
7 foreign: foreign.c
8     gcc -I$(MTI_HOME)/include -shared -g -o foreign.so foreign.c
9
10 optimize:
11     vopt +acc test -o opt_test
12
13 sim:
14     vsim opt_test test -sv_lib foreign
15
16 all:
17     worklib compile foreign optimize sim
18
19 clean:
20     rm -rf work transcript vsim.wlf foreign.so dpi_types.h
21
```

The five targets in the *Makefile* are:

Line 1 – worklib: The **vlib** command creates the *work* library where everything will be compiled to.

Lines 4-5 – compile: The **vlog** command invokes the vlog compiler on the *test.sv* source file.

Lines 7-8 – foreign: The **gcc** command invokes the gcc C compiler on the *foreign.c* source file and creates a shared object (*foreign.so*) that will be loaded during simulation. Note that this command assumes that you have the MTI_HOME environment variable set to the QuestaSim installation directory.

Lines 10-11 – optimize: The **vopt** command initiates optimization of the design. The **+acc** option provides full visibility into the design for debugging purposes. The **-o** option is required for naming the optimized design object (in this case, *opt_test*).

Lines 13-14 – sim: The **vsim** command invokes the simulator using the optimized design object *opt_test*. The **-sv_lib** option specifies the shared object to be loaded during simulation. Without this option, the simulator will not be able to find any imported (C) functions you've defined.

Explore the *windows.bat* File

A *windows.bat* file has been included for Windows users ([Figure 16-4](#)).

Figure 16-4. The *windows.bat* File for Compiling and Running in Windows - Data Passing Lab

```
1 vlib work
2
3 vlog test.sv -dpiheader dpi_types.h
4
5 vopt +acc test -o opt_test
6
7 gcc -I %MTI_HOME%\include -shared -g -o foreign.dll foreign.c -lmtipli -L
%MTI_HOME%\win32
8
9 vsim -i opt_test -sv_lib foreign -do "view source"
10
```

The *windows.bat* file compiles and runs the simulation as follows:

Line 1 – The **vlib** command creates the *work* library where everything will be compiled to.

Line 3 – The **vlog** command invokes the vlog compiler on the *test.sv* source file.

Line 5 – The **vopt** command initiates optimization of the design. The **+acc** option provides full visibility into the design for debugging purposes. The **-o** option is required for naming the optimized design object (in this case, *opt_test*).

Line 7– The **gcc** command compiles the *foreign.c* source file. The **-I** option is used to specify a directory to search for include files. The **-shared** option tells **gcc** to create a shared library as the output (i.e. compile AND link). The **-g** option adds debugging code to the output. The **-o** option creates an output library called *foreign.dll*. The **-lmtipli** option is used to specify a compiled library that is to be included when trying to resolve all the functions used in the C/C++ code being compiled. The **-L** option specifies a directory to search for libraries specified in the **-l** option.

Note

The **gcc** command assumes that you have the MTI_HOME environment variable set to the QuestaSim installation directory.

Line 9– The **vsim** command invokes the simulator using the *opt_test* optimized design object. The **-sv_lib** option tells the simulator to look in the *foreign.dll* library for C design objects that can be used in the SystemVerilog simulation. The **-do "view source"** option opens the Source window and displays the *test.sv* source code.

Run the Simulation

Once in simulation with the *test.sv* module loaded, you can use the Step Over command button to advance through the simulation. This will simply set values of different types of Verilog objects and send the data over to C for print out to the screen.



1. (For UNIX and Linux) Right-click the *test* instance in the Workspace window and select View Declaration from the popup menu that appears. This will open a Source window and display the *test.sv* source code.

- Click the Step Over button. With this first step you should be on line #12 in *test.sv* (indicated by the blue arrow in the Source window - see [Figure 16-5](#)) where we print out the value of *int_var* – which is defined as an *int* on line #6.

Figure 16-5. Line 12 of *test.sv* in the Source Window

```

6    int int_var;
7    bit bit_var;
8    logic logic_var;
9
10   initial
11   begin
12   ➡    print_int(int_var);
13       int_var = 1;
14       print_int(int_var);

```

Nothing has been assigned to *int_var* yet, so it should have its default initial value of 0. If you look in the Objects window, you should see that *int_var* is indeed equal to 0 ([Figure 16-6](#)).

Figure 16-6. The Value of *int_var* is Currently 0



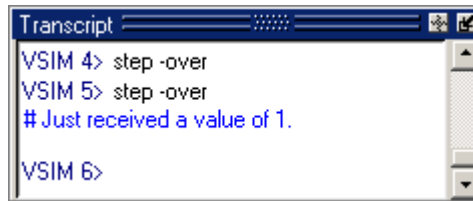
Name	Value	Kind	Mode
int_var	0	Int	Internal
bit_var	0	Bit	Internal
logic_var	x	Register	Internal

- Click the Step Over button again. This will call the imported C function *print_int* with *int_var* as its input parameter. If you look in the Transcript window after this line executes, you should see the following message:

```
Just received a value of 0.
```

That's what we expect to happen. So far, so good.

- Next we set *int_var* to a value of 1. Click the Step Over button and you will see the value of *int_var* change to 1 in the Objects window.
- Now do another Step Over and you should see a 1 being printed in the Transcript window this time ([Figure 16-7](#)).

Figure 16-7. The Value of *int_var* Printed to the Transcript Window

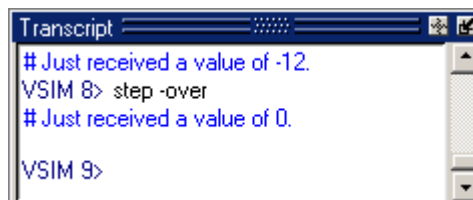
6. With the next two steps (click Step Over twice), we change *int_var* to -12 and print again. You should get the idea now. Both positive and negative integers are getting set and printed properly.

Next we are going to use the *print_int* function to print the value of *bit_var*, which is defined as a *bit* type on line #7. It also has a default initial value of 0, so you can guess what will be printed out.

7. Click Step Over again and verify the results in the Objects window (Figure 16-8) and in the Transcript window (Figure 16-9).

Figure 16-8. The Value of *bit_var* is 0.

Name	Value	Kind	Mode
int_var	-12	Int	Internal
bit_var	0	Bit	Internal
logic_var	x	Register	Internal

Figure 16-9. Transcript Shows the Value Returned for *bit_var*

8. Click Step Over twice to set *bit_var* to a 1 and print to the transcript.
9. Click Step Over to set *bit_var* to X.

Look in the Objects window. The variable didn't go to X. It went to 0. Why?

Remember that the bit type is a 2-state type. If you try to assign an X or a Z, it gets converted to 0. So we get a 0 instead, and that's what should get printed.

10. Click Step Over for the *print_int* function and verify that a value of 0 is printed.

Now let's try some 4-state values. You should be on line #22 now where *logic_var* is a 4-state "logic" type being assigned a 1.

11. Click Step Over to go to line #23. You should see the value of *logic_var* change from X to 1 in the Objects window.
12. Click Step Over to call *print_int* again and print the value of *logic_var* to the transcript.
13. Click Step Over to set *logic_var* to X.
14. Click Step Over to print *logic_var*. You should be on line #26 now. Look at the transcript and you will see that a value of 0 is printed instead of X. Why? Let's look into the source code to see what happened.

Look at the *foreign.c* file in [Figure 16-1](#), which is the C source for our imported functions. In line 3, the *print_int* function is expecting an integer (*int*) as its input. That works fine when we were sending integers. But now we are working with 4-state data types, which allow X and Z values. How is that kind of data going to cross over the boundary, and what is it going to look like when it gets over to C? What about user defined types and the many other types of data we can send back and forth? How are you supposed to know how to write your C functions to accept that kind of data properly and/or send it back to Verilog properly?

Fortunately, the answer to all these questions is that you don't really have to know the fine details. The SystemVerilog language defines this data mapping for you. Furthermore, QuestaSim will create a C header file for you during compilation that you can reference in your C code. All the function prototypes, data type definitions, and other important pieces of information are made available to you via this header file.

If you look at the **compile** target in the Makefile ([Figure 16-3](#)) you will see an option in the **vlog** command called **-dpiheader** with an output file name as its argument. As **vlog** compiles your Verilog source file, it analyzes any DPI import/export statements and creates a C header file with what it knows to be the correct way to define the prototypes for your imported/exported functions/tasks. In this lesson, we call the file *dpi_types.h* ([Figure 16-10](#)).

Figure 16-10. The *dpi_types.h* File

```

1  /* MTI_DPI */
2
3  /*
4   * Copyright 2004 Mentor Graphics Corporation.
5   *
6   * Note:
7   *   This file is automatically generated.
8   *   Please do not edit this file - you will lose your edits.
9   *
10  * Settings when this file was generated:
11  *   PLATFORM = 'win32'
12  *   Info = SE 6.1c 2005.11
13  */
14 #ifndef INCLUDED_DPI_TYPES
15 #define INCLUDED_DPI_TYPES
16
17 #ifdef __cplusplus
18 extern "C" {
19 #endif
20
21 #include "svdpi.h"
22
23 DPI_DLLESPEC
24 void
25 print_int(
26     int int_in);
27
28 DPI_DLLESPEC
29 void
30 print_logic(
31     svLogic logic_in);
32
33 #ifdef __cplusplus
34 } /* extern "C" */
35 #endif
36
37 #endif /* INCLUDED */
38

```

At the top of this file is information for internal DPI purposes. But if you go down to line 25, you'll see a function prototype for the *print_int* function. As expected, the input parameter is an *int* type.

Just below this function is the prototype for the *print_logic* function, which has an input parameter of type "svLogic" (i.e. SystemVerilog Logic). This file includes another header file called *svdpi.h*, which is part of the SystemVerilog language and is shipped in the QuestaSim installation directory (that's why we have "-I\$(MTI_HOME)/include" on the command line for C compilation in the Makefile's "foreign" target – see [Figure 16-3](#)). This svLogic type is basically an unsigned char.

When you put *#include dpi_types.h* in your C source file, all these function prototypes and data types will be made available to you. In fact, we strongly recommend that you use this file when writing the C code that will interface with Verilog via DPI.

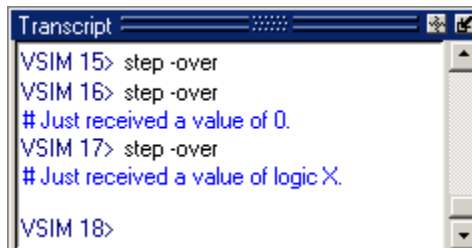
Look back at the *test.sv* file ([Figure 16-2](#)) and look for the DPI import statements. There is one for *print_int* and one for *print_logic*. The **vlog** compiler looks at these statements, sees the names of the functions being imported along with their parameters and return values (in Verilog terms), and then creates the correct DPI header file for you. In the case of the *print_logic* function, it saw that the input parameter was of type "logic". So it

put *logic*'s counterpart of "svLogic" in the header file. Now both elements of the dual personality for this particular object are defined and everything should pass over to C properly.

Let's go back to simulation. We should be on line #26, just after the point where the bad logic value of 0 got printed instead of an X. Now that we know we were using the wrong function for this particular type of data, we will use the *print_logic* function instead.

15. Click Step Over to execute this line. The X value is printed out this time (Figure 16-11). You can take a look at the *foreign.c* file to see how this was accomplished.

Figure 16-11. The Transcript Shows the Correct Value of logic X



Basically, 4-state values are represented as 0, 1, 2, and 3 in their canonical form. The values you see in the switch statement inside the *print_logic* function are *#define*'d in the *svdpi.h* file for you so that you can keep everything straight. Again, if you use the DPI header file in your C code, you can just use this stuff and everything will work properly.

Go ahead and step through a few more statements and you can see that *logic_var* gets set to some other 4-state values and we print them correctly using the *print_logic* function.

Lesson Wrap-Up

There is certainly much more involved with passing data back and forth across the boundary between C and Verilog using DPI. What about user-defined types? What about arrays? Structs? 64-bit integers? This particular subject can get into some pretty hefty detail, and we've already covered quite a bit here. Hopefully, this lesson has helped you understand the most basics of passing data through the interface. Most important of all, it should give you an understanding of how to make use of the DPI header file that **vlog** creates in order to make sure your C code is written properly to interface with SystemVerilog.

1. Select **Simulate > End Simulation**. Click Yes.

Chapter 17

Comparing Waveforms

Introduction

Waveform Compare computes timing differences between test signals and reference signals. The general procedure for comparing waveforms has four main steps:

1. Select the simulations or datasets to compare
2. Specify the signals or regions to compare
3. Run the comparison
4. View the comparison results

In this exercise you will run and save a simulation, edit one of the source files, run the simulation again, and finally compare the two runs.

Note



The functionality described in this tutorial requires a compare license feature in your QuestaSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Design Files for this Lesson

The sample design for this lesson consists of a finite state machine which controls a behavioral memory. The testbench *test_sm* provides stimulus.

The QuestaSim installation comes with Verilog and VHDL versions of this design. The files are located in the following directories:

Verilog – *<install_dir>/examples/tutorials/verilog/compare*

VHDL – *<install_dir>/examples/tutorials/vhdl/compare*

This lesson uses the Verilog version in the examples. If you have a VHDL license, use the VHDL version instead. When necessary, instructions distinguish between the Verilog and VHDL versions of the design.

Related Reading

User's Manual sections: [Waveform Compare](#) and [Recording Simulation Results With Datasets](#).

Creating the Reference Dataset

The reference dataset is the *.wlf* file that the test dataset will be compared against. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

In this exercise you will use a DO file to create the reference dataset.

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise (in case other users will be working with these lessons). Create the directory and copy all files from `<install_dir>/questasim/examples/tutorials/verilog/compare` to the new directory.

If you have a VHDL license, copy the files in `<install_dir>/questasim/examples/tutorials/vhdl/compare` instead.

2. Start QuestaSim and change to the exercise directory.

If you just finished the previous lesson, QuestaSim should already be running. If not, start QuestaSim.

- a. Type **vsim** at a UNIX shell prompt or use the QuestaSim icon in Windows.

If the Welcome to QuestaSim dialog appears, click **Close**.

- b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Execute the lesson DO file.

- a. Type **do gold_sim.do** at the QuestaSim> prompt.

The DO file does the following:

- Creates and maps the work library
- Compiles the Verilog and VHDL files
- Loads the simulator with optimizations turned off (**vsim -novopt**)
- Runs the simulation and saves the results to a dataset named *gold.wlf*
- Quits the simulation

Feel free to open the DO file and look at its contents.

Creating the Test Dataset

The test dataset is the *.wlf* file that will be compared against the reference dataset. Like the reference dataset, the test dataset can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

To simplify matters, you will create the test dataset from the simulation you just ran. However, you will edit the testbench to create differences between the two runs.

Verilog

1. Edit the testbench.
 - a. Select **File > Open** and open *test_sm.v*.
 - b. Scroll to line 122, which looks like this:

```
@ (posedge clk) wt_wd('h10,'haa);
```
 - c. Change the data pattern 'aa' to 'ab':

```
@ (posedge clk) wt_wd('h10,'hab);
```
 - d. Select **File > Save** to save the file.
2. Compile the revised file and rerun the simulation.
 - a. Type **do sec_sim.do** at the QuestaSim> prompt.

The DO file does the following:

 - Re-compiles the testbench
 - Adds waves to the Wave window
 - Loads the simulator with optimizations turned off (**vsim -novopt**)
 - Runs the simulation

VHDL

1. Edit the testbench.
 - a. Select **File > Open** and open *test_sm.vhd*.
 - b. Scroll to line 151, which looks like this:

```
wt_wd ( 16#10#, 16#aa#, clk, into );
```
 - c. Change the data pattern 'aa' to 'ab':

```
wt_wd ( 16#10#, 16#ab#, clk, into );
```
 - d. Select **File > Save** to save the file.
2. Compile the revised file and rerun the simulation.
 - a. Type **do sec_sim.do** at the QuestaSim> prompt.

The DO file does the following:

 - Re-compiles the testbench

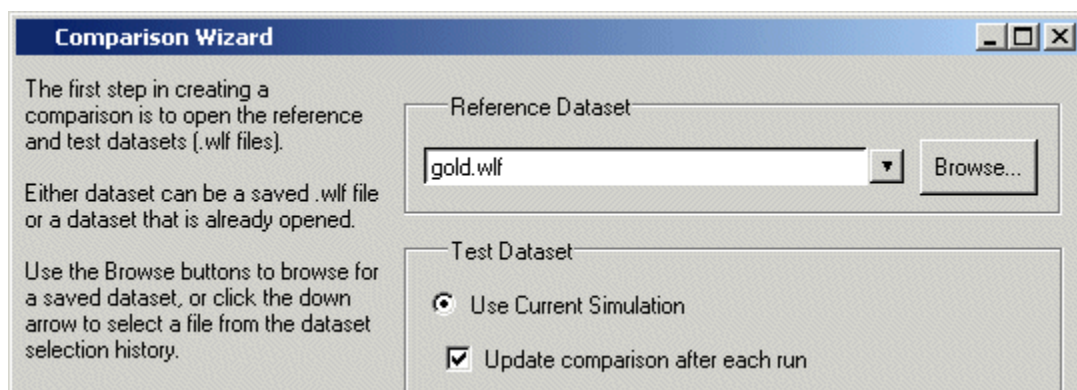
- Adds waves to the Wave window
- Loads the simulator with optimizations turned off (**vsim -novopt**)
- Runs the simulation

Comparing the Simulation Runs

QuestaSim includes a Comparison Wizard that walks you through the process. You can also configure the comparison manually with menu or command line commands.

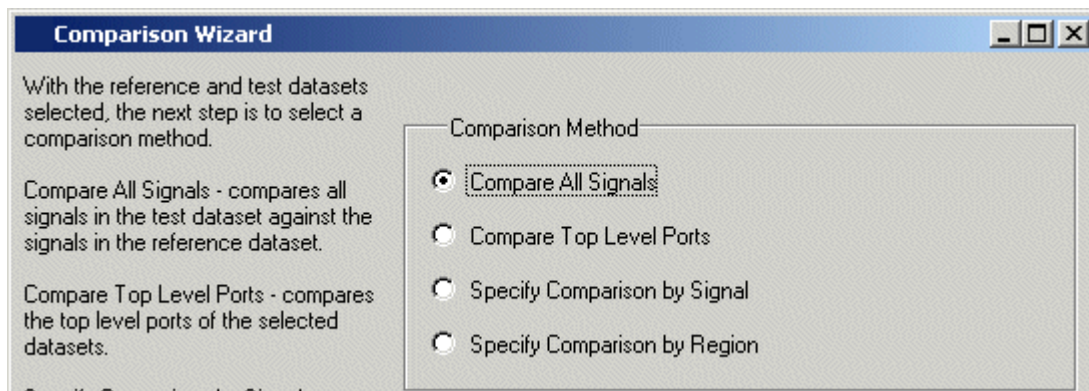
1. Create a comparison using the Comparison Wizard.
 - a. Select **Tools > Waveform Compare > Comparison Wizard**.
 - b. Click the **Browse** button and select *gold.wlf* as the reference dataset (Figure 17-1). Recall that *gold.wlf* is from the first simulation run.

Figure 17-1. First dialog of the Waveform Comparison Wizard



- c. Leaving the test dataset set to **Use Current Simulation**, click **Next**.
- d. Select **Compare All Signals** in the second dialog (Figure 17-2) and click **Next**.

Figure 17-2. Second dialog of the Waveform Comparison Wizard



- e. In the next three dialogs, click **Next**, **Compute Differences Now**, and **Finish**, respectively.

QuestaSim performs the comparison and displays the compared signals in the Wave window.

Viewing Comparison Data

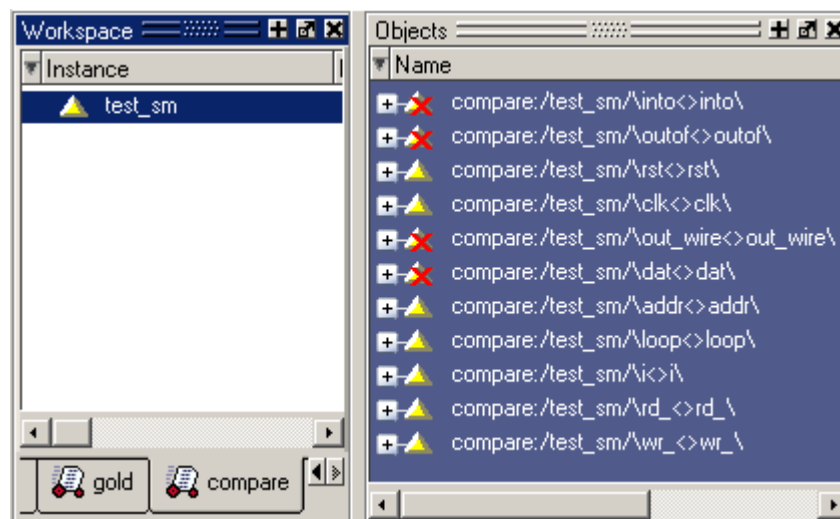
Comparison data is displayed in the Workspace, Transcript, Objects, Wave and List window panes. Compare objects are denoted by a yellow triangle.

The Compare tab in the Workspace pane shows the region that was compared;

The Transcript pane shows the number of differences found between the reference and test datasets;

The Objects pane shows comparison differences when you select the comparison object in the Compare tab of the Workspace (Figure 17-3).

Figure 17-3. Comparison information in the Workspace and Objects panes

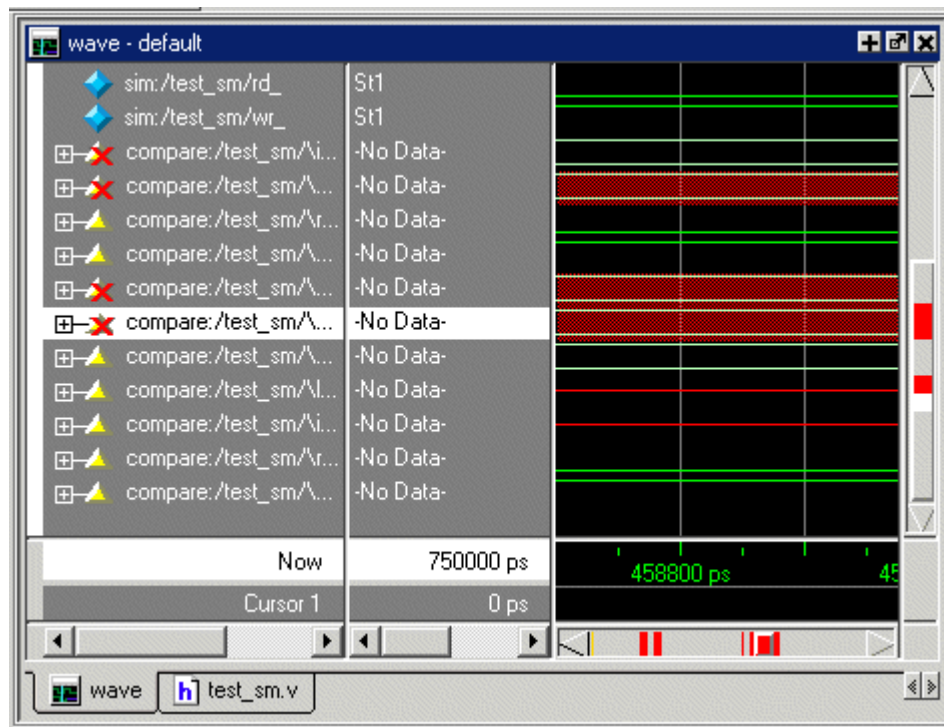


Comparison Data in the Wave Window

The Wave window displays comparison information as follows:

- timing differences are denoted by a red X's in the pathnames column (Figure 17-4),

Figure 17-4. Comparison objects in the Wave window



- red areas in the waveform view show the location of the timing differences,
- red lines in the scrollbars also show the location of timing differences,
- and, annotated differences are highlighted in blue.

The Wave window includes six compare icons that let you quickly jump between differences ([Figure 17-5](#)).

Figure 17-5. The compare icons



From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.

The compare icons cycle through differences on all signals. To view differences in only a selected signal, use <tab> and <shift> - <tab>.

Comparison Data in the List Window

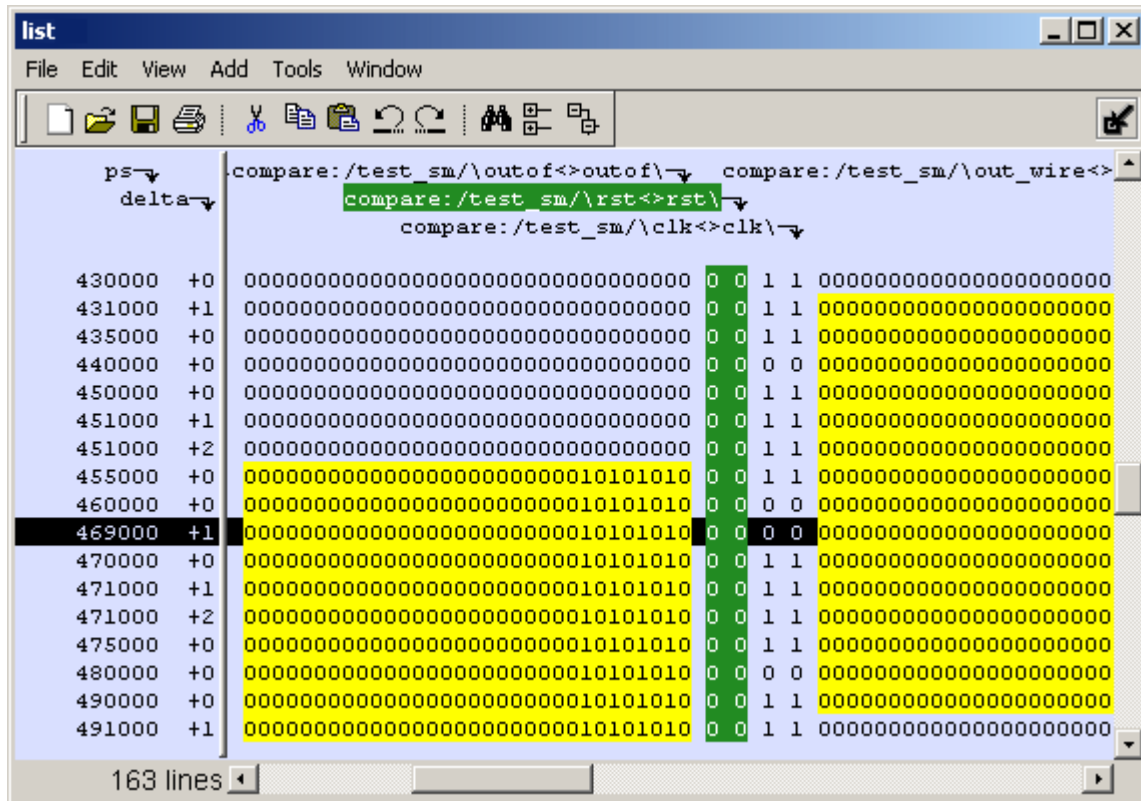
You can also view the results of your waveform comparison in the List window.

1. Add comparison data to the List window.

- a. Select **View > List** from the Main window menu bar.
- b. Drag the *test_sm* comparison object from the compare tab of the Main window to the List window.
- c. Scroll down the window.

Differences are noted with yellow highlighting (Figure 17-6). Differences that have been annotated have red highlighting.

Figure 17-6. Compare differences in the List window



Saving and Reloading Comparison Data

You can save comparison data for later viewing, either in a text file or in files that can be reloaded into Questasim.

To save comparison data so it can be reloaded into Questasim, you must save two files. First, you save the computed differences to one file; next, you save the comparison configuration rules to a separate file. When you reload the data, you must have the reference dataset open.

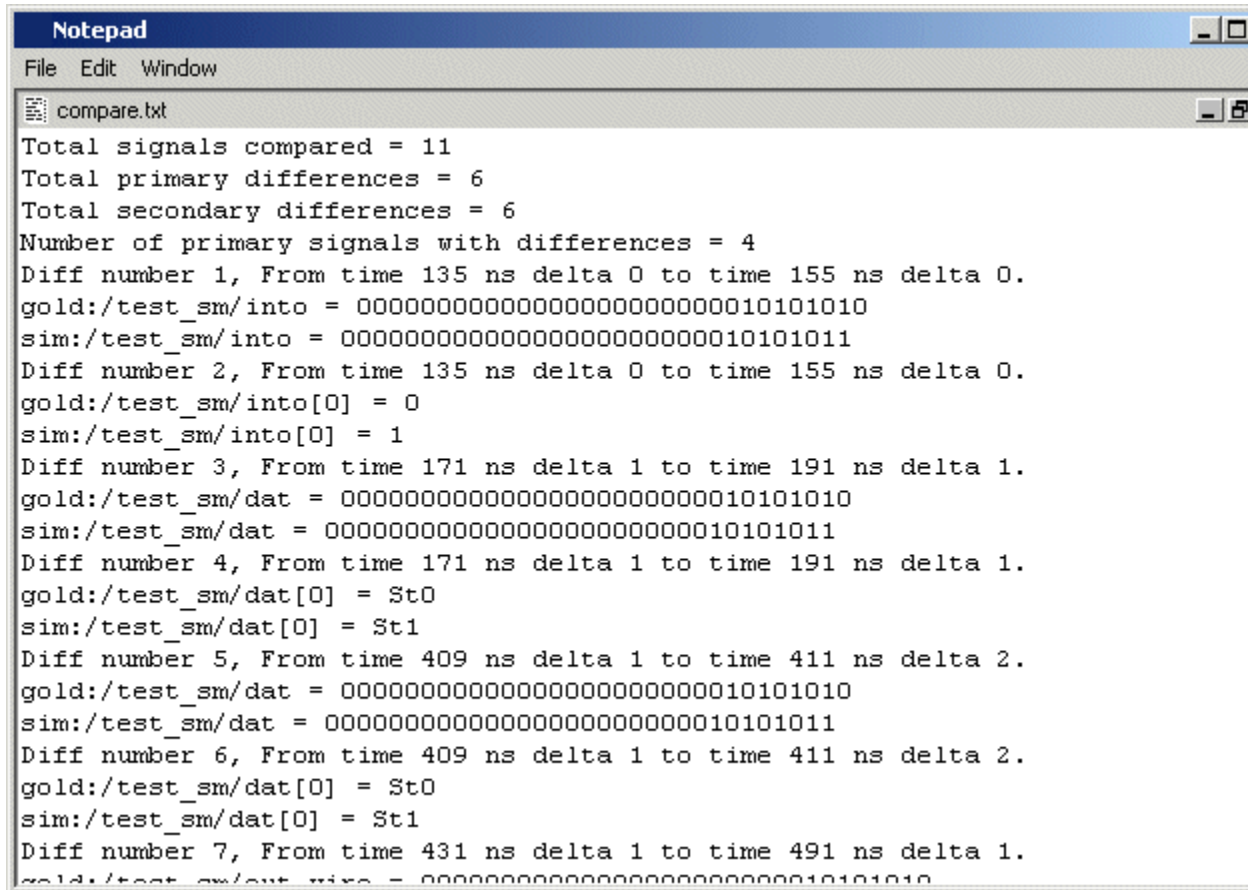
1. Save the comparison data to a text file.
 - a. In the Main window, select **Tools > Waveform Compare > Differences > Write Report**.

- b. Click **Save**.

This saves *compare.txt* to the current directory.

- c. Type **notepad compare.txt** at the VSIM> prompt to display the report (Figure 17-7).

Figure 17-7. Coverage data saved to a text file



The screenshot shows a Notepad window with the title bar 'Notepad'. The menu bar includes 'File', 'Edit', and 'Window'. The file name in the title bar is 'compare.txt'. The text content of the file is as follows:

```
Total signals compared = 11
Total primary differences = 6
Total secondary differences = 6
Number of primary signals with differences = 4
Diff number 1, From time 135 ns delta 0 to time 155 ns delta 0.
gold:/test_sm/into = 00000000000000000000000010101010
sim:/test_sm/into = 00000000000000000000000010101011
Diff number 2, From time 135 ns delta 0 to time 155 ns delta 0.
gold:/test_sm/into[0] = 0
sim:/test_sm/into[0] = 1
Diff number 3, From time 171 ns delta 1 to time 191 ns delta 1.
gold:/test_sm/dat = 00000000000000000000000010101010
sim:/test_sm/dat = 00000000000000000000000010101011
Diff number 4, From time 171 ns delta 1 to time 191 ns delta 1.
gold:/test_sm/dat[0] = St0
sim:/test_sm/dat[0] = St1
Diff number 5, From time 409 ns delta 1 to time 411 ns delta 2.
gold:/test_sm/dat = 00000000000000000000000010101010
sim:/test_sm/dat = 00000000000000000000000010101011
Diff number 6, From time 409 ns delta 1 to time 411 ns delta 2.
gold:/test_sm/dat[0] = St0
sim:/test_sm/dat[0] = St1
Diff number 7, From time 431 ns delta 1 to time 491 ns delta 1.
gold:/test_sm/out_wire = 00000000000000000000000010101010
```

- d. Close Notepad when you have finished viewing the report.
2. Save the comparison data in files that can be reloaded into QuestaSim.
 - a. Select **Tools > Waveform Compare > Differences > Save**.
 - b. Click **Save**.

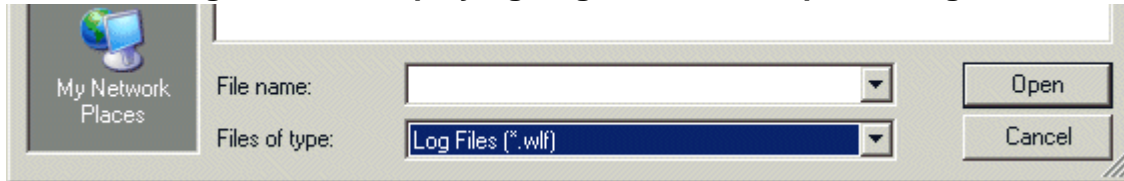
This saves *compare.dif* to the current directory.

- c. Select **Tools > Waveform Compare > Rules > Save**.
- d. Click **Save**.

This saves *compare.rul* to the current directory.

- e. Select **Tools > Waveform Compare > End Comparison**.
3. Reload the comparison data.
 - a. With the sim tab of the Workspace active, select **File > Open**.
 - b. Change the **Files of Type** to Log Files (*.wlf) (Figure 17-8).

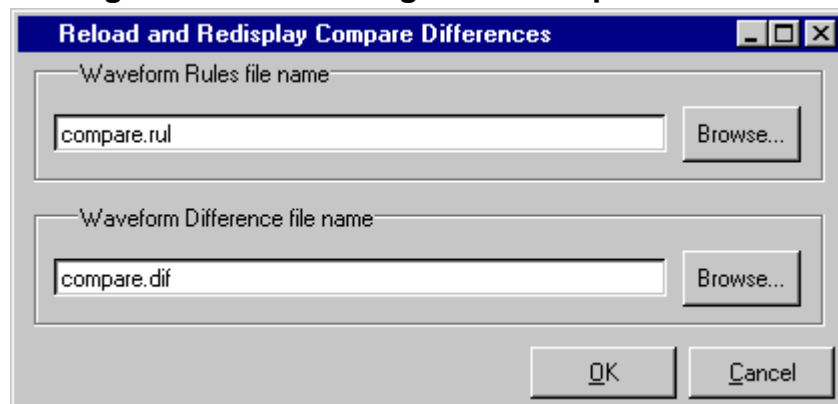
Figure 17-8. Displaying Log Files in the Open dialog



- c. Double-click *gold.wlf* to open the dataset.
- d. Select **Tools > Waveform Compare > Reload**.

Since you saved the data using default file names, the dialog should already have the correct Waveform Rules and Waveform Difference files specified (Figure 17-9).

Figure 17-9. Reloading saved comparison data



- e. Click **OK**.

The comparison reloads. You can drag the comparison object to the Wave or List window to view the differences again.

Lesson Wrap-Up

This concludes this lesson. Before continuing we need to end the current simulation and close the *gold.wlf* dataset.

1. Type **quit -sim** at the VSIM> prompt.
2. Type **dataset close gold** at the QuestaSim> prompt.

Chapter 18

Automating Simulation

Introduction

Aside from executing a couple of pre-existing DO files, the previous lessons focused on using QuestaSim in interactive mode: executing single commands, one after another, via the GUI menus or Main window command line. In situations where you have repetitive tasks to complete, you can increase your productivity with DO files.

DO files are scripts that allow you to execute many commands at once. The scripts can be as simple as a series of QuestaSim commands with associated arguments, or they can be full-blown Tcl programs with variables, conditional execution, and so forth. You can execute DO files from within the GUI or you can run them from the system command prompt without ever invoking the GUI.

Note



This lesson assumes that you have added the `<install_dir>/questasim/<platform>` directory to your PATH. If you did not, you will need to specify full paths to the tools (i.e., vlib, vmap, vlog, vcom, and vsim) that are used in the lesson.

Related Reading

User's Manual Chapter: [Tcl and Macros \(DO Files\)](#).

Practical Programming in Tcl and Tk, Brent B. Welch, Copyright 1997

Creating a Simple DO File

Creating DO files is as simple as typing the commands in a text file. Alternatively, you can save the Main window transcript as a DO file. In this exercise, you will use the commands you enter in the Main window transcript to create a DO file that adds signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

1. Load the *test_counter* design unit.
 - a. If necessary, start QuestaSim.
 - b. Change to the directory you created in the "Basic Simulation" lesson.
 - c. Enter **`vsim -voptargs="+acc" test_counter`** to load the design unit.

2. Enter commands to add signals to the Wave window, force signals, and run the simulation.
 - a. Select **File > New > Source > Do** to create a new DO file.
 - b. Enter the following commands into the source window:

```
add wave count
add wave clk
add wave reset
force -freeze clk 0 0, 1 {50 ns} -r 100
force reset 1
run 100
force reset 0
run 300
force reset 1
run 400
force reset 0
run 200
```

3. Save the file.
 - a. Select **File > Save As**.
 - b. Type **sim.do** in the File name: field and save it to the current directory.
4. Load the simulation again and use the DO file.
 - a. Enter **quit -sim** at the VSIM> prompt.
 - b. Enter **vsim -voptargs="+acc" test_counter** at the QuestaSim> prompt.

The **-voptargs="+acc"** argument for the **vsim** command provides visibility into the design for debugging purposes.

Note

By default, QuestaSim optimizations are performed on all designs (see [Optimizing Designs with vopt](#)).

- c. Enter **do sim.do** at the VSIM> prompt.

QuestaSim executes the saved commands and draws the waves in the Wave window.

5. When you are done with this exercise, select **File > Quit** to quit QuestaSim.

Running in Command-Line Mode

We use the term "command-line mode" to refer to simulations that are run from a DOS/ UNIX prompt without invoking the GUI. Several QuestaSim commands (e.g., **vsim**, **vlib**, **vlog**, etc.) are actually stand-alone executables that can be invoked at the system command prompt. Additionally, you can create a DO file that contains other QuestaSim commands and specify that file when you invoke the simulator.

1. Create a new directory and copy the tutorial files into it.

Start by creating a new directory for this exercise. Create the directory and copy the following files into it:

- `/<install_dir>/examples/tutorials/verilog/automation/counter.v`
- `/<install_dir>/examples/tutorials/verilog/automation/stim.do`

This lesson uses the Verilog file *counter.v*. If you have a VHDL license, use *the counter.vhd* and *stim.do* files in the `/<install_dir>/examples/tutorials/vhdl/automation` directory instead.

2. Create a new design library and compile the source file.

Again, enter these commands at a DOS/ UNIX prompt in the new directory you created in step 1.

- a. Type **vlib work** at the DOS/ UNIX prompt.
- b. For Verilog, type **vlog counter.v** at the DOS/ UNIX prompt. For VHDL, type **vcom counter.vhd**.

3. Create a DO file.

- a. Open a text editor.
- b. Type the following lines into a new file:

```
# list all signals in decimal format
add list -decimal *

# read in stimulus
do stim.do

# output results
write list counter.lst

# quit the simulation
quit -f
```

- c. Save the file with the name *sim.do* and place it in the current directory.

4. Run the batch-mode simulation.

- a. Type **vsim -voptargs="+acc"-c -do sim.do counter -wlf counter.wlf** at the DOS/ UNIX prompt.

The **-c** argument instructs QuestaSim not to invoke the GUI. The **-wlf** argument saves the simulation results in a WLF file. This allows you to view the simulation results in the GUI for debugging purposes.

5. View the list output.

- a. Open *counter.lst* and view the simulation results. Output produced by the Verilog version of the design should look like the following:

```

ns          /counter/count
delta       /counter/clk
           /counter/reset
0   +0      x  z  *
1   +0      0  z  *
50  +0      0  *  *
100 +0      0  0  *
100 +1      0  0  0
150 +0      0  *  0
151 +0      1  *  0
200 +0      1  0  0
250 +0      1  *  0
.
.
.

```

The output may appear slightly different if you used the VHDL version.

6. View the results in the GUI.

Since you saved the simulation results in *counter.wlf*, you can view them in the GUI by invoking VSIM with the **-view** argument.

Note

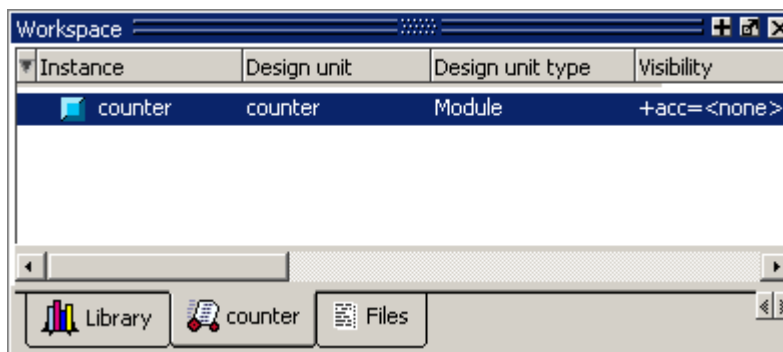


Make sure your PATH environment variable is set with the current version of QuestaSim at the front of the string.

- a. Type **vsim -view counter.wlf** at the DOS/ UNIX prompt.

The GUI opens and a dataset tab named "counter" is displayed in the Workspace (Figure 18-1).

Figure 18-1. A Dataset in the Main Window Workspace



- b. Right-click the *counter* instance and select **Add > To Wave > All items in region**.

The waveforms display in the Wave window.

7. When you finish viewing the results, select **File > Quit** to close QuestaSim.

Using Tcl with the Simulator

The DO files used in previous exercises contained only QuestaSim commands. However, DO files are really just Tcl scripts. This means you can include a whole variety of Tcl constructs such as procedures, conditional operators, math and trig functions, regular expressions, and so forth.

In this exercise, you create a simple Tcl script that tests for certain values on a signal and then adds bookmarks that zoom the Wave window when that value exists. Bookmarks allow you to save a particular zoom range and scroll position in the Wave window. The Tcl script also creates buttons in the Main window that call these bookmarks.

1. Create the script.

- a. In a text editor, open a new file and enter the following lines:

```
proc add_wave_zoom {stime num} {  
    echo "Bookmarking wave $num"  
    bookmark add wave "bk$num" "[expr $stime - 50] [expr $stime +  
100]" 0  
    add button "$num" [list bookmark goto wave bk$num]  
}
```

These commands do the following:

- Create a new procedure called "add_wave_zoom" that has two arguments, *stime* and *num*.
- Create a bookmark with a zoom range from the current simulation time minus 50 time units to the current simulation time plus 100 time units.
- Add a button to the Main window that calls the bookmark.

- b. Now add these lines to the bottom of the script:

```
add wave -r /*  
when {clk'event and clk="1"} {  
    echo "Count is [exa count]"  
    if {[examine count]== "00100111"} {  
        add_wave_zoom $now 1  
    } elseif {[examine count]== "01000111"} {  
        add_wave_zoom $now 2  
    }  
}
```

These commands do the following:

- Add all signals to the Wave window.
- Use a **when** statement to identify when *clk* transitions to 1.
- Examine the value of *count* at those transitions and add a bookmark if it is a certain value.

- c. Save the script with the name "*add_bkmrk.do*."

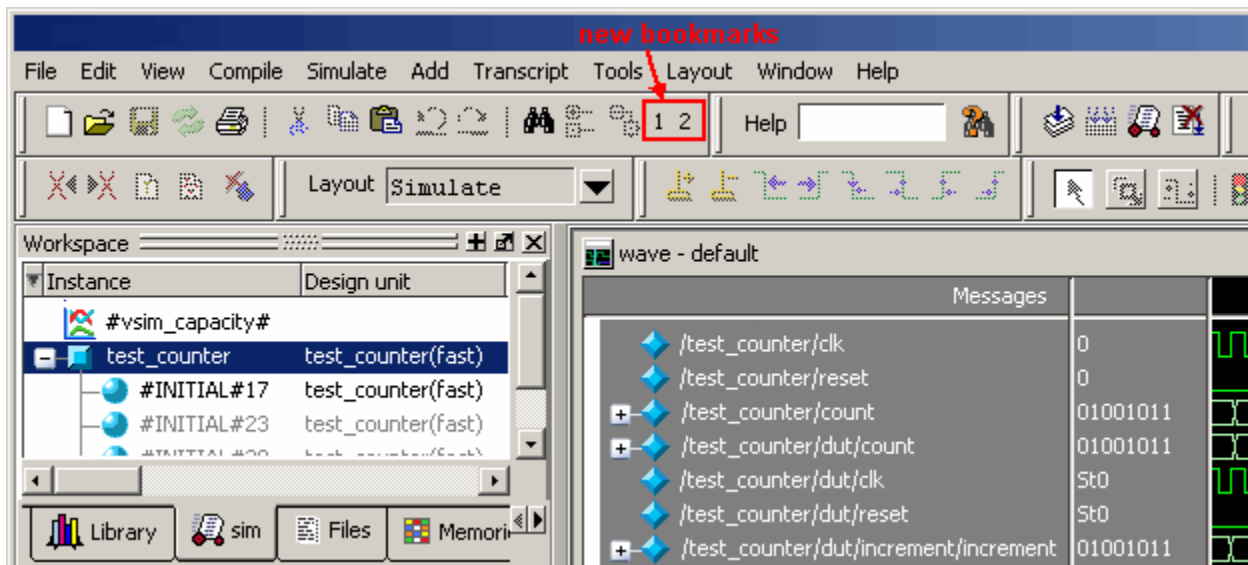
Save it into the directory you created in [Basic Simulation](#).

2. Load the *test_counter* design unit.
 - a. Start QuestaSim.
 - b. Select **File > Change Directory** and change to the directory you saved the DO file to in step 1c above.
 - c. Enter the following command at the QuestaSim> prompt:
vsim -voptargs="+acc" test_counter
3. Execute the DO file and run the design.
 - a. Type **do add_bkmrk.do** at the VSIM> prompt.
 - b. Type **run 1500 ns** at the VSIM> prompt.

The simulation runs and the DO file creates two bookmarks.

It also creates buttons (labeled "1" and "2") on the Main window toolbar that jump to the bookmarks ([Figure 18-2](#)).

Figure 18-2. Buttons Added to the Main Window Toolbar



- c. Click the buttons and watch the Wave window zoom on and scroll to the time when *count* is the value specified in the DO file.

Lesson Wrap-Up

This concludes this lesson.

1. Select **File > Quit** to close QuestaSim.

— A —

aCC, 66

add dataflow command, 112

add wave command, 83

Assertions

add to dataflow, 162

debugging failures, 160

ignore assertions during simulation, 154

-nopsl argument to vsim, 154

speeding debugging, 155

— B —

break icon, 32

breakpoints

in SystemC modules, 75

setting, 32

stepping, 35

— C —

C Debug, 75

Code Coverage

excluding lines and files, 149

reports, 150

Source window, 146

command-line mode, 226

compile order, changing, 44

compiling your design, 20, 27

-cover argument, 142

coverage report command, 152

cursors, Wave window, 84, 97

— D —

Dataflow window

displaying hierarchy, 112

expanding to drivers/readers, 104

options, 112

tracing events, 106

tracing unknowns, 110

dataset close command, 223

design library

working type, 21

design optimization, 19

documentation, 15

drivers, expanding to, 104

— E —

external libraries, linking to, 58

— F —

folders, in projects, 47

format, saving for Wave window, 87

— G —

gcc, 66

— H —

hierarchy, displaying in Dataflow window, 112

— L —

libraries

design library types, 21

linking to external libraries, 58

mapping to permanently, 62

resource libraries, 21

working libraries, 21

working, creating, 26

linking to external libraries, 58

— M —

manuals, 15

mapping libraries permanently, 62

memories

changing values, 127

initializing, 123

memory contents, saving to a file, 121

— N —

notepad command, 222

— O —

optimization, 19

options, simulation, 50

— P —

Performance Analyzer
 filtering data, [137](#)
 physical connectivity, [104](#)
 Profiler
 profile details, [136](#)
 viewing profile details, [136](#)
 projects
 adding items to, [42](#)
 creating, [41](#)
 flow overview, [20](#)
 organizing with folders, [47](#)
 simulation configurations, [50](#)

— Q —

quit command, [59](#)

— R —

radix command, [116](#)
 reference dataset, Waveform Compare, [216](#)
 reference signals, [215](#)
 run -all, [32](#)
 run command, [31](#)

— S —

saving simulation options, [50](#)
 simulation
 basic flow overview, [19](#)
 restarting, [33](#)
 running, [30](#)
 simulation configurations, [50](#)
 stepping after a breakpoint, [35](#)
 SystemC
 setting up the environment, [66](#)
 supported platforms, [66](#)
 viewing in the GUI, [74](#)

— T —

Tcl, using in the simulator, [229](#)
 test dataset, Waveform Compare, [216](#)
 test signals, [215](#)
 time, measuring in Wave window, [84](#), [97](#)
 toggle statistics, Signals window, [148](#)
 tracing events, [106](#)
 tracing unknowns, [110](#)

— U —

unknowns, tracing, [110](#)

— V —

vcom command, [116](#)
 vlib command, [116](#)
 vlog command, [116](#)
 vsim command, [26](#)

— W —

Wave window
 adding items to, [82](#), [90](#)
 cursors, [84](#), [97](#)
 measuring time with cursors, [84](#), [97](#)
 saving format, [87](#)
 zooming, [84](#), [93](#)
 Waveform Compare
 reference signals, [215](#)
 saving and reloading, [221](#)
 test signals, [215](#)
 working library, creating, [20](#), [26](#)

— X —

X values, tracing, [110](#)

— Z —

zooming, Wave window, [84](#), [93](#)

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/terms_conditions/enduser.cfm

IMPORTANT INFORMATION

USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE. USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software between you, the end user, as an authorized representative of the company acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries (collectively "Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by you and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

- GRANT OF LICENSE.** The software programs, including any updates, modifications, revisions, copies, documentation and design data ("Software"), are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; (c) for the license term; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions.
- EMBEDDED SOFTWARE.** If you purchased a license to use embedded software development ("ESD") Software, if applicable, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into your products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.
- BETA CODE.** Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this section 3 shall survive the termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than employees and on-site contractors, excluding Mentor Graphics' competitors, whose job performance requires access and who are under obligations of confidentiality. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.
5. **LIMITED WARRANTY.**
 - 5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
 - 5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 6 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.
7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 7 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.
8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS

DESCRIBED IN SECTION 7. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

9. **INFRINGEMENT.**

9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.

9.4. THIS SECTION IS SUBJECT TO SECTION 6 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30 day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.

14. **AUDIT RIGHTS.** You will monitor access to, location and use of Software. With reasonable prior notice and during your normal business hours, Mentor Graphics shall have the right to review your software monitoring system and reasonably relevant records to confirm your compliance with the terms of this Agreement, an addendum to this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet report log files that you shall capture and provide at Mentor Graphics' request. Mentor Graphics shall treat as confidential information all of your information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement or addendum to this Agreement. The provisions of this section 14 shall survive the expiration or termination of this Agreement.

15. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH OR SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia (except for Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the Chairman of the Singapore International Arbitration Centre (“SIAC”) to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section 15. This section shall not restrict Mentor Graphics’ right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Some Software may contain code distributed under a third party license agreement that may provide additional rights to you. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.