Questa® SIM Tutorial

Including Support for Questa SV/AFV

Software Version 10.4c

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/doc_feedback_form

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The Questa SIM Tutorial provides lessons for gaining a basic understanding of how to simulate your design. It includes step-by-step instruction on the basics of simulation - from creating a working library, compiling your design, and loading the simulator to running the simulation and debugging your results.

Where to Find Questa SIM Documentation

Questa SIM documentation is available in both PDF and HTML formats. Refer to the table below for a complete list.

Note

Documentation for Questa Sim also supports Questa SV/AFV products. For more complete information on current support for Questa SV/AFV, refer to the Installation and Licensing Guide.

Table 1-1. Documentation List

Document	Format	How to get it
Installation & Licensing Guide	PDF	**Help** > **PDF Bookcase**
	HTML and PDF	**Help** > **InfoHub**
Quick Guide (command and feature quick-reference)	PDF	**Help** > **PDF Bookcase** and **Help** > **InfoHub**
Tutorial	PDF	**Help** > **PDF Bookcase**
	HTML and PDF	**Help** > **InfoHub**
User's Manual	PDF	**Help** > **PDF Bookcase**
	HTML and PDF	**Help** > **InfoHub**
Command Reference Manual	PDF	**Help** > **PDF Bookcase**
	HTML and PDF	**Help** > **InfoHub**
Graphical User Interface (GUI) Reference Manual	PDF	**Help** > **PDF Bookcase**
	HTML and PDF	**Help** > **InfoHub**

Table 1-1. Documentation List

Document	Format	How to get it
Foreign Language Interface Manual	PDF	**Help > PDF Bookcase**
	HTML	**Help > InfoHub**
OVL Checkers Manager User's Guide	PDF	**Help > PDF Bookcase**
	HTML	**Help > InfoHub**
Power Aware Simulation User's Manual	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 *contents.htm* in *\modeltech\docs\tcl_help_html*
Technotes	HTML	available from the support site

Download a Free PDF Reader With Search

Questa SIM 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 product support includes software enhancements, technical support, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service.

For details, refer to the following location on the Worldwide Web:

`http://supportnet.mentor.com/about/`

If you have questions about this software release, please log in to the SupportNet web site. You can 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 can register for SupportNet by filling out the short form at:

`http://supportnet.mentor.com/user/register.cfm`

For any customer support contact information, refer to the following web site location:

`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 Questa SIM's GUI, the interface is consistent for all platforms.)

Example Designs

Questa SIM comes with Verilog and VHDL versions of the designs used in most of 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

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

This lesson provides a brief conceptual overview of the Questa SIM 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, Questa SIM 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.

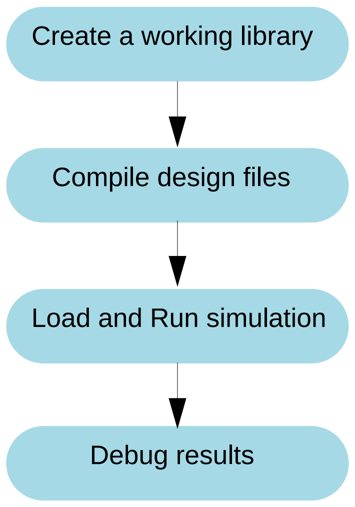
Global optimizations, however, may have an impact on the visibility of the design simulation results you can view – certain signals and processes may not be visible. If these signals and processes are important for debugging the design, it may be necessary to customize the simulation by removing optimizations 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 Questa SIM is called **vopt**. Please refer to the Optimizing Designs with vopt chapter in the Questa SIM 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 Questa SIM.

Figure 2-1. Basic Simulation Flow - Overview Lab



- Creating the Working Library

  In Questa SIM, all designs are compiled into a library. You typically start a new simulation in Questa SIM by creating a working library called "work," which is the default 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 Questa SIM 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

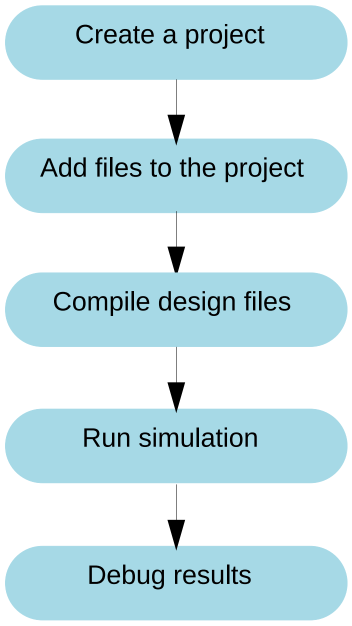
  If you don’t get the results you expect, you can use Questa SIM’s robust debugging environment to track down the cause of the problem.

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 Questa SIM, 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 Questa SIM 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 Questa SIM unless you specifically close them.

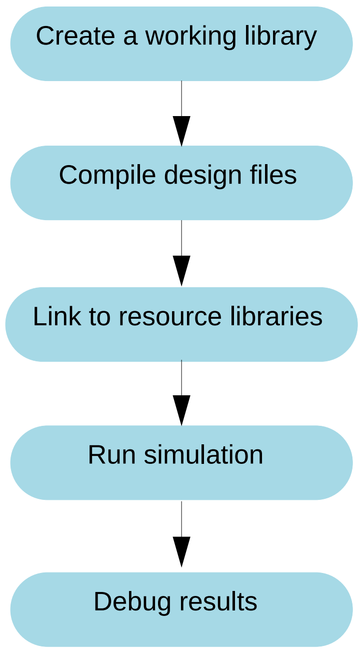
Multiple Library Flow

Questa SIM 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 test bench 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 test bench to the project.

Debugging Tools

Questa SIM 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

In this lesson you will guide you through the basic simulation flow.

You will learn to:

1. Create the Working Design Library
2. Compile the Design Units
3. Optimize the Design
4. Load the Design
5. 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 test bench.

The pathnames are as follows:

Verilog – *<install_dir>/examples/tutorials/verilog/basicSimulation/counter.v* and t*counter.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 test bench with the VHDL counter or vice versa.

Related Topics

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.

Procedure

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 Questa SIM *if necessary*.

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

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

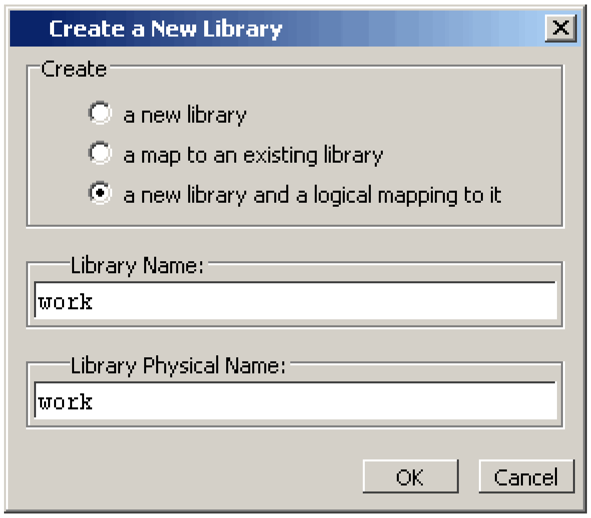
   b. Select **File > Change Directory** and change to the directory you created in step 1.

3. Create the working library.

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

      This opens a dialog box where you specify physical and logical names for the library (Figure 3-1). You can create a new library or map to an existing library. We'll be doing the former.

Figure 3-1. The Create a New Library Dialog Box


Create a New Library
Create
a new library
a map to an existing library
a new library and a logical mapping to it
OK
Cancel

Create: [✓ a new library and a logical mapping to it]

Library Name: work

Library Physical Name: work

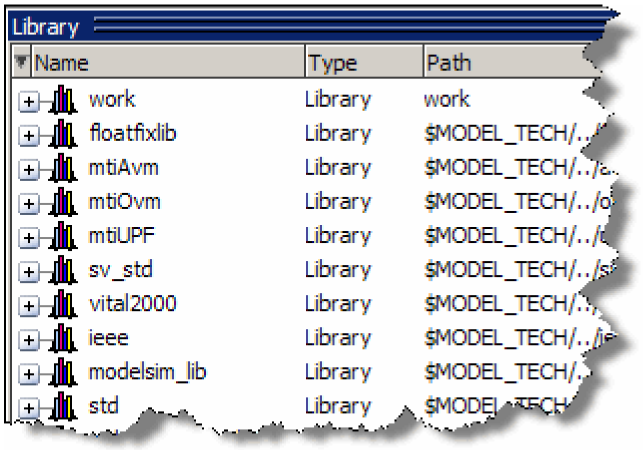
   b. Type **work** in the Library Name field (if it isn't already entered automatically).

   c. Click **OK**.

Questa SIM 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 Questa SIM library. Do not edit the folder contents from your operating system; all changes should be made from within Questa SIM.

Questa SIM also adds the library to the Library window (Figure 3-2) and records the library mapping for future reference in the Questa SIM initialization file (*modelsim.ini*).

Figure 3-2. work Library Added to the Library Window



Name	Type	Path
work	Library	work
floatfixlib	Library	$MODEL_TECH/../
mtiAvm	Library	$MODEL_TECH/../a..
mtiOvm	Library	$MODEL_TECH/../o
mtiUPF	Library	$MODEL_TECH/../
sv_std	Library	$MODEL_TECH/../s
vital2000	Library	$MODEL_TECH/..
ieee	Library	$MODEL_TECH/../ie
modelsim_lib	Library	$MODEL_TECH/..
std	Library	$MODEL_TECH

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

```
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 Units

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

You can compile your source files using the menus and dialog boxes of the graphic interface, as in the Verilog example below, or by entering a command at the Questa SIM> prompt.

Procedure

1. Compile *counter.v* and *tcounter.v*.

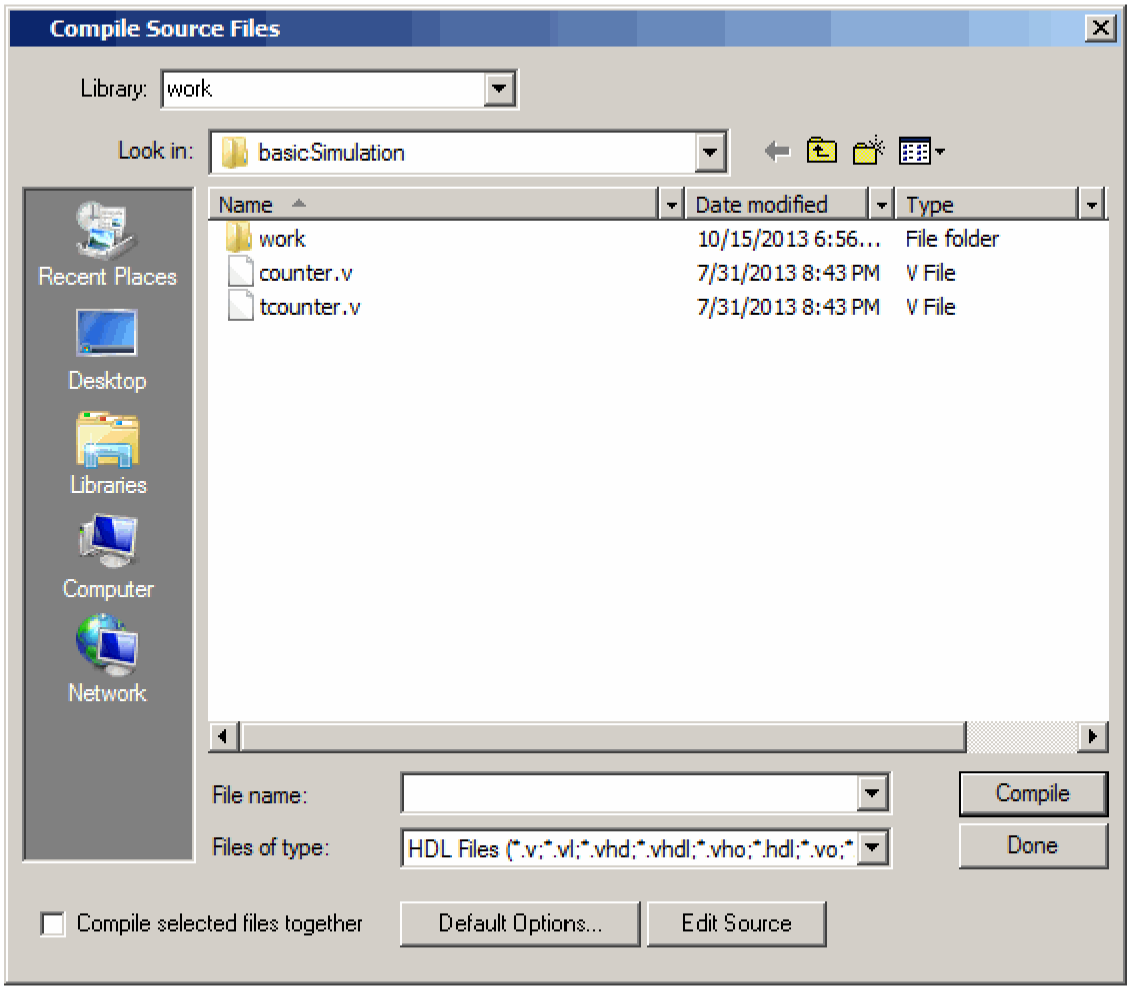
    a. Select **Compile** > **Compile**. This opens the Compile Source Files dialog box (Figure 3-3).

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

b. Select both *counter.v* and *tcounter.v* modules from the Compile Source Files dialog box and click **Compile**. The files are compiled into the *work* library.

c. When compile is finished, click **Done**.

Figure 3-3. Compile Source Files Dialog Box


Compile Source Files

Library: work

Look in: basicSimulation

Name	Date modified	Type
work	10/15/2013 6:56...	File folder
counter.v	7/31/2013 8:43 PM	V File
tcounter.v	7/31/2013 8:43 PM	V File

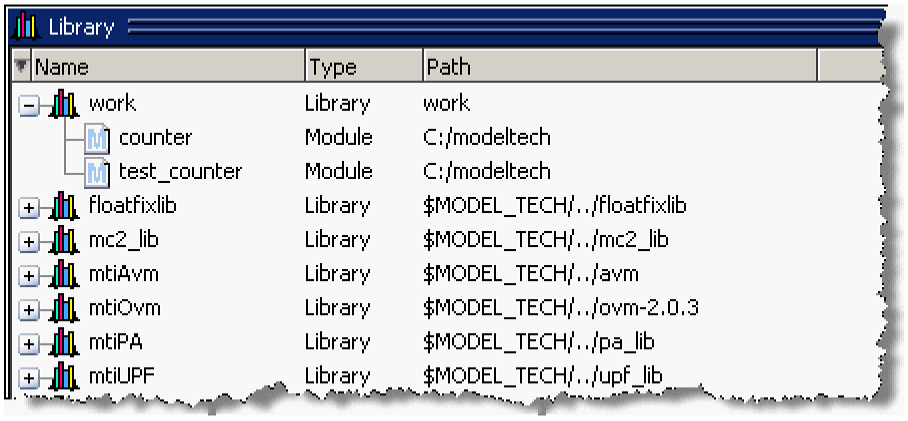
File name:

Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*

2. View the compiled design units.

a. In the Library window, click the ’+’ icon next to the *work* library and you will see two design units (Figure 3-4). You can also see their types (Modules, Entities, etc.) and the path to the underlying source files.

Figure 3-4. Verilog Modules Compiled into work Library


Library

Name	Type	Path
work	Library	work
counter	Module	C:/modeltech
test_counter	Module	C:/modeltech
floatfixlib	Library	$MODEL_TECH/../floatfixlib
mc2_lib	Library	$MODEL_TECH/../mc2_lib
mtiAvm	Library	$MODEL_TECH/../avm
mtiOvm	Library	$MODEL_TECH/../ovm-2.0.3
mtiPA	Library	$MODEL_TECH/../pa_lib
mtiUPF	Library	$MODEL_TECH/../upf_lib

Optimize the Design

Optimizing your design for simulation will speed the process.

Procedure

1. Use the vopt command to optimize the design with full visibility into all design units.

   a. Enter the following command at the Questa SIM> prompt in the Transcript window:

   ```
   vopt +acc test_counter -o testcounter_opt
   ```

   The **+acc** switch provides visibility into the design for debugging purposes.

   The **-o** switch allows you designate the name of the optimized design file (testcounter_opt).

Note
You must provide a name for the optimized design file when you use the vopt command.

Load the Design

Now you're ready to load the design into the simulator.

Procedure

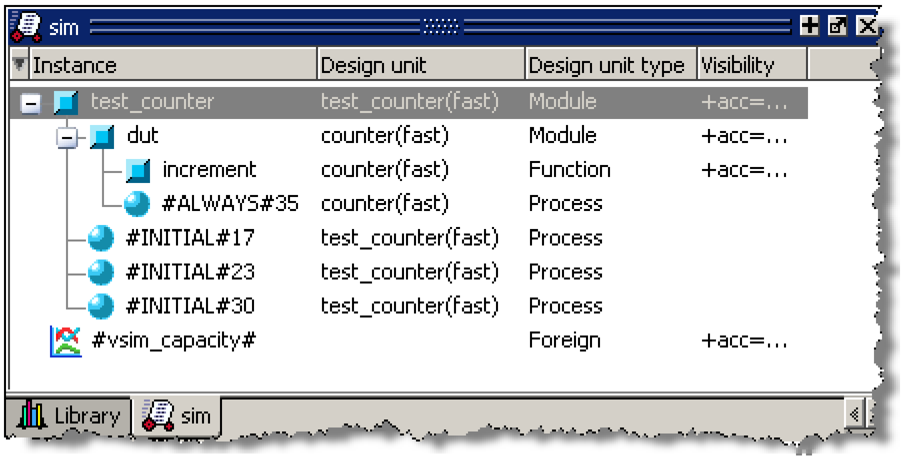
1. Load the *test_counter* module into the simulator.

   a. Use the optimized design name to load the design with the vsim command:

   ```
   vsim testcounter_opt
   ```

   When the design is loaded, a Structure window opens (labeled **sim**). This window displays the hierarchical structure of the design as shown in Figure 3-5. You can navigate within the design hierarchy in the Structure (**sim**) window by clicking on any line with a '+' (expand) or '-' (contract) icon.

Figure 3-5. The Design Hierarchy



Instance	Design unit	Design unit type	Visibility
test_counter	test_counter(fast)	Module	+acc=...
dut	counter(fast)	Module	+acc=...
increment	counter(fast)	Function	+acc=...
#ALWAYS#35	counter(fast)	Process	
#INITIAL#17	test_counter(fast)	Process	
#INITIAL#23	test_counter(fast)	Process	
#INITIAL#30	test_counter(fast)	Process	
#vsim_capacity#		Foreign	+acc=...

2. Open the Objects and Processes windows.

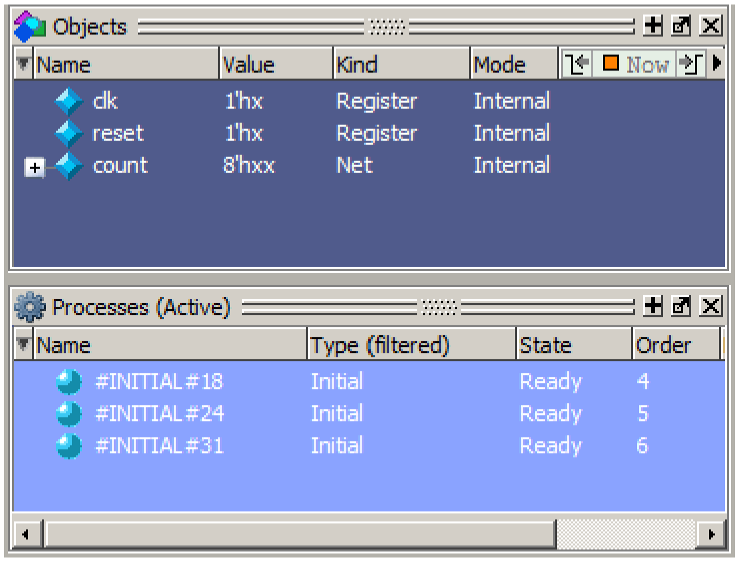
   a. Select **View** > **Objects** from the menu bar.

   b. Select **View** > **Process**.

   The Objects window shows the names and current values of data objects in the current region selected in the Structure (sim) window (Figure 3-6). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters, and member data variables of a SystemC module.

   The Processes window displays a list of processes in one of four viewing modes: Active, In Region, Design, and Hierarchical. The Design view mode is intended for primary navigation of ESL (Electronic System Level) designs where processes are a foremost consideration. By default, this window displays the active processes in your simulation (Active view mode).

Figure 3-6. The Object Window and Processes Window



Objects

Name	Value	Kind	Mode
clk	1'hx	Register	Internal
reset	1'hx	Register	Internal
count	8'hxx	Net	Internal

Processes (Active)

Name	Type (filtered)	State	Order
#INITIAL#18	Initial	Ready	4
#INITIAL#24	Initial	Ready	5
#INITIAL#31	Initial	Ready	6

Run the Simulation

We’re ready to run the simulation. But before we do, we’ll open the Wave window and add signals to it.

Procedure

1. Open the Wave window.

   a. Enter **view wave** at the command line.

   The Wave window opens in the right side of the Main window. Resize it, if necessary, so it is visible.

   You can also use the **View** > **Wave** menu selection to open a Wave window. The Wave window is just one of several debugging windows available on the **View** menu.

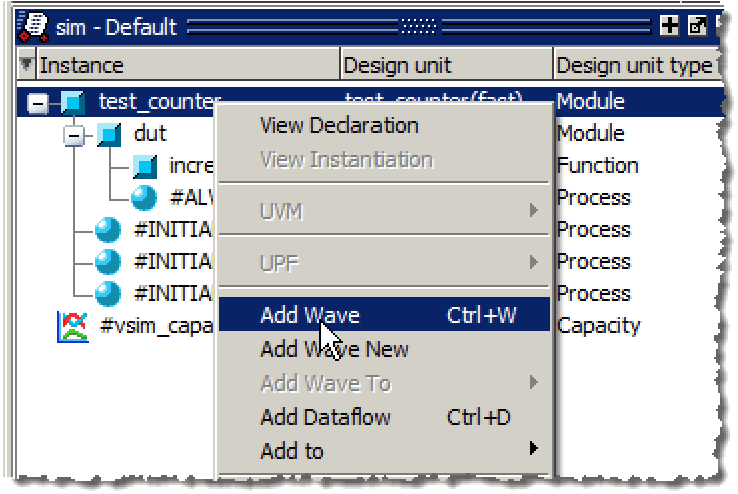
2. Add signals to the Wave window.

   a. In the Structure (sim) window, right-click *test_counter* to open a popup context menu.

   b. Select **Add Wave** (Figure 3-7).

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

Figure 3-7. Using the Popup Menu to Add Signals to Wave Window



3. Run the simulation.

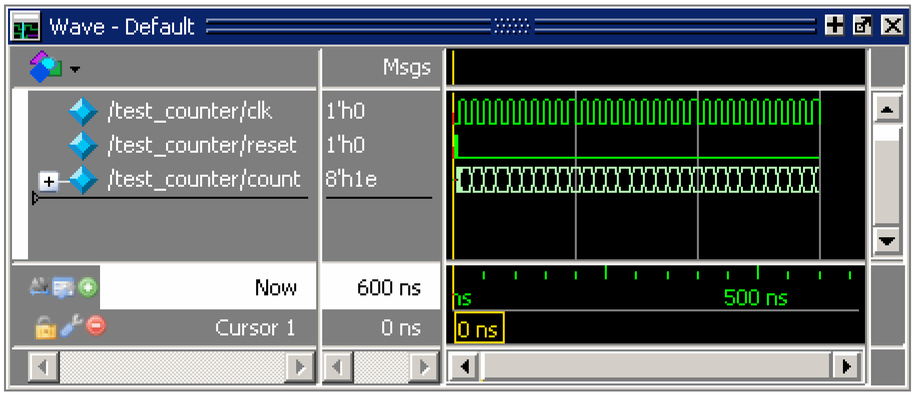
   a. Click the Run icon.

      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 Transcript window.

      The simulation advances another 500 ns for a total of 600 ns (Figure 3-8).

Figure 3-8. 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 (ie., a Verilog $stop statement) that halts the simulation.

   d. Click the Break icon to stop the simulation.

Set Breakpoints and Step through the Source

Next you will take a brief look at one interactive debugging feature of the Questa SIM 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 executable lines, which are indicated with red line numbers.

Procedure

1. Open *counter.v* in the Source window.

   a. Select **View** > **Files** to open the Files window.

   b. Click the + sign next to the *sim* filename to see the contents of *vsim.wlf* dataset.

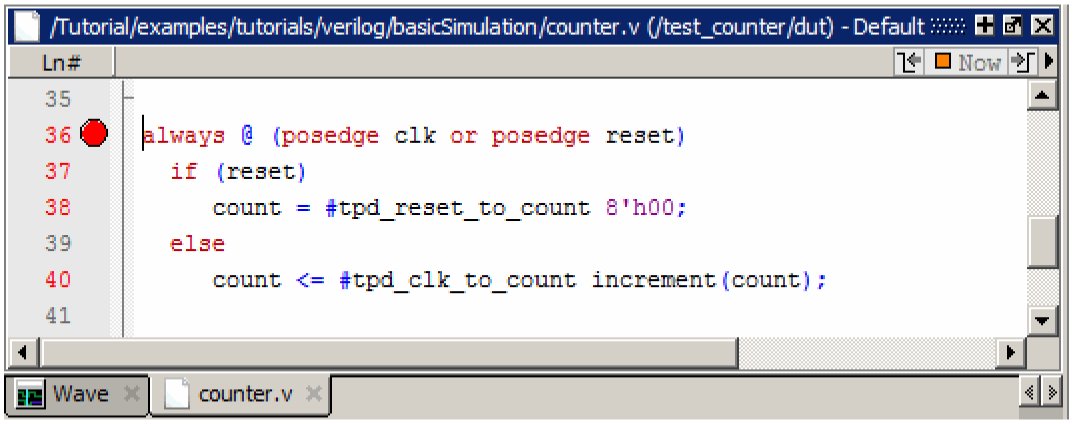
   c. Double-click *counter.v* (or *counter.vhd* if you are simulating the VHDL files) to open the file in the Source window.

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 Ln# (line number) column next to the line number.

   A red dot appears in the line number column at line number 36 (Figure 3-9), indicating that a breakpoint has been set.

Figure 3-9. Setting Breakpoint in Source Window


/Tutorial/examples/tutorials/verilog/basicSimulation/counter.v (/test_counter/dut) - Default
Ln#
Now
Wave
counter.v

```
35
36 ●  always @ (posedge clk or posedge reset)
37      if (reset)
38         count = #tpd_reset_to_count 8'h00;
39      else
40         count <= #tpd_clk_to_count increment(count);
41
```

3. Disable, enable, and delete the breakpoint.

   a. Click the red dot to disable the breakpoint. It will become a gray dot.

   b. Click the gray dot again to re-enable the breakpoint. It will become a red dot.

   c. Click the red dot with your right mouse button and select **Remove Breakpoint 36**.

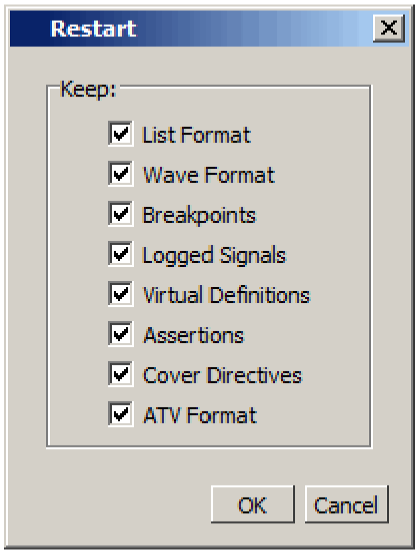
   d. Click in the line number column next to line number 36 again to re-create the breakpoint.

4. Restart the simulation.

a. Click the Restart icon to reload the design elements and reset the simulation time to zero.

The Restart dialog box that appears gives you options on what to retain during the restart (Figure 3-10).

Figure 3-10. Setting Restart Functions



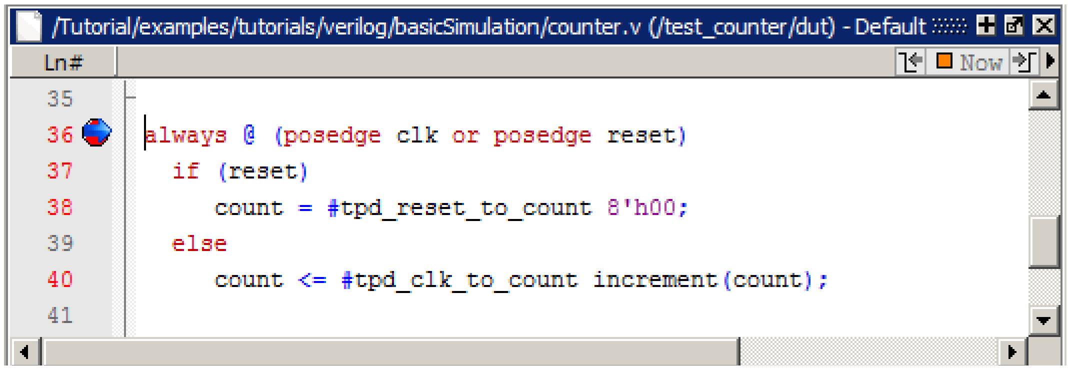
Keep:
- [✓] List Format
- [✓] Wave Format
- [✓] Breakpoints
- [✓] Logged Signals
- [✓] Virtual Definitions
- [✓] Assertions
- [✓] Cover Directives
- [✓] ATV Format

b. Click the **OK** button in the Restart dialog box.

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 3-11), and issues a Break message in the Transcript window.

Figure 3-11. Blue Arrow Indicates Where Simulation Stopped.

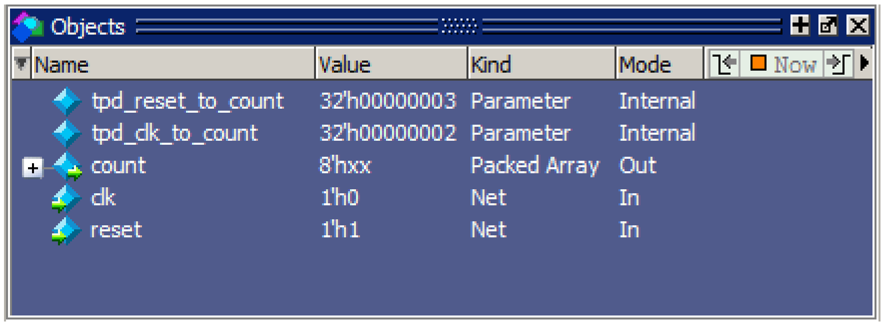

/Tutorial/examples/tutorials/verilog/basicSimulation/counter.v (/test_counter/dut) - Default
Ln#
Now

```
always @ (posedge clk or posedge reset)
  if (reset)
     count = #tpd_reset_to_count 8'h00;
  else
     count <= #tpd_clk_to_count increment(count);
```

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 3-12).

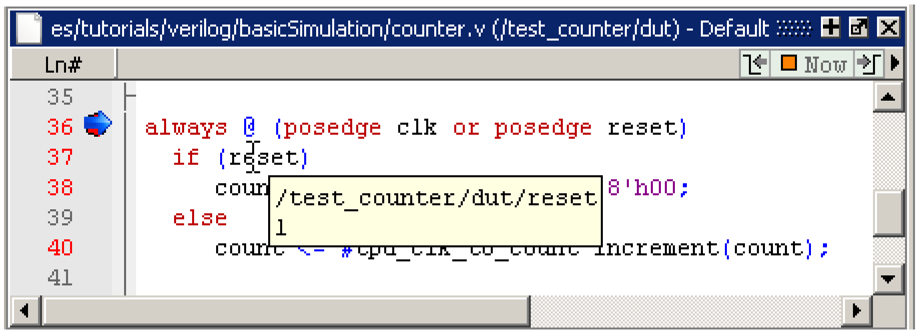
Figure 3-12. Values Shown in Objects Window



Name	Value	Kind	Mode
tpd_reset_to_count	32'h00000003	Parameter	Internal
tpd_clk_to_count	32'h00000002	Parameter	Internal
count	8'hxx	Packed Array	Out
clk	1'h0	Net	In
reset	1'h1	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 (Figure 3-13).

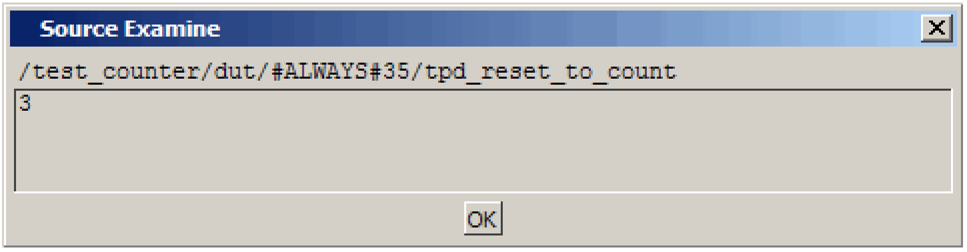
Figure 3-13. Hover Mouse Over Variable to Show Value


es/tutorials/verilog/basicSimulation/counter.v (/test_counter/dut) - Default
Ln#
Now

```
always @ (posedge clk or posedge reset)
  if (reset)
    coun /test_counter/dut/reset 8'h00;
  else 1
    count <= #tpd_clk_to_count increment(count);
```

- 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



```
/test_counter/dut/#ALWAYS#35/tpd_reset_to_count
3
```

- use the **examine** command at the VSIM> prompt to output a variable value to the Transcript window (i.e., `examine count`)

5. Try out the step commands.

   a. Click the Step Into icon on the Step 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.

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.

Chapter 4 Projects

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 an *.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 test bench.

The pathnames are as follows:

Verilog – *<install_dir>/examples/tutorials/verilog/projects/counter.v* and t*counter.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 Topics

User's Manual Chapter: Projects.

Project Work Flow

Common tasks for creating and building a new project.

Create a New Project

We'll start the process of creating a new project by defining the project settings.

Procedure

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, Questa SIM should already be running. If not, start Questa SIM.

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM 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 box 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.

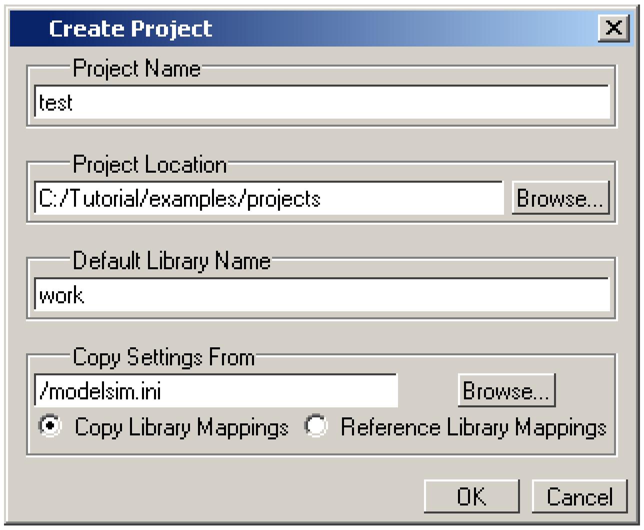
   b. Type **test** in the Project Name field.

   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 Box - Project Lab



Create Project

Project Name: test

Project Location: C:/Tutorial/examples/projects

Default Library Name: work

Copy Settings From: /modelsim.ini

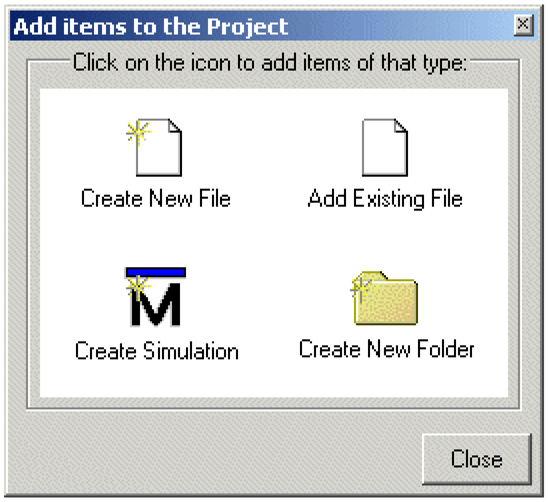
[✓ Copy Library Mappings] Reference Library Mappings

Add Objects to the Project

Once you click OK to accept the new project settings, a blank Project window and the "Add items to the Project" dialog box will appear.

From the dialog box (Figure 4-2) 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



Add items to the Project

Click on the icon to add items of that type:

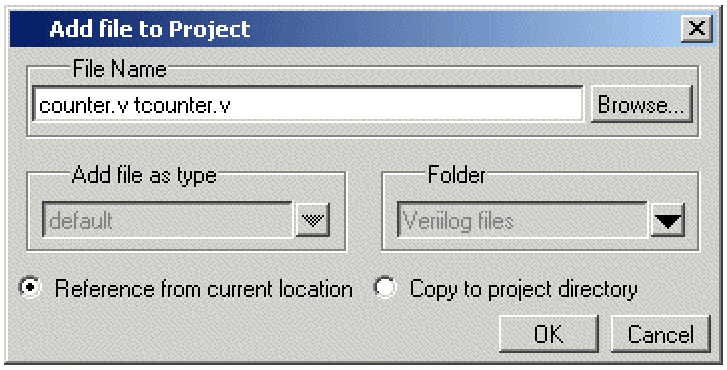
Procedure

1. Add two existing files.

    a. Click **Add Existing File**.

This opens the Add file to Project dialog box (Figure 4-3). This dialog box 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 Box


Add file to Project

File Name: counter.v tcounter.v

Add file as type: default

Folder: Verilog files

(●) Reference from current location ( ) Copy to project directory

b. Click the **Browse** button for the File Name field. This opens the “Select files to add to project” dialog box 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**.

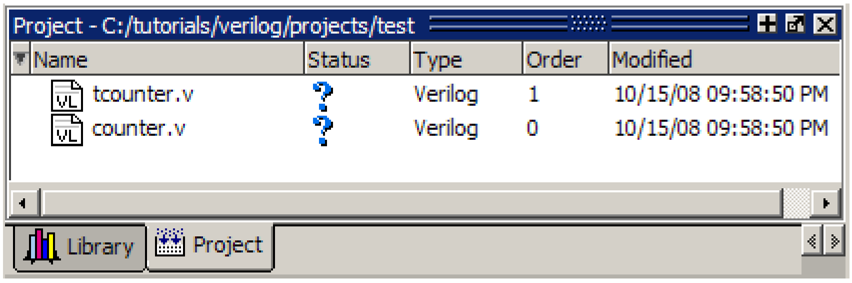
This closes the “Select files to add to project” dialog box and displays the selected files in the “Add file to Project” dialog box (Figure 4-3).

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

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

You should now see two files listed in the Project window (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


Project - C:/tutorials/verilog/projects/test

Name	Status	Type	Order	Modified
tcounter.v	?	Verilog	1	10/15/08 09:58:50 PM
counter.v	?	Verilog	0	10/15/08 09:58:50 PM

Changing Compile Order (VHDL)

By default Questa SIM performs default binding of VHDL designs when you load the design with the **vsim** command. However, you can elect to perform default binding at compile time. 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.

Procedure

1. Change the compile order.

   a. Select **Compile > Compile Order**.

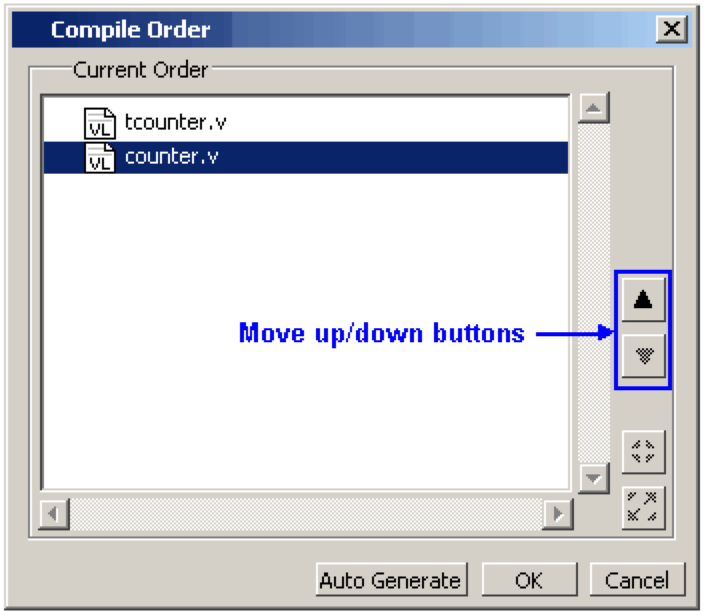
      This opens the Compile Order dialog box.

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

      Questa SIM 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 Box



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

Related Topics

For details about default binding, refer to the section Default Binding in the User’s Manual.

Compile the Design

With the Project settings defined and objects added to the project, you are ready to compile the design.

Procedure

1. Compile the files.

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

   Questa SIM 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 window.

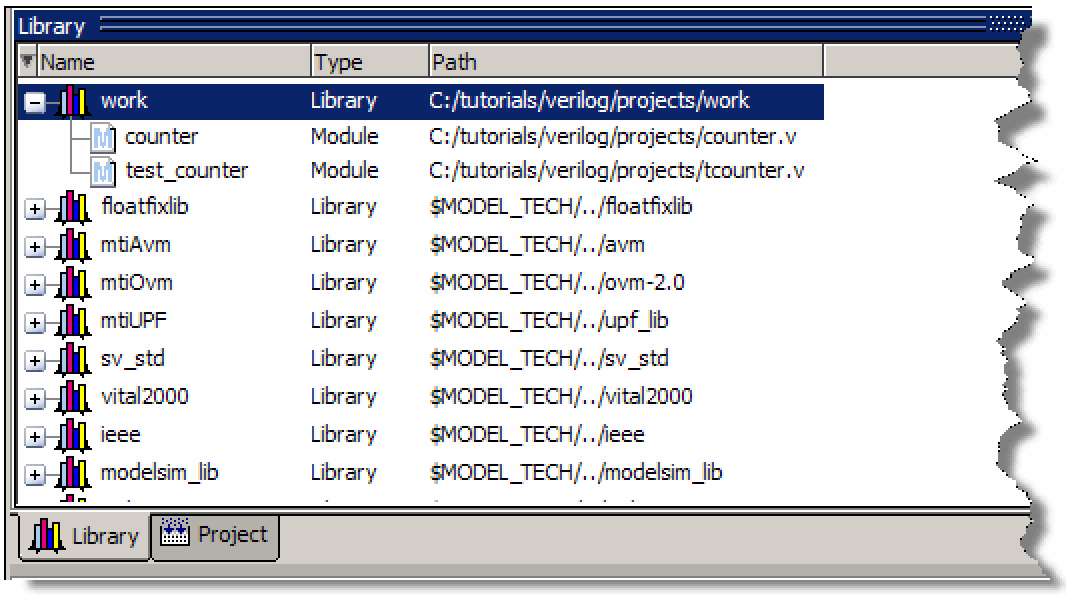
2. View the design units.

   a. Click the **Library** tab (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 Window with Expanded Library



Name	Type	Path
work	Library	C:/tutorials/verilog/projects/work
counter	Module	C:/tutorials/verilog/projects/counter.v
test_counter	Module	C:/tutorials/verilog/projects/tcounter.v
floatfixlib	Library	$MODEL_TECH/../floatfixlib
mtiAvm	Library	$MODEL_TECH/../avm
mtiOvm	Library	$MODEL_TECH/../ovm-2.0
mtiUPF	Library	$MODEL_TECH/../upf_lib
sv_std	Library	$MODEL_TECH/../sv_std
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib

Optimize for Design Visibility

Design optimization helps to decrease simulation time.

Procedure

1. Use the vopt command to optimize the design with full visibility into all design units.
   a. Enter the following command at the QuestaSim> prompt in the Transcript window:

   ```
   vopt +acc test_counter -o testcounter_opt
   ```

   The **+acc** switch provides visibility into the design for debugging purposes.

   The **-o** switch allows you designate the name of the optimized design file (testcounter_opt).

Note
You must provide a name for the optimized design file when you use the vopt command.

Load the Design

Now we’re ready to load the design into the simulator.

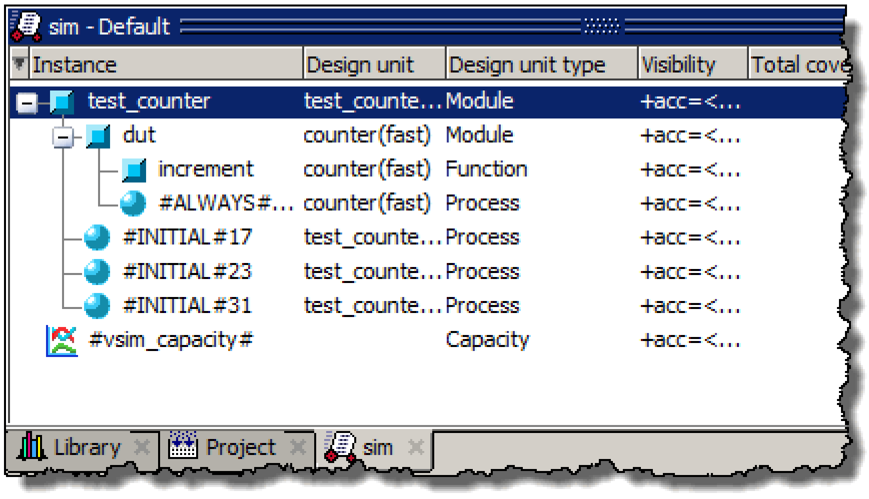
Procedure

1. Load the *test_counter* design unit.
   a. Use the optimized design name to load the design with the vsim command:

   ```
   vsim testcounter_opt
   ```

   The Structure (sim) window appears as part of the tab group with the Library and Project windows (Figure 4-7).

Figure 4-7. Structure(sim) window for a Loaded Design



Instance	Design unit	Design unit type	Visibility	Total cove
test_counter	test_counte...	Module	+acc=<...	
dut	counter(fast)	Module	+acc=<...	
increment	counter(fast)	Function	+acc=<...	
#ALWAYS#...	counter(fast)	Process	+acc=<...	
#INITIAL#17	test_counte...	Process	+acc=<...	
#INITIAL#23	test_counte...	Process	+acc=<...	
#INITIAL#31	test_counte...	Process	+acc=<...	
#vsim_capacity#		Capacity	+acc=<...	

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.

Adding Folders

As shown previously, the Add items to the Project dialog box has an option for adding folders. If you have already closed that dialog box, you can use a menu command to add a folder.

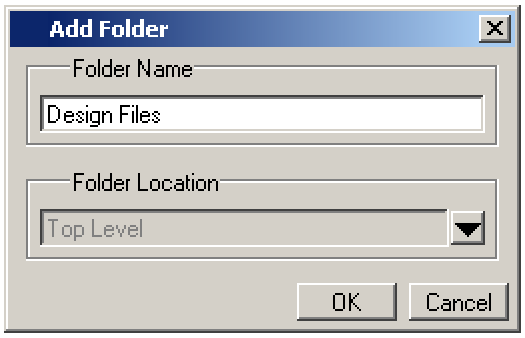
Procedure

1. Add a new folder.

   a. Right-click in the Projects window and select **Add to Project** > **Folder**.

b. Type **Design Files** in the **Folder Name** field (Figure 4-8).

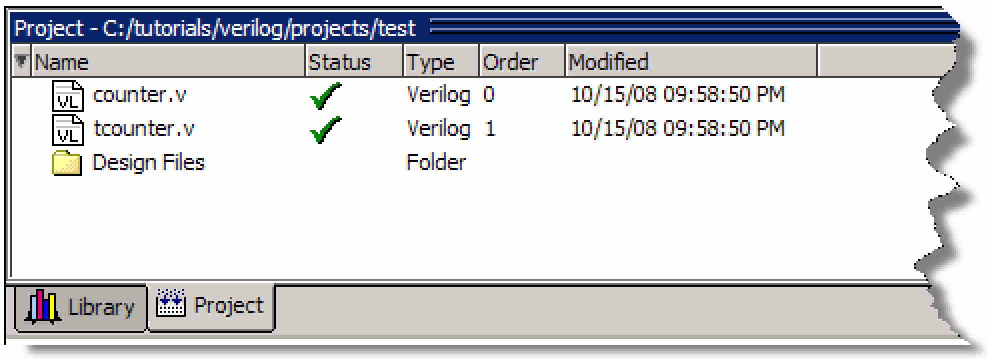
Figure 4-8. Adding New Folder to Project



c. Click **OK**.

The new Design Files folder is displayed in the Project window (Figure 4-9).

Figure 4-9. A Folder Within a Project



Project - C:/tutorials/verilog/projects/test

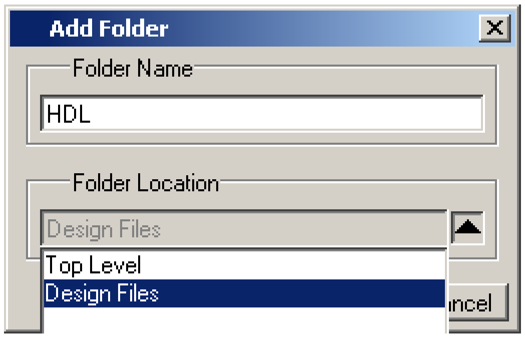
Name	Status	Type	Order	Modified
counter.v	✓	Verilog	0	10/15/08 09:58:50 PM
tcounter.v	✓	Verilog	1	10/15/08 09:58:50 PM
Design Files		Folder		

2. Add a sub-folder.

a. Right-click anywhere in the Project window and select **Add to Project > Folder**.

b. Type **HDL** in the **Folder Name** field (Figure 4-10).

Figure 4-10. Creating Subfolder

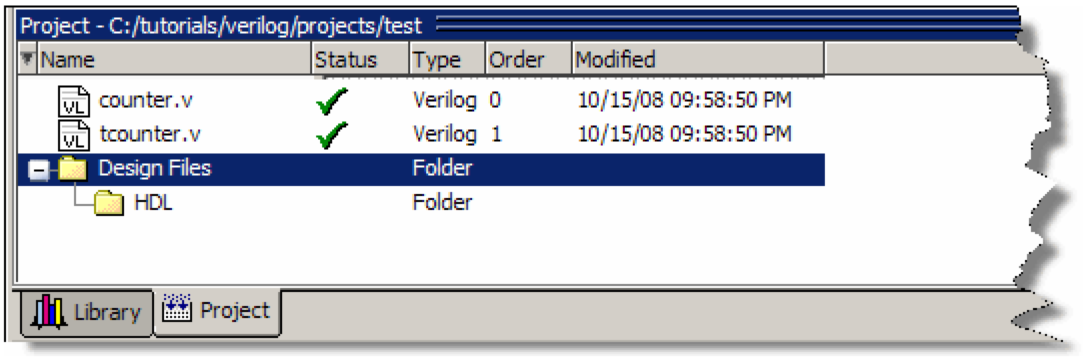


c. Click the **Folder Location** drop-down arrow and select *Design Files*.

d. Click **OK**.

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

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


Project - C:/tutorials/verilog/projects/test

Name	Status	Type	Order	Modified
counter.v	✓	Verilog	0	10/15/08 09:58:50 PM
tcounter.v	✓	Verilog	1	10/15/08 09:58:50 PM
Design Files		Folder		
HDL		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 Project Compiler Settings dialog box.

Procedure

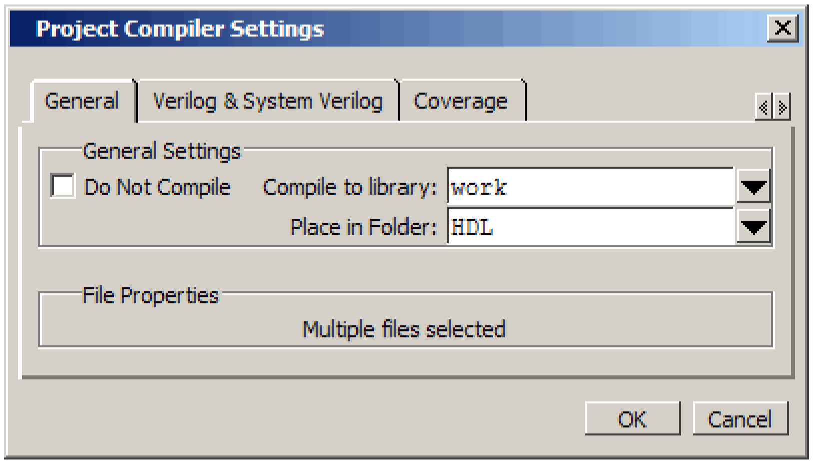
1. Move *tcounter.v* and *counter.v* to the *HDL* folder.

   a. Select both *counter.v* and *tcounter.v* in the Project window.

   b. Right-click either file and select **Properties**.

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

Figure 4-12. Changing File Location



General Settings
Do Not Compile
Compile to library: work
Place in Folder: HDL

File Properties
Multiple files selected

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.

Using 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 window and you can double-click it to load *tcounter.v* along with its options.

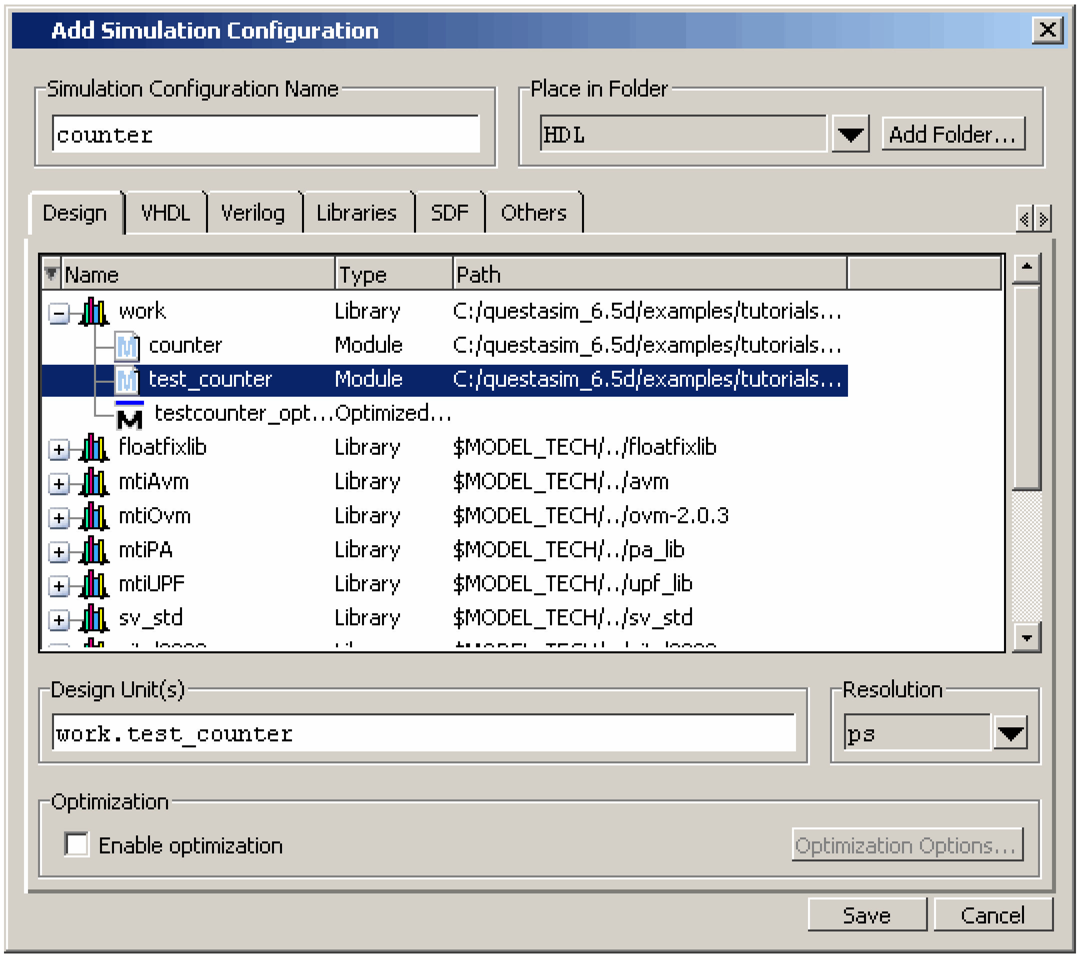
Procedure

1. Create a new Simulation Configuration.

   a. Right-click in the Project window and select **Add to Project > Simulation Configuration** from the popup menu.

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

Figure 4-13. Simulation Configuration Dialog Box



Simulation Configuration Name: counter

Place in Folder: HDL

Name	Type	Path
work	Library	C:/questasim_6.5d/examples/tutorials...
counter	Module	C:/questasim_6.5d/examples/tutorials...
test_counter	Module	C:/questasim_6.5d/examples/tutorials...
testcounter_opt...	Optimized...	
floatfixlib	Library	$MODEL_TECH/../floatfixlib
mtiAvm	Library	$MODEL_TECH/../avm
mtiOvm	Library	$MODEL_TECH/../ovm-2.0.3
mtiPA	Library	$MODEL_TECH/../pa_lib
mtiUPF	Library	$MODEL_TECH/../upf_lib
sv_std	Library	$MODEL_TECH/../sv_std

Design Unit(s): work.test_counter

Resolution: ps

Optimization
☐ Enable optimization

b. Type **counter** in the **Simulation Configuration Name** field.

c. Select *HDL* from the **Place in Folder** drop-down.

d. Click the '+' icon next to the *work* library and select *test_counter*.

e. Click the **Resolution** drop-down and select *ps*.

f. Uncheck the **Enable optimization** selection box.

g. For Verilog, click the Verilog tab and check **Enable hazard checking (-hazards)**.

h. Click **Save**.

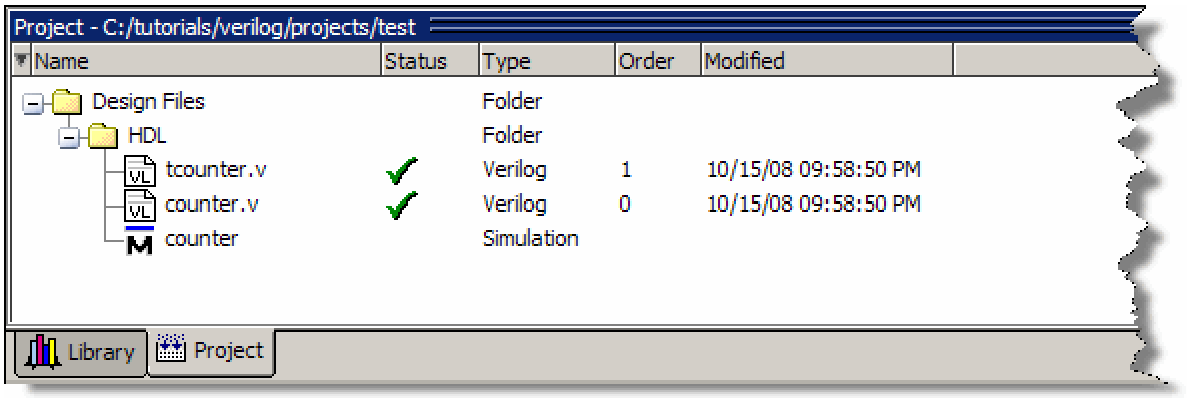
The files *tcounter.v* and *counter.v* show question mark icons in the status column because they have changed location since they were last compiled and need to be recompiled.

i. Select one of the files, *tcounter.v* or *counter.v*.

j. Select **Compile > Compile All**.

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

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



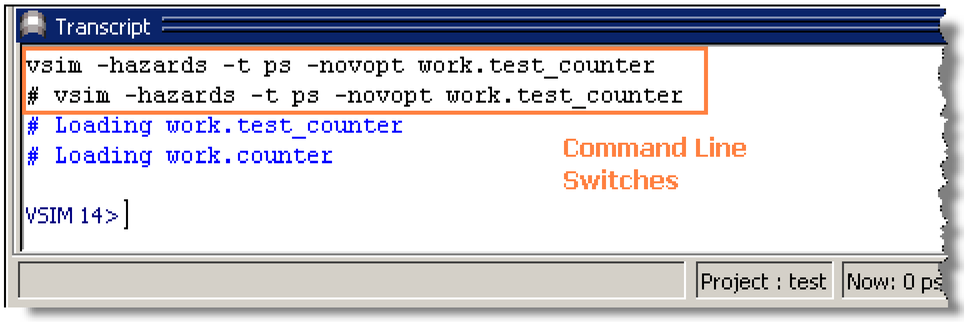
Name	Status	Type	Order	Modified
Design Files		Folder		
HDL		Folder		
tcounter.v	✓	Verilog	1	10/15/08 09:58:50 PM
counter.v	✓	Verilog	0	10/15/08 09:58:50 PM
counter		Simulation		

2. Load the Simulation Configuration.

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

   In the Transcript window of the Main window, the **vsim** (the Questa SIM 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 box.

Figure 4-15. Transcript Shows Options for Simulation Configurations



```
vsim -hazards -t ps -novopt work.test_counter
# vsim -hazards -t ps -novopt work.test_counter
# Loading work.test_counter
# Loading work.counter

VSIM 14>
```

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. In the Project window, right-click and select **Close Project**.

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

Chapter 5
Working With Multiple Libraries

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 test bench 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 test bench.

The pathnames are as follows:

Verilog – *<install_dir>/examples/tutorials/verilog/libraries/counter.v* and t*counter.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 Topics

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.

For additional information, see Permanently Mapping VHDL Resource Libraries.

Procedure

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 test bench.

   Create a new directory called *testbench* that will hold the test bench 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 Questa SIM and change to the *resource_library* directory.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box appears, click **Close**.

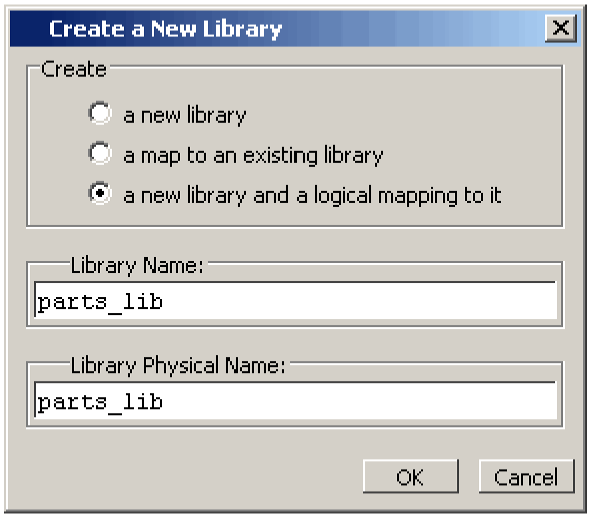
   b. Select **File > Change Directory** and change to the *resource_library* directory you created in step 1.

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



Create: [✓ a new library and a logical mapping to it]

Library Name: parts_lib

Library Physical Name: parts_lib

The Library Physical Name field is filled out automatically.

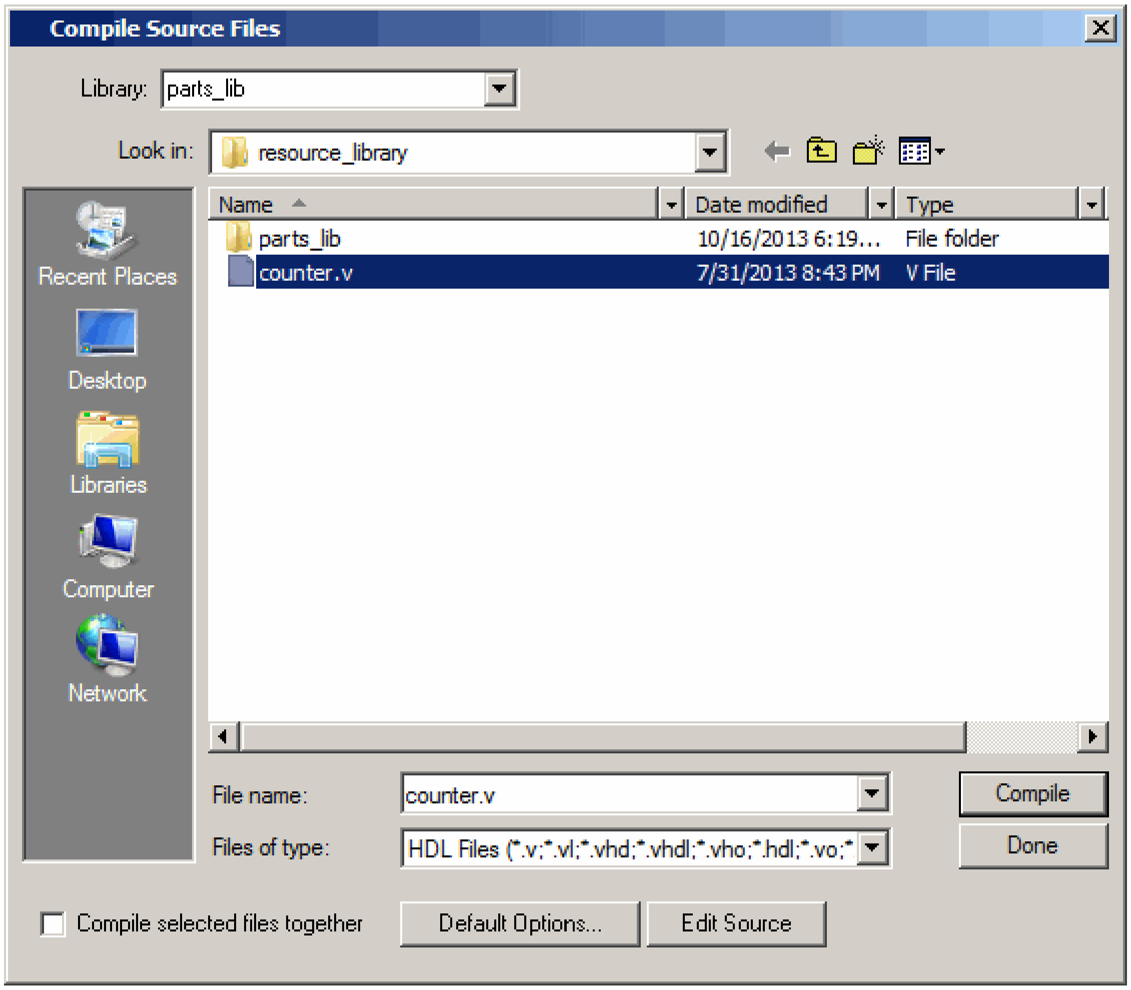
Once you click OK, Questa SIM creates a directory for the library, lists it in the Library window, and modifies the *modelsim.ini* file to record this new library for the future.

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


Compile Source Files
Library: parts_lib
Look in: resource_library
Name
Date modified
Type
parts_lib 10/16/2013 6:19... File folder
counter.v 7/31/2013 8:43 PM V File
Recent Places
Desktop
Libraries
Computer
Network
File name: counter.v
Files of type: HDL Files (*.v;*.vl;*.vhd;*.vhdl;*.vho;*.hdl;*.vo;*
Compile
Done
Compile selected files together
Default Options...
Edit Source

   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 test bench.

Procedure

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 test bench to the project.
   a. Click **Add Existing File** in the Add items to the Project dialog box.
   b. Click the **Browse** button and select *tcounter.v* in the "Select files to add to project" dialog box.
   c. Click **Open**.
   d. Click **OK**.
   e. Click **Close** to dismiss the "Add items to the Project" dialog box.

      The *tcounter.v* file is listed in the Project window.
3. Compile the test bench.
   a. Right-click *tcounter.v* and select **Compile** > **Compile Selected**.

Loading Without Linking Libraries

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

Questa SIM responds differently for Verilog and VHDL in this situation.

Verilog

Optimize the Verilog Design for Debug Visibility

Optimizing the design speeds simulation throughput.

Procedure

1. Use the vopt command to optimize with full debug visibility into all design units.

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

      vopt +acc test_counter -o testcounter_opt

      The **+acc** switch provides visibility into the design for debugging purposes.

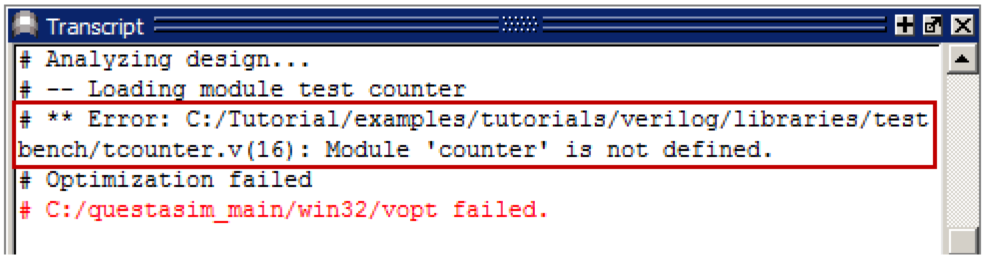
      The **-o** switch allows you designate the name of the optimized design file (testcounter_opt).

Note

You must provide a name for the optimized design file when you use the vopt command.

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

Figure 5-3. Error Reported Because Module Not Defined



```
# Analyzing design...
# -- Loading module test counter
# ** Error: C:/Tutorial/examples/tutorials/verilog/libraries/test
bench/tcounter.v(16): Module 'counter' is not defined.
# Optimization failed
# C:/questasim_main/win32/vopt failed.
```

VHDL

Optimize the VHDL Design for Debug Visibility

Optimizing the design speeds simulation throughput.

Procedure

1. Use the vopt command to optimize with full debug visibility into all design units.

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

      vopt +acc test_counter -o testcounter_opt

The **+acc** switch provides visibility into the design for debugging purposes.

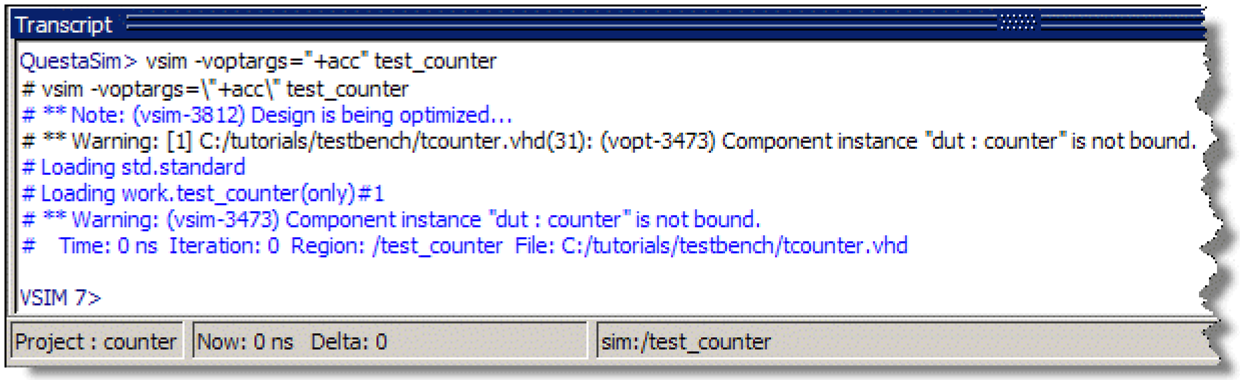
The **-o** switch allows you designate the name of the optimized design file (testcounter_opt).

Note

You must provide a name for the optimized design file when you use the vopt command.

The Main window Transcript reports a warning (Figure 5-4). When you see a message that contains text like "Warning: (vsim-3473)", you can view more detail by using the **verror** command.

Figure 5-4. VHDL Simulation Warning Reported in Main Window



```
QuestaSim> vsim -voptargs="+acc" test_counter
# vsim -voptargs=\"+acc\" test_counter
# ** Note: (vsim-3812) Design is being optimized...
# ** Warning: [1] C:/tutorials/testbench/tcounter.vhd(31): (vopt-3473) Component instance "dut : counter" is not bound.
# Loading std.standard
# Loading work.test_counter(only)#1
# ** Warning: (vsim-3473) Component instance "dut : counter" is not bound.
#    Time: 0 ns  Iteration: 0  Region: /test_counter  File: C:/tutorials/testbench/tcounter.vhd

VSIM 7>
```

```
Project : counter | Now: 0 ns   Delta: 0 | sim:/test_counter
```

b. Type **verror 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 to the Resource Library. If you are using VHDL, follow the steps in Permanently Mapping VHDL Resource Libraries one page later.

Linking to the Resource Library

Linking to a resource library requires that you specify a "search library" when you invoke the simulator.

Procedure

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.

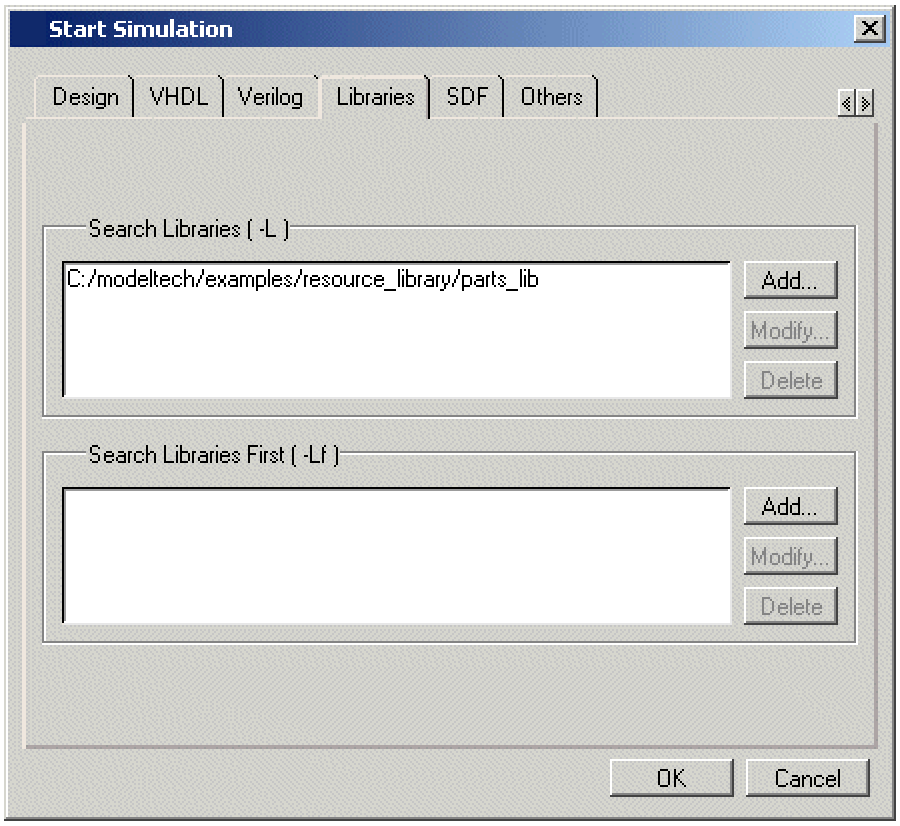
f. Click OK.

The dialog box should have *parts_lib* listed in the Search Libraries field (Figure 5-5).

g. Click OK.

The design loads without errors.

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



C:/modeltech/examples/resource_library/parts_lib

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:

Procedure

1. Locate the *modelsim.ini* file in the Questa SIM 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 window to make it active.
3. Select **File** > **Close**. Click **OK**.

Chapter 6
Simulating SystemC Designs

Questa SIM 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 Questa SIM 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 test bench 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 Topics

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 Questa SIM license file to simulate SystemC designs. Please contact your Mentor Graphics sales representatives if you currently do not have such a feature.

SystemC runs on a subset of Questa SIM supported platforms: one subset exists for SystemC-2.3 (Table 6-1) and another for SystemC-2.2 (Table 6-2).

Table 6-1. Supported Platforms for SystemC-2.3

Platform/OS	Supported compiler versions[1]	32-bit	64-bit	TLM
linux, linux_x86_64	gcc-4.7.4 gcc 4.5.0 VCO is linux (32-bit binary) VCO is linux_x86_64 (64-bit binary)	yes	yes	2.0.2
Windows[2] 7 and 8	gcc 4.2.1— VCO is win32	yes	no	2.0.2

1. Header files location: <path to questa install tree>/include/systemc/sc
2. 32-bit executable and 32-bit gcc can be used with 64-bit Windows systems, though they only run as 32-bit binaries.

Table 6-2. Supported Platforms for SystemC-2.2

Platform/OS	Supported compiler versions[1]	32-bit	64-bit	TLM
linux, linux_x86_64	gcc 4.3.3 and 4.5.0 VCO is linux (32-bit binary) VCO is linux_x86_64 (64-bit binary)	yes	yes	2.0.1
Windows[2] XP, Vista and 7	Minimalist GNU for Windows (MinGW) gcc 4.2.1— VCO is win32	yes	no	2.0.1

1. Header files location: <path to questa install tree>/include/systemc/sc22
2. 32-bit executable and 32-bit gcc can be used with 64-bit Windows systems, though they only run as 32-bit binaries.

Refer to Supported Platforms and Compiler Versions in the *Questa SIM User's Manual* for further details.

Preparing an OSCI SystemC Design

There are a few steps you must take to prepare a SystemC design to run on Questa SIM.

Prerequisite

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

- Replace **sc_main()** with an SC_MODULE, potentially adding a process to contain any test bench 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 Questa SIM simulations, we recommend that you preserve the original code by using #ifdef to add the Questa SIM-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.

Procedure

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 Questa SIM and change to the exercise directory.

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

   a. Type vsim at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box appears, click **Close**.

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

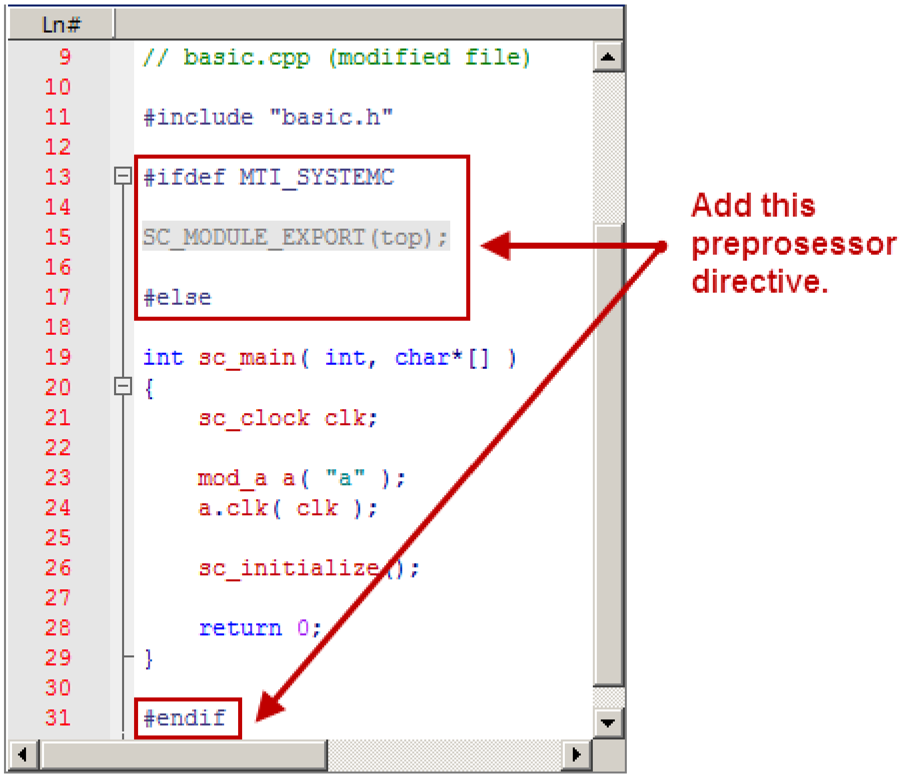
3. Use a text editor to view and edit the *basic_orig.cpp* file. To use Questa SIM's editor, from the Main Menu select **File > Open**. Change the files of type to C/C++ files then double-click *basic_orig.cpp*.

   a. If you are using Questa SIM's editor, right-click in the source code view of the *basic_orig.cpp* file and uncheck the Read Only option in the popup menu.

   b. Using the **#ifdef MTI_SYSTEMC** preprocessor directive, add the **SC_MODULE_EXPORT(top);** to the design as shown in Figure 6-1.

   c. Save the file as *basic.cpp*.

Figure 6-1. The SystemC File After Modifications



```
// basic.cpp (modified file)

#include "basic.h"

#ifdef MTI_SYSTEMC

SC_MODULE_EXPORT(top);

#else

int sc_main( int, char*[] )
{
    sc_clock clk;

    mod_a a( "a" );
    a.clk( clk );

    sc_initialize();

    return 0;
}

#endif
```

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

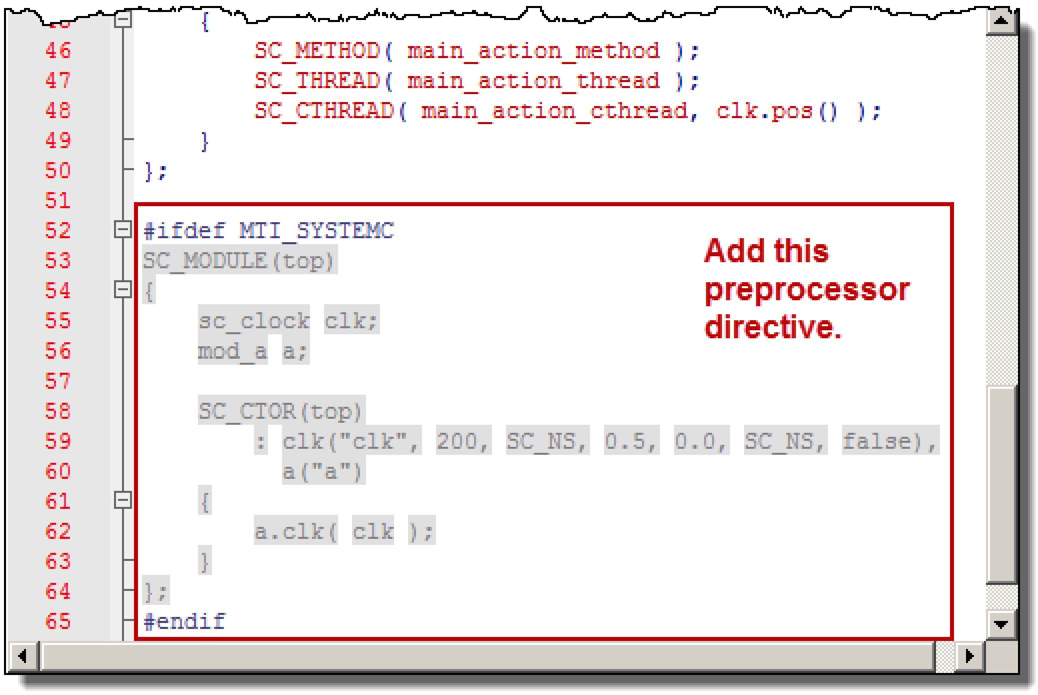
4. Edit the *basic_orig.h* header file as shown in Figure 6-2.

   a. If you are using Questa SIM's editor, right-click in the source code view of the *basic_orig.h* file and uncheck the Read Only option in the popup menu.

   b. Add a Questa SIM 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.



```
        {
            SC_METHOD( main_action_method );
            SC_THREAD( main_action_thread );
            SC_CTHREAD( main_action_cthread, clk.pos() );
        }
};

#ifdef MTI_SYSTEMC
SC_MODULE(top)
{
    sc_clock clk;
    mod_a a;

    SC_CTOR(top)
        : clk("clk", 200, SC_NS, 0.5, 0.0, SC_NS, false),
          a("a")
    {
        a.clk( clk );
    }
};
#endif
```

c. 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 the **sccom** command.

Procedure

1. Create a work library.

   a. Type **vlib work** at the Questa SIM> prompt in the Transcript window.

2. Compile and link all SystemC files.

   a. Type **sccom -g basic.cpp** at the Questa SIM> prompt.

   The **-g** argument compiles the design for debug.

   b. Type **sccom -link** at the Questa SIM> 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 test bench that instantiates an HDL module. In order for the SystemC test bench 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**.

Procedure

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 Questa SIM and change to the exercise directory.

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

   a. Type vsim at a command shell prompt.

      If the Welcome to Questa SIM dialog box 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 Questa SIM Transcript window to create the working library.

4. Compile the design.

   a. **Verilog:**
      Type **vlog *.v** in the Questa SIM Transcript window to compile all Verilog source files.

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

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

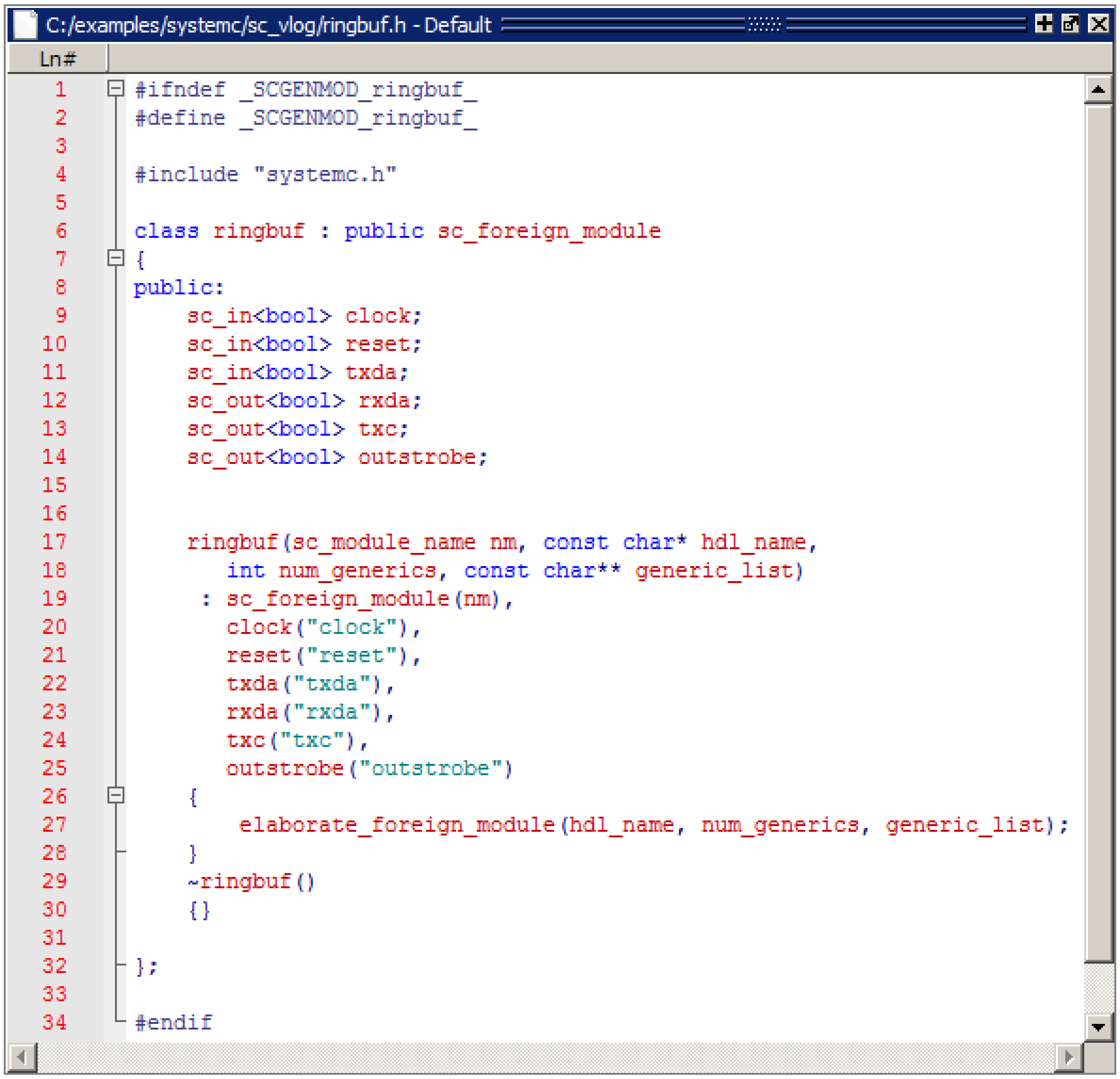
   a. **Verilog:**
   Type **scgenmod -map "scalar=bool" ringbuf > ringbuf.h** at the Questa SIM> prompt.

   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 Questa SIM> prompt.

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

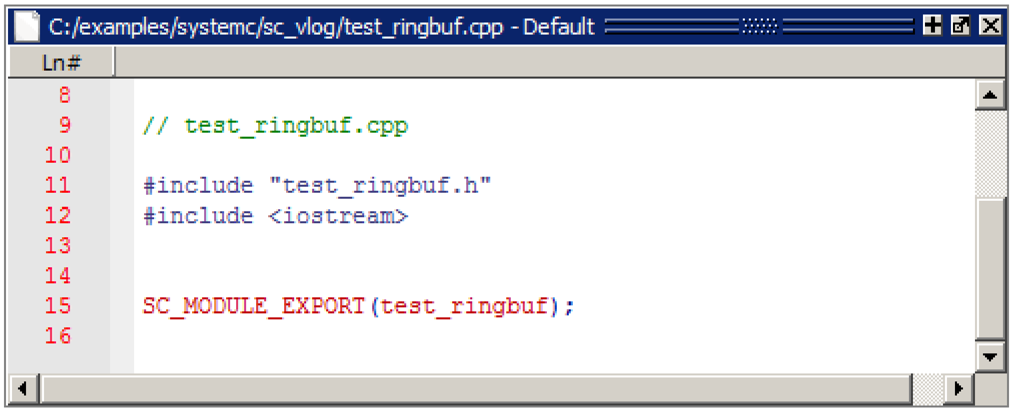
Figure 6-3. The ringbuf.h File.



```
#ifndef _SCGENMOD_ringbuf_
#define _SCGENMOD_ringbuf_

#include "systemc.h"

class ringbuf : public sc_foreign_module
{
public:
    sc_in<bool> clock;
    sc_in<bool> reset;
    sc_in<bool> txda;
    sc_out<bool> rxda;
    sc_out<bool> txc;
    sc_out<bool> outstrobe;


    ringbuf(sc_module_name nm, const char* hdl_name,
       int num_generics, const char** generic_list)
     : sc_foreign_module(nm),
       clock("clock"),
       reset("reset"),
       txda("txda"),
       rxda("rxda"),
       txc("txc"),
       outstrobe("outstrobe")
    {
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~ringbuf()
    {}

};

#endif
```

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



```

// test_ringbuf.cpp

#include "test_ringbuf.h"
#include <iostream>


SC_MODULE_EXPORT(test_ringbuf);

```

6. Compile and link all SystemC files, including the generated *ringbuf.h*.

   a. Type **sccom -g test_ringbuf.cpp** at the Questa SIM> prompt.

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

   b. Type **sccom -link** at the Questa SIM> prompt to perform the final link on the SystemC objects.

7. Optimize the design with full debug visibility.

   a. Enter the following command at the Questa SIM> prompt:

      vopt +acc test_ringbuf -o test_ringbuf_opt

      The **+acc** switch for the vopt command provides full visibility into the design for debugging purposes.

      The **-o** switch designates the name of the optimized design (test_ringbuf_opt).

Note
You must provide a name for the optimized design file when you use the vopt command.

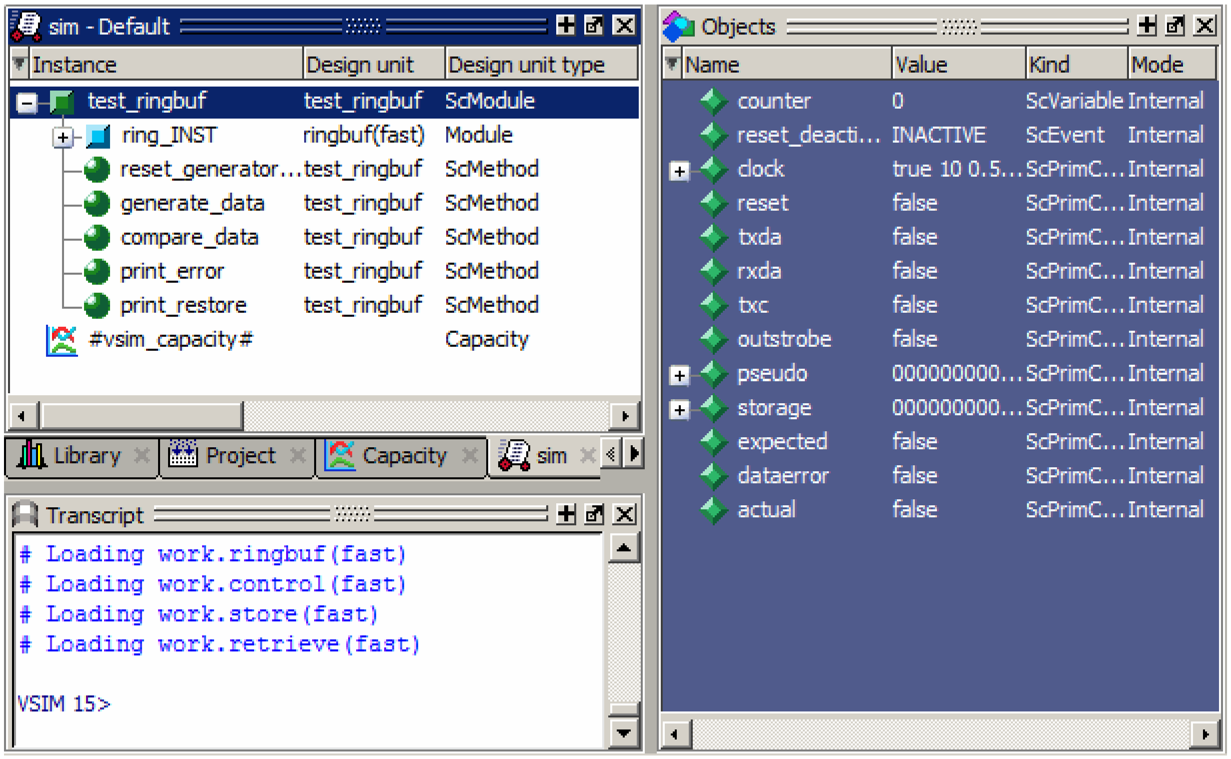
8. Load the design.

   a. Load the design using the optimized design name.

      vsim test_ringbuf_opt

9. Make sure the Objects window is open as shown in Figure 6-5. To open this window, use the **View > Objects** menu selection.

Figure 6-5. The test_ringbuf Design



Instance	Design unit	Design unit type
test_ringbuf	test_ringbuf	ScModule
ring_INST	ringbuf(fast)	Module
reset_generator...	test_ringbuf	ScMethod
generate_data	test_ringbuf	ScMethod
compare_data	test_ringbuf	ScMethod
print_error	test_ringbuf	ScMethod
print_restore	test_ringbuf	ScMethod
#vsim_capacity#		Capacity

```
# Loading work.ringbuf(fast)
# Loading work.control(fast)
# Loading work.store(fast)
# Loading work.retrieve(fast)

VSIM 15>
```

Name	Value	Kind	Mode
counter	0	ScVariable	Internal
reset_deacti...	INACTIVE	ScEvent	Internal
clock	true 10 0.5...	ScPrimC...	Internal
reset	false	ScPrimC...	Internal
txda	false	ScPrimC...	Internal
rxda	false	ScPrimC...	Internal
txc	false	ScPrimC...	Internal
outstrobe	false	ScPrimC...	Internal
pseudo	000000000...	ScPrimC...	Internal
storage	000000000...	ScPrimC...	Internal
expected	false	ScPrimC...	Internal
dataerror	false	ScPrimC...	Internal
actual	false	ScPrimC...	Internal

Viewing SystemC Objects in the GUI

SystemC objects are denoted in the Questa SIM GUI with a green ’S’ in the Library window and a green square, circle, or diamond icon elsewhere.

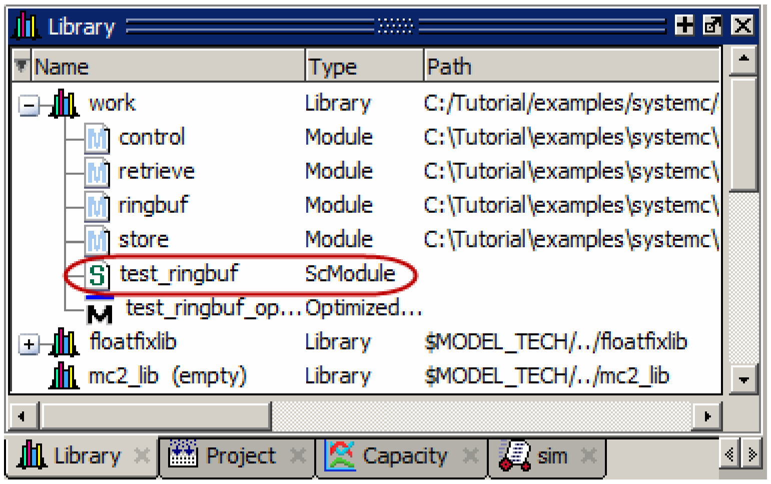
Procedure

1. View objects in the Library window.

   a. Click on the Library tab and expand the work library.

      SystemC objects have a green ’S’ next to their names (Figure 6-6).

Figure 6-6. SystemC Objects in the work Library



Name	Type	Path
work	Library	C:/Tutorial/examples/systemc/
control	Module	C:\Tutorial\examples\systemc\
retrieve	Module	C:\Tutorial\examples\systemc\
ringbuf	Module	C:\Tutorial\examples\systemc\
store	Module	C:\Tutorial\examples\systemc\
test_ringbuf	ScModule	
test_ringbuf_op...	Optimized...	
floatfixlib	Library	$MODEL_TECH/../floatfixlib
mc2_lib (empty)	Library	$MODEL_TECH/../mc2_lib

2. Add objects to the Wave window.

   a. In the Structure window (sim tab), right-click *test_ringbuf* and select **Add Wave** from the popup menu.

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, Questa SIM 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.

Procedure

1. Before we set a breakpoint, we must enable the Auto Lib Step Out feature.

   a. Select **Tools > C Debug > Allow lib step** from the Main menus.

2. Set a breakpoint.

   a. Double-click *test_ringbuf* in the Structure (sim) window 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 the red line number of the line containing (shown in Figure 6-7):

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

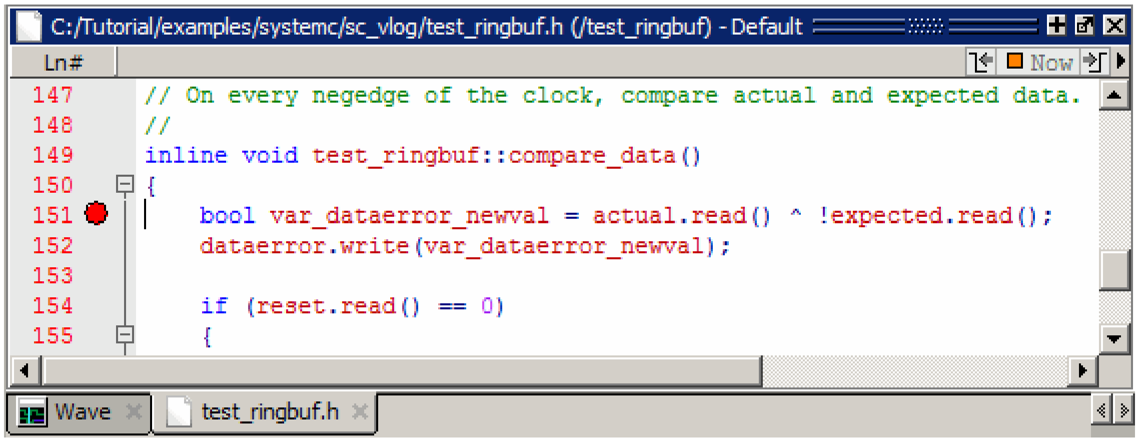
VHDL: `sc_logic var_dataerror_newval = acutal.read ...`

Note

Questa SIM 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, Questa SIM places a solid red dot next to the line number (Figure 6-7).

Figure 6-7. Active Breakpoint in a SystemC File


C:/Tutorial/examples/systemc/sc_vlog/test_ringbuf.h (/test_ringbuf) - Default
Ln#
Now
Wave
test_ringbuf.h

```
147      // On every negedge of the clock, compare actual and expected data.
148      //
149      inline void test_ringbuf::compare_data()
150      {
151 ●        bool var_dataerror_newval = actual.read() ^ !expected.read();
152          dataerror.write(var_dataerror_newval);
153
154          if (reset.read() == 0)
155          {
```

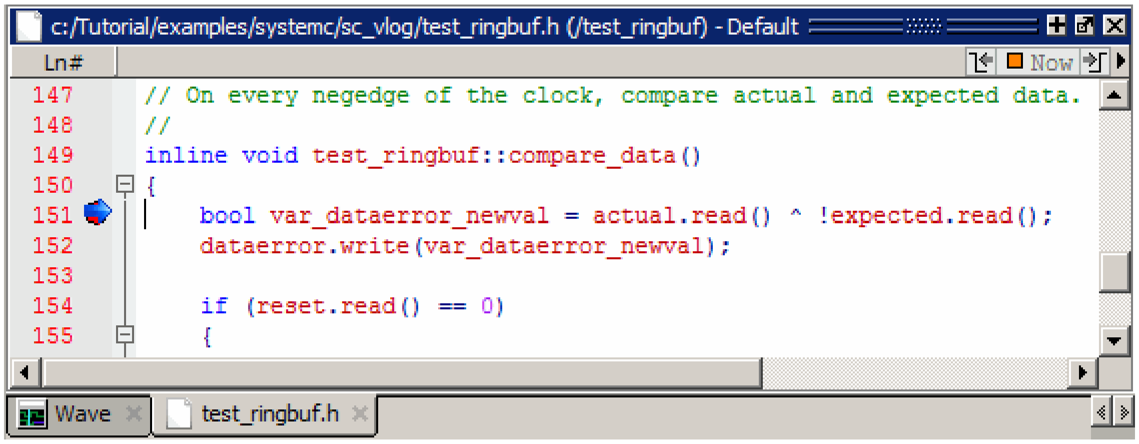
3. Run and step through the code.

   a. Type **run 500** at the VSIM> prompt and press the Enter key.

   When the simulation hits the breakpoint it stops running, highlights the line with a blue arrow in the Source window (Figure 6-8), and issues a message like this in the Transcript:

```
# C breakpoint c.1
# test_ringbuf::compare_data (this=0x27c4d08) at test_ringbuf.h:151
```

Figure 6-8. Simulation Stopped at Breakpoint

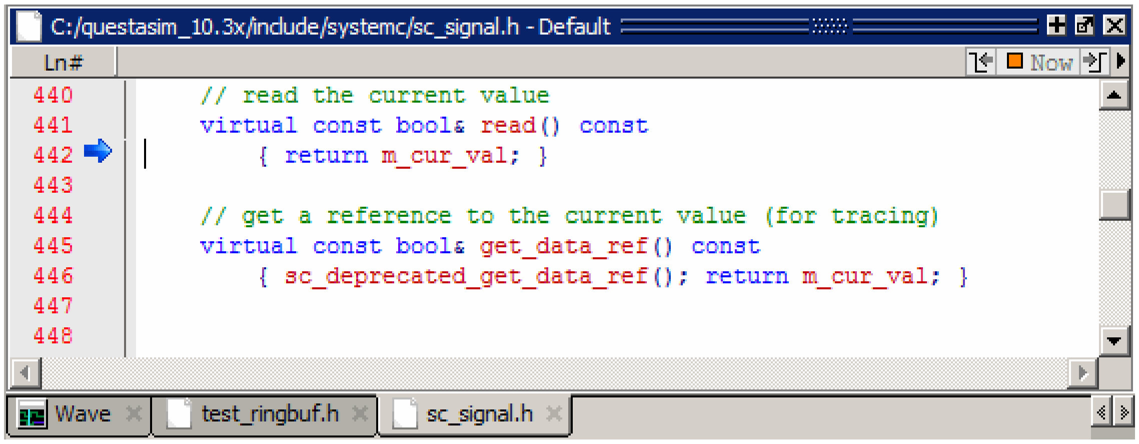

c:/Tutorial/examples/systemc/sc_vlog/test_ringbuf.h (/test_ringbuf) - Default
Ln#
Now
Wave
test_ringbuf.h

```
// On every negedge of the clock, compare actual and expected data.
//
inline void test_ringbuf::compare_data()
{
    bool var_dataerror_newval = actual.read() ^ !expected.read();
    dataerror.write(var_dataerror_newval);

    if (reset.read() == 0)
    {
```

b. Click the Step Into icon on the Step Toolbar.

This steps the simulation to the next statement. Because the next statement is a function call, Questa SIM steps into the function, which is in a separate file — *sc_signal.h* (Figure 6-9).

Figure 6-9. Stepping into a Separate File



```
    // read the current value
    virtual const bool& read() const
        { return m_cur_val; }

    // get a reference to the current value (for tracing)
    virtual const bool& get_data_ref() const
        { sc_deprecated_get_data_ref(); return m_cur_val; }
```

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.

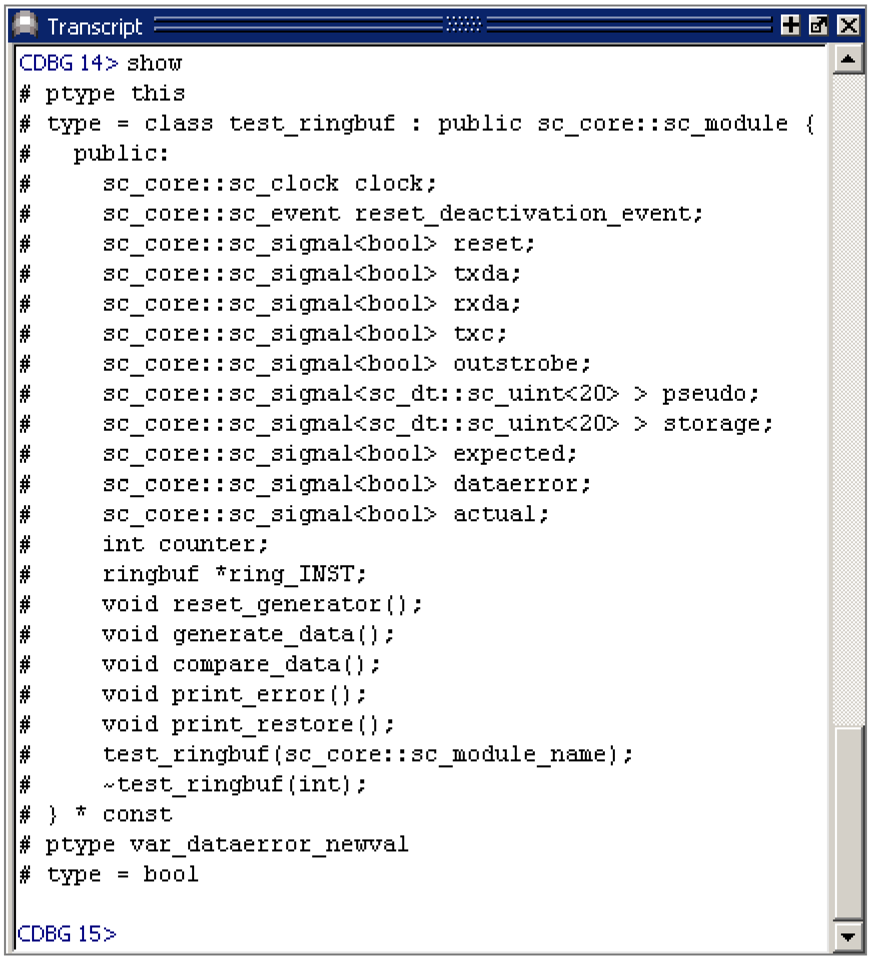
Procedure

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 6-10).

Figure 6-10. Output of show Command



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

Removing a Breakpoint

You can easily remove a breakpoint and rerun the simulation.

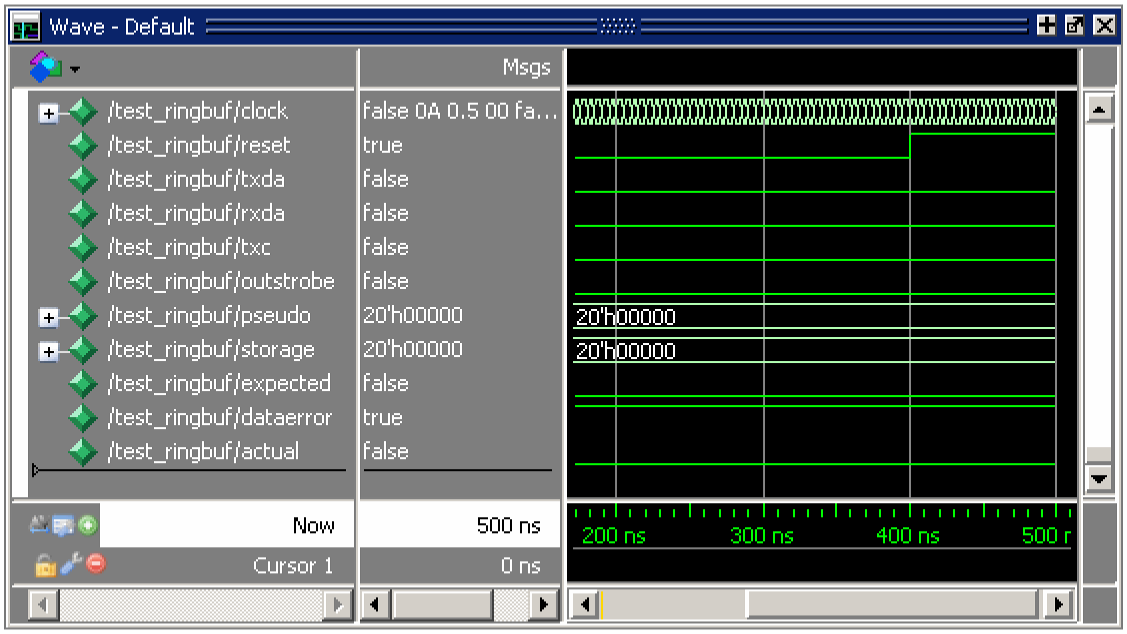
Procedure

1. Return to the Source window for test_ringbuf.h and right-click the red dot 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 6-11).

   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 6-11. SystemC Primitive Channels in the Wave Window



	Msgs
/test_ringbuf/clock	false 0A 0.5 00 fa...
/test_ringbuf/reset	true
/test_ringbuf/txda	false
/test_ringbuf/rxda	false
/test_ringbuf/txc	false
/test_ringbuf/outstrobe	false
/test_ringbuf/pseudo	20'h00000
/test_ringbuf/storage	20'h00000
/test_ringbuf/expected	false
/test_ringbuf/dataerror	true
/test_ringbuf/actual	false
Now	500 ns
Cursor 1	0 ns

Lesson Wrap-up

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

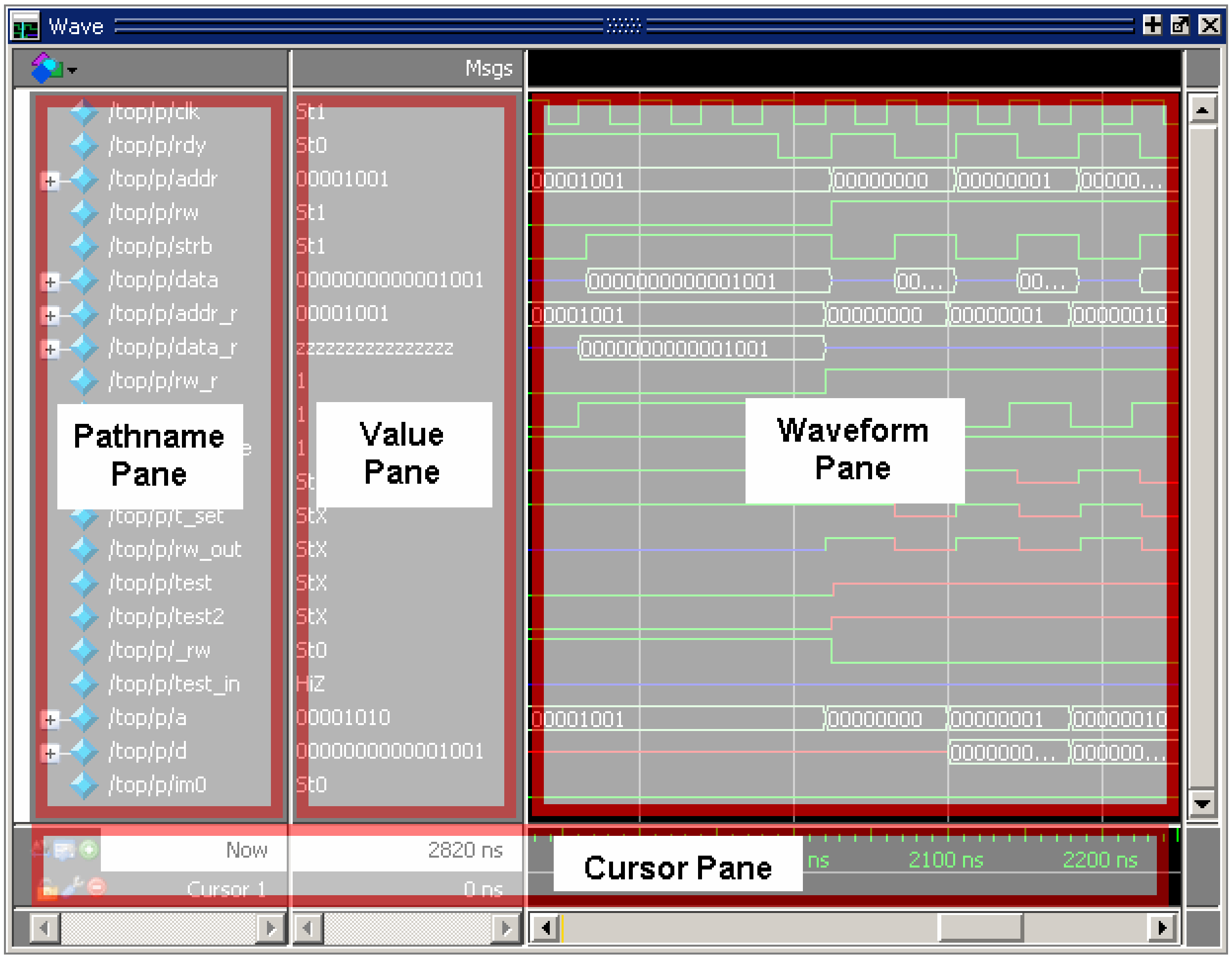
1. Select **Tools > C Debug > Quit C Debug**.
2. Select **Simulate > End Simulation**. Click **Yes** when prompted to confirm that you wish to quit simulating.

Chapter 7
Analyzing Waveforms

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



Loading a Design

For the examples in this exercise, we will use the design simulated in the Basic Simulation lesson.

Procedure

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

    a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

    If the Welcome to Questa SIM dialog box 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. Use the optimized design name to load the design with vsim.

    ```
    vsim testcounter_opt
    ```

    Questa SIM loads the design and opens a Structure (sim) window.

Add Objects to the Wave Window

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

Procedure

1. Add objects from the Objects window.

    a. Open an Objects window by selecting **View** > **Objects**.

    b. Select an item in the Objects window, right-click, and then select **Add to** > **Wave** > **Signals in Region**. Questa SIM opens a Wave window and displays signals in the region.

    - Or, place the cursor over an object and click the right mouse button to open a context menu. Then click **Add Wave** to place the object in the Wave window.

    - Or, select a group of objects then click the right mouse button to open the context menu and click **Add Wave**.

2. Undock the Wave window.

By default Questa SIM opens the Wave window in the right side of the Main window. You can change the default via the Preferences dialog box (**Tools** > **Edit Preferences**). Refer to the Setting GUI Preferences section in the Questa SIM Graphical User Interface (GUI) Reference Manual for more information.

a. Click the undock icon on the Wave window.

The Wave window becomes a standalone, un-docked window. Resize the window as needed.

3. Add objects using drag-and-drop.

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

a. In the Wave window, select **Edit** > **Select All** and then **Edit** > **Delete**.

b. Drag an instance from the Structure (sim) window to the Wave window.

Questa SIM adds the objects for that instance to the Wave window.

c. Drag a signal from the Objects window to the Wave window.

d. In the Wave window, select **Edit** > **Select All** and then **Edit** > **Delete**.

4. Add objects using the add wave command.

a. Type the following at the VSIM> prompt.

```
add wave *
```

Questa SIM adds all objects from the current selected region.

b. Run the simulation for 500 ns so you can see waveforms.

Zooming the Waveform Display

There are numerous methods for zooming the Waveform display. This exercise will show you how to zoom using various techniques.

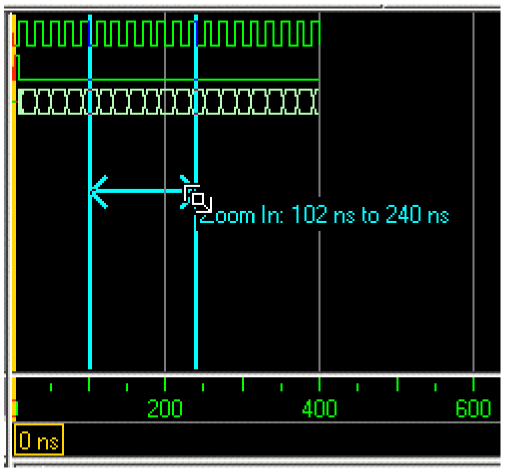
Procedure

1. Click the Zoom Mode icon on the Wave window toolbar.

a. In the waveform display, click and drag down and to the right.

b. You should see blue vertical lines and numbers defining an area to zoom in (Figure 7-2).

Figure 7-2. Zooming in with the Zoom Mode Mouse Pointer



2. Select **View** > **Zoom** > **Zoom Last**.
   a. The waveform display restores the previous display range.
3. Click the Zoom In icon a few times.
4. In the waveform display, click and drag up and to the right.

   You should see a blue line and numbers defining an area to zoom out.
5. Select **View** > **Zoom** > **Zoom Full**.

Using Cursors in the Wave Window

Cursors mark simulation time in the Wave window. When Questa SIM first draws the Wave window, it places one cursor at time zero. Clicking in the cursor timeline brings the 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

Let's look at the information provided when using a single cursor.

Procedure

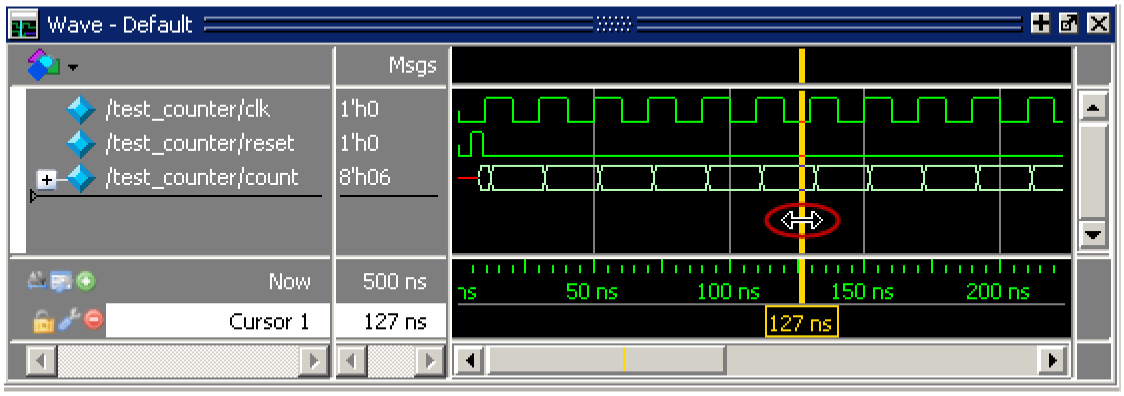
1. Position the cursor by clicking in the cursor timeline then dragging.

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

   b. Click anywhere in the cursor timeline.

   The cursor snaps to the time where you clicked (Figure 7-3).

Figure 7-3. 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, position the mouse pointer over the cursor line. When the pointer changes to a two headed arrow (Figure 7-3), click and hold the left mouse button to select the cursor. Drag the cursor to the right of a transition.

   The cursor "snaps" to the nearest transition to the left when you release the mouse button. Cursors "snap" to a waveform edge when you drag a cursor to within ten pixels of an edge. You can set the snap distance in the Window Preferences dialog box (select **Tools** > **Window Preferences**).

   e. In the cursor timeline pane, select the yellow timeline indicator box then drag the cursor to the right of a transition (Figure 7-3).

   The cursor does not snap to a transition when you drag in the timeline pane.

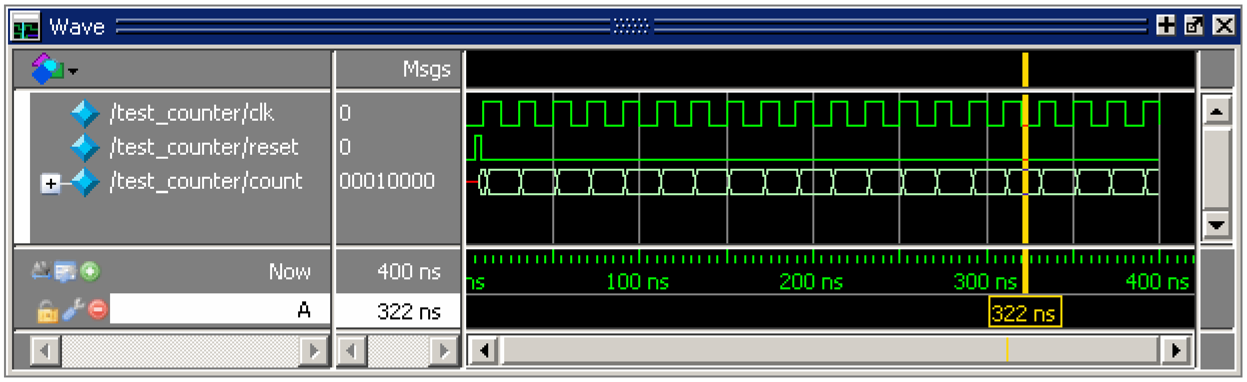
2. Rename the cursor.

   a. Right-click "Cursor 1" in the cursor pane, then select and delete the text.

   b. Type **A** and press Enter.

   The cursor name changes to "A" (Figure 7-4).

Figure 7-4. 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 selected signal.

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

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

Working with Multiple Cursors

Even more information is available when workinng with multiple cursors.

Procedure

1. Add a second cursor.

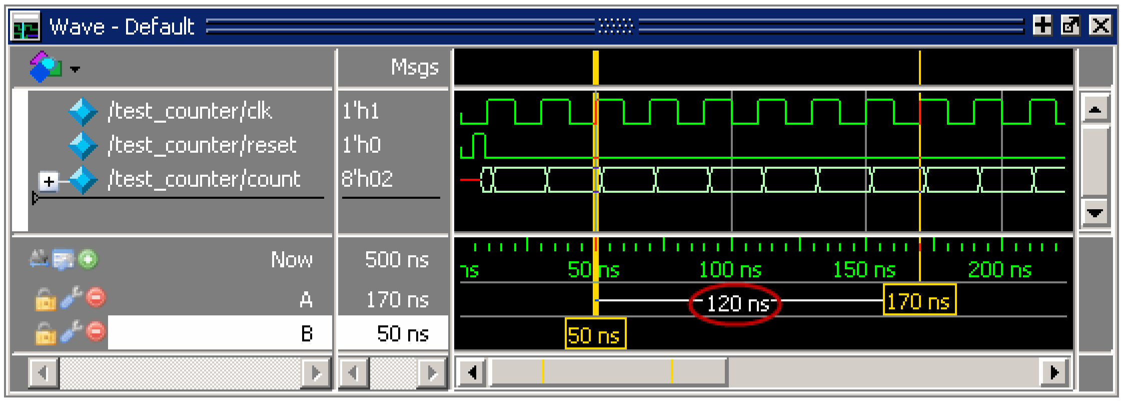
   a. Click the Insert Cursor icon on the Wave window toolbar.

   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 7-5).

Figure 7-5. Interval Measurement Between Two Cursors



	Msgs
/test_counter/clk	1'h1
/test_counter/reset	1'h0
/test_counter/count	8'h02
Now	500 ns
A	170 ns
B	50 ns

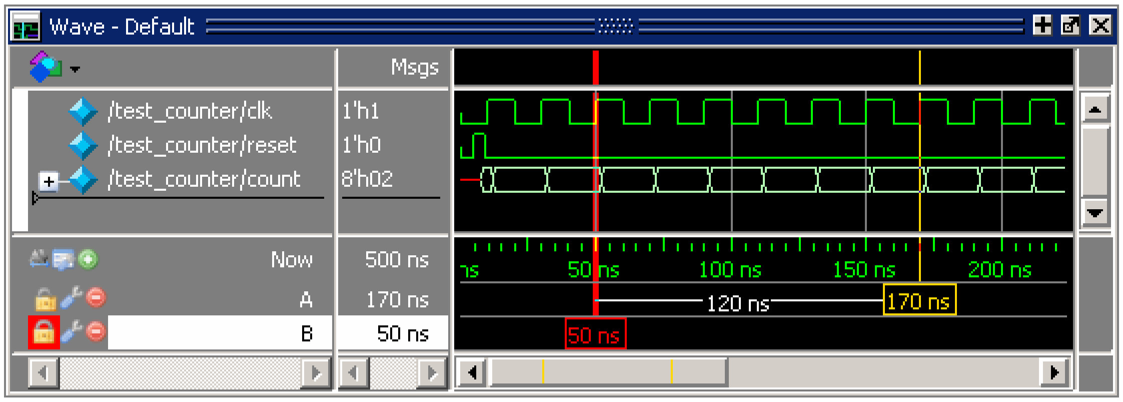
2. Lock cursor *B*.

   a. Right-click the yellow time indicator box associated with cursor *B* (at 56 ns).

   b. Select **Lock B** from the popup menu.

      The cursor color changes to red and you can no longer drag the cursor (Figure 7-6).

Figure 7-6. A Locked Cursor in the Wave Window



	Msgs
/test_counter/clk	1'h1
/test_counter/reset	1'h0
/test_counter/count	8'h02
Now	500 ns
A	170 ns
B	50 ns

3. Delete cursor *B*.

   a. Right-click cursor *B* (the red box at 56 ns) 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.

Procedure

1. Save a format file.

   a. In the Wave window, select **File** > **Save Format**.

   b. In the Pathname field of the Save Format dialog box, 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 box, select *wave.do* and click **Open**.

   Questa SIM 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.

Related Topics

User's Manual sections: Wave Window and Recording Simulation Results With Datasets

Chapter 8
Creating Stimulus With Waveform Editor

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:

- Create a new directory and copy the *counter* design unit into it.
- Load the *counter* design unit without a test bench.
- Create waves via a wizard.
- Edit waves interactively in the Wave window.
- Export the waves to an HDL test bench and extended VCD file.
- Run the simulation.
- Re-simulate using the exported test bench and VCD file.

Design Files for this Lesson

The sample design for this lesson is a simple 8-bit, binary up-counter that was used in the Basic Simulation lesson.

The pathnames are as follows:

Verilog - *<install_dir>/examples/tutorials/verilog/basicSimulation*

VHDL - *<install_dir>/examples/tutorials/vhdl/basicSimulation*

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 Topics

User’s Manual Section: Generating Stimulus with Waveform Editor and Wave Window in the Questa SIM Graphical User Reference (GUI) Reference Manual.

Compile and Load the Design

Before using the Waveform Editor we'll compile and load a design.

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.

Procedure

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 the file *counter.v* from *<install_dir>/examples/tutorials/verilog/basicSimulation* to the new directory.

   If you have a VHDL license, copy the file *counter.vhd* from *<install_dir>/examples/tutorials/vhdl/basicSimulation* to the new directory.

2. Start Questa SIM and change to the directory you created for this lesson in step 1.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box 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 Questa SIM> prompt.

   b. Compile the design file:

      Verilog:
      Type **vlog counter.v** at the Questa SIM> prompt.

      VHDL:
      Type **vcom counter.vhd** at the Questa SIM> prompt.

4. Load the design unit.

   a. Select Simulate > Start Simulation to open the Start Simulation dialog box.

   b. Click the '+' sign next to the work library to open it.

   c. Select the **counter** module.

   d. Uncheck the **Enable optimization** box.

   e. Click the OK button.

5. Open a Wave and an Objects window. (If these windows are already open, skip this step.)

   a. Select **View** > **Wave** from the Main window menus.

   b. Select **View** > **Objects**.

Create Graphical Stimulus with a Wizard

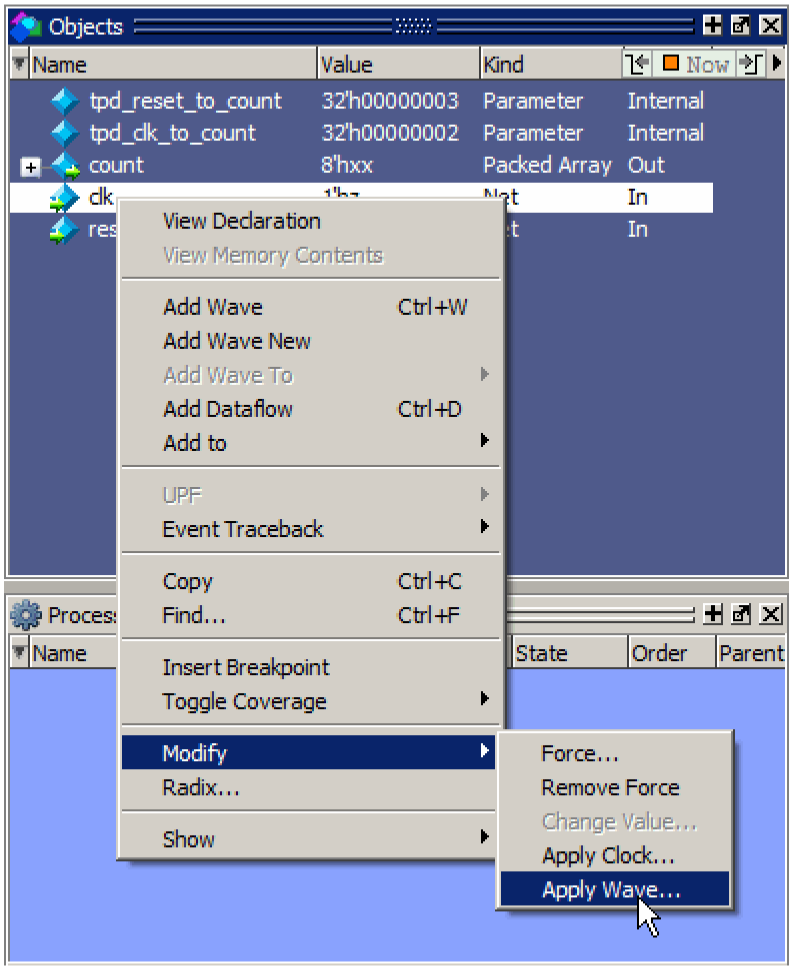
Waveform Editor includes a Create Pattern Wizard that walks you through the process of creating editable waveforms.

Procedure

1. Use the Create Pattern Wizard to create a clock pattern.

   a. In the Objects window, right click the *clk* signal and select **Modify** > **Apply Wave** (Figure 8-1).

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

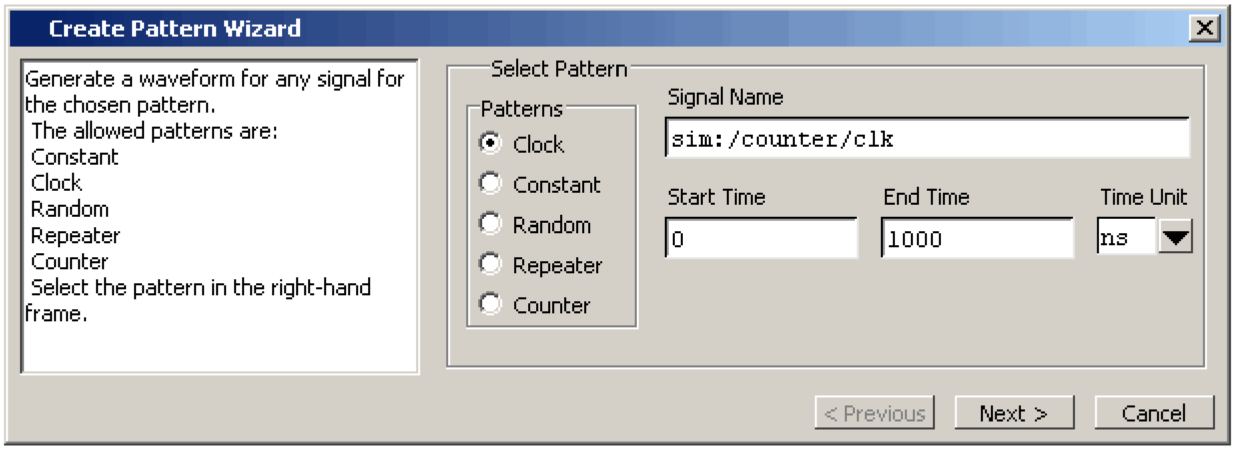


Name	Value	Kind	
tpd_reset_to_count	32'h00000003	Parameter	Internal
tpd_clk_to_count	32'h00000002	Parameter	Internal
count	8'hxx	Packed Array	Out
clk	1'hz	Net	In
res		t	In

This opens the Create Pattern Wizard dialog box 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



Generate a waveform for any signal for the chosen pattern.
The allowed patterns are:
Constant
Clock
Random
Repeater
Counter
Select the pattern in the right-hand frame.

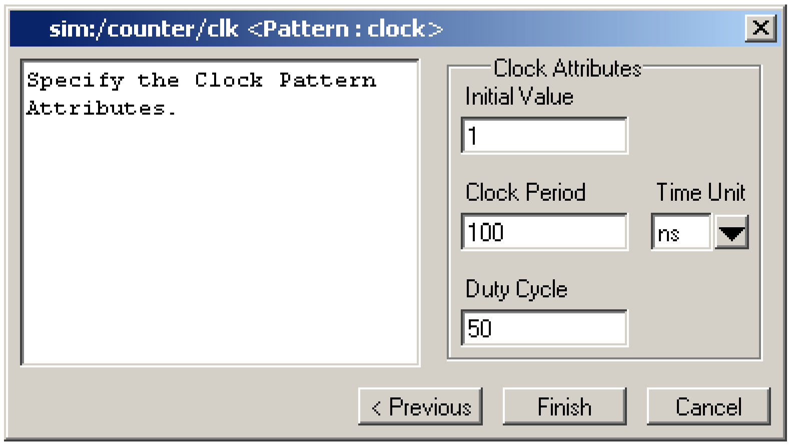
Patterns: [✓ Clock] Constant Random Repeater Counter

Signal Name: sim:/counter/clk

Start Time: 0
End Time: 1000
Time Unit: ns

c. In the second dialog box 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

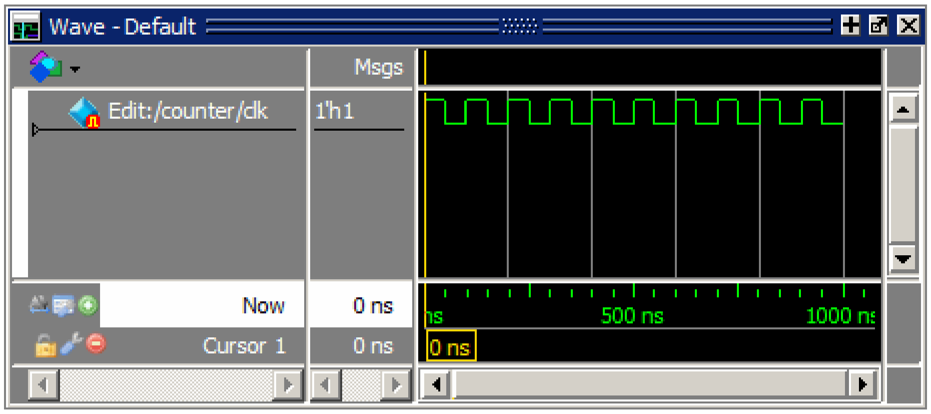

sim:/counter/clk <Pattern : clock>

Specify the Clock Pattern Attributes.

Clock Attributes
Initial Value: 1
Clock Period: 100
Time Unit: ns
Duty Cycle: 50

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.

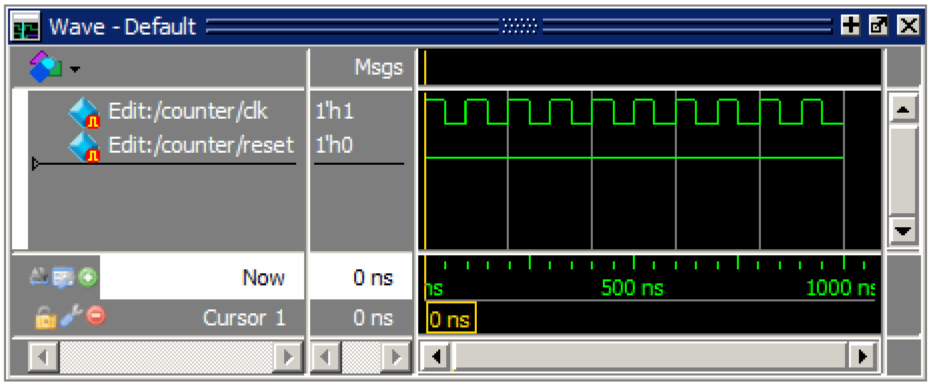
   a. Right-click signal *reset* in the Objects window and select **Modify > Apply Wave** from the popup menu.

   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.

Procedure

1. Insert a pulse on signal *reset*.

   a. Click the Wave window title bar to make the Wave window active.

   b. Click the Edit Mode icon in the toolbar.

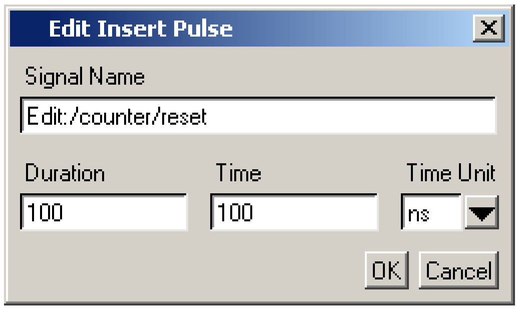
   c. In the Wave window Pathnames column, click the *reset* signal so it is selected.

   d. Click the Insert Pulse icon in the Wave Edit Toolbar.

   Or, in the Wave window, right-click on the *reset* signal waveform (not the pathname or value) and select **Wave > Wave Editor > Insert Pulse**.

   e. In the Edit Insert Pulse dialog box, 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 Box


Edit Insert Pulse

Signal Name: Edit:/counter/reset

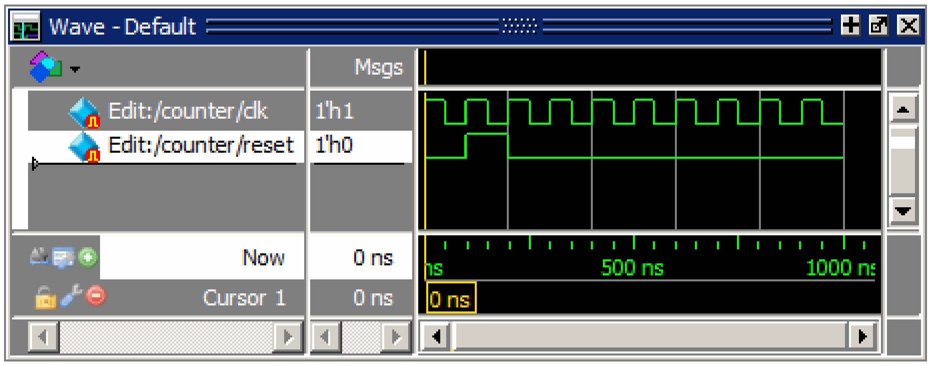
Duration: 100

Time: 100

Time Unit: ns

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. Select the *clk* signal by clicking on its name in the Pathnames column.

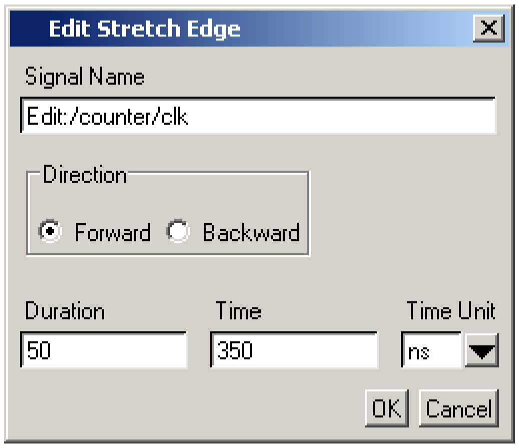
   b. In the waveform pane, click the *clk* waveform slightly to the right of the transition at 350 ns of the signal *clk*. The cursor should snap to the transition at 350 ns. If the yellow cursor line is not visible, click anywhere in the cursor timeline to move the cursor into the current view.

   c. Right-click that same transition and select **Wave Editor > Stretch Edge** from the popup menu.

      If the command is dimmed out, the cursor probably is not on the edge at 350 ns.

   d. In the Edit Stretch Edge dialog box, 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 Box


Edit Stretch Edge

Signal Name: Edit:/counter/clk

Direction: [✓ Forward] Backward

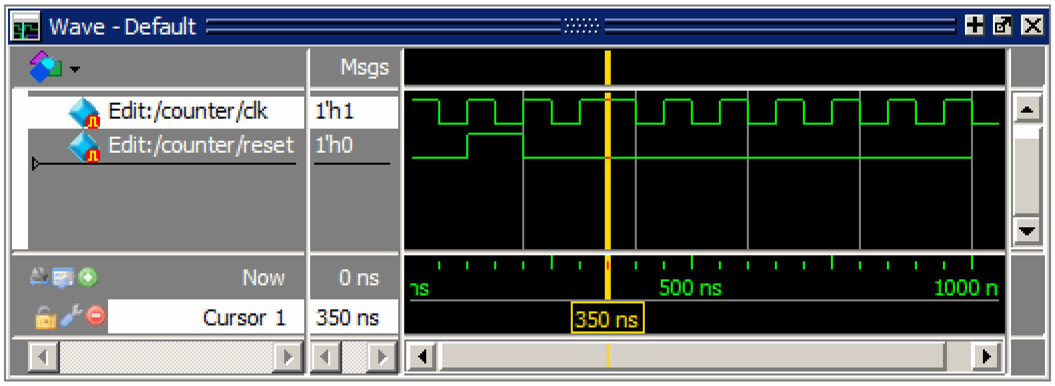
Duration: 50

Time: 350

Time Unit: ns

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.

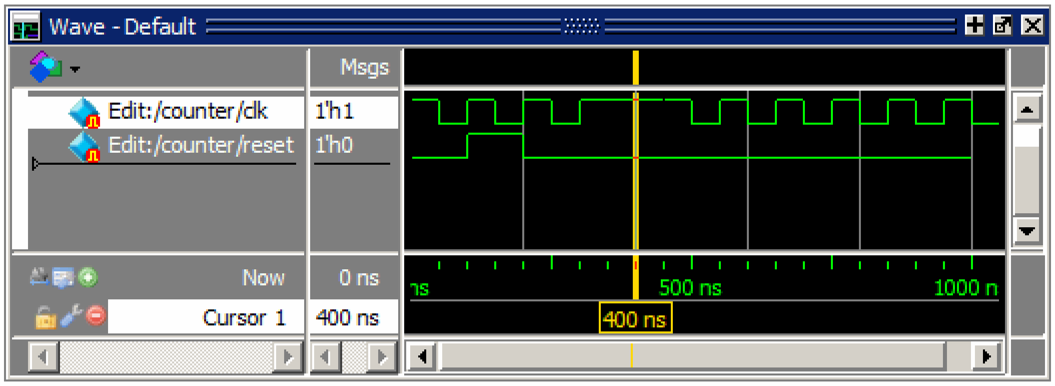
3. Delete an edge.

   a. Click the *clk* waveform 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 box. 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 box 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 in the waveform display.

b. Reselect the *clk* signal to activate the Redo icon.

c. Click the Redo icon.

d. Click **OK** in the Edit Redo dialog box.

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

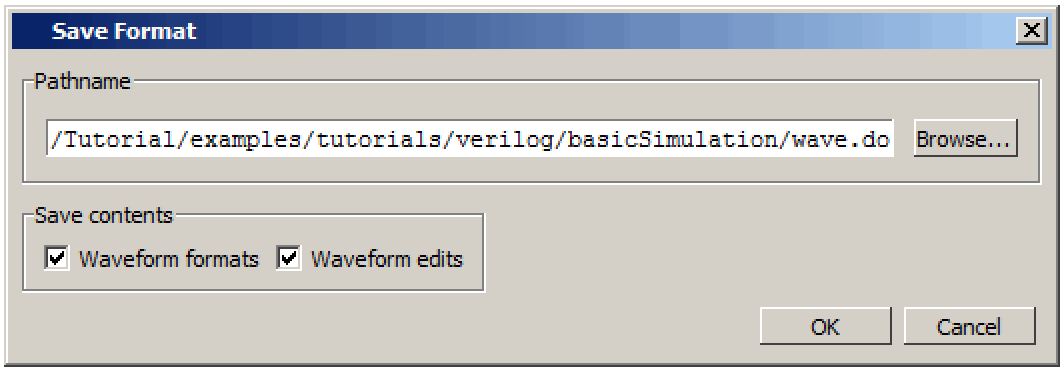
You can save the commands that Questa SIM used to create the waveforms. You can load this "format" file at a later time to re-create the waves. In this exercise, we will save the commands, quit and reload the simulation, and then open the format file.

Procedure

1. Save the wave commands to a format file.

   a. Select **File > Save Format** in the menu bar to open the Save Format dialog box.

Figure 8-11. Save Format Dialog Box



Save Format

Pathname

/Tutorial/examples/tutorials/verilog/basicSimulation/wave.do

Browse...

Save contents

✓ Waveform formats ✓ Waveform edits

OK Cancel

   The default file name is *wave.do.*

   b. Click the OK button to save a DO file named *wave.do* to the current directory.

   c. Close the Wave window by clicking Close icon (x) in the top right corner or by selecting **View > Wave** in the menus.

2. Quit and then reload the design.

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

   b. Enter the following command at the Questa SIM> prompt.

   ```
   vsim counter
   ```

3. Open the format file.

   a. Select **View** > **Wave** to open the Wave window.

   b. Select **File** > **Load** > **Macro File** from the menu bar.

   c. Double-click *wave.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 export the created waveforms so you can use them later in the lesson.

Procedure

1. Export the created waveforms in an HDL test bench format.

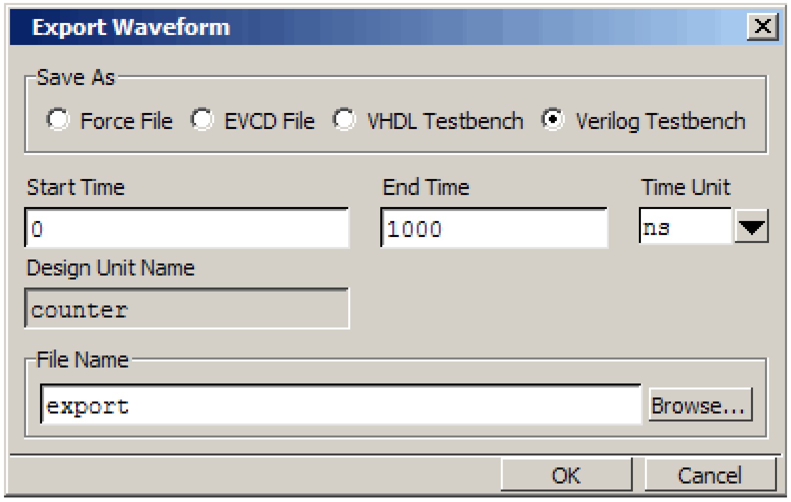
   a. Select **File** > **Export** > **Waveform**.

   b. Select **Verilog Testbench** (or **VHDL Testbench** if you are using the VHDL sample files).

   c. Enter **1000** for End Time if necessary.

   d. Type "export" in the File Name field and click **OK** (Figure 8-12).

Figure 8-12. The Export Waveform Dialog Box



Export Waveform

Save As: Force File / EVCD File / VHDL Testbench / [✓ Verilog Testbench]

Start Time: 0
End Time: 1000
Time Unit: ns
Design Unit Name: counter
File Name: export

Questa SIM 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.

   Questa SIM 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.

Procedure

1. Add a design signal.

   a. In the Objects window, right-click *count* and select **Add Wave**.

   The signal is added to the Wave window.

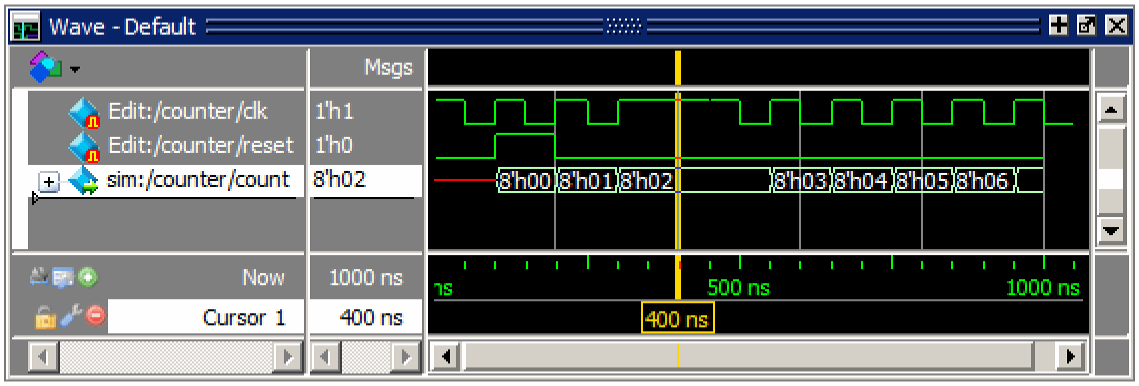
2. Run the simulation.

   a. Enter the following command at the Questa SIM> prompt.

   run 1000

   The simulation runs for 1000 ns and the waveform is drawn for *sim:/counter/count* (Figure 8-13).

Figure 8-13. 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. Click **No** if you are asked to save the wave commands.

Simulating with the Test Bench File

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

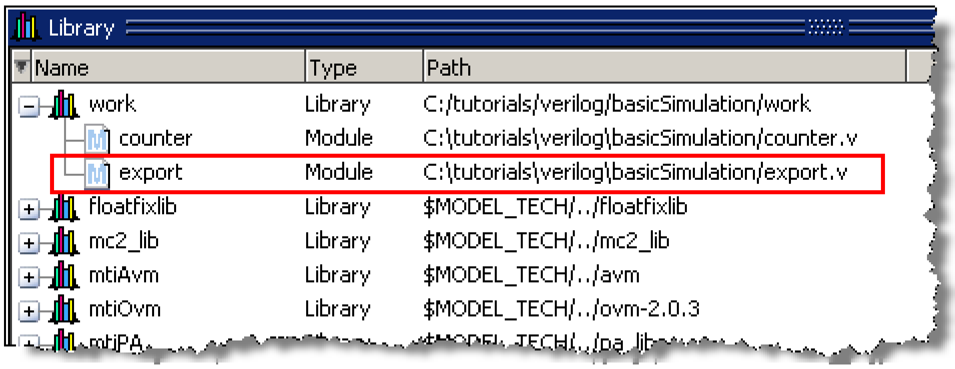
Procedure

1. Compile and load the test bench.

   a. At the Questa SIM 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 work library (Figure 8-14).

Figure 8-14. The export Test Bench Compiled into the work Library



Name	Type	Path
work	Library	C:/tutorials/verilog/basicSimulation/work
counter	Module	C:\tutorials\verilog\basicSimulation/counter.v
export	Module	C:\tutorials\verilog\basicSimulation/export.v
floatfixlib	Library	$MODEL_TECH/../floatfixlib
mc2_lib	Library	$MODEL_TECH/../mc2_lib
mtiAvm	Library	$MODEL_TECH/../avm
mtiOvm	Library	$MODEL_TECH/../ovm-2.0.3
mtiPA		$MODEL_TECH/../pa_lib

   b. Enter the following command at the Questa SIM> prompt.

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

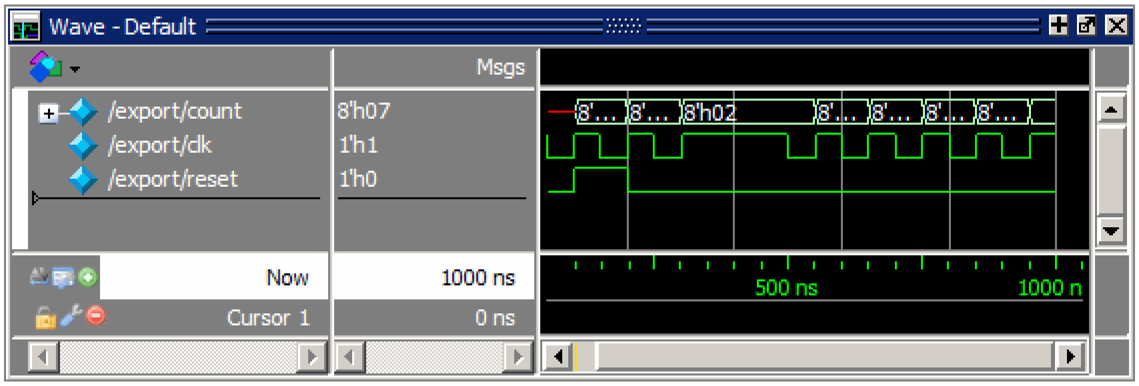
2. Add waves and run the design.

   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 8-15).

Figure 8-15. Waves from Newly Created Test Bench



3. Quit the simulation.

   a. At the VSIM> prompt, type **quit -sim**. 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.

Procedure

1. Load the *counter* design unit and add waves.

   a. Enter the following command at the Questa SIM> prompt.

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

   b. In the Objects window, right-click *count* and select **Add Wave**.

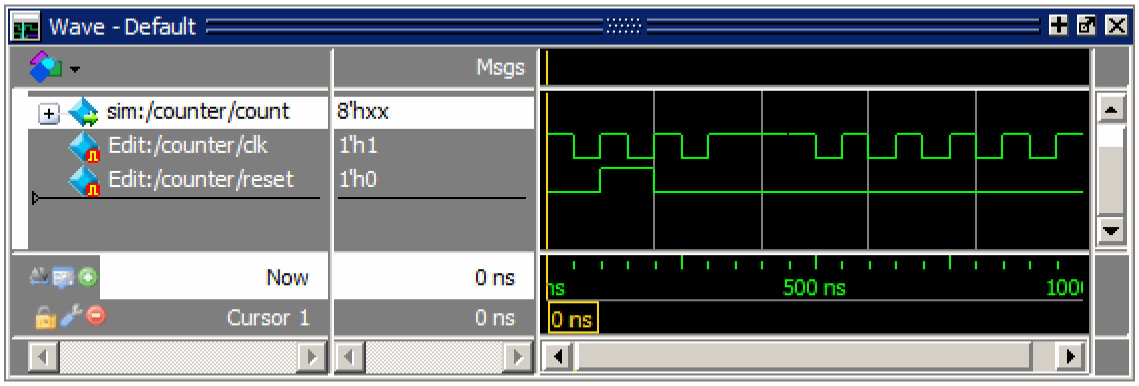
2. Import the VCD file.

   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 8-16).

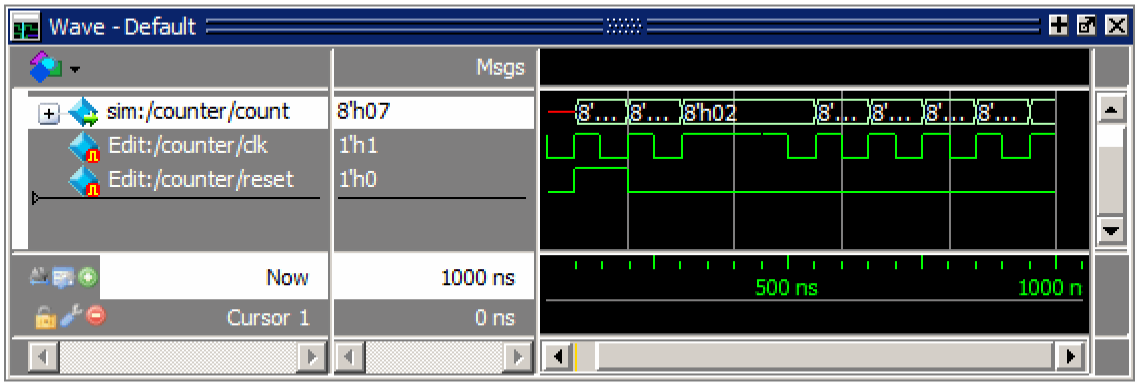
Figure 8-16. EVCD File Loaded in Wave Window



c. Click the Run -All icon.

The simulation runs for 1000 ns and the waveform is drawn for *sim:/counter/count* (Figure 8-17).

Figure 8-17. 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. At the VSIM> prompt, type quit -sim. Click No if you are asked to save the wave commands.

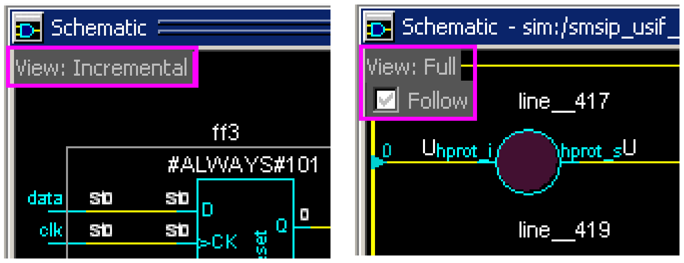
Chapter 9
Debugging With The Schematic Window

The Schematic 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, registers, VHDL architectures, and Verilog modules.

The Schematic window provides two views of the design — a Full View, which is a structural overview of design hierarchy; and an Incremental View, which uses click-and-sprout actions to incrementally add to the selected net's fanout. The Incremental view displays the logical gate equivalent of the RTL portion of the design, making it easier to understand the intent of the design.

A “View” indicator is displayed in the top left corner of the window (Figure 9-1). You can toggle back and forth between views by simply clicking this “View” indicator.

Figure 9-1. Schematic View Indicator



The Incremental View is ideal for design debugging. It allows you to explore design connectivity by tracing signal readers/drivers to determine where and why signals change values at various times.

Note

The Schematic window will not function without an extended dataflow license. If you attempt to create the debug database (vsim -debugdb) without this license the following error message will appear: “Error: (vsim-3304) You are not authorized to use -debugdb, no extended dataflow license exists.”

Design Files for this Lesson

The sample design for this lesson is a test bench 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/schematic*

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

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 Topics

User's Manual section: Schematic Window.

Compile and Load the Design

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

Procedure

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/schematic* to the new directory.

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

2. Start Questa SIM and change to the exercise directory.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box appears, click **Close**.

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

3. Change your WildcardFilter settings.

   Execute the following command:

   ```
   set WildcardFilter "Variable Constant Generic Parameter SpecParam Memory
   Assertion Endpoint ImmediateAssert"
   ```

   With this command, you remove "CellInternal" from the default list of Wildcard filters. This allows all signals in cells to be logged by the simulator so they will be visible in the debug environment.

4. Execute the lesson DO file.

a. Type **do run.do** at the Questa SIM> prompt.

The DO file does the following:

- Creates the working library — vlib work
- Compiles the design files — vlog or vcom
- Optimizes the design — vopt +acc top -debugdb -o top_opt
- Loads the design into the simulator — vsim -debugdb top_opt
- Adds signals to the Wave window — add wave /top/p/*
- Logs all signals in the design — log -r /*
- Runs the simulation — run -all

5. Change the radix to Symbolic.

   a. Type **radix -symbolic** at the Questa SIM> prompt and press enter.

Exploring Connectivity

A primary use of the incremental view of the Schematic window is exploring the physical connectivity of your design. You do this by expanding the view from process to process, to display the drivers/receivers of a particular signal, net, register, process, module or architecture.

Procedure

1. Open the Schematic window.

   a. Select **View > Schematic** from the menus or use the view schematic command at the VSIM prompt in the Transcript window.

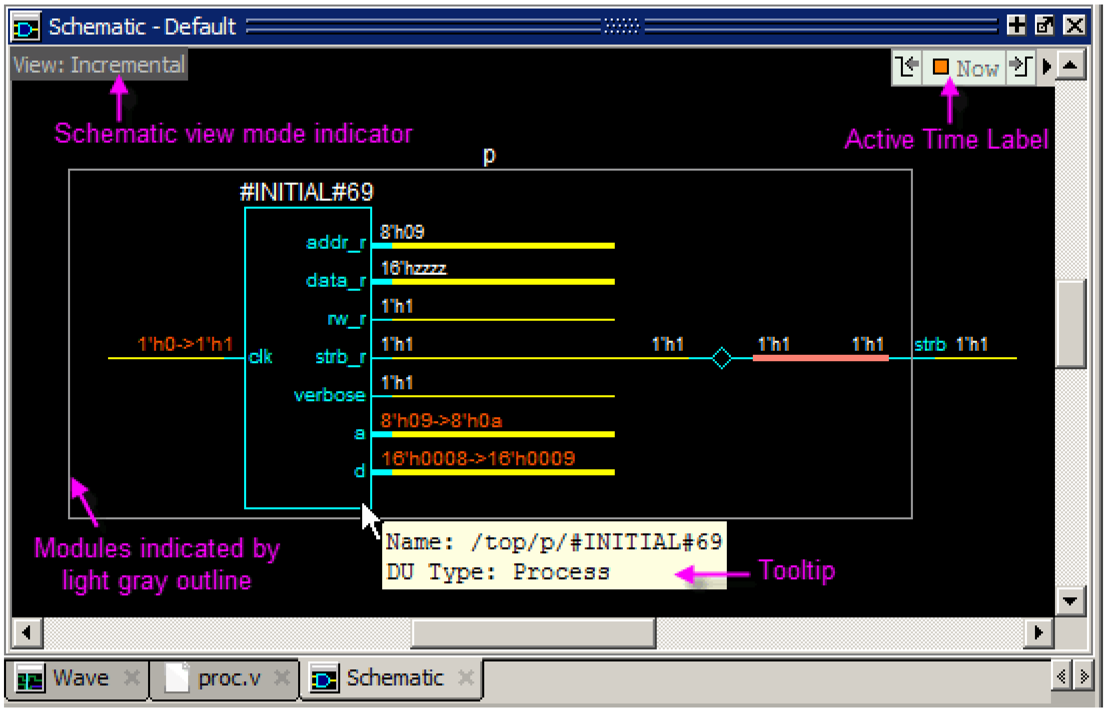
   The Schematic window opens to the Incremental view.

2. Add a signal to the Schematic window.

   a. Make sure instance *p* is selected in the Structure (sim) window.

   b. Drag the *strb* signal from the Objects window to the Schematic window (Figure 9-2).

Figure 9-2. A Signal in the Schematic Window



```
Name: /top/p/#INITIAL#69
DU Type: Process
```

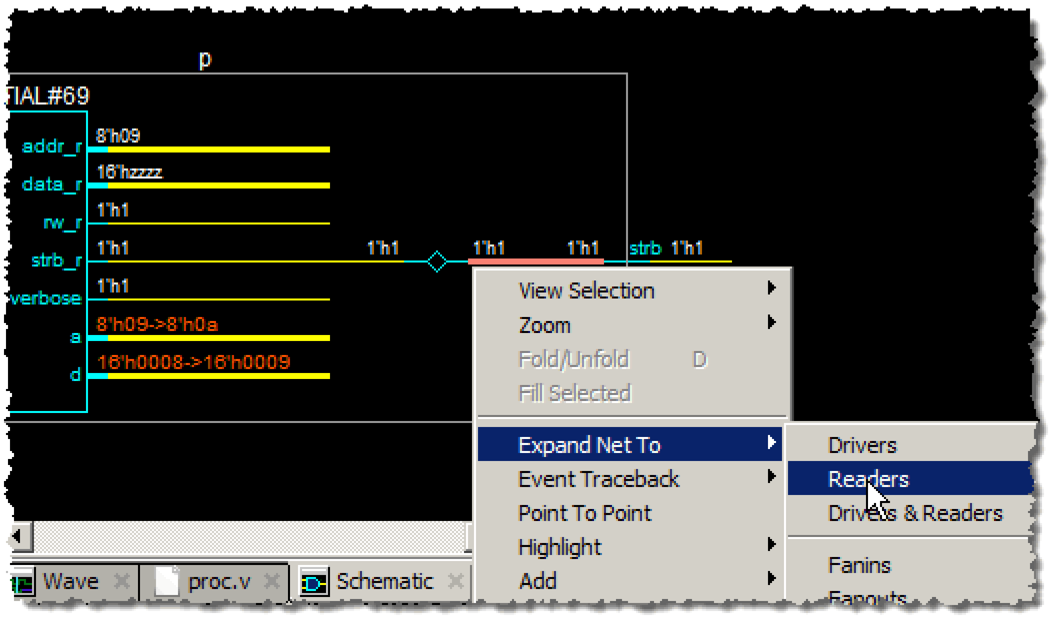
The Incremental view shows the *strb* signal, highlighted in orange. You can display a tooltip - a text information box – as shown in Figure 9-2 – by hovering the mouse cursor over any design object in the schematic. In this case, the tooltip shows details about the *p* module, denoted by the light gray box.

Signal values are displayed at the ends of each signal net. You can toggle signals values on and off with the 'v' key on your keyboard when the Schematic window is active.

3. Find the readers of the *strb* signal inside the *p* module.

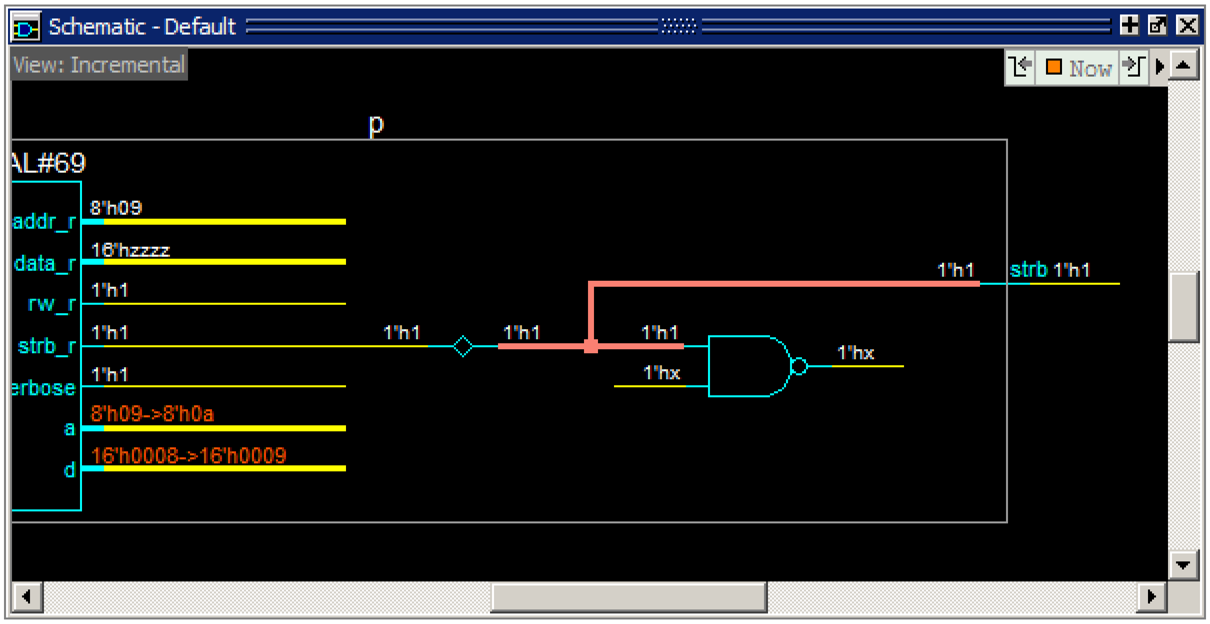
   a. Right-click the highlighted *strb* signal inside the *p* module and select **Expand Net to > Readers** from the popup menu (Figure 9-3).

Figure 9-3. Expand Net to > Readers



The *p* module will be displayed as shown in Figure 9-4.

Figure 9-4. The p Module

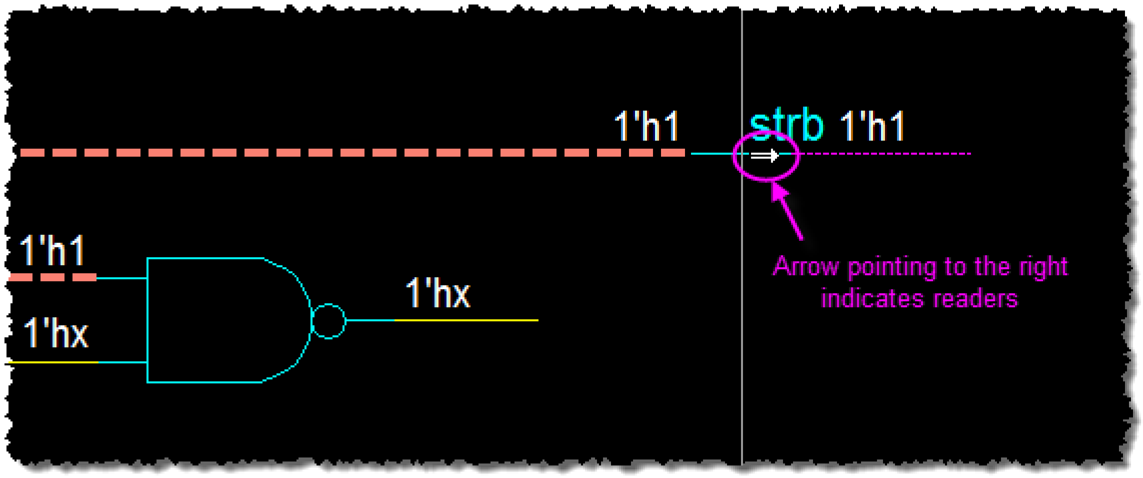


4. Find the readers of the *strb* signal outside the *p* module.

   When you mouse-over any signal pin the mouse cursor will change to a right-pointing arrow, a left-pointing arrow, or a double-headed arrow. If the arrow points to the right, you can double-click the pin to expand the signal fanout to its readers. If the arrow points left, you can double-click to expand the signal fanout to its drivers. Double-clicking a double-headed arrow will expand to drivers and readers.

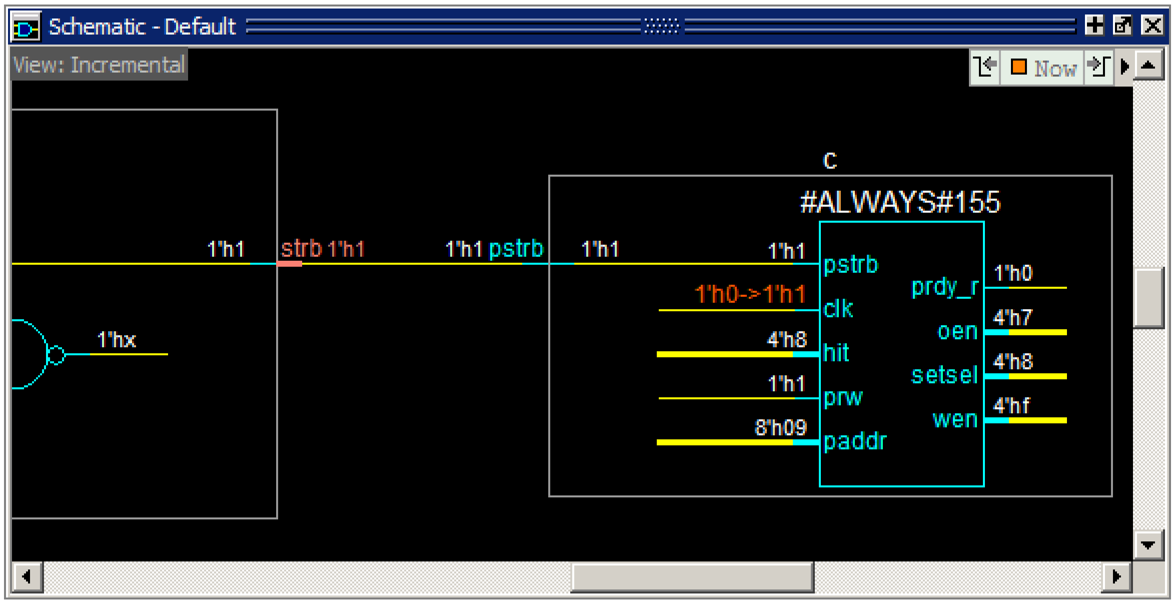
a. Place the cursor over the *strb* signal as shown in Figure 9-5, so you see a right pointing arrow indicating readers, and double click.

Figure 9-5. Right Pointing Arrow Indicates Readers



This sprouts all readers of *strb* (Figure 9-6).

Figure 9-6. Expanding the View to Display Readers of strb Signal



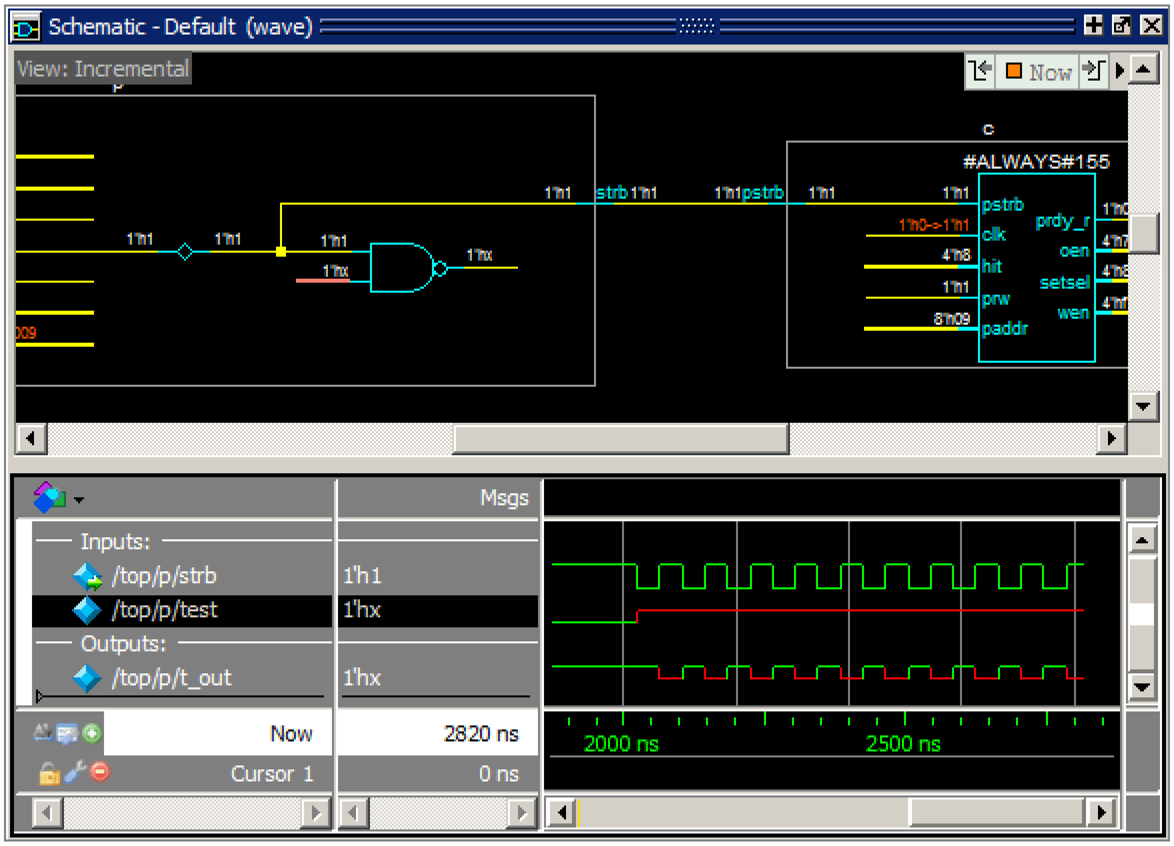
5. Find the drivers of the *test* signal on the NAND gate in the *p* module.

a. Click the **Show Wave** button to open the Schematic Window's embedded Wave Viewer. You may need to increase the size of the schematic window to see everything

b. Select the NAND gate in the schematic. This loads the wave signals for the inputs and outputs for this gate into the Wave Viewer and highlights the gate.

c. Select the *test* signal in the Wave Viewer. This highlights the *test* input in the schematic (Figure 9-7).

Figure 9-7. Select test signal



Notice that the title of the Schematic window is “Schematic -Default (wave)” when the embedded Wave Viewer is active, and “Schematic -Default (schematic)” when the Incremental View is active. In the next step we have to select a pin in the schematic to make the Incremental View and associated toolbar buttons active.

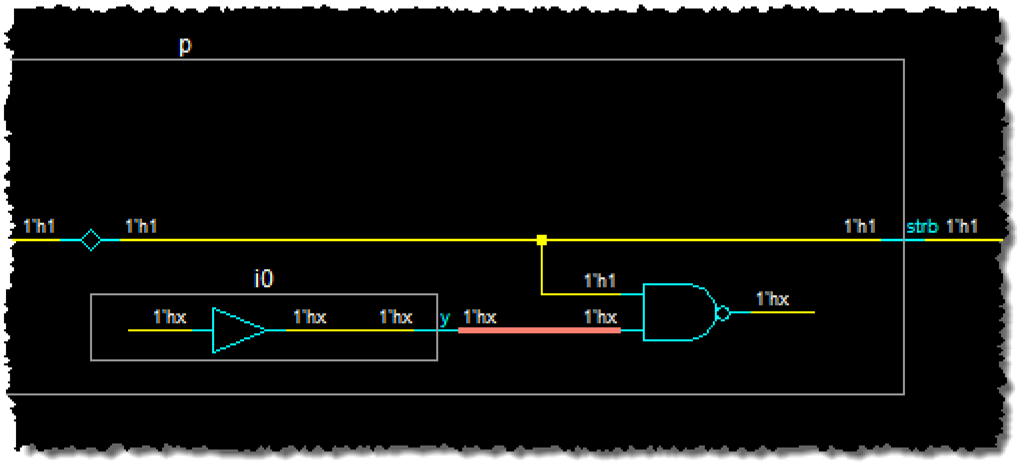
d. Select the pin for the highlighted signal – *test* – in the schematic. This makes the schematic view active.

e. Click the **Expand net to all drivers** icon.

This opens a Source window for the *proc.v* file.

f. Click the Schematic tab to switch back to the Schematic window. You can see in Figure 9-8 that the driving process of the *test* signal is an *i0* module, which is included in the *p* module.

Figure 9-8. The test Net Expanded to Show All Drivers

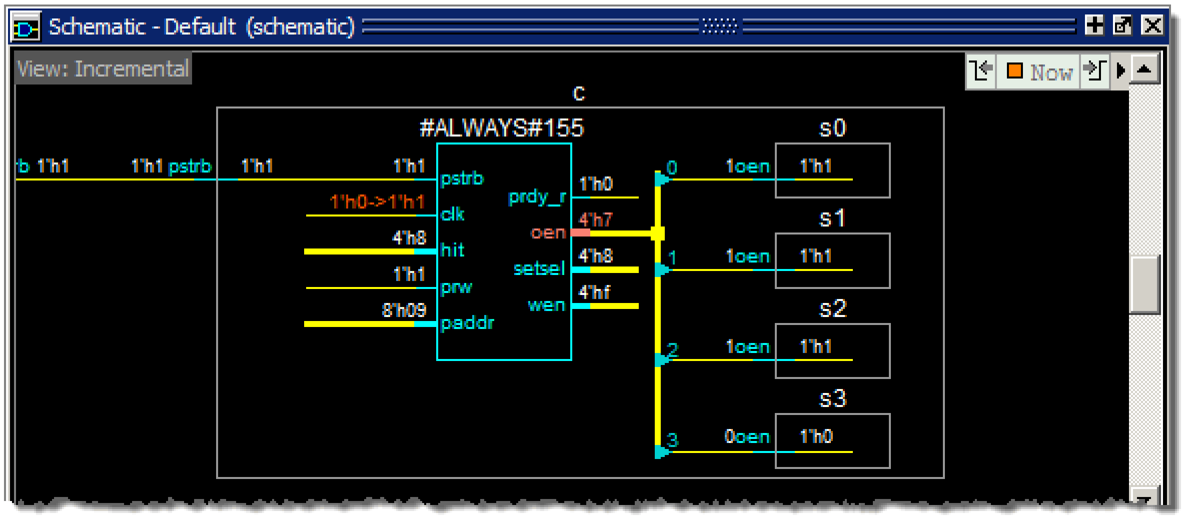


6. Open the readers for signal *oen* on process *#ALWAYS#155* in the *c* module (labeled *line_84* in the VHDL version).

   a. Click the *oen* pin to make it active.

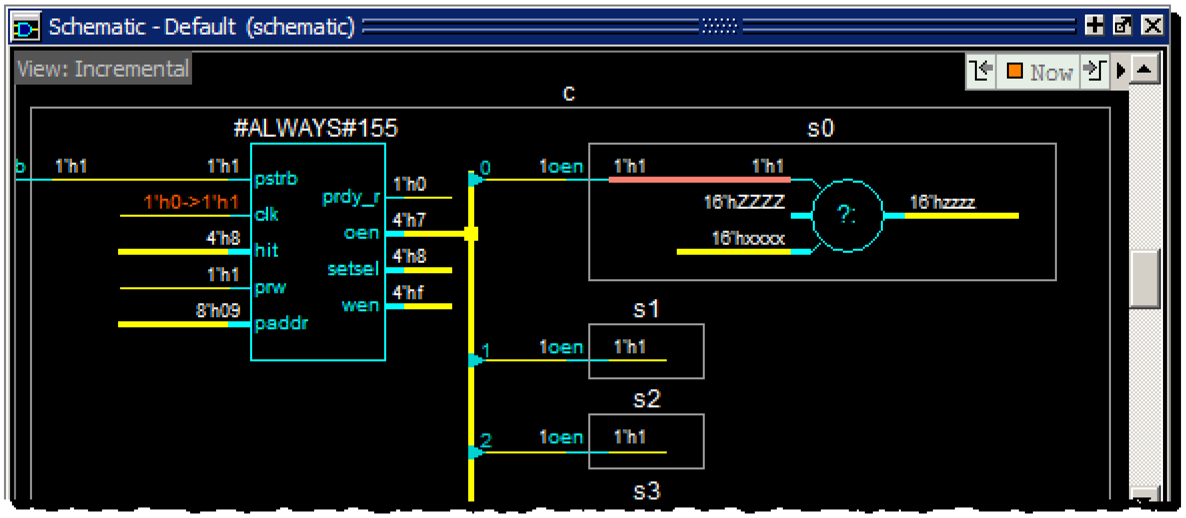
   b. Right-click anywhere in the schematic to open the popup menu and select **Expand Net To** > **Readers**. Figure Figure 9-9 shows the results.

Figure 9-9. Signal oen Expanded to Readers



Notice, expansion of *oen* to its readers stops at the boundaries of the s0-s3 instances. To see inside any instance, double-click the *oen* net inside the instance to sprout a tri-state device as shown in Figure 9-10.

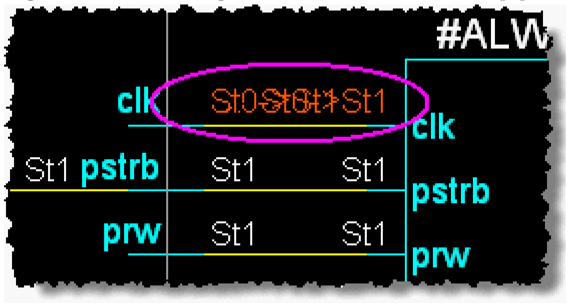
Figure 9-10. Sprout oen in the s0 Instance



Continue exploring the design with any of the methods discussed above – double-click signal pins or nets, use the toolbar buttons, or use menu selections from the right-click popup menu.

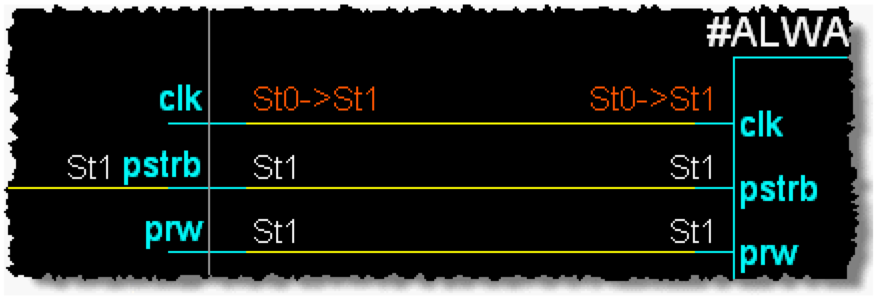
The signal values for the signals may not be easily distinguished when the values at each end of the net overlap.

Figure 9-11. Signal Values Overlapped



Click the Regenerate button to redraw the Schematic with all design elements, signal values, and pin names clearly displayed (Figure 9-12).

Figure 9-12. Signal Values After Regenerate



When you are finished, click and hold the **Delete Content** button until the popup menu appears, then click **Delete All** to clear the schematic viewer.

Click the **Show Wave** button to close the embedded Wave Viewer.

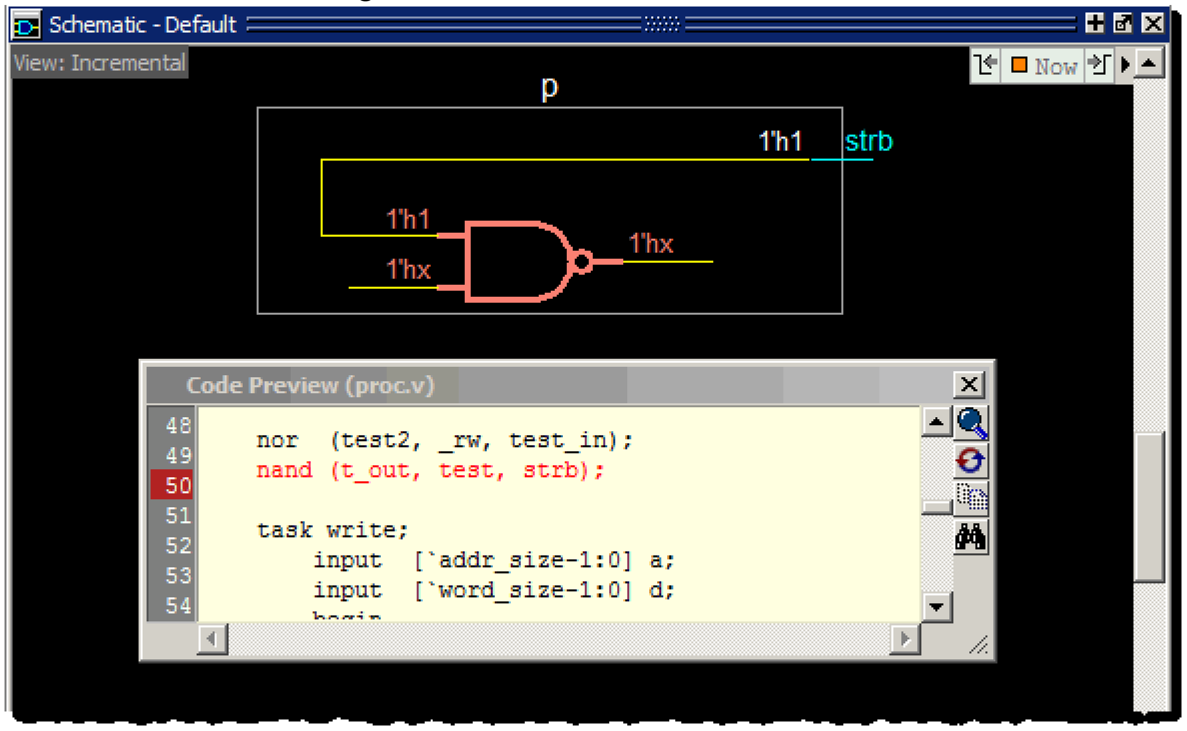
Viewing Source Code from the Schematic

The Schematic window allows you to display a source code preview of any design object.

Procedure

1. Add a signal to the Schematic window.
   a. Make sure instance *p* is selected in the Structure (sim) window.
   b. Drag signal *t_out* from the Objects window to the Schematic window.
   c. Double-click the NAND gate to display a Code Preview window (Figure 9-13). The source code for the selected object is highlighted.

Figure 9-13. Code Preview Window



```
    nor  (test2, _rw, test_in);
    nand (t_out, test, strb);

    task write;
        input  [`addr_size-1:0] a;
        input  [`word_size-1:0] d;
```

The Code Preview window provides a four-button toolbar that allows you to take the following actions:

- view the source code in a Source Editor

- recenter the selected code in the Code Preview window if you have scrolled it out of the display
- copy selected code so it can be pasted elsewhere
- open the Find toolbar at the bottom of the Code Preview window so you can search for specific code

d. Experiment with the Code Preview toolbar buttons to see how they work.

When you are finished, close the Code Preview window, then press and hold the **Delete Content** button until the popup menu appears and select **Delete All** to clear the schematic viewer.

Unfolding and Folding Instances

Contents of complex instances are folded (hidden) in the Incremental view to maximize screen space and improve the readability of the schematic.

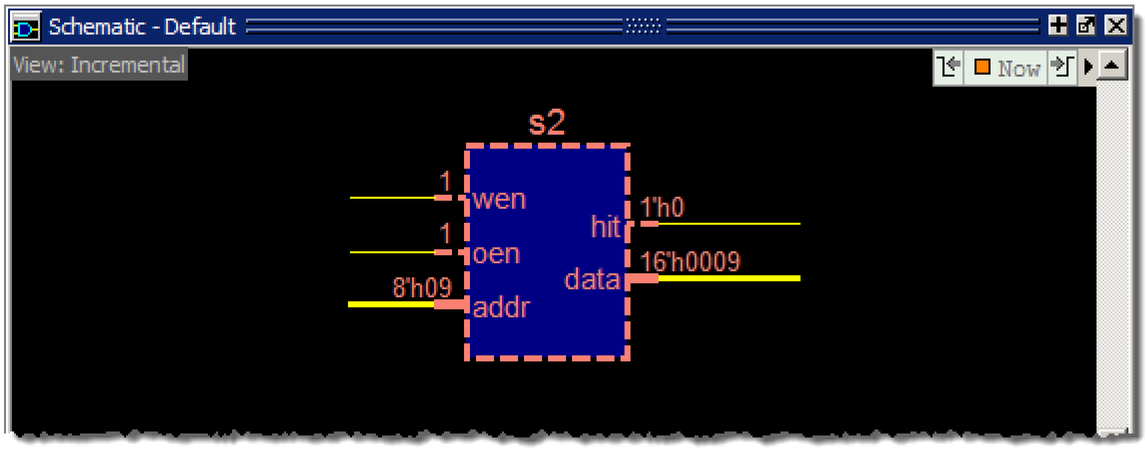
Procedure

1. Display a folded instance in the Incremental view of the schematic.

   a. Expand the hierarchy of the *c* module in the Structure window.

   b. Drag the *s2* module instance (in the *c* module) from the Structure window to the Schematic.

Figure 9-14. Folded Instance



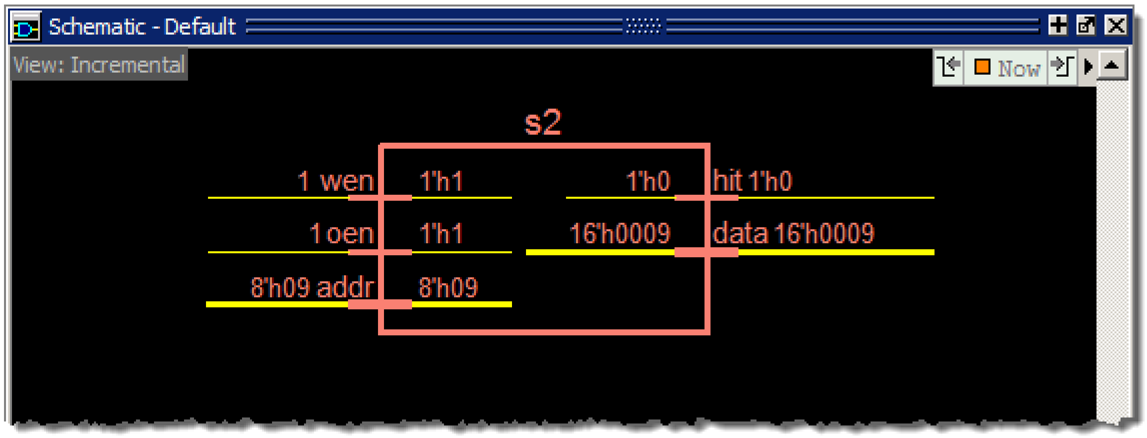
The folded instance is indicated by a dark blue square with dashed borders (Figure 9-14). When you hover the mouse cursor over a folded instance, the tooltip (text box popup) will show that it is **FOLDED**.

2. Unfold the folded instance.

a. Right-click inside the folded instance to open a popup menu.

b. Select **Fold/Unfold** to unfold the instance as shown in Figure 9-15.

Figure 9-15. Unfolded Instance

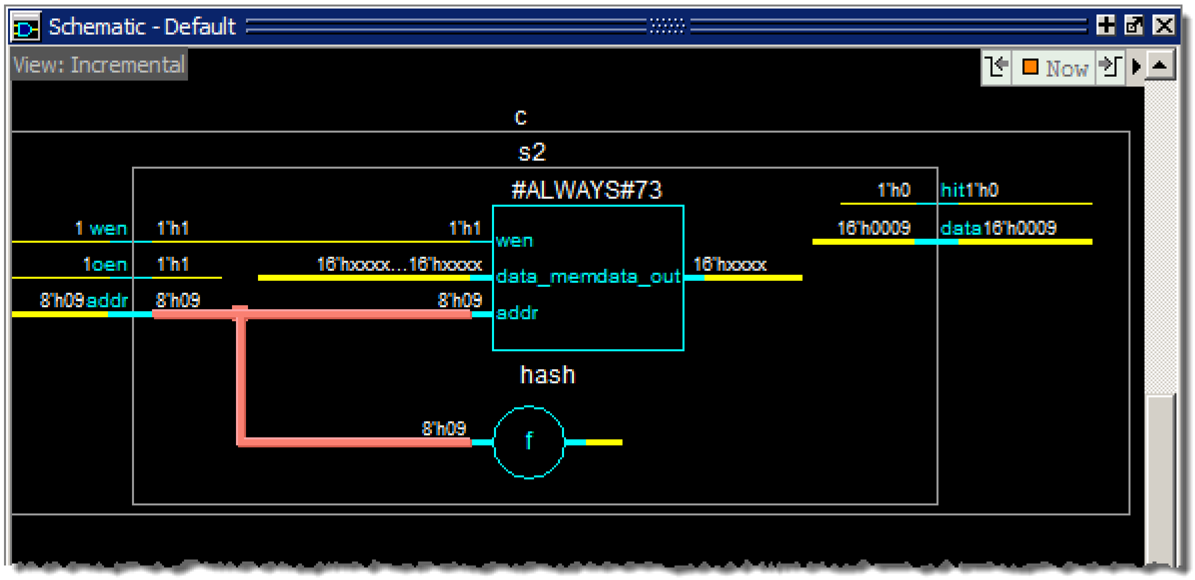


Since we have not traced any signals into the folded instance (we simply dragged it into the Incremental view), we cannot see the contents of the *s2* instance.

3. Display the contents of the *s2* instance.

   a. Double-click the *addr* net inside the *s2* instance to cause the connected gates and internal instances to appear (Figure 9-16).

Figure 9-16. Contents of Unfolded Instance s2

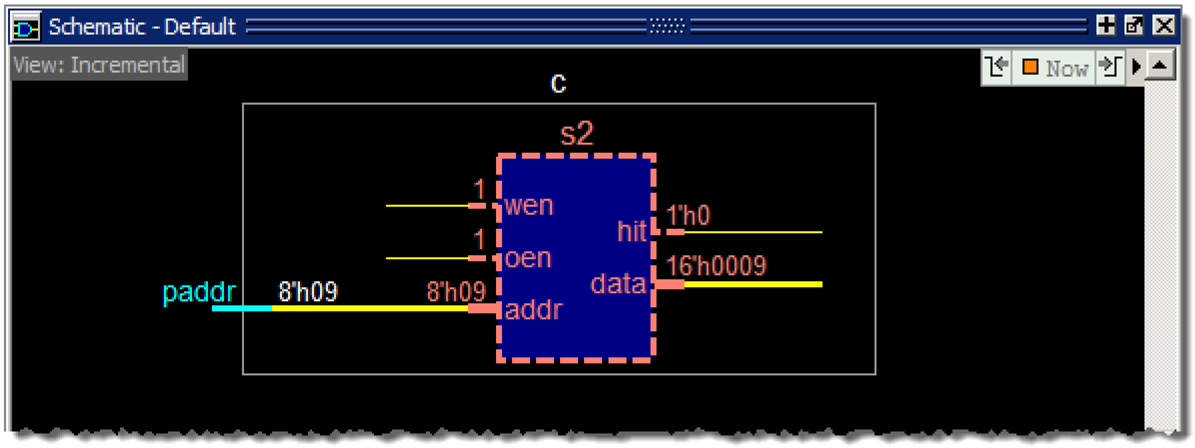


4. Fold instance *s2*.

   a. Left-click the *s2* instance border so it is highlighted.

b. Right-click to open the popup menu and select **Fold/Unfold** to fold the instance.

Figure 9-17. Instance s2 Refolded



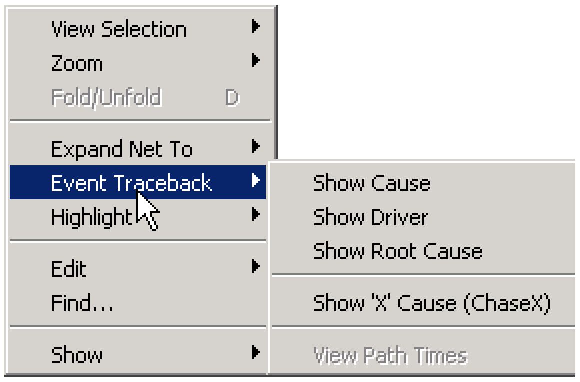
Experiment with other folded instances (s0, s1, s3). When you are finished, use the **Delete Content** button to clear the schematic.

Tracing Events

The Schematic window gives you the ability to trace events to their cause.

Event traceback options are available when you right-click anywhere in the Incremental View and select Event Traceback from the popup menu (Figure 9-18).

Figure 9-18. Event Traceback Menu Options



The event trace begins at the current “active time,” which is set:

- by the selected cursor in the Wave window
- by the selected cursor in the Schematic window’s embedded Wave viewer
- with the Current Time label in the Schematic window.

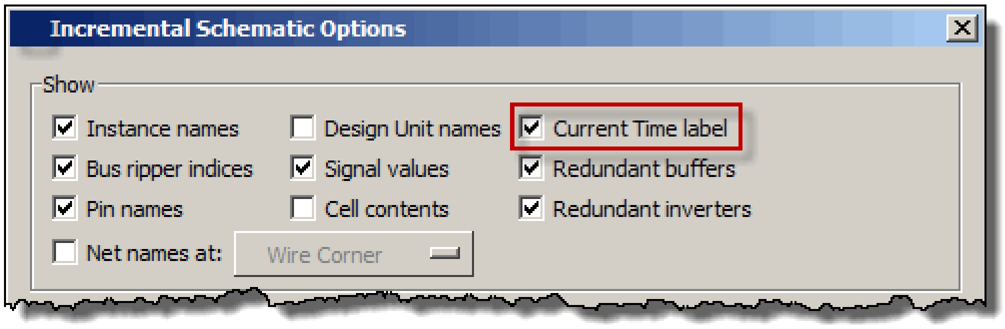
Turn on the Current Time Label

We will use the current time set by the cursor in the embedded Wave viewer. The Current Time Label is on by default, the following instructions allow you to turn it off or on in the Incremental View.

Procedure

1. With the Incremental view active, select **Schematic** > **Preferences** to open the Incremental Schematic Options dialog box.
2. In the Show section of the dialog box, click the **Current Time label** box so a checkmark appears, then click the OK button to close the dialog box.

Figure 9-19. Selecting Current Time Label Display Option

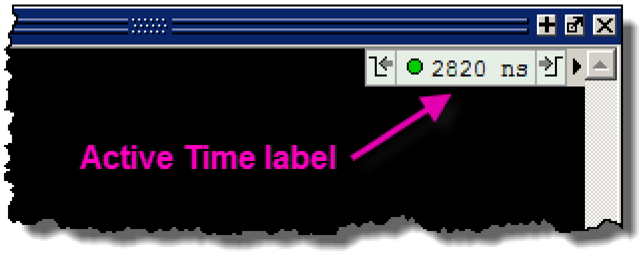


Incremental Schematic Options

Show
- [✓] Instance names
- [ ] Design Unit names
- [✓] Current Time label
- [✓] Bus ripper indices
- [✓] Signal values
- [✓] Redundant buffers
- [✓] Pin names
- [ ] Cell contents
- [✓] Redundant inverters
- [ ] Net names at: Wire Corner

The Current Time label appears in the upper right corner of Incremental view.

Figure 9-20. CurrentTime Label in the Incremental View



Trace to an Event

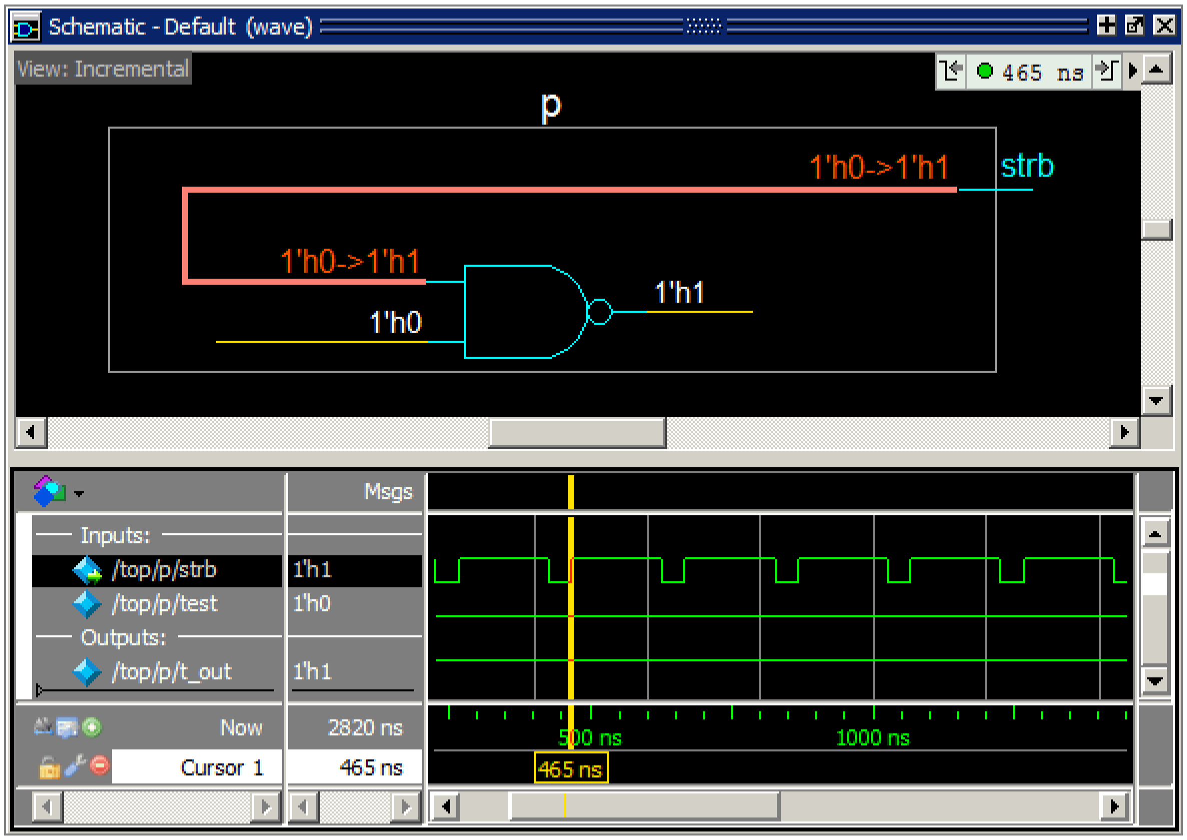
Now we'll trace an event.

Procedure

1. Add an object to the schematic window.
   a. Make sure instance *p* is selected in the Structure (sim) window.
   b. Drag signal *t_out* from the Objects window to the schematic window.

2. Open the Schematic window's Wave viewer.
   a. Click the Show Wave button in the toolbar.
3. Show signals for a process in the Schematic window's Wave viewer.
   a. Select the *NAND* gate in the schematic. This loads the wave signals for the inputs and outputs for this gate into the Wave viewer.
4. Place a cursor in the Wave viewer to designate the Current Time.
   a. Locate the cursor and drag it to the transition at 465 ns on the *strb* waveform in the Wave viewer.
   b. Select the *strb* signal pathname in the Wave viewer. This highlights the *strb* signal net in the schematic (Figure 9-21).

Figure 9-21. The Embedded Wave Viewer

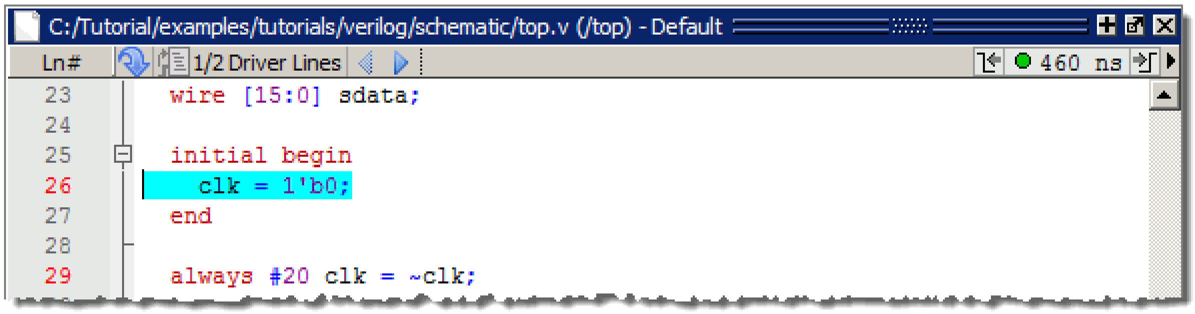


Notice that the Current Time label in the upper right corner of the schematic displays the time of the selected cursor, 465 ns.

5. Trace to the cause of the event.
   a. Right-click the highlighted signal in the schematic to open the popup menu.

b. Select **Event Traceback** > **Show Cause**. This will open a Source window where the immediate driving process will be highlighted (Figure 9-22).

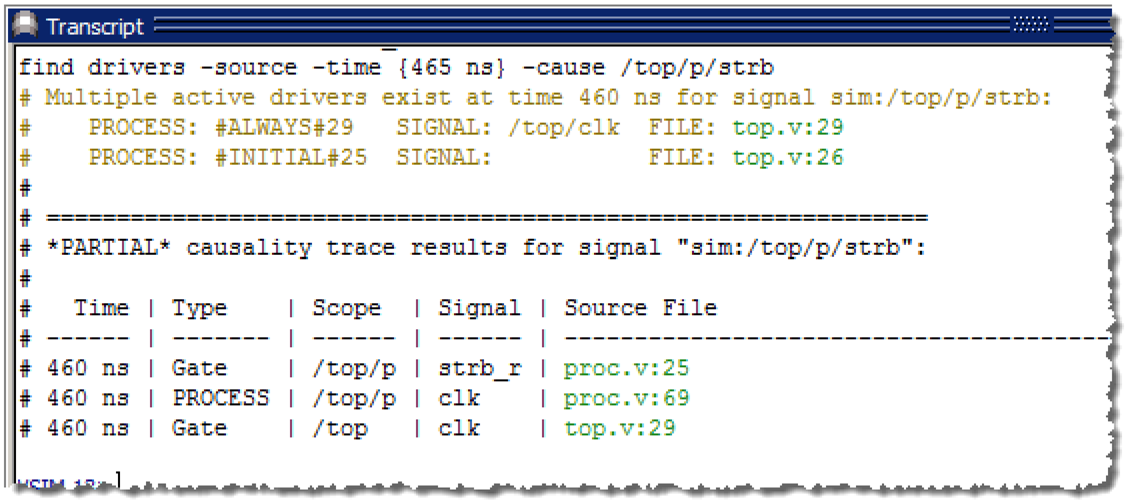
Figure 9-22. Immediate Driving Process in the Source Window


C:/Tutorial/examples/tutorials/verilog/schematic/top.v (/top) - Default
Ln#
1/2 Driver Lines
460 ns

```
wire [15:0] sdata;

initial begin
  clk = 1'b0;
end

always #20 clk = ~clk;
```

In addition, the Transcript window displays the result of the trace as shown in Figure 9-23.

Figure 9-23. Result of Trace in Transcript

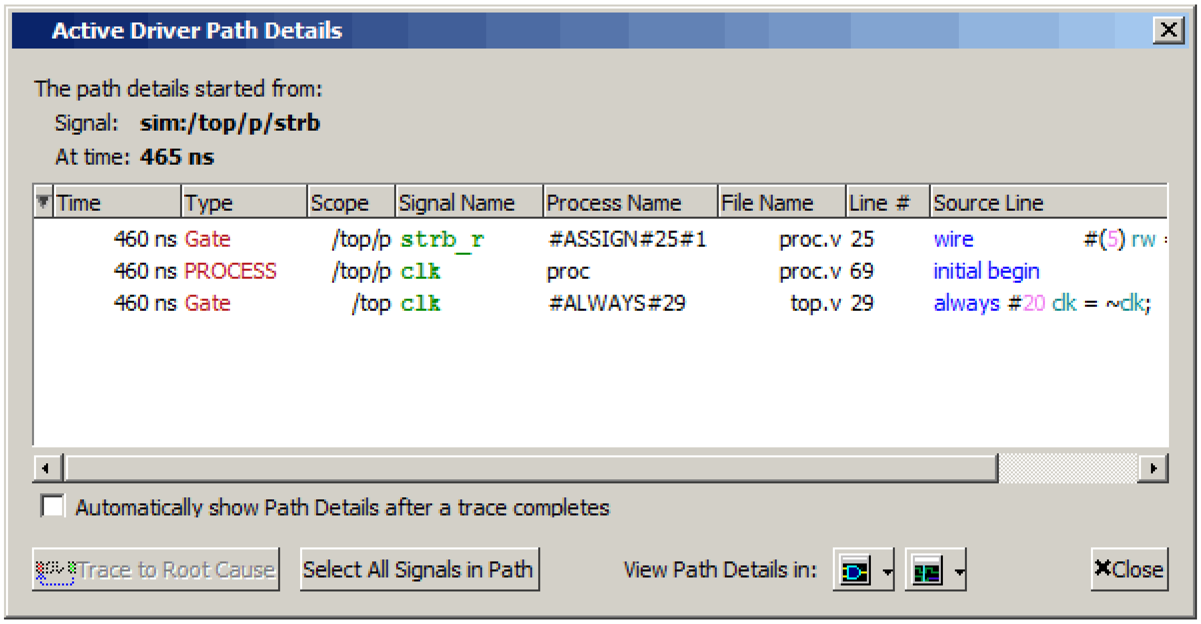


```
find drivers -source -time {465 ns} -cause /top/p/strb
# Multiple active drivers exist at time 460 ns for signal sim:/top/p/strb:
#    PROCESS: #ALWAYS#29   SIGNAL: /top/clk   FILE: top.v:29
#    PROCESS: #INITIAL#25  SIGNAL:            FILE: top.v:26
#
# ================================================================
# *PARTIAL* causality trace results for signal "sim:/top/p/strb":
#
#   Time | Type    | Scope  | Signal | Source File
# ------ | ------- | ------ | ------ | ----------------------------------------
# 460 ns | Gate    | /top/p | strb_r | proc.v:25
# 460 ns | PROCESS | /top/p | clk    | proc.v:69
# 460 ns | Gate    | /top   | clk    | top.v:29
```

c. To see path details, open the Active Driver Path Details window by clicking and holding the Event Traceback toolbar button until the popup menu appears, then selecting **View Path Details**.

The Active Driver Path Details window (Figure 9-24) displays information about the sequential process(es) that caused the selected event. It shows the selected signal name, the time of each process in the causality path to the first sequential process, and details about the location of the causal process in the code.

Figure 9-24. Active Driver Path Details Window



The path details started from:
Signal: **sim:/top/p/strb**
At time: **465 ns**

Time	Type	Scope	Signal Name	Process Name	File Name	Line #	Source Line
460 ns	Gate	/top/p	strb_r	#ASSIGN#25#1	proc.v	25	wire #(5) rw
460 ns	PROCESS	/top/p	clk	proc	proc.v	69	initial begin
460 ns	Gate	/top	clk	#ALWAYS#29	top.v	29	always #20 clk = ~clk;

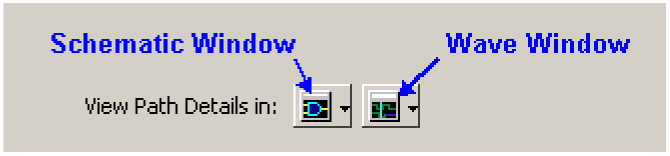
Active Driver Path Details
Automatically show Path Details after a trace completes
Trace to Root Cause
Select All Signals in Path
View Path Details in:
Close

6. View path details for *strb_r* from the #ASSIGN#25#1 process in the Schematic window.

   a. Click the top line in the Active Driver Path Details window to select the *strb_r* signal driver.

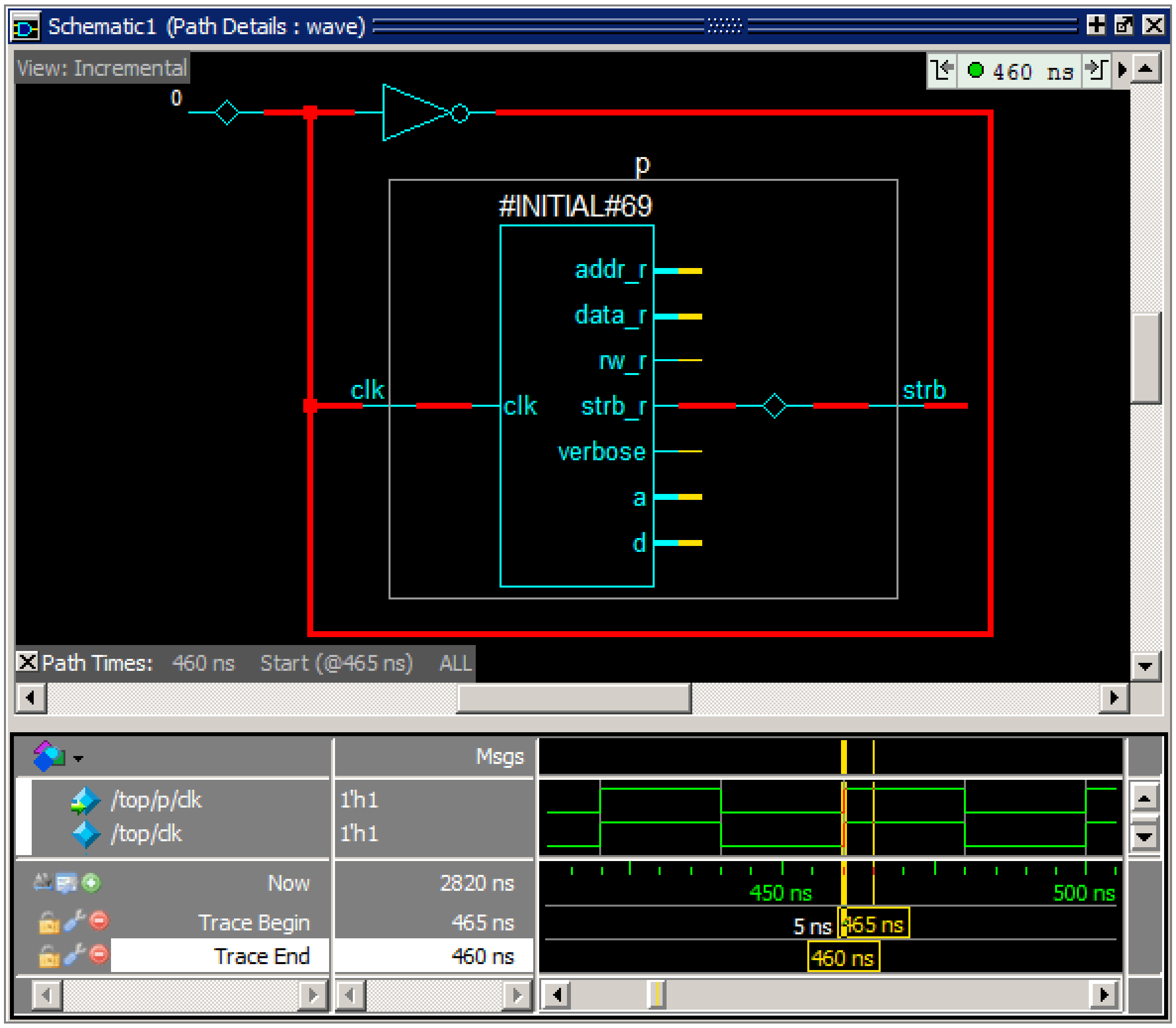
   b. Click the **Schematic Window** button in the View Path Details section of the Active Driver Path Details dialog box (Figure 9-25).

Figure 9-25. Schematic Window Button



This will open a dedicated Schematic (Path Details) window that displays the path details for the selected driver of the signal (Figure 9-26).

Figure 9-26. Schematic Path Details Window



The Wave viewer section of the dedicated Schematic (Path Details) window displays a Trace Begin and a Trace End cursor.

Experiment with tracing other events and viewing path details in the dedicated Schematic and Wave windows.

7. Clear the Schematic window before continuing.

   a. Close the Active Driver Path Details window.

   b. Close the Schematic (Path Details) window.

   c. Select the original Schematic window by clicking the Schematic tab.

   d. Use the **Delete Content** button to clear the Schematic Viewer.

   e. Click the **Show Wave** icon to close the Wave view of the schematic 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.

To return the wildcard filter to its factory default settings, enter:

```
set WildcardFilter "default"
```

Chapter 10
Debugging With The Dataflow Window

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.

Design Files for this Lesson

The sample design for this lesson is a test bench 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 Topics

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.

Procedure

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 Questa SIM and change to the exercise directory.

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

a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

   If the Welcome to Questa SIM dialog box appears, click **Close**.

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

3. Change your WildcardFilter settings.

   Execute the following command:

   ```
   set WildcardFilter "Variable Constant Generic Parameter SpecParam Memory
   Assertion Endpoint ImmediateAssert"
   ```

   With this command, you remove “CellInternal” from the default list of Wildcard filters. This allows all signals in cells to be logged by the simulator so they will be visible in the debug environment.

4. Execute the lesson DO file.

   a. Type **do run.do** at the Questa SIM> prompt.

      The DO file does the following:

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

5. Open a Dataflow window.

   a. Type **view dataflow** at VSIM> prompt and press the Enter key.

6. Change the radix to Symbolic.

   a. Type **radix -symbolic** at the VSIM> prompt

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.

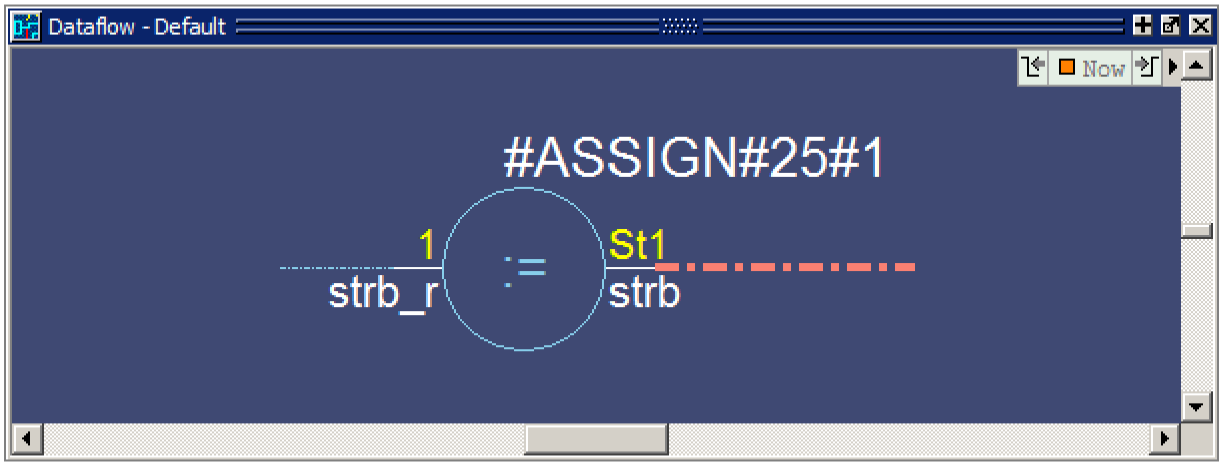
Procedure

1. Add a signal to the Dataflow window.

   a. Make sure instance *p* is selected in the Structure (sim) window.

   b. Drag signal *strb* from the Objects window to the Dataflow window (Figure 10-1).

Figure 10-1. A Signal in the Dataflow Window

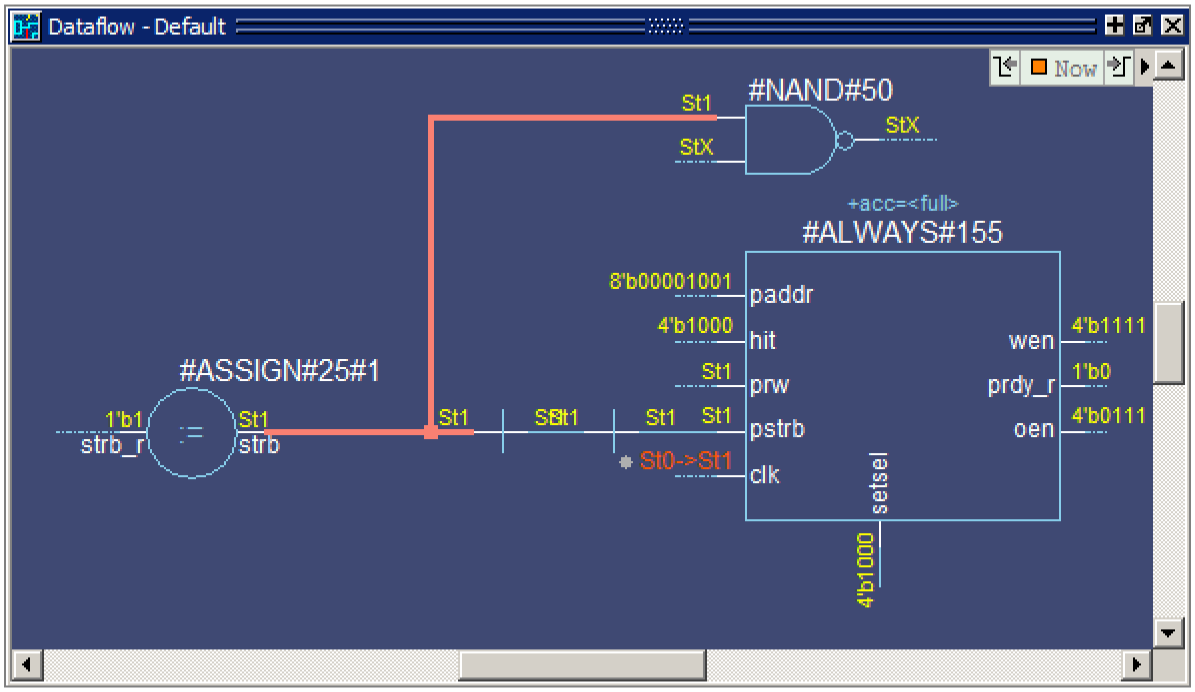


2. Explore the design.

   a. Click the **Expand net to all readers** icon.

   The view expands to display the processes that are connected to *strb* (Figure 10-2).

Figure 10-2. Expanding the View to Display Connected Processes



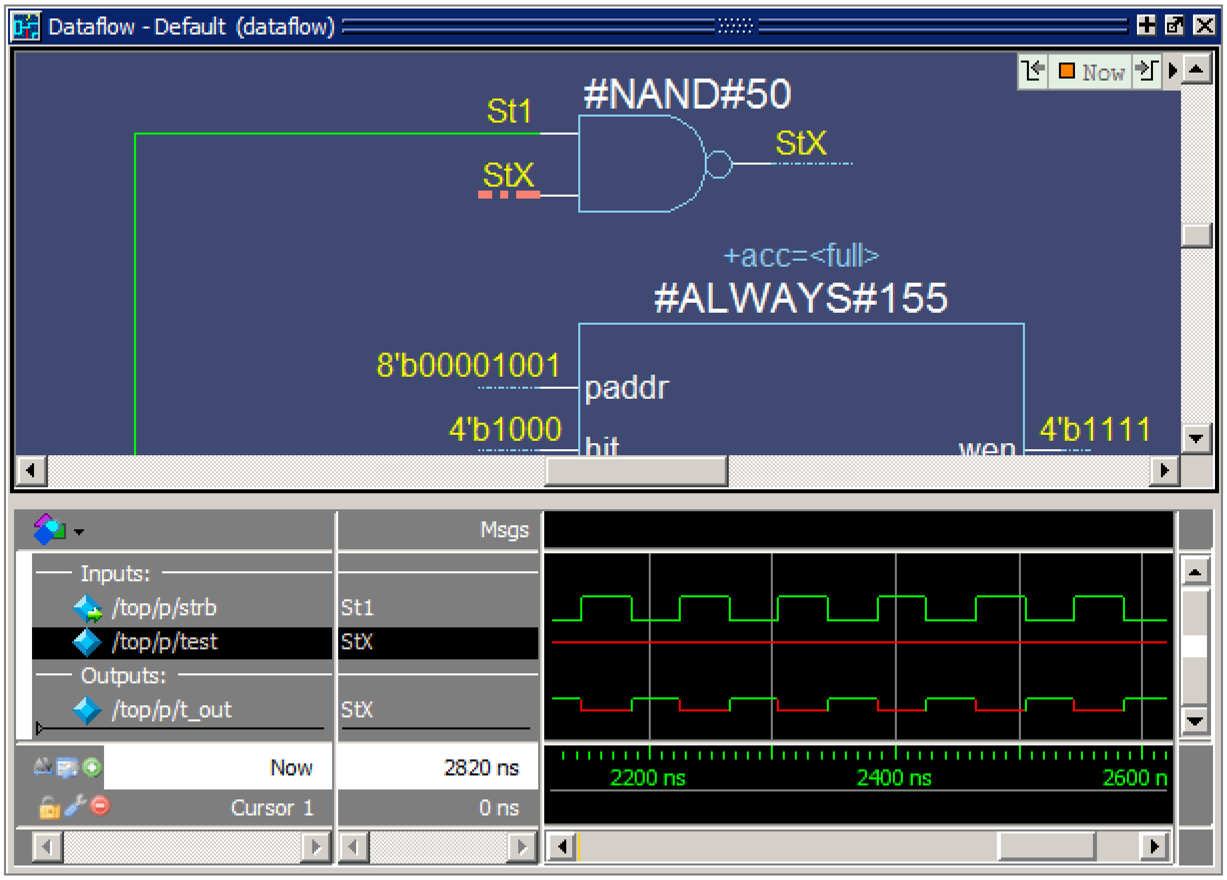
b. Find the drivers of the signal *test* on process *#NAND#50* (labeled *line_71* in the VHDL version).

i. Click the **Show Wave** icon to open the Wave Viewer. You may need to increase the size of the Dataflow window to see everything

ii. Select the *#NAND#50* gate (labeled *t_out_asgn* in the VHDL version) in the Dataflow Viewer. This loads the wave signals for the inputs and outputs for this gate into the Wave Viewer and highlights the gate.

iii. Select the *test* signal in the Wave Viewer. This highlights the *test* input in the Dataflow Viewer. (Figure 10-3)

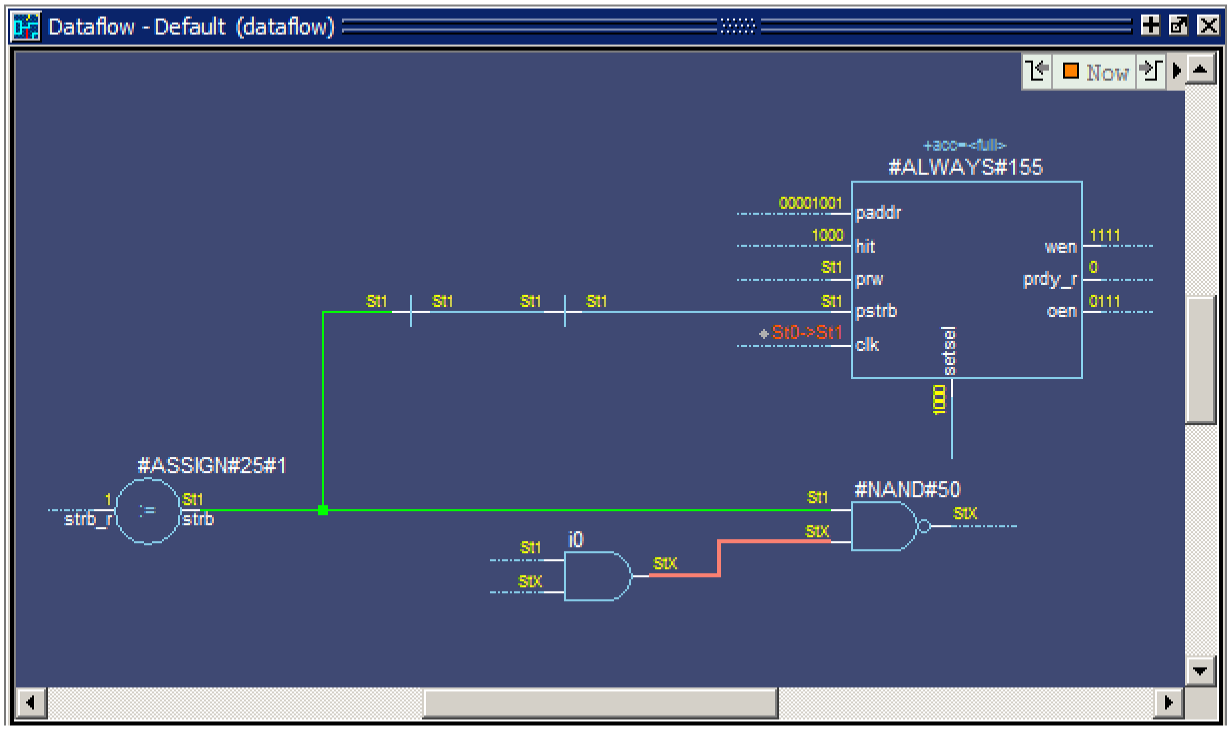
Figure 10-3. Select test signal



iv. Select the highlighted signal in the Dataflow Viewer (this makes the Dataflow Viewer portion of the Dataflow window active) then click the **Expand net to all drivers** icon.

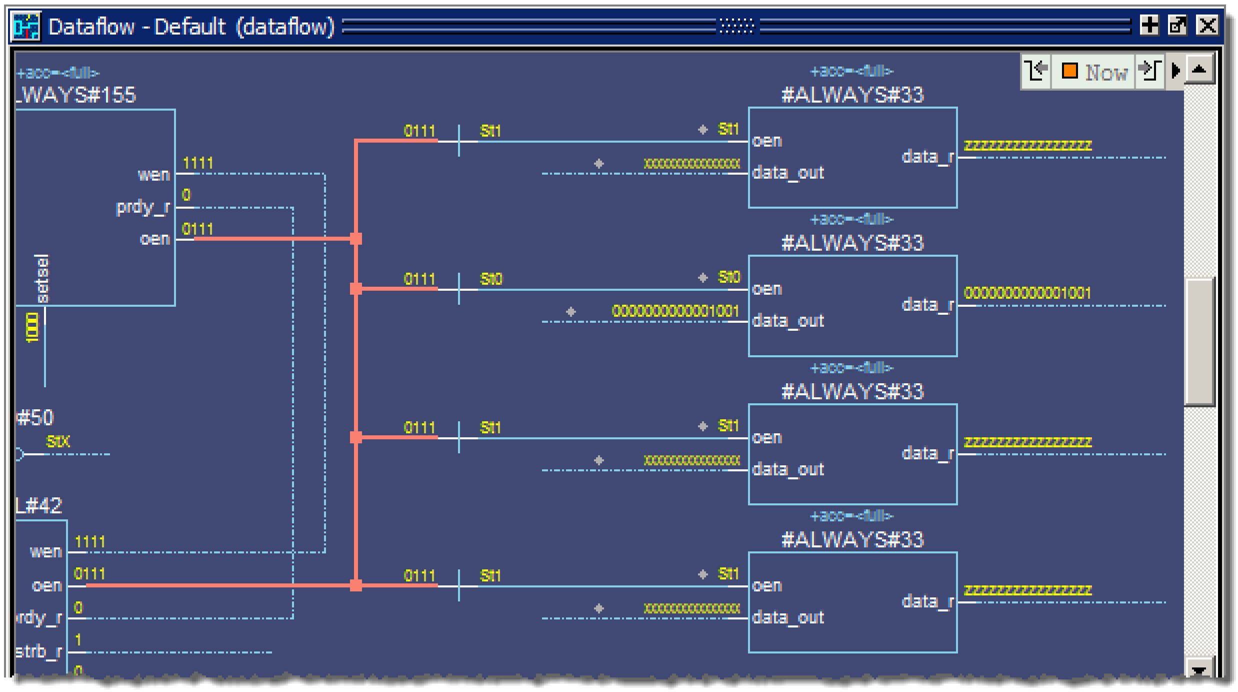
In Figure 10-4, the green highlighting indicates the path you have traversed in the design.

Figure 10-4. The test Net Expanded to Show All Drivers



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

Figure 10-5. The oen Net Expanded to Show All Readers



Continue exploring if you wish.

When you are finished, click and hold the **Delete Content** button until a drop-down list appears; then select **Delete All** to clear the Dataflow Viewer.

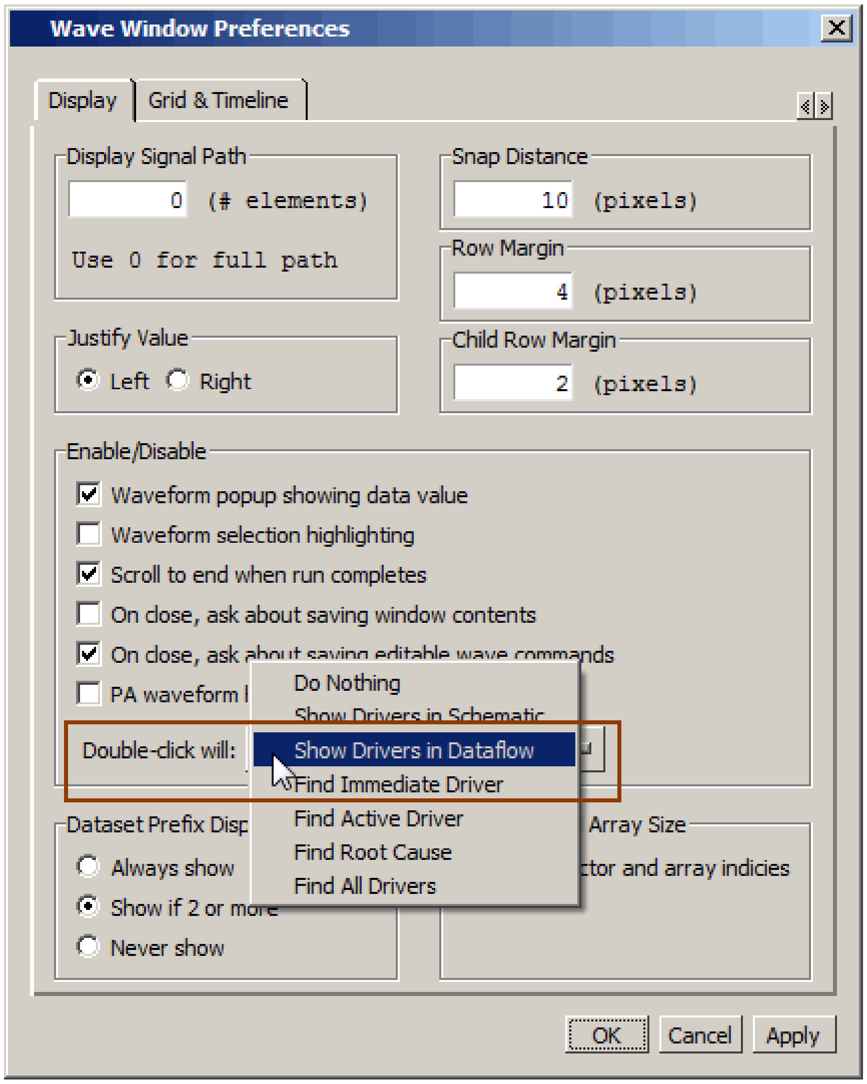
Tracing Events

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 a process or signal that caused the unexpected output.

Procedure

1. Set the default behavior to show drivers in the Dataflow window when double-clicking a signal in the Wave window.
   a. Click the Wave window tab to make the Wave window active.
   b. Select **Wave > Wave Preferences.** This opens the **Wave Window Preferences** dialog box.
   c. Select **Show Drivers in Dataflow** in the "Double-click will:" menu, then click **OK**. (Figure 10-6)

Figure 10-6. Wave Window Preferences Dialog Box



Display Signal Path
0 (# elements)
Use 0 for full path

Snap Distance
10 (pixels)

Row Margin
4 (pixels)

Child Row Margin
2 (pixels)

Justify Value
[✓ Left] Right

Enable/Disable
[✓] Waveform popup showing data value
[ ] Waveform selection highlighting
[✓] Scroll to end when run completes
[ ] On close, ask about saving window contents
[✓] On close, ask about saving editable wave commands
[ ] PA waveform l

Double-click will: Show Drivers in Dataflow

Dataset Prefix Disp
Always show
[✓ Show if 2 or more]
Never show

Array Size
tor and array indicies

2. Add an object to the Dataflow window.

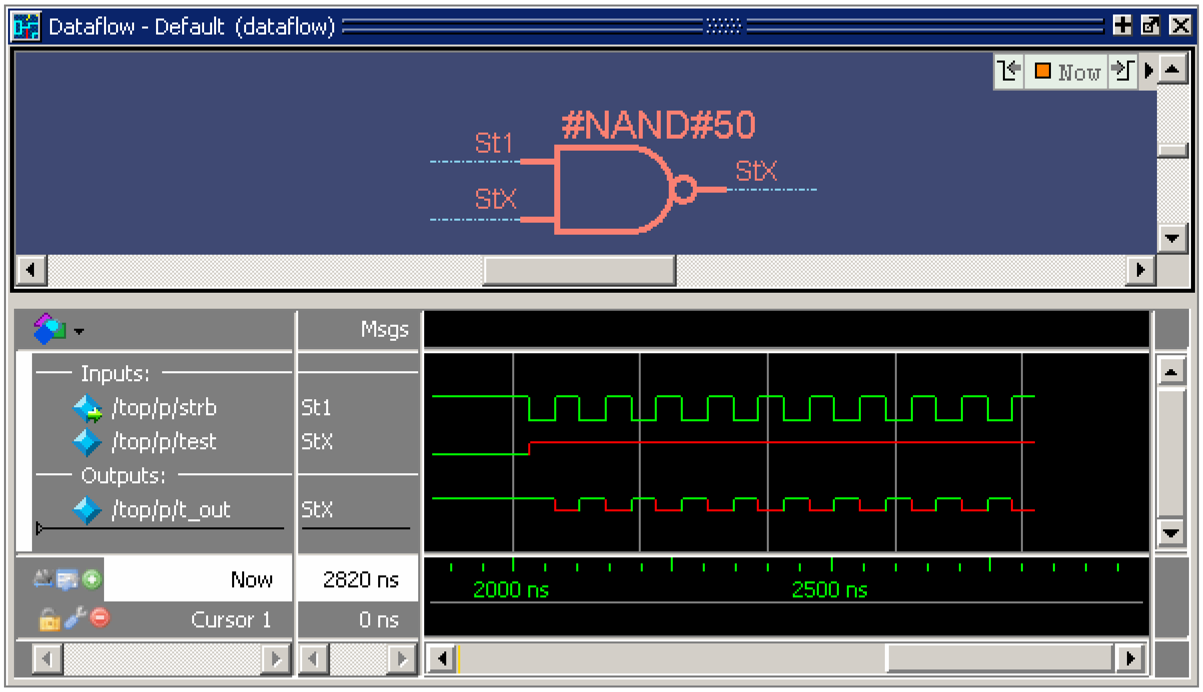
   a. Double-click anywhere on the *t_out* waveform in the Wave window. The Source window will open to show the source code for that signal.

   b. Click the Dataflow tab to open the Dataflow window.

   c. Click the **Show Wave** icon to open the Wave Viewer if it is not already open. You may need to increase the size of the Dataflow window to see everything .

   d. Click the *#NAND#50* gate to display its inputs and outputs in the Wave Viewer (Figure 10-7).

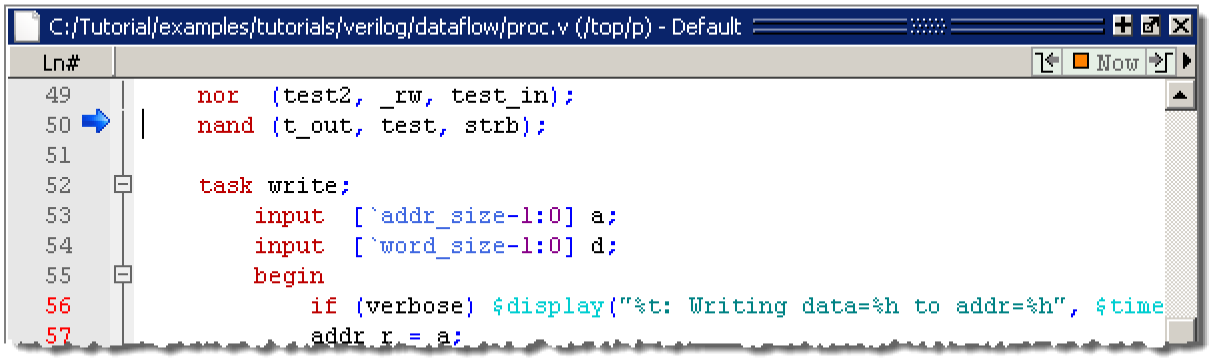
Figure 10-7. The Embedded Wave Viewer



3. Trace the inputs of the nand gate.

   a. Double-click process *#NAND#50* (labeled *line_71* in the VHDL version) in the Dataflow Viewer. The active display jumps to the source code view of the *proc.v* file with a blue arrow pointing to the declaration of the NAND gate (Figure 10-8).

Figure 10-8. Source Code for the NAND Gate


C:/Tutorial/examples/tutorials/verilog/dataflow/proc.v (/top/p) - Default
Ln#
Now

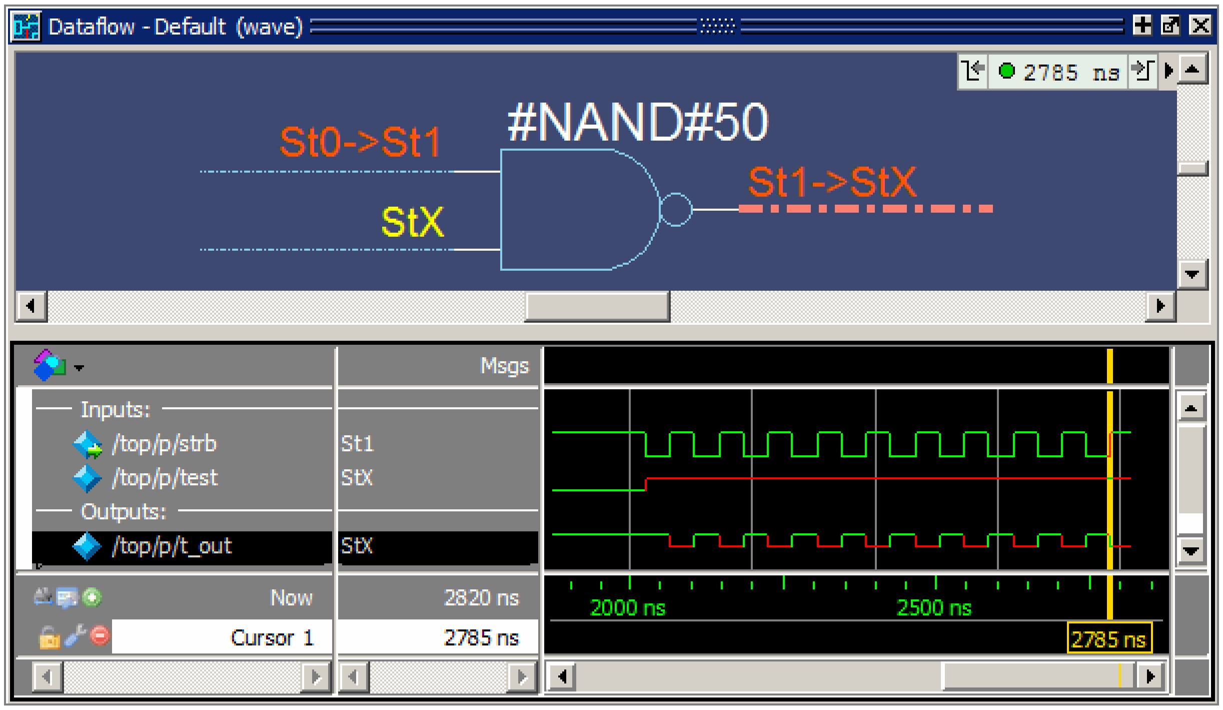
```
    nor  (test2, _rw, test_in);
    nand (t_out, test, strb);

    task write;
        input  [`addr_size-1:0] a;
        input  [`word_size-1:0] d;
        begin
            if (verbose) $display("%t: Writing data=%h to addr=%h", $time
            addr_r = a;
```

   b. Click the Dataflow tab to go back to the Dataflow window.

   c. In the Wave Viewer, scroll to the last transition of signal *t_out*.

   d. Click just to the right of the last transition of signal *t_out*. The cursor should snap to time 2785 ns. (Figure 10-9)

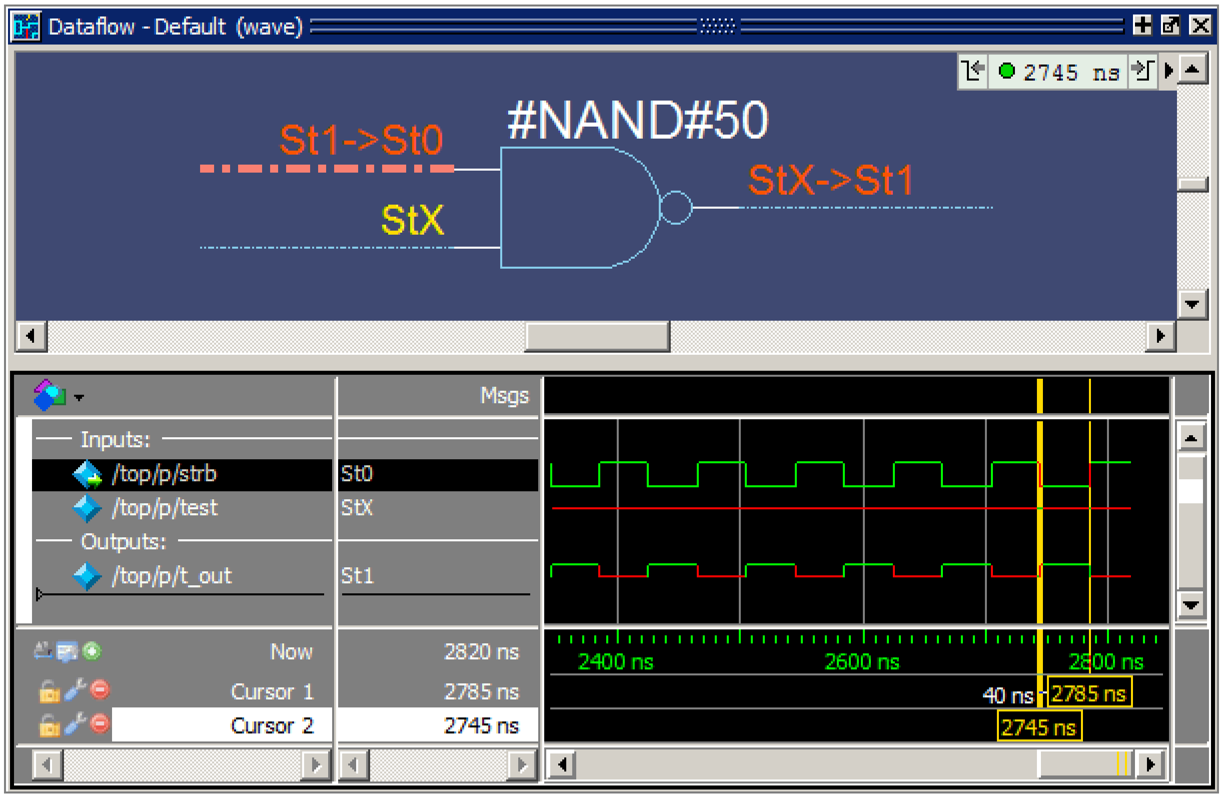
Figure 10-9. Signals Added to the Wave Viewer Automatically



e. The signal *t_out* in the Dataflow Viewer should be highlighted red. Click on the highlighted signal to make the signal active, then select **Tools > Trace > Trace next event** to trace the first contributing event.

   Questa SIM adds a cursor to the Wave Viewer to mark the last event - the transition of the strobe to St0 at 2745 ns - which caused the output of St1 on *t_out* (Figure 10-10).

Figure 10-10. Cursor in Wave Viewer Marks Last Event

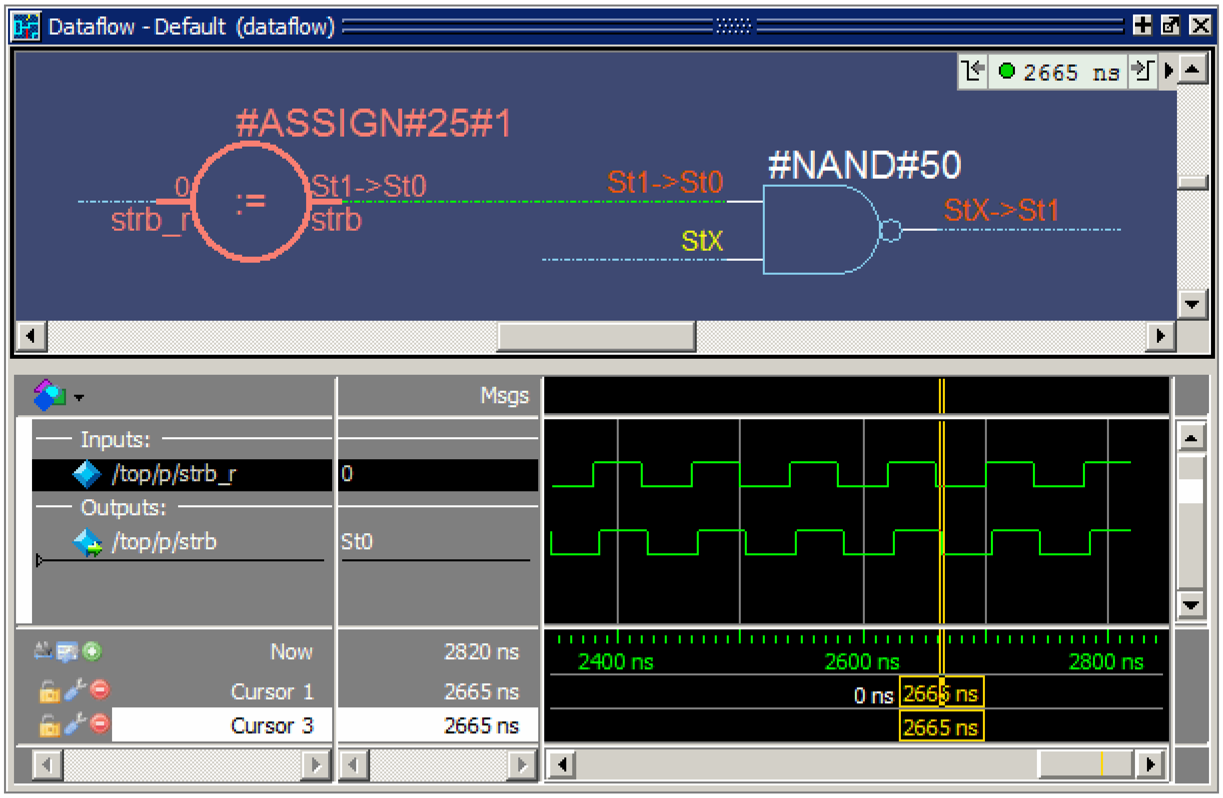


f. Select **Tools** > **Trace** > **Trace next event** two more times and watch the cursor jump to the next event.

g. Select **Tools** > **Trace** > **Trace event set**.

The Dataflow flow diagram sprouts to the preceding process and shows the input driver of the *strb* signal (Figure 10-11). Notice, also, that the Wave Viewer now shows the input and output signals of the newly selected process.

Figure 10-11. 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 flow diagram.

4. When you are finished, 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 dynamically 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.

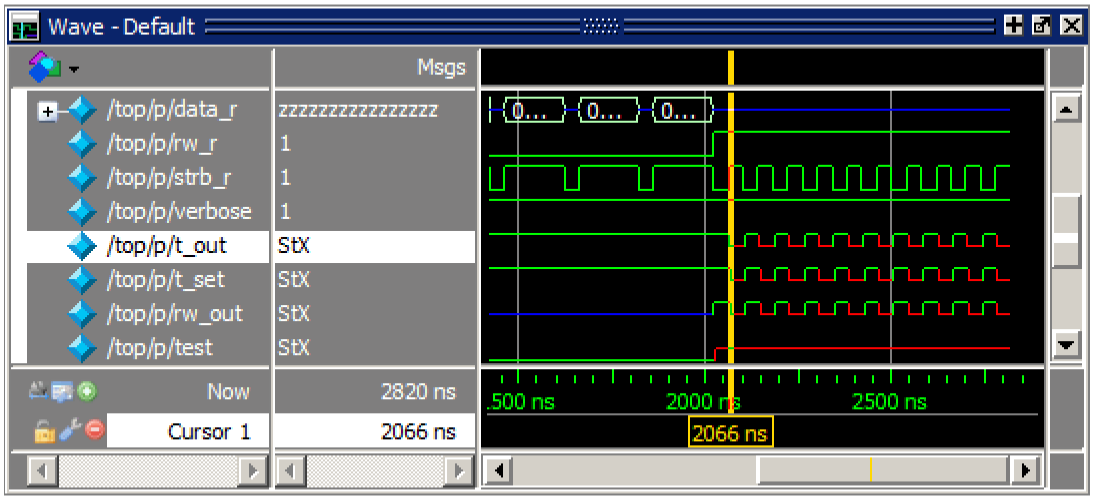
Procedure

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 2066 ns and continues transitioning between 1 and unknown for the rest of the run (Figure 10-12). The red color of the waveform indicates an unknown value.

Figure 10-12. A Signal with Unknown Values



b. Double-click the *t_out* waveform at the last transition of signal *t_out* at 2786 ns.

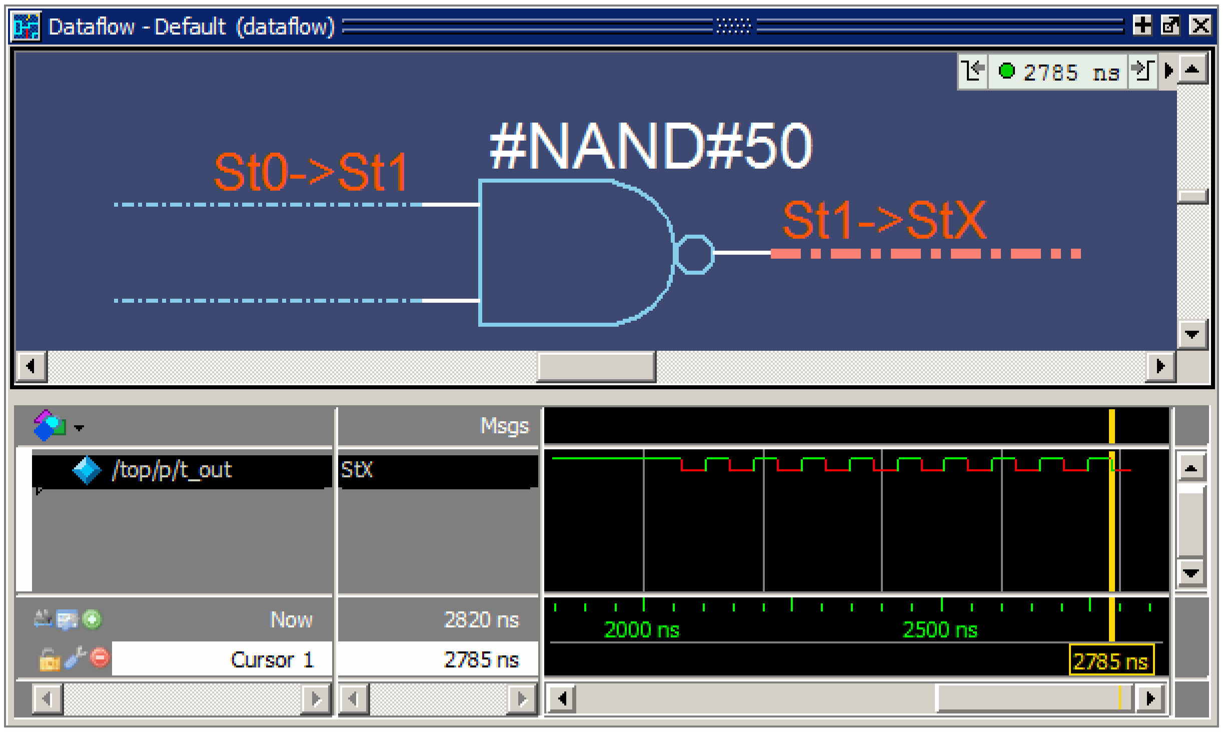
Once again, the source code view is opened and indicates the *t_out* signal.

Double-clicking the waveform in the Wave window also automatically opens a Dataflow window and displays *t_out*, its associated process, and its waveform.

c. Click the Dataflow tab to make the Dataflow window active.

Since the Wave Viewer was open when you last closed the window, it opens again inside the Dataflow window with the *t_out* signal highlighted (Figure 10-13).

Figure 10-13. Dataflow Window with Wave Viewer



d. Position the cursor at a time when *t_out* is unknown (for example, 2725 ns).

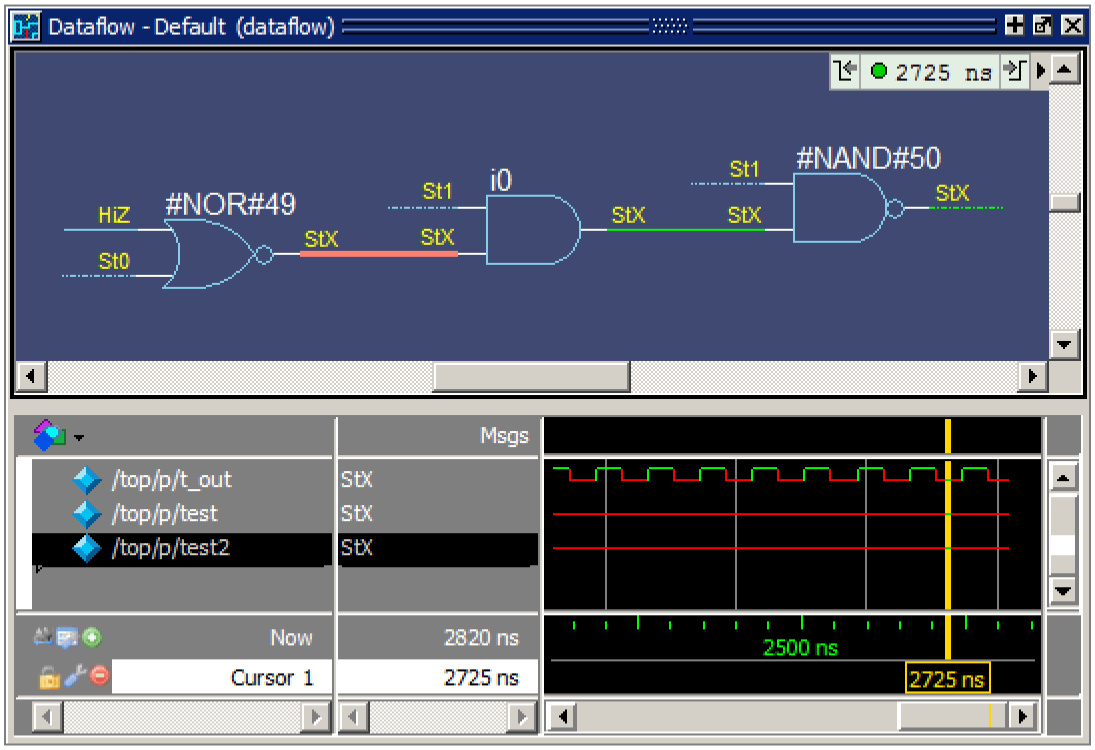
2. Trace the unknown.

   a. In the Dataflow Viewer, click the highlighted signal to make the Viewer active. (A black frame appears around the Dataflow Viewer when it is active. The signal will be orange when selected.)

   b. Select **Tools > Trace > ChaseX** from the menus.

   The design expands to show the source of the unknown state for *t_out* (Figure 10-14). In this case there is a HiZ value (U in the VHDL version) on input signal *test_in* and a St0 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 10-14).

Figure 10-14. ChaseX Identifies Cause of Unknown on t_out



3. Clear the Dataflow window before continuing.

   a. Click the **Delete All** icon to clear the Dataflow Viewer.

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

Displaying Hierarchy in the Dataflow Window

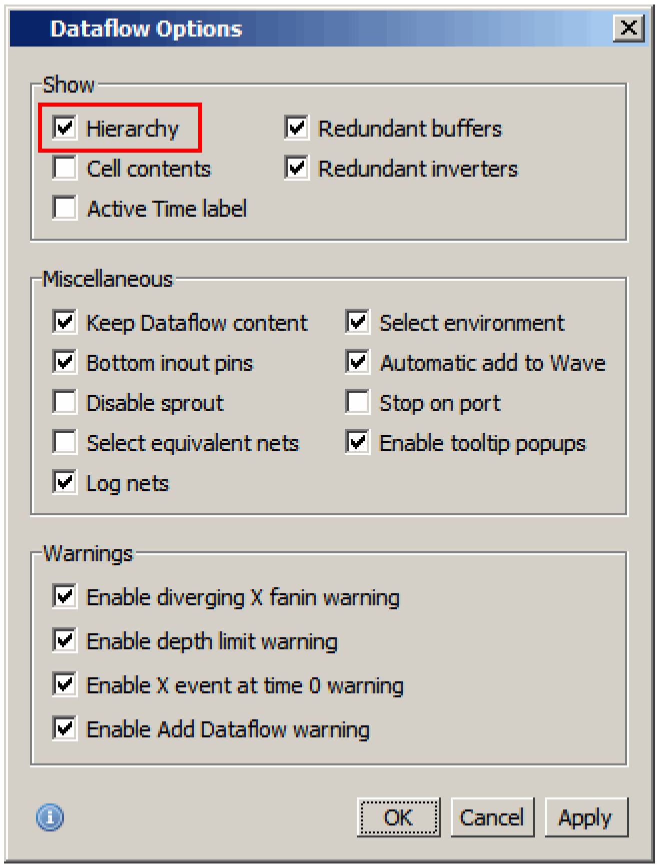
You can display connectivity in the Dataflow window using hierarchical instances. You enable this by modifying the options prior to adding objects to the window.

Procedure

1. Change options to display hierarchy.

   a. Select **Dataflow** > **Dataflow Preferences** > **Options** from the Main window menus. (When the Dataflow window is undocked, select **Tools** > **Options** from the Dataflow window menu bar.) This will open the Dataflow Options dialog box (Figure 10-15).

Figure 10-15. Dataflow Options Dialog Box



Show
[✓] Hierarchy
[ ] Cell contents
[ ] Active Time label
[✓] Redundant buffers
[✓] Redundant inverters

Miscellaneous
[✓] Keep Dataflow content
[✓] Bottom inout pins
[ ] Disable sprout
[ ] Select equivalent nets
[✓] Log nets
[✓] Select environment
[✓] Automatic add to Wave
[ ] Stop on port
[✓] Enable tooltip popups

Warnings
[✓] Enable diverging X fanin warning
[✓] Enable depth limit warning
[✓] Enable X event at time 0 warning
[✓] Enable Add Dataflow warning

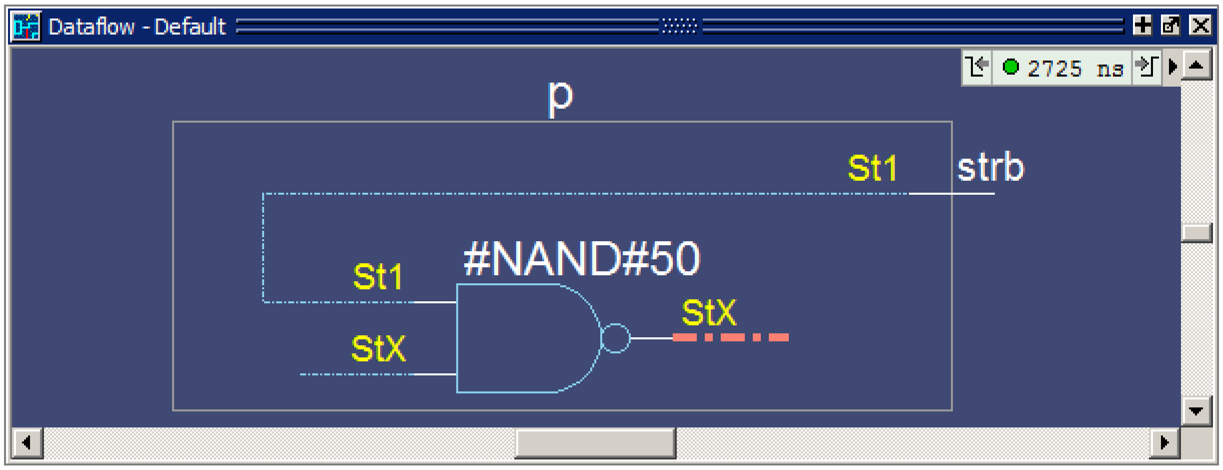
b. Select **Show: Hierarchy** and then click **OK**.

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 10-16).

Figure 10-16. 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.

To return the wildcard filter to its factory default settings, enter:

```
set WildcardFilter "default"
```

Chapter 11
Viewing And Initializing Memories

In this lesson you will learn how to view and initialize memories.

Questa SIM defines and lists any of the following as memories:

- 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 installation comes with Verilog and VHDL versions of the example design.

Example 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 Topics

User's Manual Section: Memory List Window.

Reference Manual commands: mem display, mem load, mem save, and radix.

Compile and Load the Design

Before viewing and initializing memories we need to comple and load a design.

Procedure

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 Questa SIM and change to the exercise directory.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box 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 Questa SIM> prompt.

   b. **Verilog**:
      Type **vlog *.v** at the Questa SIM> prompt to compile all verilog files in the design.

      VHDL:
      Type **vcom -93 sp_syn_ram.vhd dp_syn_ram.vhd ram_tb.vhd** at the Questa SIM> prompt.

4. Optimize the design

   a. Enter the following command at the Questa SIM> prompt:

      ```
      vopt +acc ram_tb -o ram_tb_opt
      ```

      The +acc switch for the vopt command provides visibility into the design for debugging purposes.

      The -o switch allows you designate the name of the optimized design file (ram_tb_opt).

Note

You must provide a name for the optimized design file when you use the vopt command.

5. Load the design.

   a. On the Library tab of the Main window Workspace, click the "+" icon next to the *work* library.

   b. Use the optimized design name to load the design with the vsim command:

      ```
      vsim ram_tb_opt
      ```

View a Memory and its Contents

The Memory List window lists all memory instances in the design, showing for each instance the range, depth, and width. Double-clicking an instance opens a window displaying the memory data.

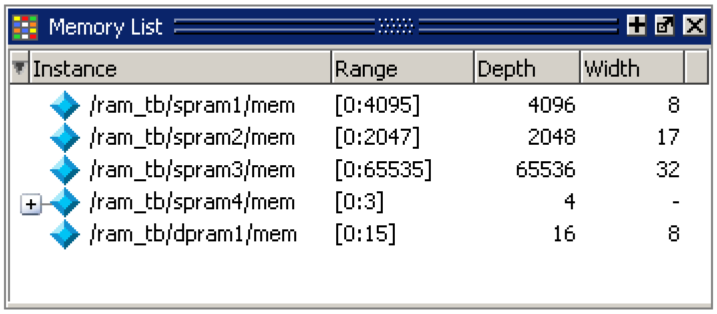
Procedure

1. Open the Memory List window and view the data of a memory instance

   a. If the Memory List window is not already open, select **View > Memory List**.

      A Memory List window is shown in Figure 11-1.

Figure 11-1. The Memory List Window



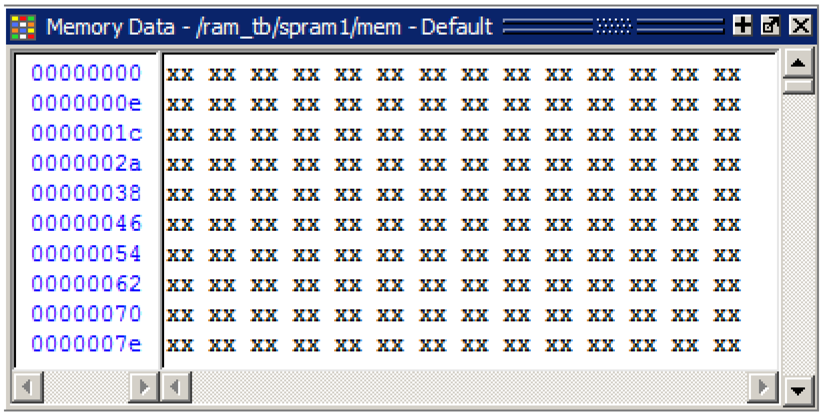
Instance	Range	Depth	Width
/ram_tb/spram1/mem	[0:4095]	4096	8
/ram_tb/spram2/mem	[0:2047]	2048	17
/ram_tb/spram3/mem	[0:65535]	65536	32
/ram_tb/spram4/mem	[0:3]	4	-
/ram_tb/dpram1/mem	[0:15]	16	8

   b. Double-click the */ram_tb/spram1/mem* instance in the memory list to view its contents.

      A Memory Data window opens displaying the contents of spram1. The first column (blue hex characters) lists the addresses, and the remaining columns show the data values.

      If you are using the Verilog example design, the data is all **X** (Figure 11-2) because you have not yet simulated the design.

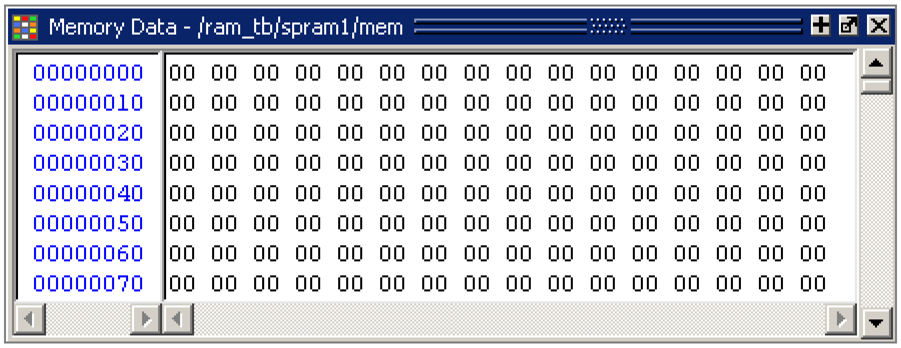
Figure 11-2. Verilog Memory Data Window



```
00000000  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000000e  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000001c  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000002a  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
00000038  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
00000046  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
00000054  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
00000062  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
00000070  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
0000007e  xx xx xx xx xx xx xx xx xx xx xx xx xx xx
```

If you are using the VHDL example design, the data is all zeros (Figure 11-3).

Figure 11-3. VHDL Memory Data Window



```
00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

c. Double-click the instance */ram_tb/spram2/mem* in the Memory List window. This opens a second Memory Data window that contains the addresses and data for the *spram2* instance. For each memory instance that you click in the Memory List window, a new Memory Data window opens.

2. Simulate the design.

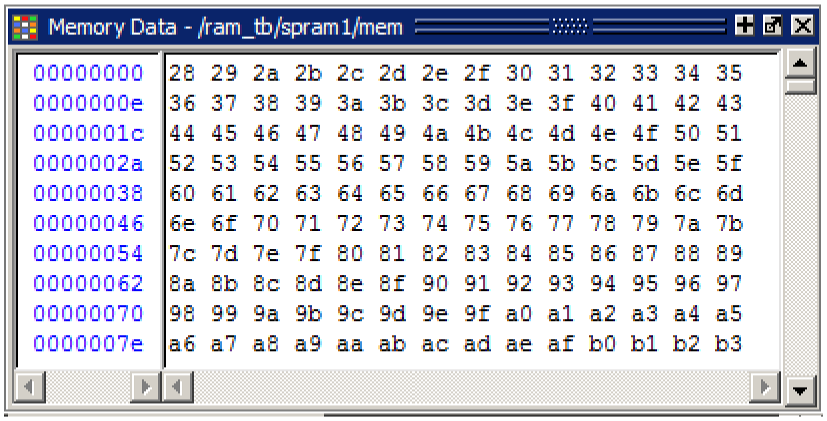
a. Click the **run -all** icon in the Main window.

A Source window opens showing the source code for the *ram_tb* file at the point where the simulation stopped.

VHDL:
In the Transcript window, 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.

a. Click the **Memory ...spram1/mem** tab to bring that Memory data window to the foreground. The Verilog data fields are shown in Figure 11-4.

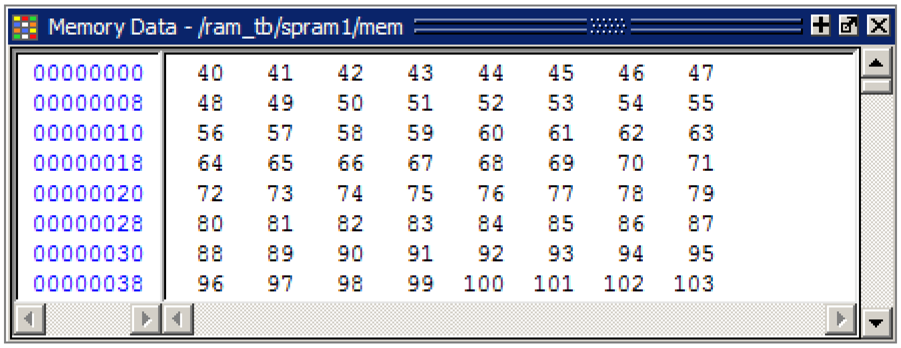
Figure 11-4. Verilog Data After Running Simulation



```
00000000 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35
0000000e 36 37 38 39 3a 3b 3c 3d 3e 3f 40 41 42 43
0000001c 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51
0000002a 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
00000038 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
00000046 6e 6f 70 71 72 73 74 75 76 77 78 79 7a 7b
00000054 7c 7d 7e 7f 80 81 82 83 84 85 86 87 88 89
00000062 8a 8b 8c 8d 8e 8f 90 91 92 93 94 95 96 97
00000070 98 99 9a 9b 9c 9d 9e 9f a0 a1 a2 a3 a4 a5
0000007e a6 a7 a8 a9 aa ab ac ad ae af b0 b1 b2 b3
```

The VHDL data fields are show in Figure 11-5.

Figure 11-5. VHDL Data After Running Simulation


Memory Data - /ram_tb/spram1/mem

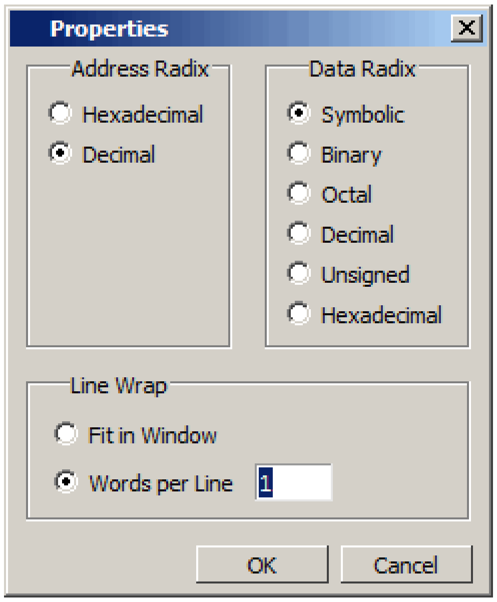
```
00000000   40   41   42   43   44   45   46   47
00000008   48   49   50   51   52   53   54   55
00000010   56   57   58   59   60   61   62   63
00000018   64   65   66   67   68   69   70   71
00000020   72   73   74   75   76   77   78   79
00000028   80   81   82   83   84   85   86   87
00000030   88   89   90   91   92   93   94   95
00000038   96   97   98   99  100  101  102  103
```

3. Change the address radix and the number of words per line for instance */ram_tb/spram1/mem*.

   a. Right-click anywhere in the spram1 Memory Data window and select **Properties**.

   b. The Properties dialog box opens (Figure 11-6).

Figure 11-6. Changing the Address Radix


Properties

Address Radix: Hexadecimal [✓ Decimal]

Data Radix: [✓ Symbolic] Binary Octal Decimal Unsigned Hexadecimal

Line Wrap: Fit in Window [✓ Words per Line] 1

   c. For the **Address Radix, select Decimal**. This changes the radix for the addresses only.

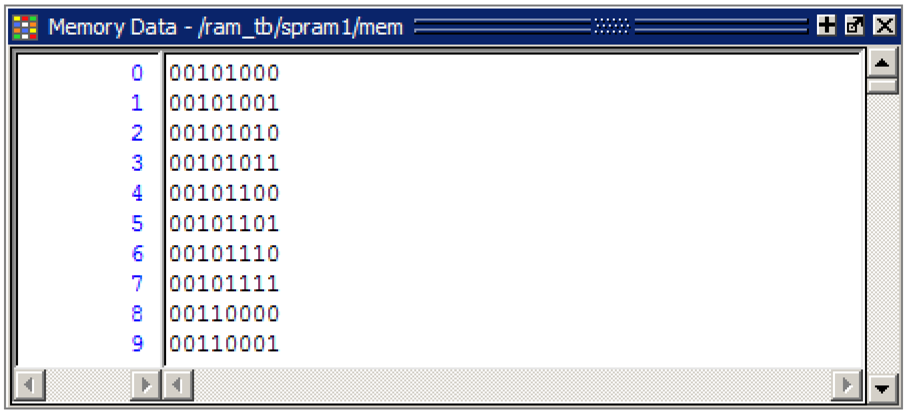
   d. Change the **Data Radix** to **Symbolic**.

   e. Select **Words per line** and type **1** in the field.

   f. Click OK.

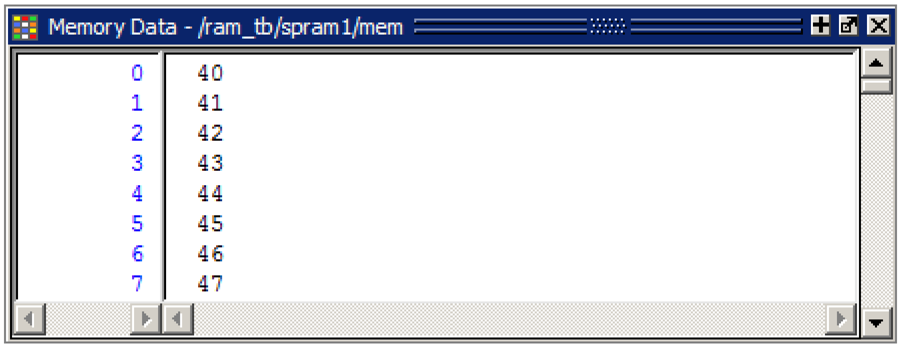
You can see the Verilog results of the settings in Figure 11-7 and the VHDL results in Figure 11-8. If the figure doesn't match what you have in your Questa SIM session, check to make sure you set the Address Radix rather than the Data Radix. Data Radix should still be set to Symbolic, the default.

Figure 11-7. New Address Radix and Line Length (Verilog



```
0 00101000
1 00101001
2 00101010
3 00101011
4 00101100
5 00101101
6 00101110
7 00101111
8 00110000
9 00110001
```

Figure 11-8. New Address Radix and Line Length (VHDL)



```
0  40
1  41
2  42
3  43
4  44
5  45
6  46
7  47
```

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.

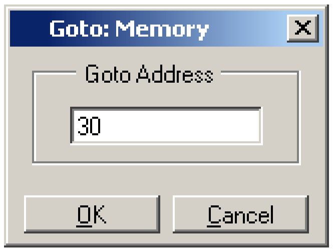
Procedure

1. Use Goto to find a specific address.

   a. Right-click anywhere in address column and select **Goto** (Figure 11-9).

      The Goto dialog box opens in the data pane.

Figure 11-9. Goto Dialog Box



b. Type **30** in the Goto Address field.

c. Click **OK**.

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

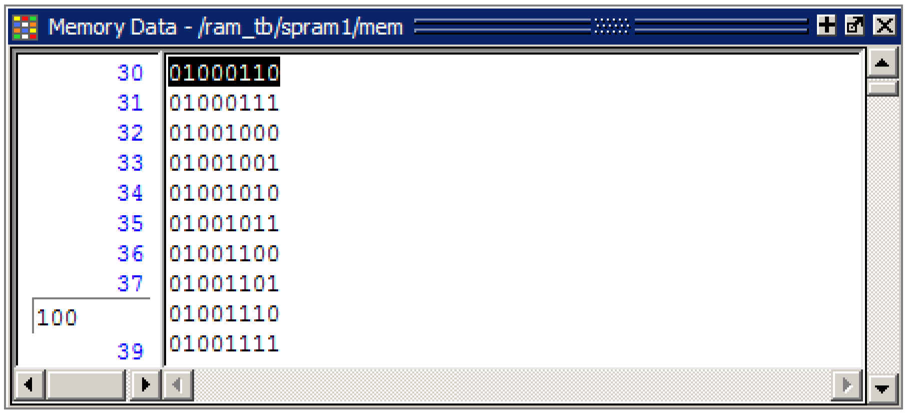
2. Edit the address location directly.

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 11-10).

Figure 11-10. Editing the Address Directly



```
 30  01000110
 31  01000111
 32  01001000
 33  01001001
 34  01001010
 35  01001011
 36  01001100
 37  01001101
100  01001110
 39  01001111
```

iii. Press the Enter or Return key on your keyboard.

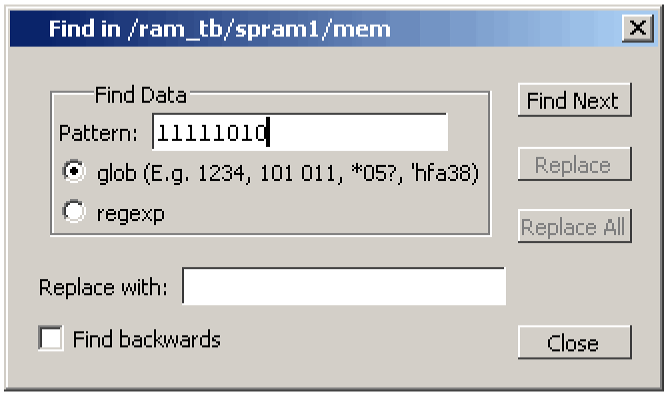
The pane jumps to address 100.

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 11-11).

Figure 11-11. Searching for a Specific Data Value



Find Data
Pattern: 11111010
glob (E.g. 1234, 101 011, *05?, 'hfa38) [✓]
regexp
Replace with:
Find backwards

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

VHDL: Type **250** 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.

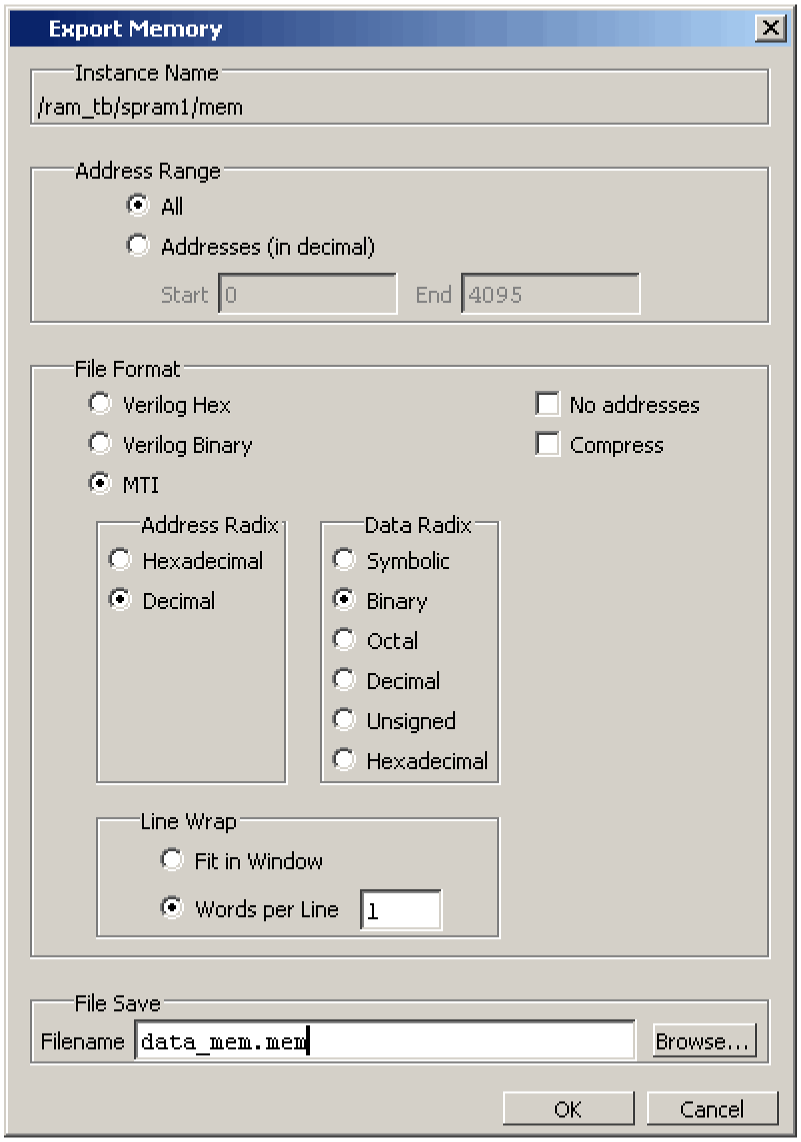
Procedure

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.

   b. Select **File > Export > Memory Data** to bring up the Export Memory dialog box (Figure 11-12).

Figure 11-12. Export Memory Dialog Box



Export Memory

Instance Name
/ram_tb/spram1/mem

Address Range
[✓ All] Addresses (in decimal)
Start 0 End 4095

File Format
Verilog Hex Verilog Binary [✓ MTI]
No addresses ☐
Compress ☐

Address Radix
Hexadecimal [✓ Decimal]

Data Radix
Symbolic [✓ Binary] Octal Decimal Unsigned Hexadecimal

Line Wrap
Fit in Window [✓ Words per Line] 1

File Save
Filename data_mem.mem

c. For the Address Radix, select **Decimal**.

d. For the Data Radix, select **Binary**.

e. For the Words per Line, set to 1.

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 Memory Data window 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 box, 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 box.

    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 **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 Words per Line, set to 1.

    i. Enter the file name as **reloc.mem**, then click OK to save the memory contents and close the dialog box. You will use this file for initialization in the next section.

Initialize a Memory

In Questa SIM, 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 the one you exported previously, *data_mem.mem*.

Procedure

1. View instance */ram_tb/spram3/mem*.

    a. Double-click the */ram_tb/spram3/mem* instance in the Memory List window.

       This will open a new Memory Data window to display the contents of */ram_tb/spram3/mem*. Familiarize yourself with the contents so you can identify changes once the initialization is complete.

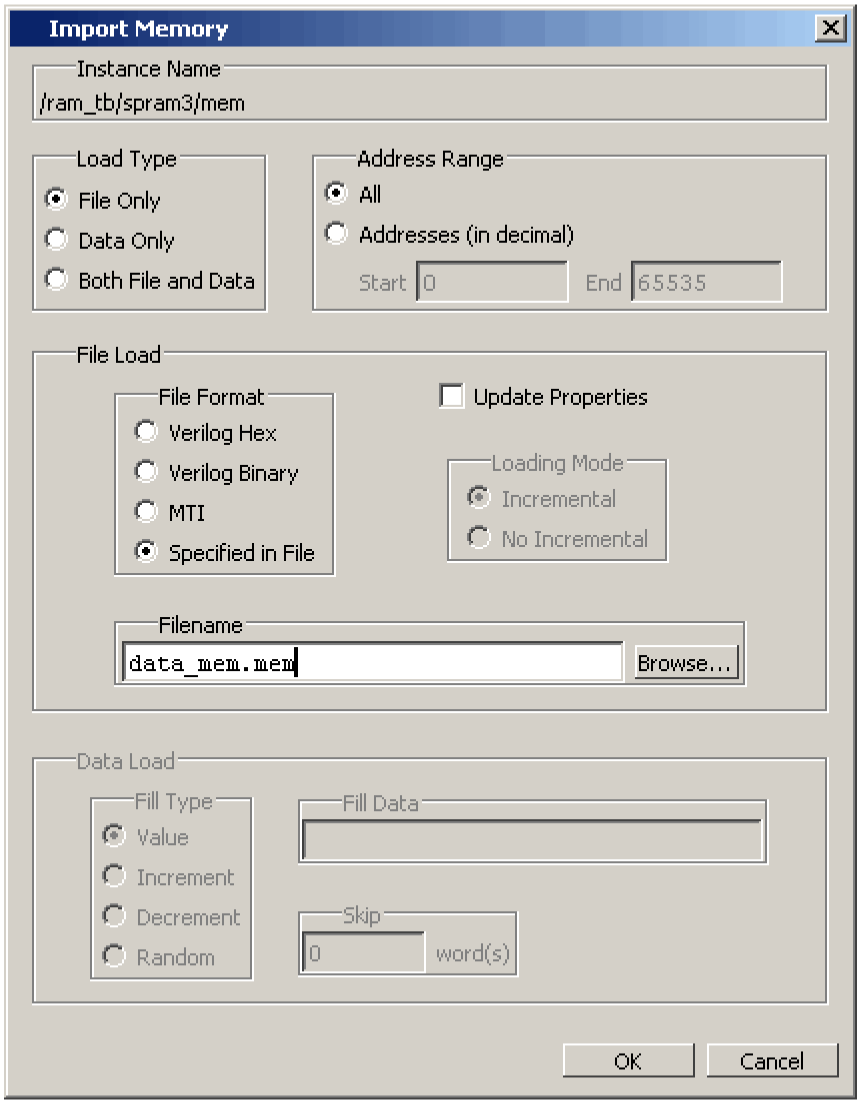
    b. Right-click and select **Properties** to bring up the Properties dialog box.

    c. Change the Address Radix to **Decimal**, Data Radix to **Binary, Words per Line to 1,** and click OK.

2. Initialize *spram3* from a file.

    a. Right-click anywhere in the data column and select **Import Data Patterns** to bring up the Import Memory dialog box (Figure 11-13).

Figure 11-13. Import Memory Dialog Box



Instance Name: /ram_tb/spram3/mem

Load Type: [✓ File Only] Data Only Both File and Data

Address Range: [✓ All] Addresses (in decimal) Start 0 End 65535

File Load

File Format: Verilog Hex Verilog Binary MTI [✓ Specified in File]

Update Properties

Loading Mode: [✓ Incremental] No Incremental

Filename: data_mem.mem

Data Load

Fill Type: [✓ Value] Increment Decrement Random

Fill Data

Skip 0 word(s)

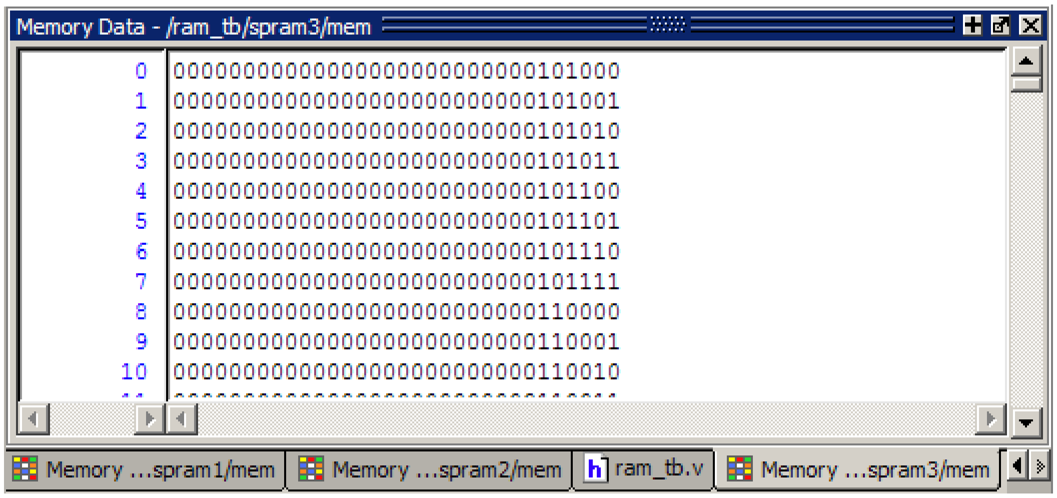
The default Load Type is File Only.

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 11-14).

Figure 11-14. Initialized Memory from File and Fill Pattern



```
 0  00000000000000000000000000101000
 1  00000000000000000000000000101001
 2  00000000000000000000000000101010
 3  00000000000000000000000000101011
 4  00000000000000000000000000101100
 5  00000000000000000000000000101101
 6  00000000000000000000000000101110
 7  00000000000000000000000000101111
 8  00000000000000000000000000110000
 9  00000000000000000000000000110001
10  00000000000000000000000000110010
```

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 *spram3* and select **Import Data Patterns** 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**.

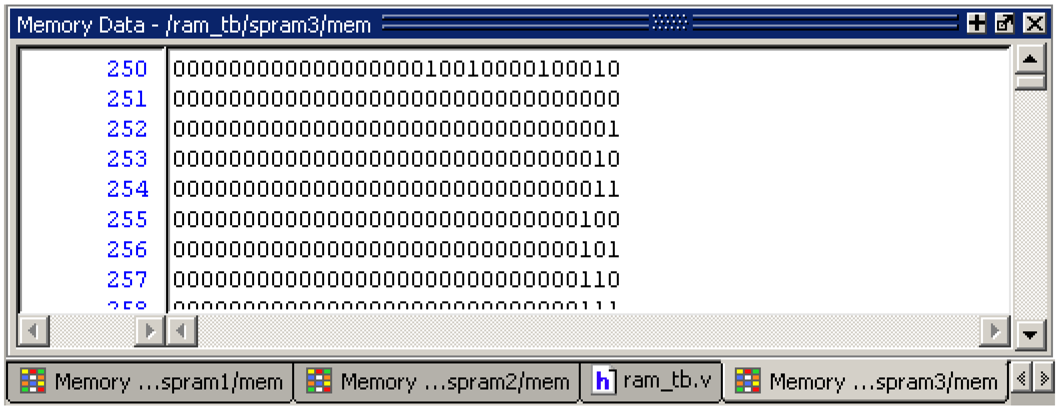
   f. In the Fill Data field, set the seed value of **0** for the incrementing data.

   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 11-15).

Figure 11-15. Data Increments Starting at Address 251



```
250 00000000000000000010010000100010
251 00000000000000000000000000000000
252 00000000000000000000000000000001
253 00000000000000000000000000000010
254 00000000000000000000000000000011
255 00000000000000000000000000000100
256 00000000000000000000000000000101
257 00000000000000000000000000000110
```

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

4. Right-click in one of the Memory Data windows and select **Close All**.

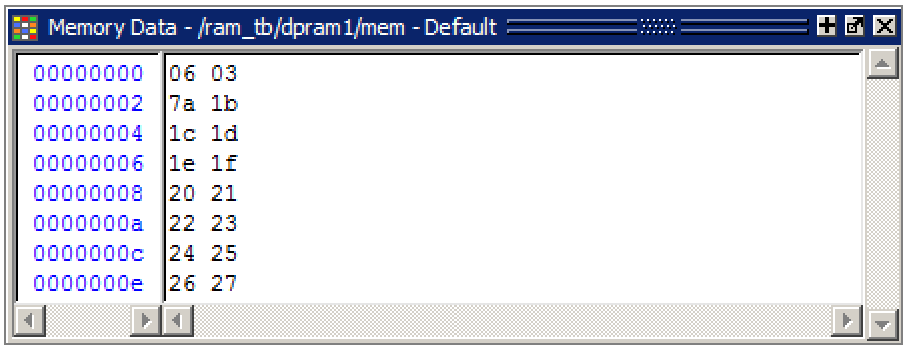
Interactive Debugging Commands

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

Procedure

1. Open a memory instance and change its display characteristics.
   a. Double-click instance */ram_tb/dpram1/mem* in the Memory List window.
   b. Right-click in the *dpram1* Memory Data window 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 11-16.

Figure 11-16. Original Memory Content

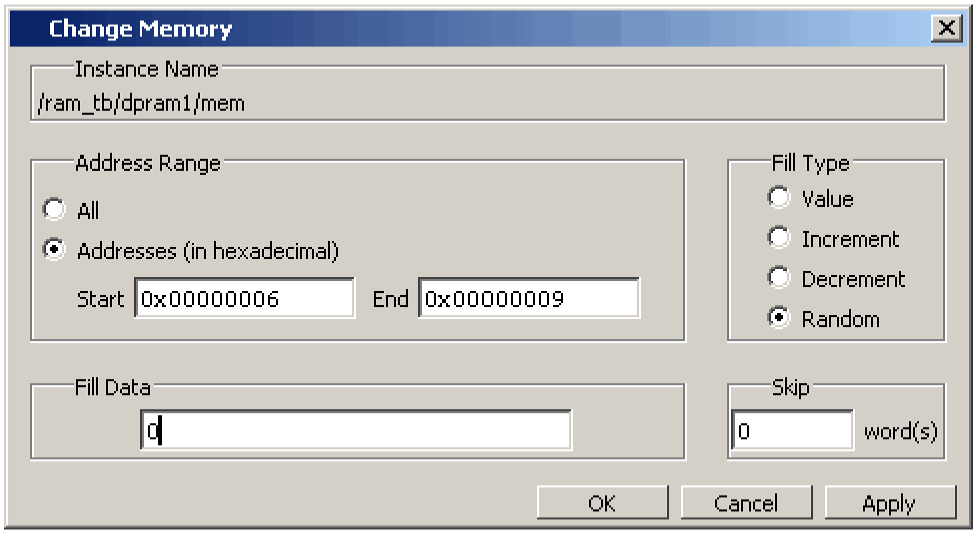


```
00000000 06 03
00000002 7a 1b
00000004 1c 1d
00000006 1e 1f
00000008 20 21
0000000a 22 23
0000000c 24 25
0000000e 26 27
```

2. Initialize a range of memory addresses from a fill pattern.

   a. Right-click in the data column of */ram_tb/dpram1/mem* and select **Change** to open the Change Memory dialog box (Figure 11-17).

Figure 11-17. Changing Memory Content for a Range of AddressesOK**



Instance Name: /ram_tb/dpram1/mem

Address Range: All [✓ Addresses (in hexadecimal)]

Start 0x00000006 End 0x00000009

Fill Type: Value Increment Decrement [✓ Random]

Fill Data: 0

Skip: 0 word(s)

   b. Select **Addresses** and enter the start address as **0x00000006** and the end address as **0x00000009**. The "0x" hex notation is optional.

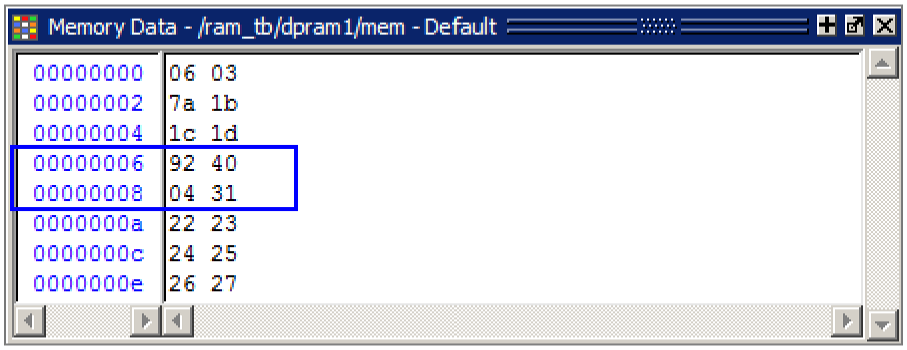
   c. Select **Random** as the **Fill Type**.

   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 11-18).

Figure 11-18. Random Content Generated for a Range of Addresses



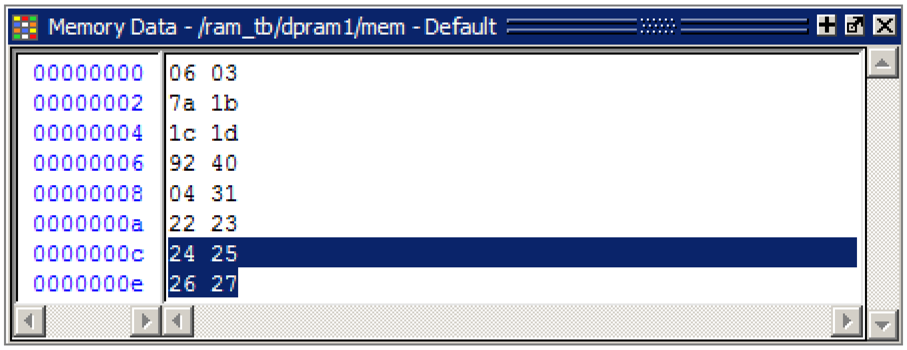
```
00000000 06 03
00000002 7a 1b
00000004 1c 1d
00000006 92 40
00000008 04 31
0000000a 22 23
0000000c 24 25
0000000e 26 27
```

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 11-19.

Figure 11-19. Changing Memory Contents by Highlighting



```
00000000  06 03
00000002  7a 1b
00000004  1c 1d
00000006  92 40
00000008  04 31
0000000a  22 23
0000000c  24 25
0000000e  26 27
```

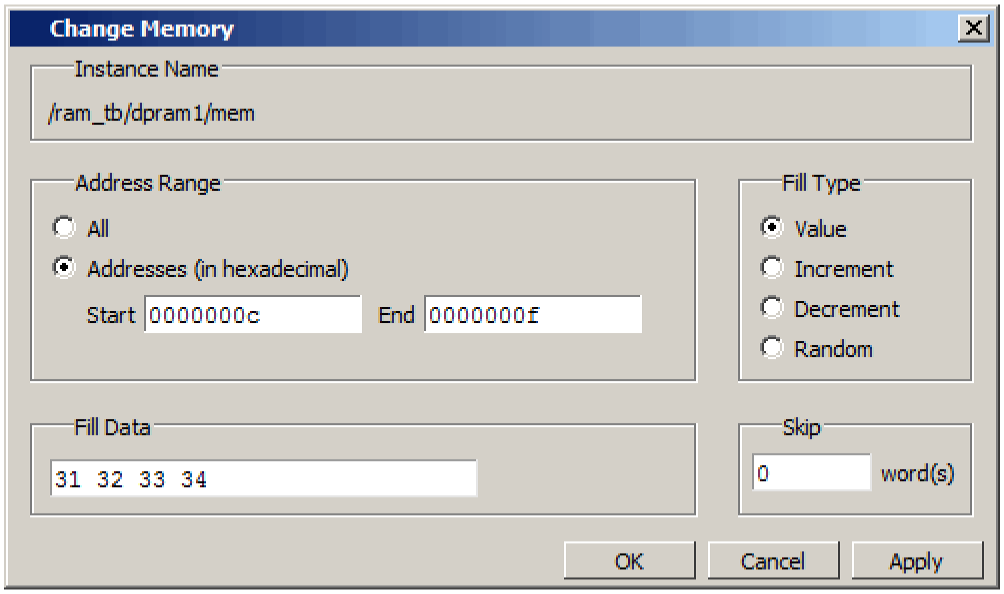
b. Right-click the highlighted data and select **Change**.

This brings up the Change memory dialog box. Note that the Addresses field is already populated with the range you highlighted.

c. Select **Value** as the Fill Type. (Refer to Figure 11-20)

d. Enter the data values into the Fill Data field as follows: **31 32 33 34.**

Figure 11-20. Entering Data to ChangeOK**



Instance Name
/ram_tb/dpram1/mem

Address Range
( ) All
(●) Addresses (in hexadecimal)
Start 0000000c End 0000000f

Fill Type
(●) Value
( ) Increment
( ) Decrement
( ) Random

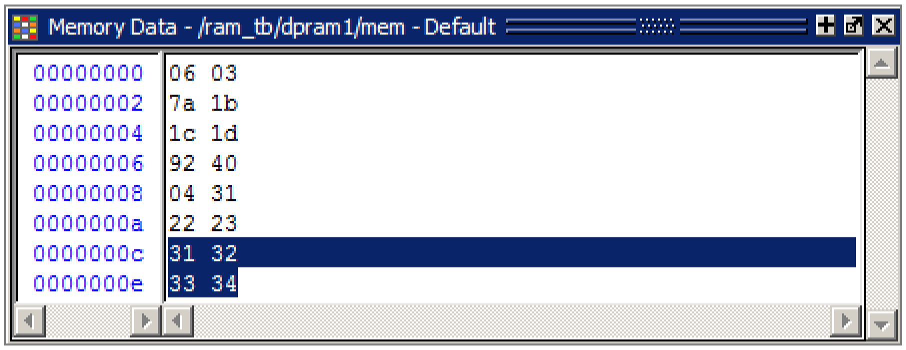
Fill Data
31 32 33 34

Skip
0 word(s)

e. Click **OK**.

The data in the address locations change to the values you entered (Figure 11-21).

Figure 11-21. Changed Memory Contents for the Specified Addresses



```
00000000 06 03
00000002 7a 1b
00000004 1c 1d
00000006 92 40
00000008 04 31
0000000a 22 23
0000000c 31 32
0000000e 33 34
```

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 the Enter or Return key 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 12
Analyzing Performance With The Profiler

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 Questa SIM 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 test bench *test_sm* provides stimulus.

The Questa SIM 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 Topics

User's Manual Chapters: Profiling Performance and Memory Use and Tcl and DO Files.

Compile and Load the Design

Before we can use the Profiler we must compile a design and load it into the simulator.

Procedure

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 Questa SIM and change to the exercise directory.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box 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 Questa SIM> prompt.

4. Compile the design files.

   a. Verilog: Type **vlog *.v** at the Questa SIM> prompt.

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

5. Optimize the design.

   a. Enter the following command at the Questa SIM> prompt in the Transcript window:

      ```
      vopt +acc test_sm -o test_sm_opt
      ```

      The **+acc** switch for the vopt command provides visibility into the design for debugging purposes.

      The **-o** switch allows you designate the name of the optimized design file (test_sm_opt).

Note
You must provide a name for the optimized design file when you use the vopt command.

6. Load the optimized design unit.

   a. Enter **vsim test_sm_opt** at the Questa SIM> prompt.

Run the Simulation

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

Procedure

1. Enable the statistical sampling profiler.

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

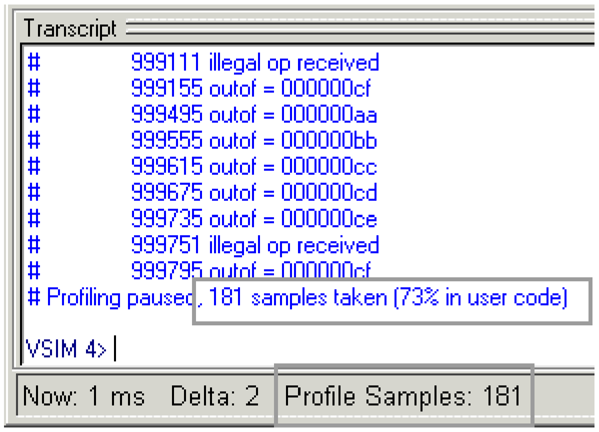
   This must be done prior to running the simulation. Questa SIM is now ready to collect performance data when the simulation is run.

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 12-1). (Your results may not match those in the figure.) Also, Questa SIM reports the percentage of samples that were taken in your design code (versus in internal simulator code).

Figure 12-1. Sampling Reported in the Transcript



```
#           999111 illegal op received
#           999155 outof = 000000cf
#           999495 outof = 000000aa
#           999555 outof = 000000bb
#           999615 outof = 000000cc
#           999675 outof = 000000cd
#           999735 outof = 000000ce
#           999751 illegal op received
#           999795 outof = 000000cf
# Profiling paused, 181 samples taken (73% in user code)

VSIM 4> |
```

Now: 1 ms   Delta: 2   Profile Samples: 181

View Performance Data in Profile Windows

Statistical performance data is displayed in four profile windows: Ranked, Call Tree, Structural, and Design Unit. Additional profile details about those statistics are displayed in the Profile Details window. All of these windows are accessible through the **View > Profiling** menu selection in the Main GUI window.

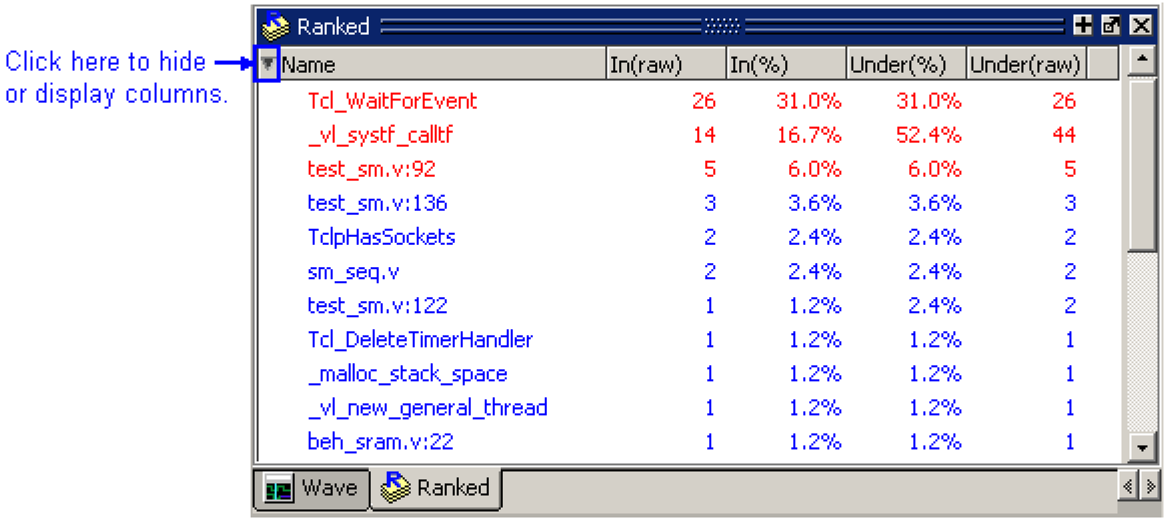
Procedure

1. View ranked performance profile data.

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

The Ranked window displays the results of the statistical performance profiler and the memory allocation profiler for each function or instance (Figure 12-2). By default, ranked performance data is sorted by values in the In% column, which shows the percentage of the total samples collected for each function or instance. (Your results may not match those in the figure.)

Figure 12-2. The Ranked Window



Name	In(raw)	In(%)	Under(%)	Under(raw)
Tcl_WaitForEvent	26	31.0%	31.0%	26
_vl_systf_calltf	14	16.7%	52.4%	44
test_sm.v:92	5	6.0%	6.0%	5
test_sm.v:136	3	3.6%	3.6%	3
TclpHasSockets	2	2.4%	2.4%	2
sm_seq.v	2	2.4%	2.4%	2
test_sm.v:122	1	1.2%	2.4%	2
Tcl_DeleteTimerHandler	1	1.2%	1.2%	1
_malloc_stack_space	1	1.2%	1.2%	1
_vl_new_general_thread	1	1.2%	1.2%	1
beh_sram.v:22	1	1.2%	1.2%	1

You can sort ranked results by any other column by simply clicking the column heading. Or, click the down arrow to the left of the Name column to open a Configure Columns dialog box, which allows you to select which columns are to be hidden or displayed.

The use of colors in the display provides an immediate visual indication of where your design is spending most of its simulation time. By default, red text indicates functions or instances that are consuming 5% or more of simulation time.

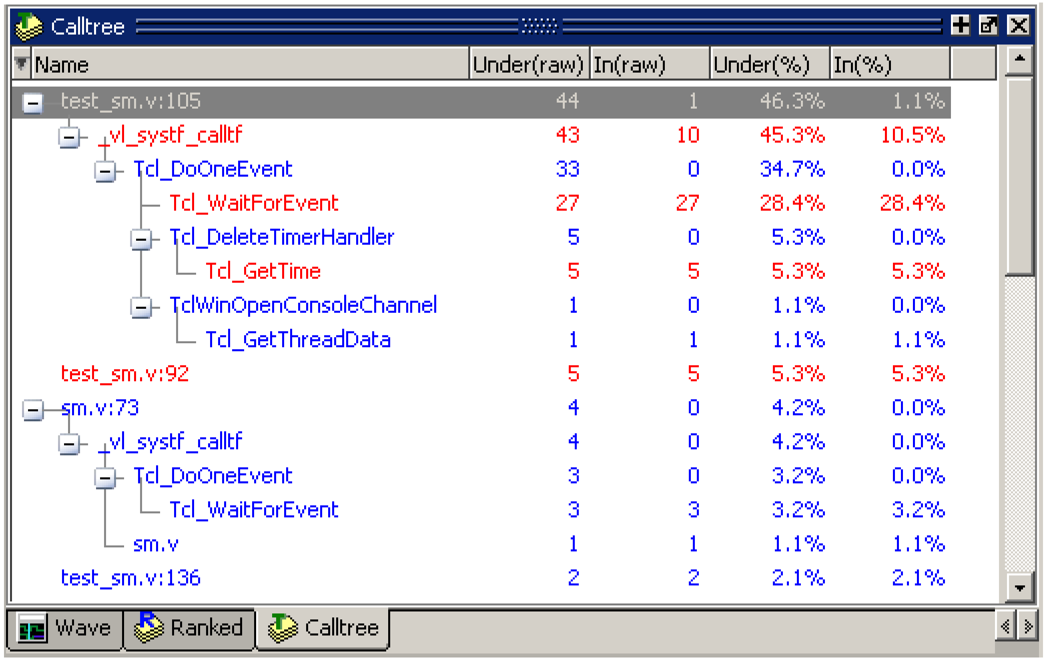
The Ranked tab does not provide hierarchical, function-call information.

2. View performance profile data in a hierarchical, function-call tree display.

a. Select **View** > **Profiling** > **Call Tree Profile**.

b. Right-click in the Calltree window and select **Expand All** from the popup window. This displays the hierarchy of function calls (Figure 12-3). Data is sorted (by default) according to the Under(%) column.

Figure 12-3. Expand the Hierarchical Function Call Tree



Name	Under(raw)	In(raw)	Under(%)	In(%)
test_sm.v:105	44	1	46.3%	1.1%
vl_systf_calltf	43	10	45.3%	10.5%
Tcl_DoOneEvent	33	0	34.7%	0.0%
Tcl_WaitForEvent	27	27	28.4%	28.4%
Tcl_DeleteTimerHandler	5	0	5.3%	0.0%
Tcl_GetTime	5	5	5.3%	5.3%
TclWinOpenConsoleChannel	1	0	1.1%	0.0%
Tcl_GetThreadData	1	1	1.1%	1.1%
test_sm.v:92	5	5	5.3%	5.3%
sm.v:73	4	0	4.2%	0.0%
vl_systf_calltf	4	0	4.2%	0.0%
Tcl_DoOneEvent	3	0	3.2%	0.0%
Tcl_WaitForEvent	3	3	3.2%	3.2%
sm.v	1	1	1.1%	1.1%
test_sm.v:136	2	2	2.1%	2.1%

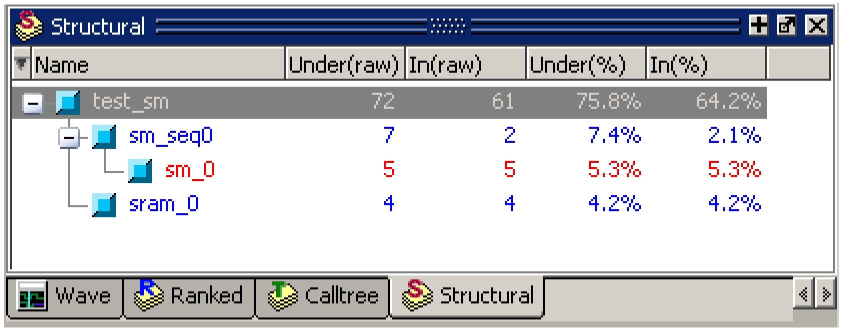
Note
Your results may look slightly different as a result of the computer you're using and different system calls that occur during the simulation. Also, the line number reported may be one or two lines off in the actual source file. This happens due to how the stacktrace is decoded on different platforms.

3. View instance-specific performance profile data in a hierarchical format.

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

   b. Right-click in the Structural profile window and select Expand All from the popup menu. Figure 12-4 displays information found in the Calltree window but adds an additional dimension with which to categorize performance samples. Data is sorted (by default) according to the Under(%) column.

Figure 12-4. Structural Profile Window



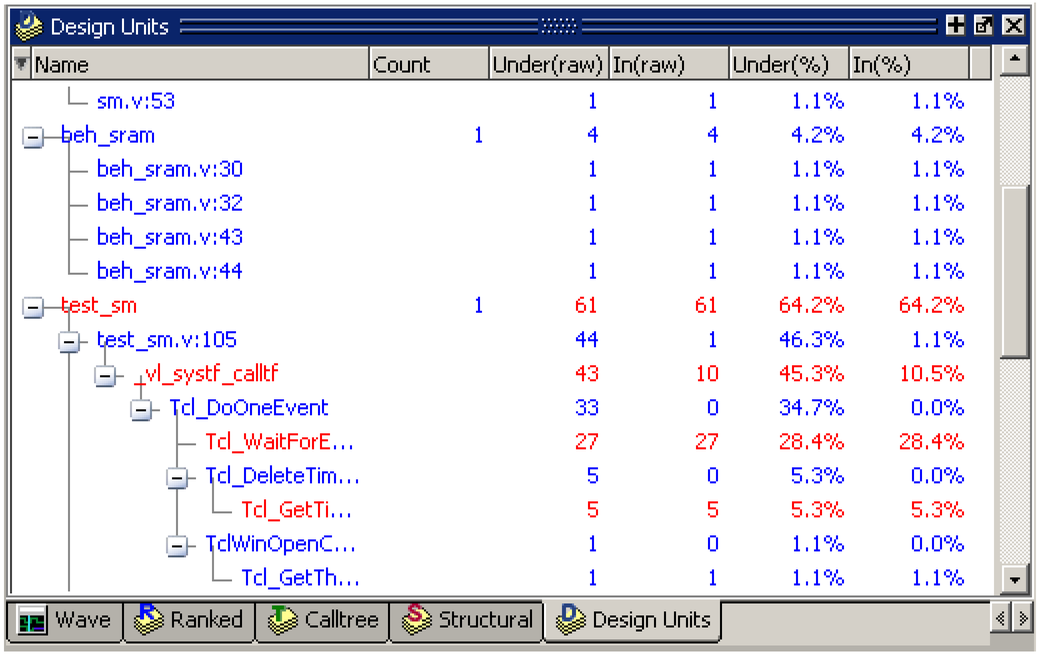
Name	Under(raw)	In(raw)	Under(%)	In(%)
test_sm	72	61	75.8%	64.2%
sm_seq0	7	2	7.4%	2.1%
sm_0	5	5	5.3%	5.3%
sram_0	4	4	4.2%	4.2%

4. View performance profile data organized by design unit.

   a. Select **View** > **Profiling** > **Design Unit Profile**.

      The Design Units profile window provides information similar to the Structural profile window, but organized by design unit, rather than hierarchically. Data is sorted (by default) according to the Under(%) column.

Figure 12-5. Design Unit Performance Profile



Name	Count	Under(raw)	In(raw)	Under(%)	In(%)
sm.v:53		1	1	1.1%	1.1%
beh_sram	1	4	4	4.2%	4.2%
beh_sram.v:30		1	1	1.1%	1.1%
beh_sram.v:32		1	1	1.1%	1.1%
beh_sram.v:43		1	1	1.1%	1.1%
beh_sram.v:44		1	1	1.1%	1.1%
test_sm	1	61	61	64.2%	64.2%
test_sm.v:105		44	1	46.3%	1.1%
vl_systf_calltf		43	10	45.3%	10.5%
Tcl_DoOneEvent		33	0	34.7%	0.0%
Tcl_WaitForE...		27	27	28.4%	28.4%
Tcl_DeleteTim...		5	0	5.3%	0.0%
Tcl_GetTi...		5	5	5.3%	5.3%
TclWinOpenC...		1	0	1.1%	0.0%
Tcl_GetTh...		1	1	1.1%	1.1%

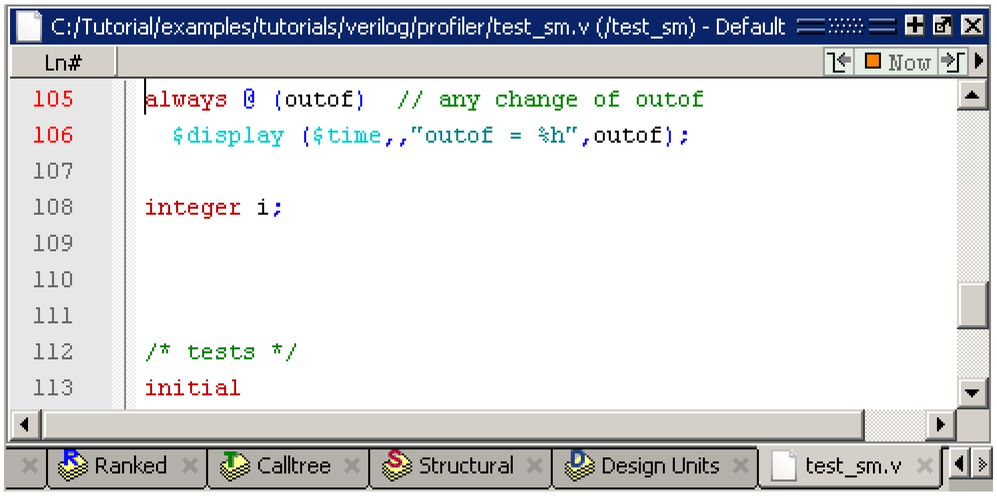
5. View Source Code by Clicking in Profile Window

   The performance profile windows are dynamically linked to the Source window. You can double-click a specific instance, function, design unit, or line and jump directly to the relevant source code in a Source window. You can perform the same task by right-clicking any function, instance, design unit, or line in any of the profile windows and selecting **View Source** from the popup menu.

   a. **Verilog:** Double-click *test_sm.v:105* in the Design Units profile window. The Source window opens with line 105 displayed (Figure 12-6).

      VHDL: Double-click *test_sm.vhd:201*. The Source window opens with line 201 displayed.

Figure 12-6. Source Window Shows Line from Profile Data


C:/Tutorial/examples/tutorials/verilog/profiler/test_sm.v (/test_sm) - Default

```
always @ (outof)  // any change of outof
  $display ($time,,"outof = %h",outof);

integer i;



/* tests */
initial
```

View Profile Details

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

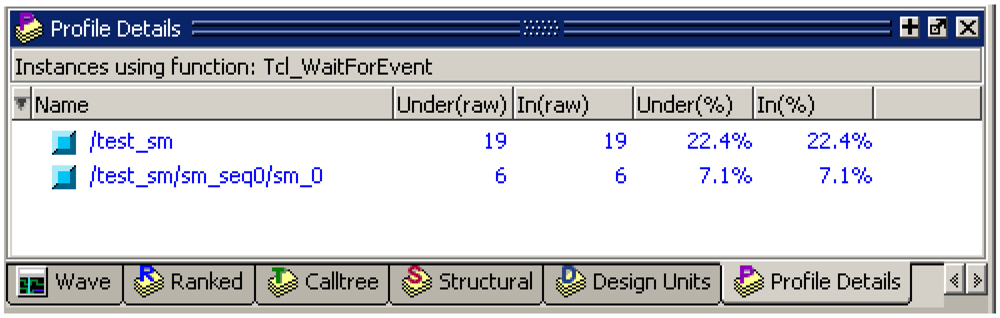
Procedure

1. View the Profile Details of a function in the Call Tree window.

   a. Right-click the *Tcl_WaitForEvent* function and select **Function Usage** from the popup menu.

   The Profile Details window displays all instances using function *Tcl_WaitForEvent* (Figure 12-7). The statistical performance data show how much simulation time is used by *Tcl_WaitForEvent* in each instance.

Figure 12-7. Profile Details of the Function Tcl_Close


Profile Details

Instances using function: Tcl_WaitForEvent

Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm	19	19	22.4%	22.4%
/test_sm/sm_seq0/sm_0	6	6	7.1%	7.1%

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

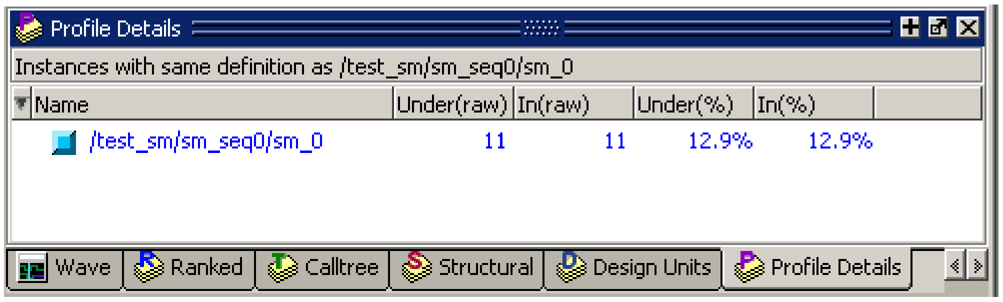
2. View the Profile Details of an instance in the Structural window.

   a. Select the **Structural** tab to change to the Structural profiling window.

   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 12-8).

Figure 12-8. Profile Details of Function sm_0



Instances with same definition as /test_sm/sm_seq0/sm_0

Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm/sm_seq0/sm_0	11	11	12.9%	12.9%

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 the Data

As a last step, you will filter out lines that take less than 3% of the simulation time using the Profiler toolbar.

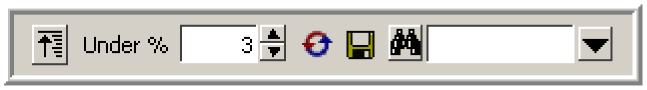
Procedure

1. Filter lines that take less than 3% of the simulation time.

   a. Click the **Calltree** tab to change to the Calltree window.

   b. Change the **Under(%)** field to 3 (Figure 12-9).

Figure 12-9. The Profile Toolbar

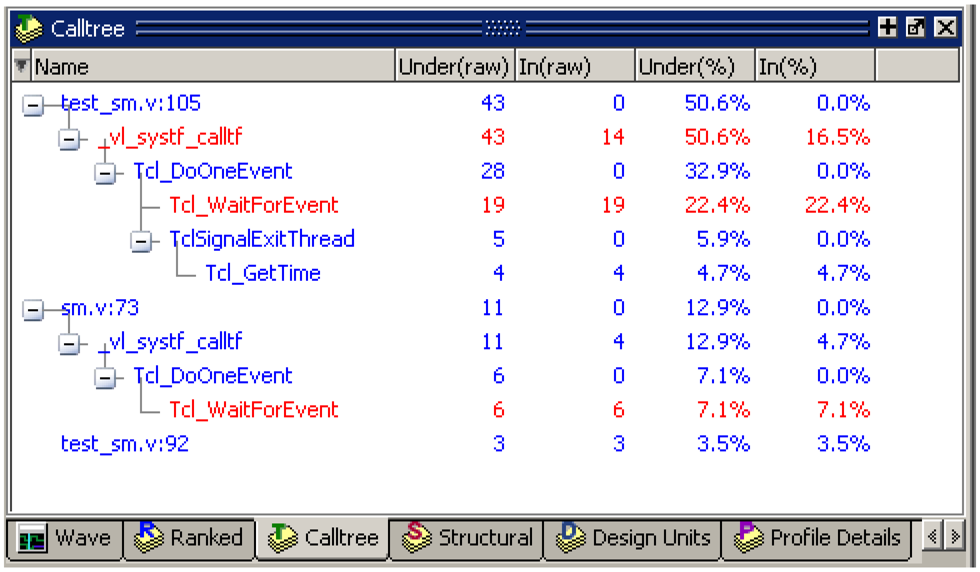


If you do not see these toolbar buttons, right-click in a blank area of the toolbar and select Profile from the popup menu of available toolbars.

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

Questa SIM filters the list to show only those lines that take 3% or more of the simulation time (Figure 12-10).

Figure 12-10. The Filtered Profile Data



Name	Under(raw)	In(raw)	Under(%)	In(%)
test_sm.v:105	43	0	50.6%	0.0%
vl_systf_calltf	43	14	50.6%	16.5%
Tcl_DoOneEvent	28	0	32.9%	0.0%
Tcl_WaitForEvent	19	19	22.4%	22.4%
TclSignalExitThread	5	0	5.9%	0.0%
Tcl_GetTime	4	4	4.7%	4.7%
sm.v:73	11	0	12.9%	0.0%
vl_systf_calltf	11	4	12.9%	4.7%
Tcl_DoOneEvent	6	0	7.1%	0.0%
Tcl_WaitForEvent	6	6	7.1%	7.1%
test_sm.v:92	3	3	3.5%	3.5%

Creating a Performance Profile Report

Questa SIM allows you to create different profile reports based on the profiler data.

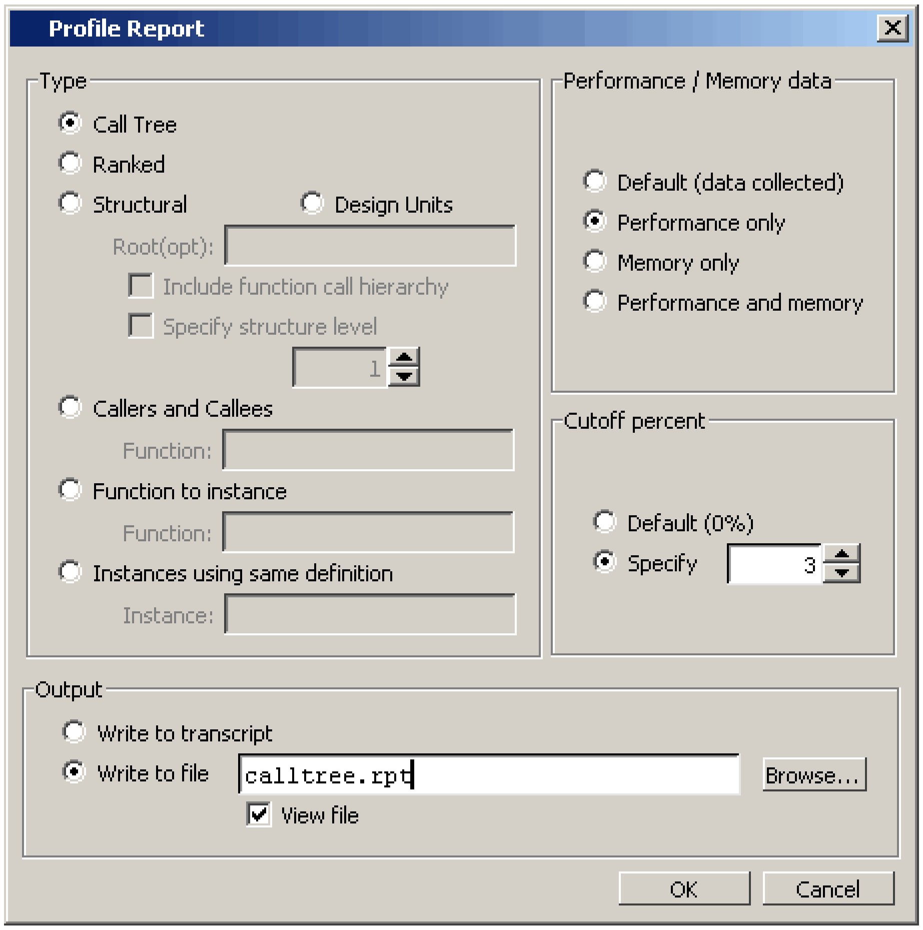
Procedure

1. Create a call tree type report of the performance profile.

   a. With the Calltree window open, select **Tools > Profile > Profile Report** from the menus to open the Profile Report dialog box.

   b. In the Profile Report dialog box (Figure 12-11), select the **Call Tree** Type.

Figure 12-11. The Profile Report Dialog Box



Profile Report

Type
- [● Call Tree]
- ○ Ranked
- ○ Structural ○ Design Units
  - Root(opt):
  - ☐ Include function call hierarchy
  - ☐ Specify structure level 1
- ○ Callers and Callees
  - Function:
- ○ Function to instance
  - Function:
- ○ Instances using same definition
  - Instance:

Performance / Memory data
- ○ Default (data collected)
- [● Performance only]
- ○ Memory only
- ○ Performance and memory

Cutoff percent
- ○ Default (0%)
- [● Specify] 3

Output
- ○ Write to transcript
- [● Write to file] calltree.rpt Browse...
- ☑ View file

OK Cancel

c. In the Performance/Memory data section select **Performance only**.

d. Specify the **Cutoff percent** as 3%.

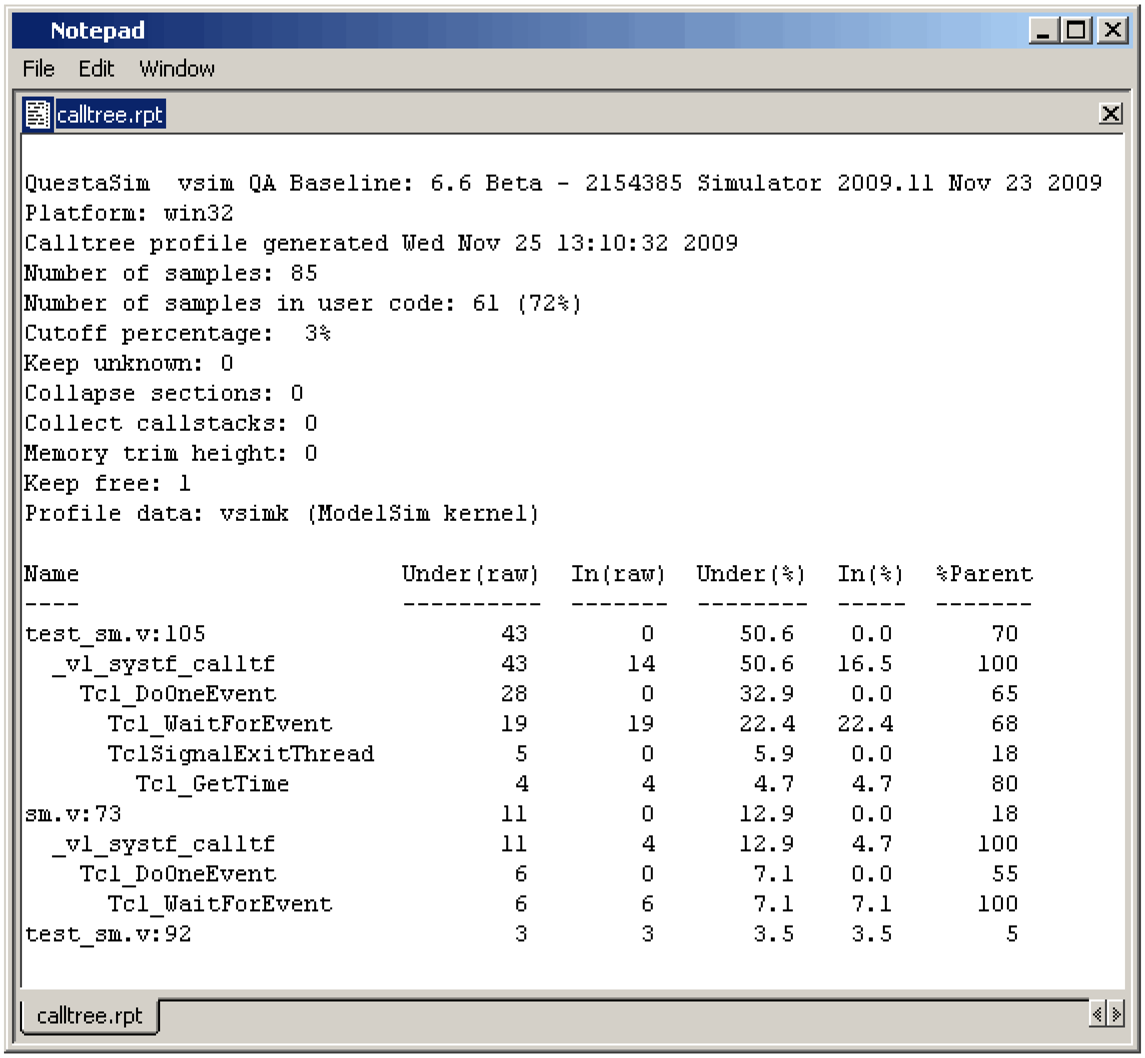
e. Select **Write to file** and type **calltree.rpt** in the file **name** field.

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 12-12).

Figure 12-12. The calltree.rpt Report



```
QuestaSim  vsim QA Baseline: 6.6 Beta - 2154385 Simulator 2009.11 Nov 23 2009
Platform: win32
Calltree profile generated Wed Nov 25 13:10:32 2009
Number of samples: 85
Number of samples in user code: 61 (72%)
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                       43        0      50.6    0.0       70
  _vl_systf_calltf                  43       14      50.6   16.5      100
    Tcl_DoOneEvent                  28        0      32.9    0.0       65
      Tcl_WaitForEvent              19       19      22.4   22.4       68
      TclSignalExitThread            5        0       5.9    0.0       18
        Tcl_GetTime                  4        4       4.7    4.7       80
sm.v:73                             11        0      12.9    0.0       18
  _vl_systf_calltf                  11        4      12.9    4.7      100
    Tcl_DoOneEvent                   6        0       7.1    0.0       55
      Tcl_WaitForEvent               6        6       7.1    7.1      100
test_sm.v:92                         3        3       3.5    3.5        5
```

You can also output this report from the command line using the **profile report** command. See the *Questa SIM 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 13
Simulating With Code Coverage

Questa SIM 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.

Design Files for this Lesson

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

The Questa SIM 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 Topics

User's Manual Chapter: Code Coverage.

Compile the Design

Enabling Code Coverage is a simple process: You compile the design files and identify which coverage statistics you want to collect. Then you load the design and tell Questa SIM to produce those statistics.

Procedure

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 Questa SIM and change to the exercise directory.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box 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 Questa SIM> prompt.

4. Compile all design files.

   a. For Verilog – Type **vlog *.v** at the Questa SIM> prompt.

      For VHDL – Type **vcom *.vhd** at the Questa SIM> prompt.

5. Designate the coverage statistics you want to collect.

   a. Type **vopt +cover=bcesxf test_sm -o test_sm_opt** at the Questa SIM> prompt.

      The **+cover=bcesxf** argument instructs Questa SIM to collect branch, condition, expression statement, extended toggle, and finite state machine coverage statistics. Refer to the Overview of Code Coverage Types in the User's Manual for more information on the available coverage types.

      The **-o** argument is used to designate a name (in this case, *test_sm_opt*) for the optimized design. This argument is required with the vopt command.

Note

By default, Questa SIM optimizations are performed on all designs (see Optimizing Designs with vopt).

Load and Run the Design

After designating the type of coverage statistics we want to collect, we're ready to load and run the design.

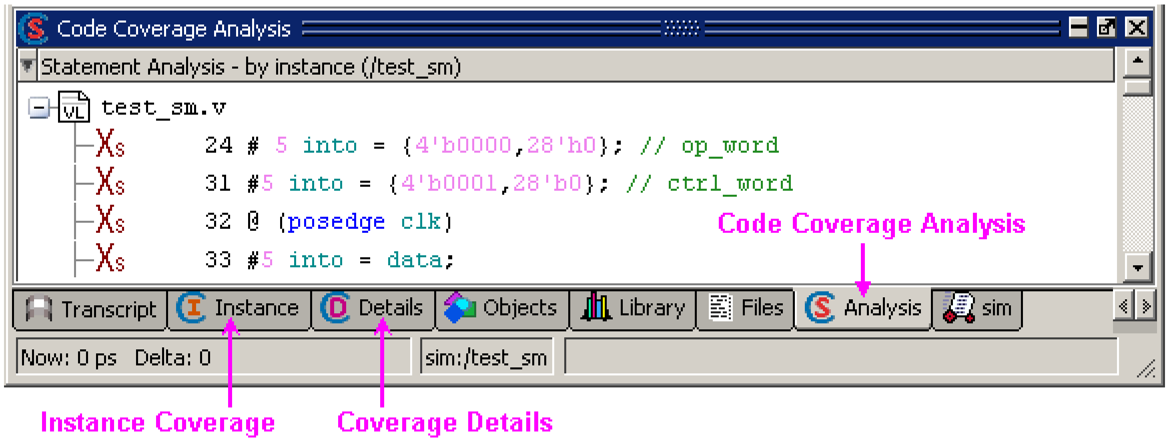
Procedure

1. Load the design.

   a. Enter **vsim -coverage test_sm_opt** at the Questa SIM> prompt. (The optimized design is loaded.)

      Three code coverage windows will open: Code Coverage Analysis, Instance Coverage, and Coverage Details (Figure 13-1).

Figure 13-1. Code Coverage Windows



```
test_sm.v
Xs    24 # 5 into = {4'b0000,28'h0}; // op_word
Xs    31 #5 into = {4'b0001,28'b0}; // ctrl_word
Xs    32 @ (posedge clk)
Xs    33 #5 into = data;
```

Within the Code Coverage Analysis window you can perform statement, branch, condition, expression, FSM, and toggle coverage analysis. Each line in the Code Coverage analysis window includes an icon that indicates whether elements in the line (statements, branches, conditions, or expressions) were executed, not executed, or excluded. Table 13-1 displays the Code Coverage icons.

Table 13-1. Code Coverage Icons

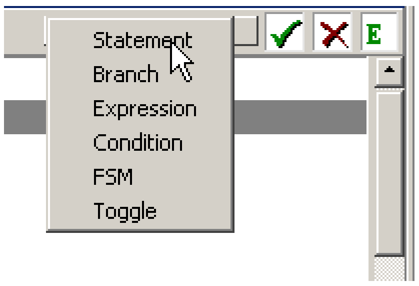
Icon	Description/Indication
✓	All statements, branches, conditions, or expressions on a particular line have been executed
X	Multiple kinds of coverage on the line were not executed
XT	True branch not executed (BC column)
XF	False branch not executed (BC column)
XC	Condition not executed (Hits column)
XE	Expression not executed (Hits column)
XB	Branch not executed (Hits column)
XS	Statement not executed (Hits column)
E	Indicates a line of code to which active coverage exclusions have been applied. Every item on the line is excluded; none are hit.

Table 13-1. Code Coverage Icons

Icon	Description/Indication
E☆	Some excluded items are hit
E✓	Some items are excluded, and all items not excluded are hit
E✗	Some items are excluded, and some items not excluded have missing coverage
EA	Auto exclusions have been applied to this line. Hover the cursor over the E_A and a tool tip balloon appears with the reason for exclusion,

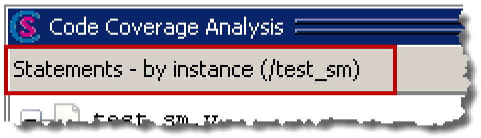
You can select the analysis you want to perform in the Analysis toolbar (Figure 13-2).

Figure 13-2. Analysis Toolbar



You can identify which analysis is currently open by the title bar in the Code Coverage Analysis window (Figure 13-3).

Figure 13-3. Title Bar Displays Current Analysis



Code Coverage Analysis

Statements - by instance (/test_sm)

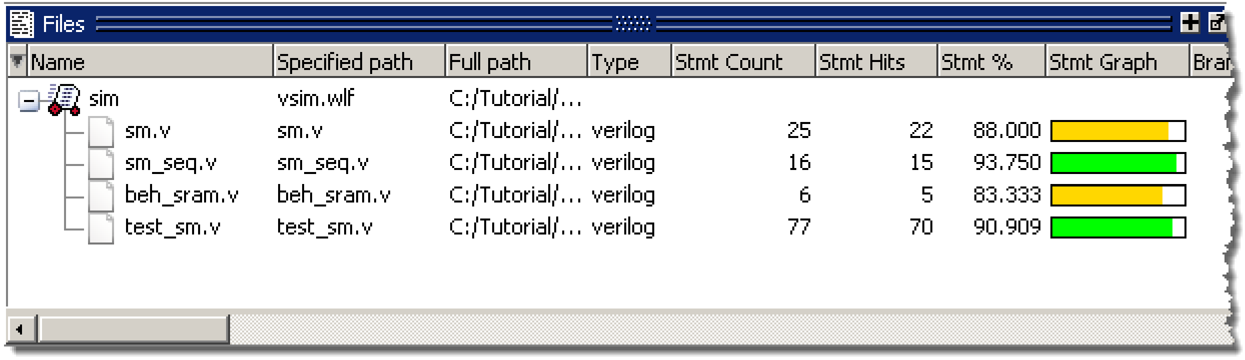
By default, Statement Analysis is displayed the first time the Code Coverage Analysis window opens. For subsequent invocations, the last-chosen analysis window is displayed.

2. Run the simulation

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

When you load a design with Code Coverage enabled, Questa SIM adds several coverage data columns to the Files and Structure (sim) windows (Figure 13-4). Use the horizontal scroll bar to see more coverage data columns. (Your results may not match those shown in the figure.)

Figure 13-4. Code Coverage Columns in the Structure (sim) Window

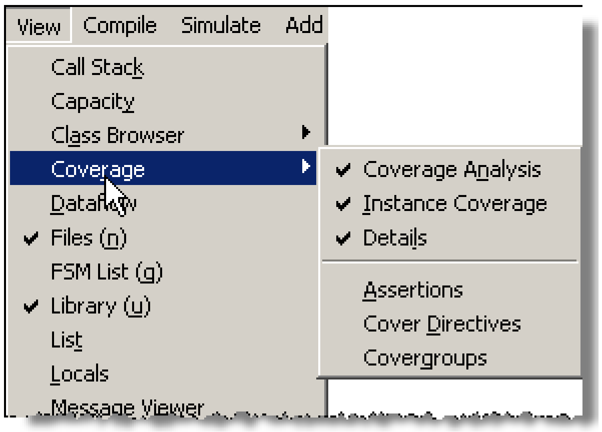


Files

Name	Specified path	Full path	Type	Stmt Count	Stmt Hits	Stmt %	Stmt Graph	Bran
sim	vsim.wlf	C:/Tutorial/...						
sm.v	sm.v	C:/Tutorial/...	verilog	25	22	88.000	🟨	
sm_seq.v	sm_seq.v	C:/Tutorial/...	verilog	16	15	93.750	🟩	
beh_sram.v	beh_sram.v	C:/Tutorial/...	verilog	6	5	83.333	🟨	
test_sm.v	test_sm.v	C:/Tutorial/...	verilog	77	70	90.909	🟩	

You can open and close coverage windows with the **View** > **Coverage** menu selection.

Figure 13-5. Coverage Menu



All coverage windows can be re-sized, rearranged, and undocked to make the data more easily viewable. To resize a window, click-and-drag on any border. To move a window, click-and-drag on the header handle (three rows of dots in the middle of the header) or click and drag the tab. To undock a window you can select it then drag it out of the Main window, or you can click the Dock/Undock button in the header bar (top right). To redock the window, click the Dock/Undock button again.

We will look at some of the coverage windows more closely in the next exercise.

Viewing Coverage Data

Let's take a look at the coverage data displayed in different coverage windows.

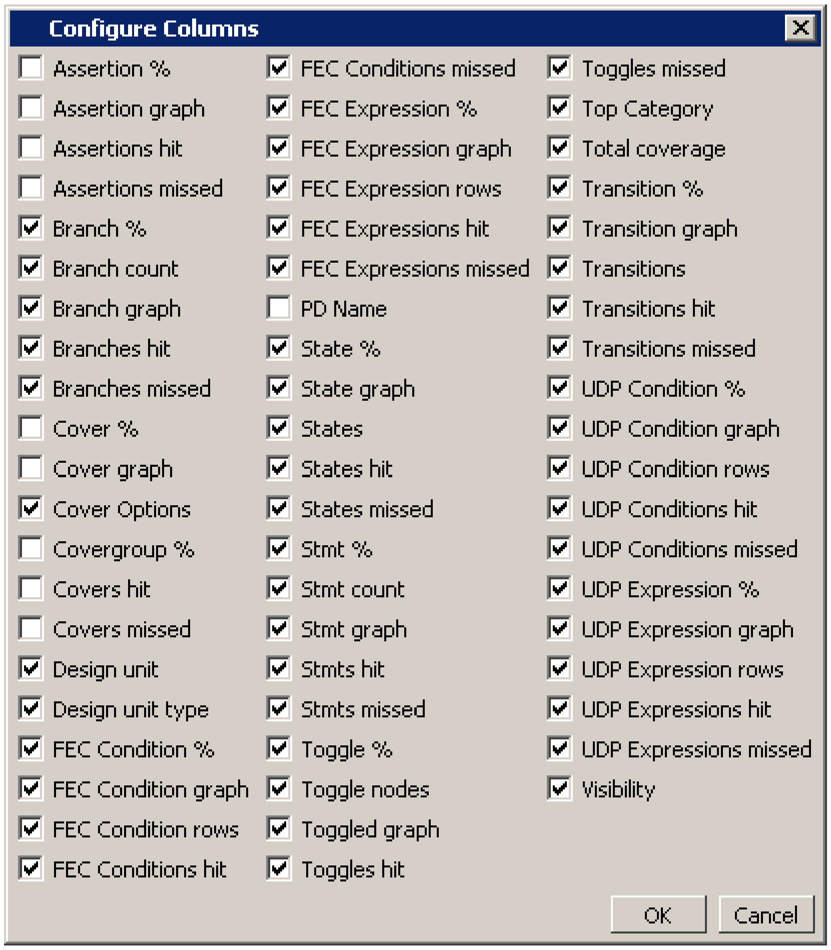
Procedure

1. View coverage data in the Structure (sim) window.

a. Select the **sim** tab and use the horizontal scroll bar to view coverage data in the coverage columns. Coverage data is shown for each object in the design.

b. Select the **Files** tab to switch to the Files window and scroll to the right. You can change which coverage data columns are displayed by right clicking on any column name, selecting **Change Column Visibility**, and selecting columns from the popup list.

Figure 13-6. Right-click a Column Heading to Show Column List



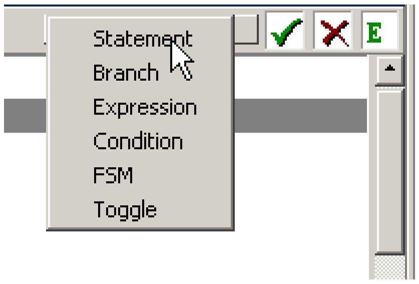
☐ Assertion %	☑ FEC Conditions missed	☑ Toggles missed
☐ Assertion graph	☑ FEC Expression %	☑ Top Category
☐ Assertions hit	☑ FEC Expression graph	☑ Total coverage
☐ Assertions missed	☑ FEC Expression rows	☑ Transition %
☑ Branch %	☑ FEC Expressions hit	☑ Transition graph
☑ Branch count	☑ FEC Expressions missed	☑ Transitions
☑ Branch graph	☐ PD Name	☑ Transitions hit
☑ Branches hit	☑ State %	☑ Transitions missed
☑ Branches missed	☑ State graph	☑ UDP Condition %
☐ Cover %	☑ States	☑ UDP Condition graph
☐ Cover graph	☑ States hit	☑ UDP Condition rows
☑ Cover Options	☑ States missed	☑ UDP Conditions hit
☐ Covergroup %	☑ Stmt %	☑ UDP Conditions missed
☐ Covers hit	☑ Stmt count	☑ UDP Expression %
☐ Covers missed	☑ Stmt graph	☑ UDP Expression graph
☑ Design unit	☑ Stmts hit	☑ UDP Expression rows
☑ Design unit type	☑ Stmts missed	☑ UDP Expressions hit
☑ FEC Condition %	☑ Toggle %	☑ UDP Expressions missed
☑ FEC Condition graph	☑ Toggle nodes	☑ Visibility
☑ FEC Condition rows	☑ Toggled graph	
☑ FEC Conditions hit	☑ Toggles hit	

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

2. View coverage data in the Statement Analysis view of the Code Coverage Analysis window.

   a. If the Statement Analysis view is not displayed in the Code Coverage Analysis window, select Statement Analysis from the Analysis toolbar (Figure 13-7).

Figure 13-7. Select Statement Analysis



b. Select different files from the Files window. The Code Coverage Analysis window updates to show coverage data for the selected file in the Statement Analysis view.

c. Double-click any entry in the Statement Analysis view to display that line in a Source window.

3. View toggle coverage details in the Coverage Details window.

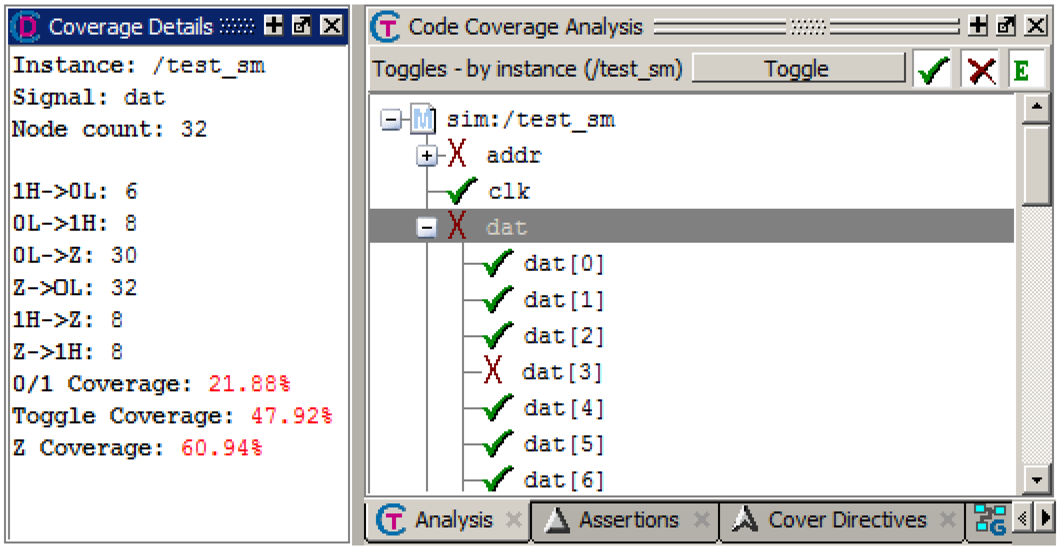
a. Switch to the Toggle Analysis view in the Code Coverage Analysis window by selecting the Toggle Analysis in the Analysis Toolbar (Figure 13-7).

b. Click the Details tab to open the Coverage Details window.

If the Details tab is not visible, select **View > Coverage > Details** from the Main menu.

c. Select any object in the Toggle Analysis and view its coverage details in the Coverage Details window (Figure 13-8).

Figure 13-8. Coverage Details Window Undocked



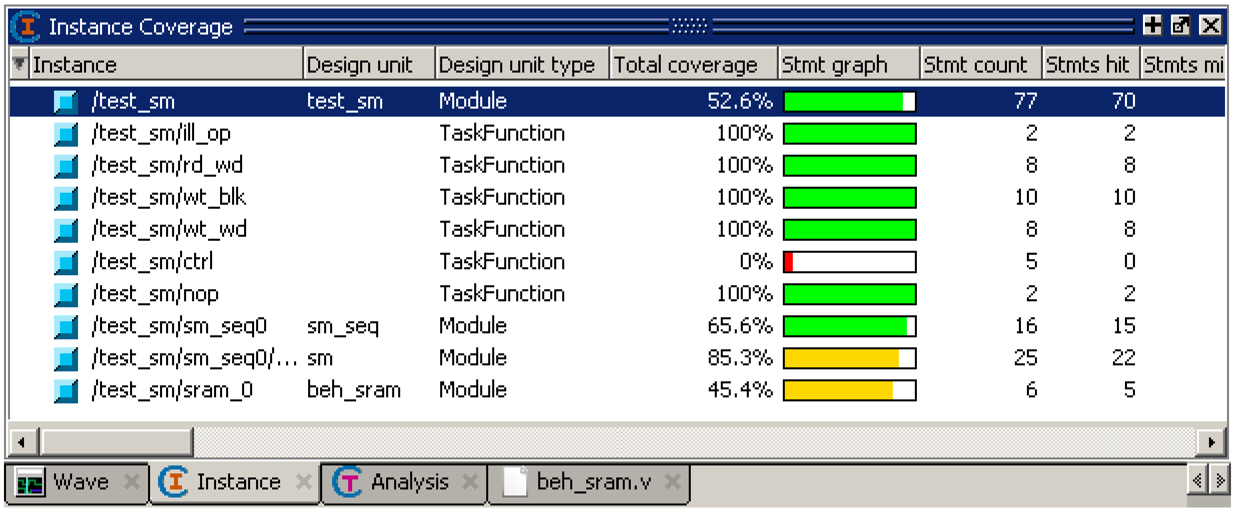
```
Instance: /test_sm
Signal: dat
Node count: 32

1H->0L: 6
0L->1H: 8
0L->Z: 30
Z->0L: 32
1H->Z: 8
Z->1H: 8
0/1 Coverage: 21.88%
Toggle Coverage: 47.92%
Z Coverage: 60.94%
```

4. View instance coverage data.

a. Click the Instance tab to switch to the Instance Coverage window. If the Instance tab is not visible, select **View** > **Coverage** > **Instance Coverage** from the Main menu.

The Instance Coverage window displays coverage statistics for each instance in a flat, non-hierarchical view. Double-click any instance in the Instance Coverage window to see its source code displayed in the Source window.

Figure 13-9. Instance Coverage Window



Instance	Design unit	Design unit type	Total coverage	Stmt graph	Stmt count	Stmts hit	Stmts mi
/test_sm	test_sm	Module	52.6%		77	70	
/test_sm/ill_op		TaskFunction	100%		2	2	
/test_sm/rd_wd		TaskFunction	100%		8	8	
/test_sm/wt_blk		TaskFunction	100%		10	10	
/test_sm/wt_wd		TaskFunction	100%		8	8	
/test_sm/ctrl		TaskFunction	0%		5	0	
/test_sm/nop		TaskFunction	100%		2	2	
/test_sm/sm_seq0	sm_seq	Module	65.6%		16	15	
/test_sm/sm_seq0/...	sm	Module	85.3%		25	22	
/test_sm/sram_0	beh_sram	Module	45.4%		6	5	

Coverage Statistics in the Source Window

The Source window contains coverage statistics of its own.

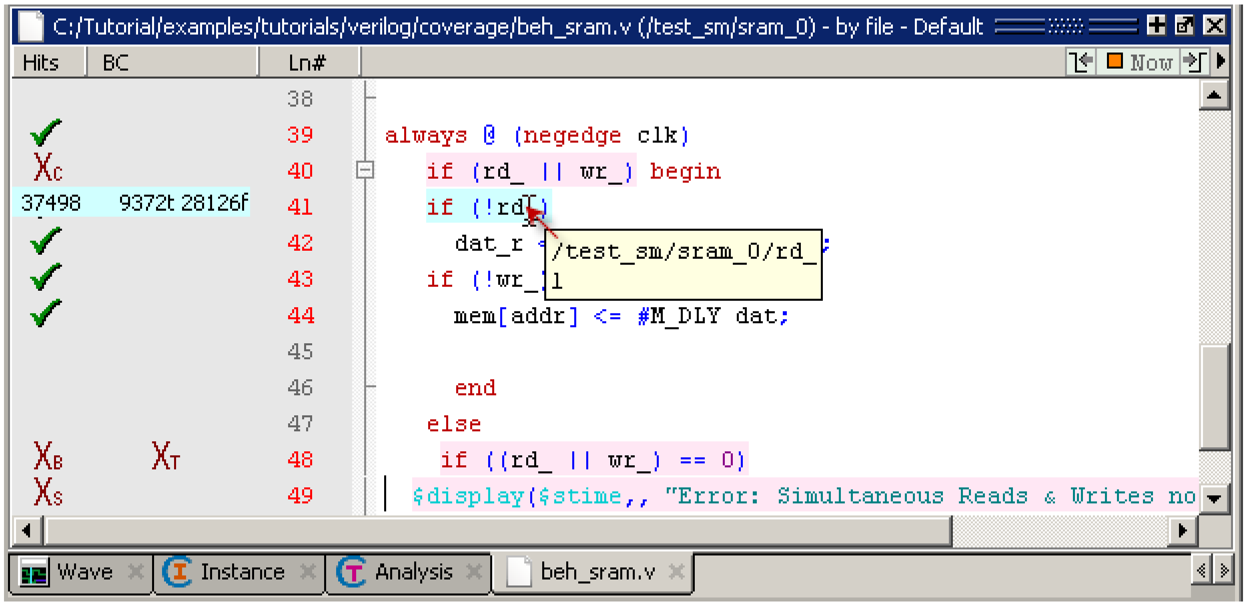
Procedure

1. View coverage statistics for *beh_sram* in the Source window.

   a. Double-click *beh_sram.v* in the **Files** window to open a source code view in the Source window.

   b. Scroll the Source window to view the code shown in Figure 13-10.

Figure 13-10. Coverage Statistics in the Source Window


C:/Tutorial/examples/tutorials/verilog/coverage/beh_sram.v (/test_sm/sram_0) - by file - Default
Now
Wave
Instance
Analysis
beh_sram.v

```
Hits    BC              Ln#
                        38
✓                       39   always @ (negedge clk)
XC                      40     if (rd_ || wr_) begin
37498   9372t 28126f    41     if (!rd_)
✓                       42       dat_r <  /test_sm/sram_0/rd_
✓                       43     if (!wr_  1
✓                       44       mem[addr] <= #M_DLY dat;
                        45
                        46       end
                        47     else
XB      XT              48      if ((rd_ || wr_) == 0)
XS                      49     $display($stime,, "Error: Simultaneous Reads & Writes no
```

The Source window includes a Hits and a BC column to display statement Hits and Branch Coverage, respectively. In Figure 13-10, the mouse cursor is hovering over the source code in line 41. This causes the coverage icons HIts and BC columns to change to coverage numbers. Table 13-2 describes the various coverage icons.

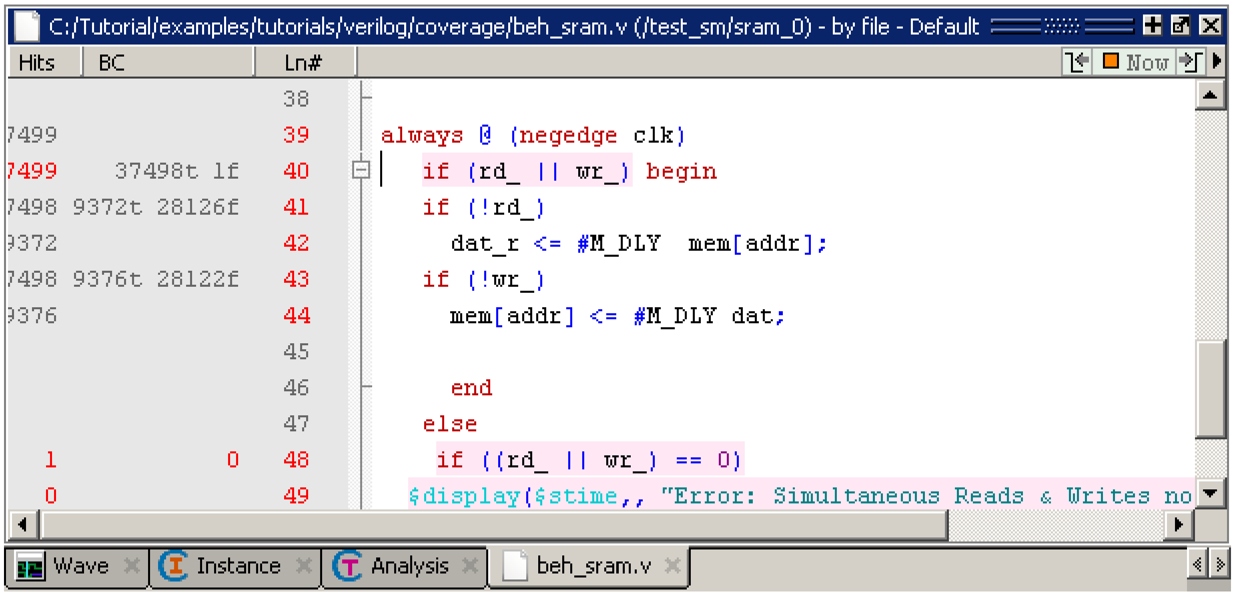
Table 13-2. Coverage Icons in the Source Window

Icon	Description
green checkmark	Indicates a statement that has been executed
green E	Indicates a line that has been excluded from code coverage statistics
red X	An X in the Hits column indicates a missed (unexecuted) statement (X_S), branch (X_B), or condition (X_C). An X in the BC column indicates a missed true (X_T) or false (X_F) branch.

c. Select **Tools > Code Coverage > Show coverage numbers**.

The coverage icons in the Hits and BC columns are replaced by execution counts on every line. Red numbers indicate missed coverage in that line of code. An ellipsis (...) is displayed whenever there are multiple statements on the line.

Figure 13-11. Coverage Numbers Shown by Hovering the Mouse Pointer



```
C:/Tutorial/examples/tutorials/verilog/coverage/beh_sram.v (/test_sm/sram_0) - by file - Default

Hits   BC            Ln#
                     38
7499                 39   always @ (negedge clk)
7499     37498t 1f   40     if (rd_ || wr_) begin
7498 9372t 28126f    41     if (!rd_)
9372                 42       dat_r <= #M_DLY  mem[addr];
7498 9376t 28122f    43     if (!wr_)
9376                 44       mem[addr] <= #M_DLY dat;
                     45
                     46       end
                     47     else
   1            0    48      if ((rd_ || wr_) == 0)
   0                 49    $display($stime,, "Error: Simultaneous Reads & Writes no
```

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

Toggle Statistics in the Objects Window

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**, **vcom**, or **vopt** command.

Refer to the section Toggle Coverage in the User's Manual for more information.

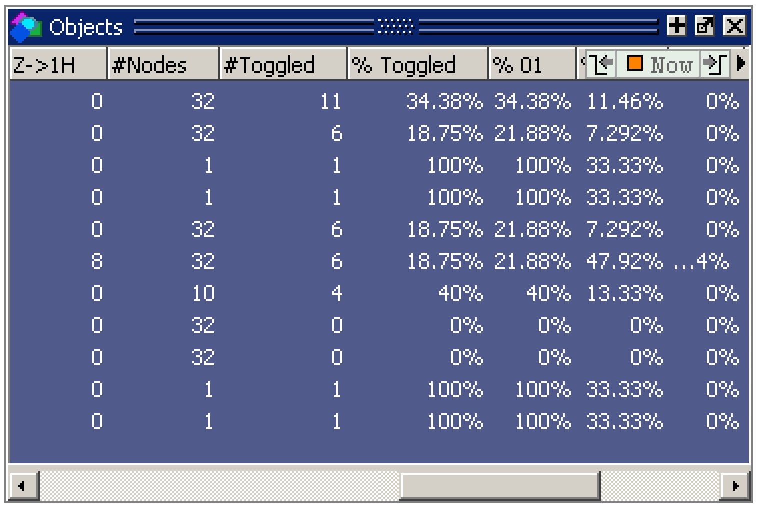
Procedure

1. View toggle data in the Objects window.

   a. Select *test_sm* in the Structure (sim) window.

   b. If the Objects window isn't open already, select **View > Objects**. Scroll to the right to see the various toggle coverage columns (Figure 13-12), or undock and expand the window until all columns are displayed. If you do not see the toggle coverage columns, simply right-click the column title bar and select **Show All Columns** from the popup menu.

Figure 13-12. Toggle Coverage in the Objects Window



Z->1H	#Nodes	#Toggled	% Toggled	% 01		
0	32	11	34.38%	34.38%	11.46%	0%
0	32	6	18.75%	21.88%	7.292%	0%
0	1	1	100%	100%	33.33%	0%
0	1	1	100%	100%	33.33%	0%
0	32	6	18.75%	21.88%	7.292%	0%
8	32	6	18.75%	21.88%	47.92%	...4%
0	10	4	40%	40%	13.33%	0%
0	32	0	0%	0%	0%	0%
0	32	0	0%	0%	0%	0%
0	1	1	100%	100%	33.33%	0%
0	1	1	100%	100%	33.33%	0%

Excluding Lines and Files from Coverage Statistics

Questa SIM allows you to exclude lines and files from code coverage statistics. You can set exclusions with GUI menu selections, with a text file called an "exclusion filter file", or with "pragmas" in your source code. Pragmas are statements that instruct Questa SIM to ignore coverage statistics for the bracketed code.

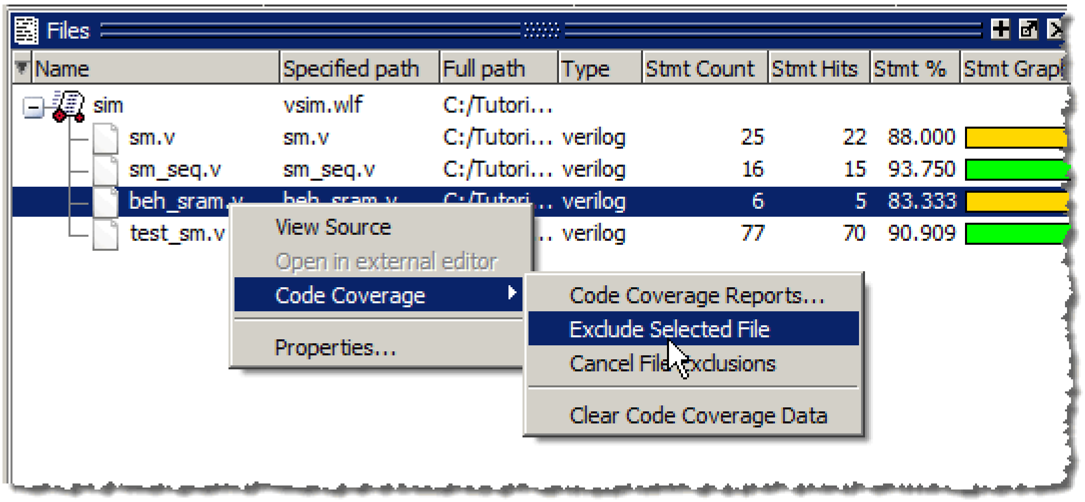
Refer to the section Coverage Exclusions in the User's Manual for more details on exclusion filter files and pragmas.

Procedure

1. Exclude a line in the Statement Analysis view of the Code Coverage Analysis window.
   a. Change the Analysis Type to Statement in the Code Coverage Analysis window.
   b. Right click a line in the Statement Analysis view and select **Exclude Selection** from the popup menu. (You can also exclude the selection for the current instance only by selecting Exclude Selection For Instance <inst_name>.)
2. Cancel the exclusion of the excluded statement.
   a. Right-click the line you excluded in the previous step and select **Cancel Selected Exclusions**.
3. Exclude an entire file.
   a. In the Files window, locate the *beh_sram.v* file (or the *beh_sram.vhd* file if you are using the VHDL example).

b. Right-click the file name and select **Code Coverage > Exclude Selected File** (Figure 13-13).

Figure 13-13. Excluding a File Using GUI Menus



Name	Specified path	Full path	Type	Stmt Count	Stmt Hits	Stmt %	Stmt Grap
sim	vsim.wlf	C:/Tutori...					
sm.v	sm.v	C:/Tutori...	verilog	25	22	88.000	🟨
sm_seq.v	sm_seq.v	C:/Tutori...	verilog	16	15	93.750	🟩
beh_sram.v	beh_sram.v	C:/Tutori...	verilog	6	5	83.333	🟨
test_sm.v		..	verilog	77	70	90.909	🟩

c. You can cancel all file exclusions by right-clicking anywhere in the Files window and selecting **Code Coverage > Cancel File Exclusions** from the popup menu.

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 window. You can also create textual reports of coverage exclusions using menu selections.

To create textual coverage reports using GUI menu selections, do one of the following:

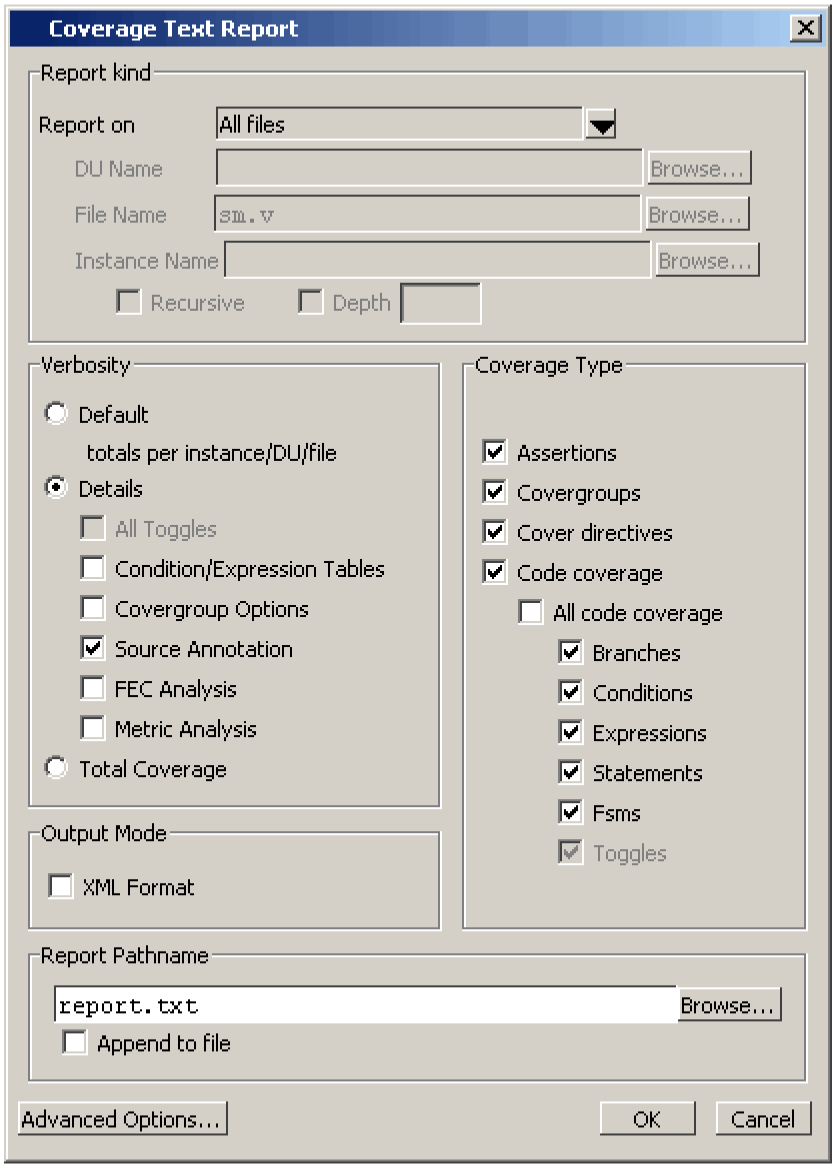
- Select **Tools > Coverage Report > Text** from the Main window menu bar.
- Right-click any object in the **sim** or **Files** windows and select **Code Coverage > Code Coverage Reports** from the popup context menu.
- Right-click any object in the Instance Coverage window and select **Code coverage 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 window is active.

This will open the Coverage Text Report dialog box (Figure 13-14) where you can elect to report on:

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

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

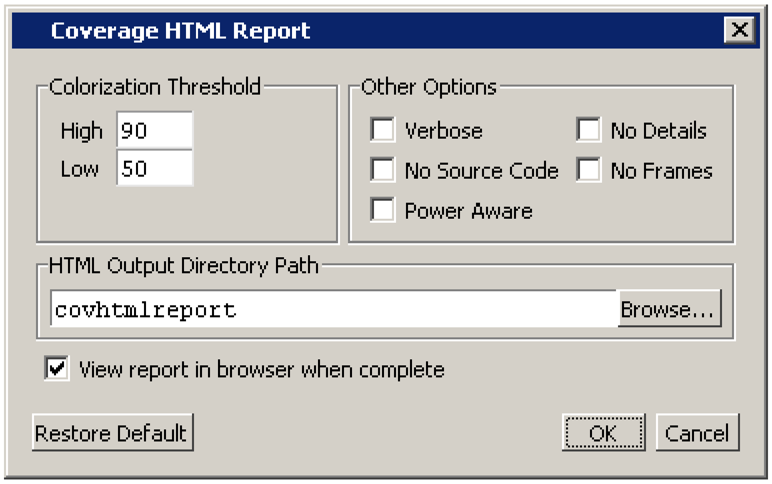
Figure 13-14. Coverage Text Report Dialog Box



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

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 box where you can designate an output directory path for the HTML report.

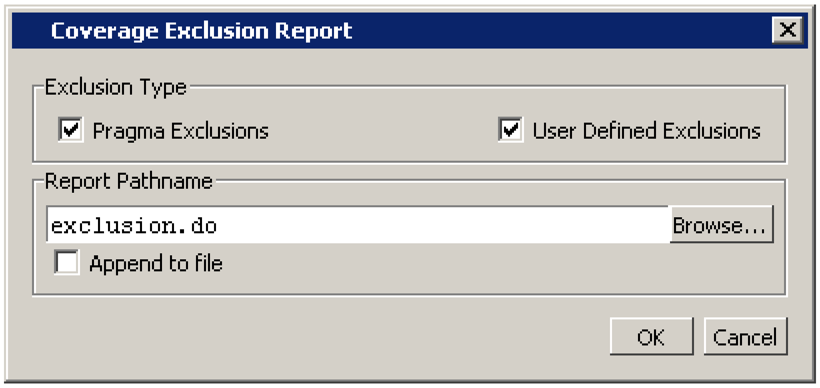
Figure 13-15. Coverage HTML Report Dialog Box



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 menu bar. This opens the Coverage Exclusions Report dialog box where you can elect to show only pragma exclusions, only user defined exclusions, or both.

Figure 13-16. Coverage Exclusions Report Dialog Box



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 14
Debugging With PSL Assertions

Using assertions in your HDL code increases visibility into your design and improves verification productivity. Questa SIM 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 Questa SIM. 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 Questa SIM 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 test bench. 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 Questa SIM installation comes with Verilog and VHDL versions of this design. The files are located in the following directories:

Verilog – *<install_dir>/examples/tutorials/psl/verilog/modeling/dram_controller*

VHDL – *<install_dir>/examples/tutorials/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 Topics

User’s Manual Chapters: Verification with Assertions and Cover Directives and the Assertions Window section in the *Questa SIM Graphical User Interface (GUI) Reference Manual*.

Run the Design without PSL Assertions

In this exercise you will use a *.do* file to run the design without PSL Assertions.

Procedure

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/tutorials/psl/verilog/modeling/dram_controller* to the new directory.

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

2. Start Questa SIM and change to the exercise directory you created.

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

   a. To start Questa SIM, type vsim at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box appears, click **Close**.

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

3. Execute the *.do* file.

   a. Type **do run_nopsl.do** at the command prompt.

      The DO file does the following:

      - Creates the working library
      - Compiles the design files and assertions
      - Optimizes the design
      - Loads (elaborates) the design
      - Runs the simulation

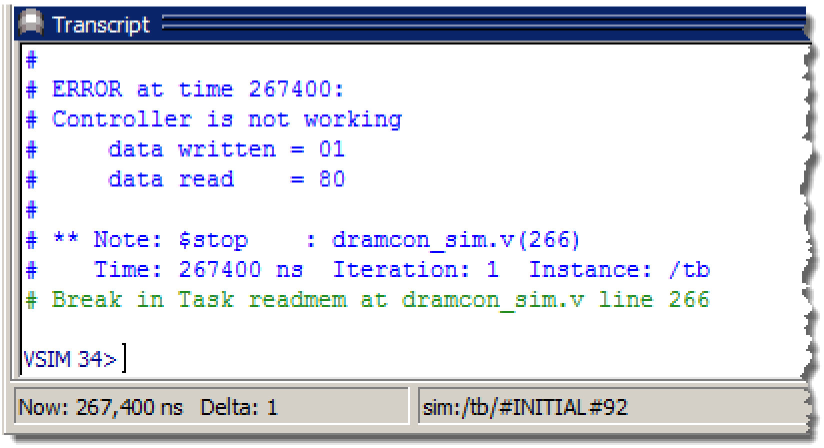
      Feel free to open the *.do* file and look at its contents. In the optimization step, the **vopt+acc tb -o dram_opt** command provides visibility into all design units for debugging purposes and creates an optimized file called “dram_opt.” During elaboration, the **-nopsl** argument instructs the compiler to ignore PSL assertions.

      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 14-1).

Figure 14-1. Transcript After Running Simulation Without Assertions



```
#
# ERROR at time 267400:
# Controller is not working
#     data written = 01
#     data read    = 80
#
# ** Note: $stop    : dramcon_sim.v(266)
#    Time: 267400 ns  Iteration: 1  Instance: /tb
# Break in Task readmem at dramcon_sim.v line 266

VSIM 34>
```

```
Now: 267,400 ns  Delta: 1        sim:/tb/#INITIAL#92
```

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.

4. End the simulation.

   a. Select **Simulate** > **End Simulation** from the menus to end this simulation.

Using Assertions to Speed Debugging

To see how assertions can speed debugging, reload the design with assertion failure tracking enabled.

Procedure

1. Reload the design.

   a. Type **vsim -msgmode both -assertdebug dram_opt** at the command prompt.

   The **-msgmode both** argument outputs messages to both the transcript and the WLF file.

   "dram_opt" is the name of the optimized file created when we ran the simulation without assertions. The **-assertdebug** option gives you another tool for debugging failed assertions, as we'll see in a moment.

2. Change your WildcardFilter settings.

Execute the following command:

```
set WildcardFilter "Variable Constant Generic Parameter SpecParam Memory
Endpoint CellInternal ImmediateAssert"
```

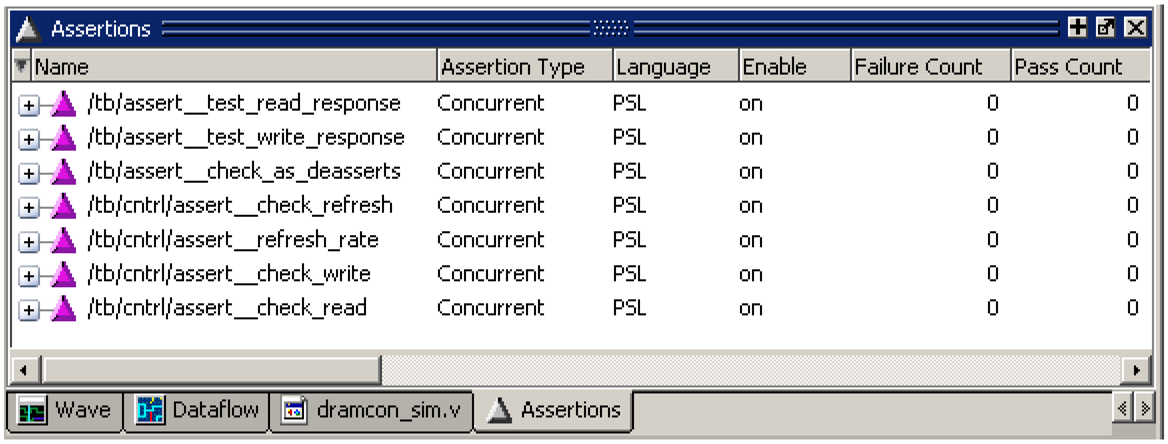
With this command, you remove “Assertion,” “Cover,” and “ScVariable” from the default list of Wildcard filters. This allows Assertions, Cover Directives, and SystemC variables to be logged by the simulator so they will be visible in the debug environment.

3. View all PSL assertions in the Assertions window.

   a. Type **view assertions** at the command prompt.

      This opens the Assertions window with all PSL assertions displayed (Figure 14-2).

Figure 14-2. Assertions Window Displays PSL Assertions



Name	Assertion Type	Language	Enable	Failure Count	Pass Count
/tb/assert__test_read_response	Concurrent	PSL	on	0	0
/tb/assert__test_write_response	Concurrent	PSL	on	0	0
/tb/assert__check_as_deasserts	Concurrent	PSL	on	0	0
/tb/cntrl/assert__check_refresh	Concurrent	PSL	on	0	0
/tb/cntrl/assert__refresh_rate	Concurrent	PSL	on	0	0
/tb/cntrl/assert__check_write	Concurrent	PSL	on	0	0
/tb/cntrl/assert__check_read	Concurrent	PSL	on	0	0

Note
You can open and close the Assertions window by selecting **View** > **Coverage** > **Assertions** from the Main menu.

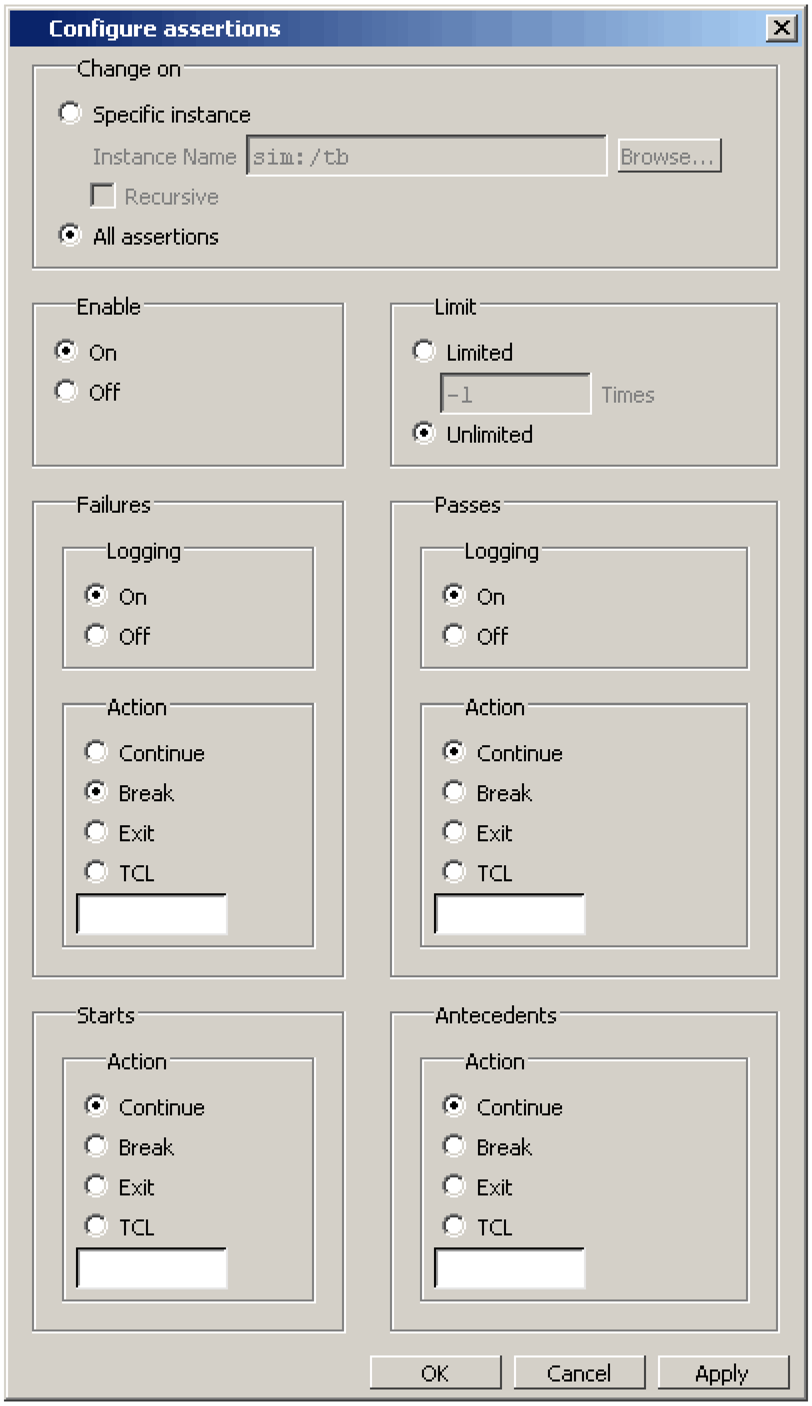
You may need to resize windows to better view the data.

4. Set all assertions to Break on Failures.

   a. Select the Assertions window to make it active.

   b. Select **Assertions** > **Configure** from the main menu to open the Configure assertions dialog box (Figure 14-3).

Figure 14-3. Configure Assertions Dialog Box



Configure assertions

Change on
- Specific instance
- Instance Name: sim:/tb
- Recursive
- [✓ All assertions]

Enable: [✓ On] Off

Limit: Limited (-1 Times) [✓ Unlimited]

Failures
- Logging: [✓ On] Off
- Action: Continue [✓ Break] Exit TCL

Passes
- Logging: [✓ On] Off
- Action: [✓ Continue] Break Exit TCL

Starts
- Action: [✓ Continue] Break Exit TCL

Antecedents
- Action: [✓ Continue] Break Exit TCL

c. In the Change on section, select **All assertions**.

d. In the Enable section, select **On**.

e. In the 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 box.

The command line equivalents for these actions are as follows:

```
assertion action -cond fail -exec break -r *
assertion pass -log on -r *
```

5. Add assertion signals to the Wave window

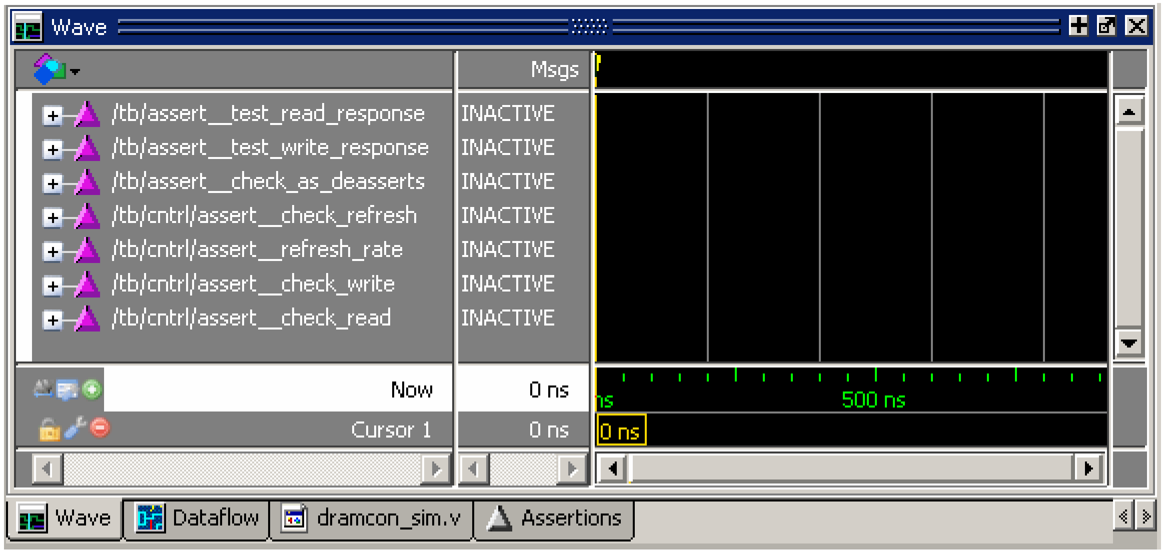
a. Select all assertions in the Assertions window.

b. Right-click on the selected assertions to open a popup menu.

c. Select **Add Wave > Selected Objects**.

The Wave window displays the assertion signals (denoted by magenta triangles) as shown in Figure 14-4.

Figure 14-4. Assertions in the Wave Window



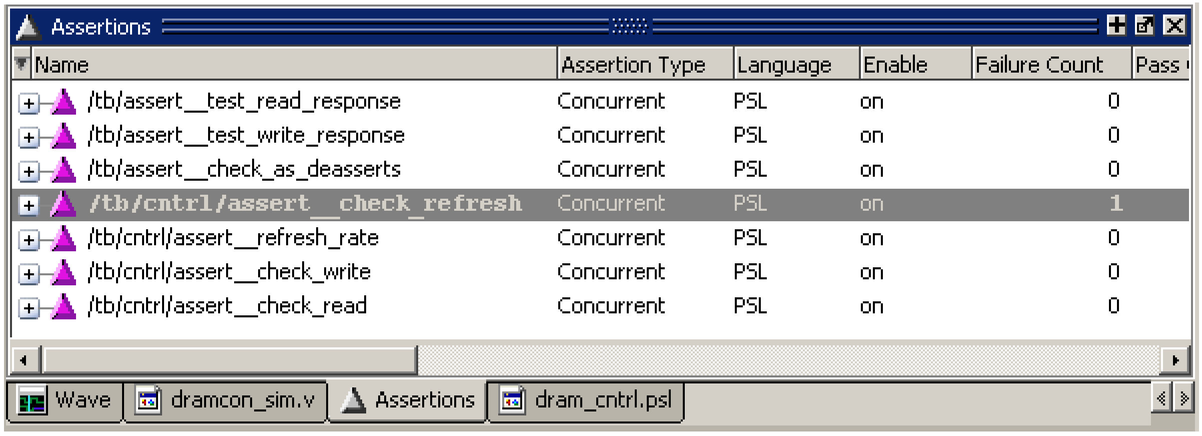
	Msgs
/tb/assert__test_read_response	INACTIVE
/tb/assert__test_write_response	INACTIVE
/tb/assert__check_as_deasserts	INACTIVE
/tb/cntrl/assert__check_refresh	INACTIVE
/tb/cntrl/assert__refresh_rate	INACTIVE
/tb/cntrl/assert__check_write	INACTIVE
/tb/cntrl/assert__check_read	INACTIVE

6. Run the simulation and view the results.

a. Type **run -all** at the command prompt.

Verilog: In the Assertion window, the *assert_check_refresh* assertion is highlighted, and a failure count of 1 appears in the Failure Count column (Figure 14-5).

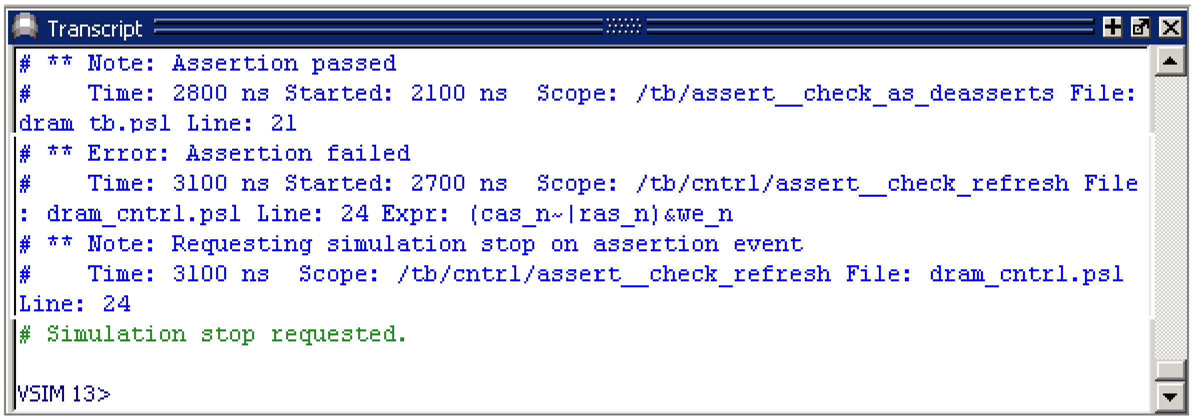
Figure 14-5. Assertion Failure Indicated in the Assertions window



Name	Assertion Type	Language	Enable	Failure Count	Pass
/tb/assert__test_read_response	Concurrent	PSL	on	0	
/tb/assert__test_write_response	Concurrent	PSL	on	0	
/tb/assert__check_as_deasserts	Concurrent	PSL	on	0	
/tb/cntrl/assert__check_refresh	Concurrent	PSL	on	1	
/tb/cntrl/assert__refresh_rate	Concurrent	PSL	on	0	
/tb/cntrl/assert__check_write	Concurrent	PSL	on	0	
/tb/cntrl/assert__check_read	Concurrent	PSL	on	0	

The Main window transcript shows that the *assert__check_refresh* assertion in the *dram_cntrl.psl* file failed at 3100 ns (Figure 14-6). The simulation is stopped at that time. Remember, with no assertions the test bench did not report a failure until 267,400 ns — over 80x the simulation time required for a failure to be reported with assertions.

Figure 14-6. Assertion Failure Shown in the Transcript Window



```
# ** Note: Assertion passed
#    Time: 2800 ns Started: 2100 ns  Scope: /tb/assert__check_as_deasserts File:
dram tb.psl Line: 21
# ** Error: Assertion failed
#    Time: 3100 ns Started: 2700 ns  Scope: /tb/cntrl/assert__check_refresh File
: dram_cntrl.psl Line: 24 Expr: (cas_n~|ras_n)&we_n
# ** Note: Requesting simulation stop on assertion event
#    Time: 3100 ns  Scope: /tb/cntrl/assert__check_refresh File: dram_cntrl.psl
Line: 24
# Simulation stop requested.

VSIM 13>
```

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. Remember, with no assertions the test bench did not report a failure until 246,800 ns — over 60x the simulation time required for a failure to be reported with assertions.

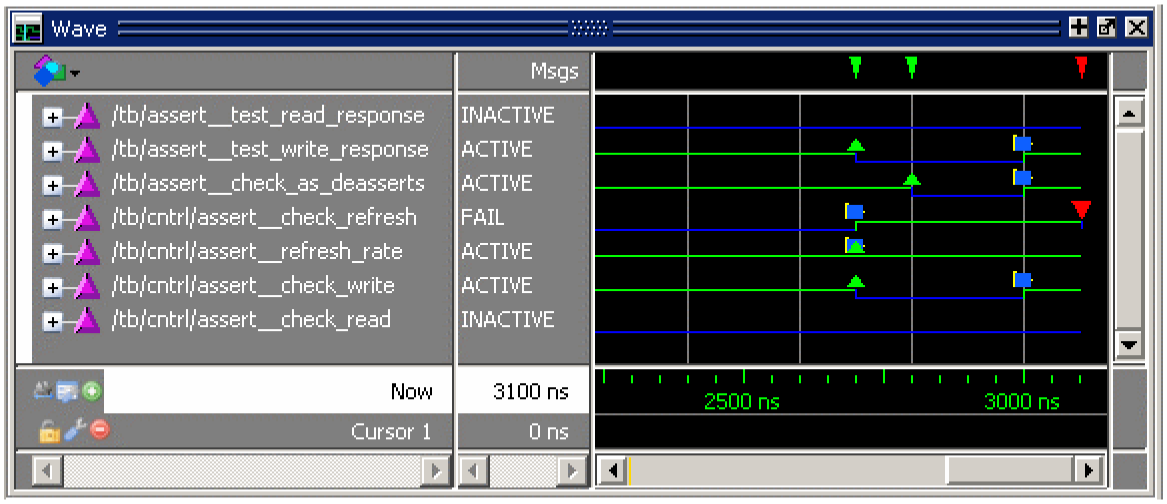
b. Click the **dram_cntrl.psl** tab to open the Source window.

The blue arrow in the Source window (*dram_cntrl.psl*) shows where the simulation stopped—at the *check_refresh* assertion on line 24.

c. Click the **Wave** tab to open the Wave window.

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 14-7). Green triangles indicate assertion passes.

Figure 14-7. Assertion Failure Indicated in Wave Window



```
Wave                                  Msgs
/tb/assert__test_read_response        INACTIVE
/tb/assert__test_write_response       ACTIVE
/tb/assert__check_as_deasserts        ACTIVE
/tb/cntrl/assert__check_refresh       FAIL
/tb/cntrl/assert__refresh_rate        ACTIVE
/tb/cntrl/assert__check_write         ACTIVE
/tb/cntrl/assert__check_read          INACTIVE
                              Now     3100 ns
                         Cursor 1     0 ns
```

The blue sections of the assertion waveforms indicate inactive assertions; green indicates active assertions.

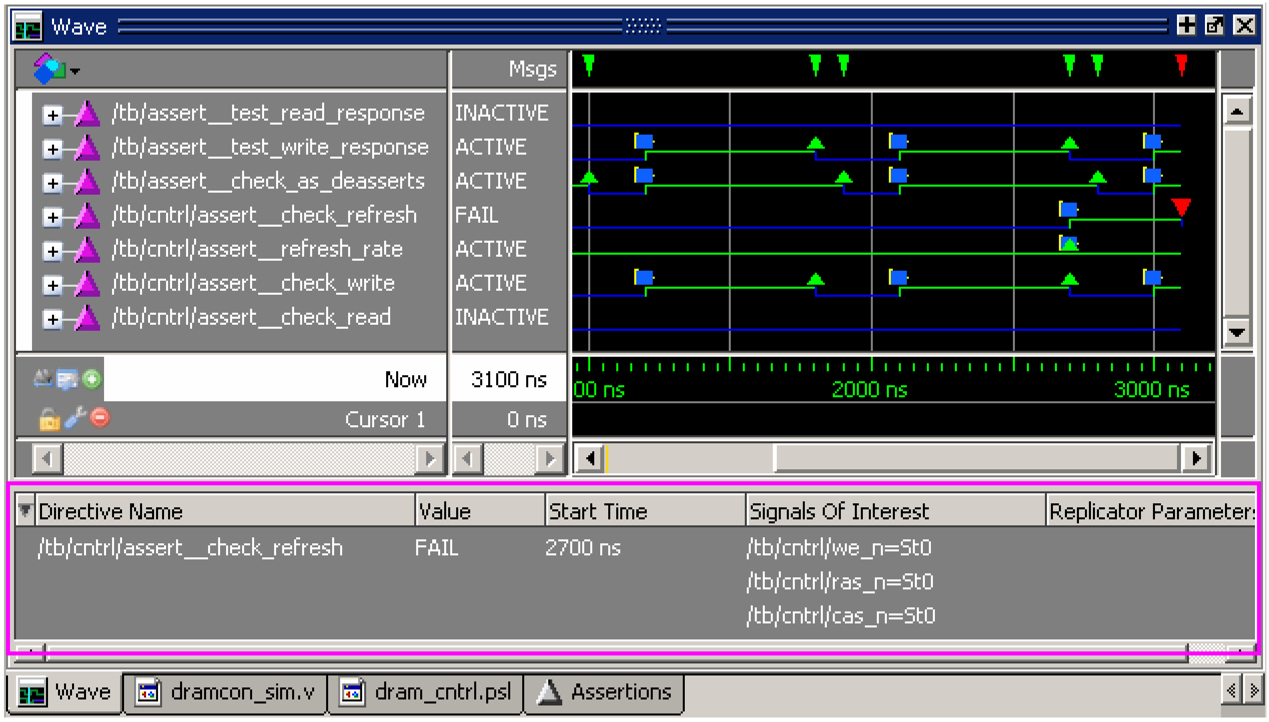
7. View the assertion failure in the Assertion Debug pane of the Wave window.

   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. Select **Wave > Assertion Debug**.

      Verilog: The Assertion Debug pane of the Wave window shows that the *assert_check_refresh* assertion failed. The Start Time of the assertion is 2700 ns. And signals of interest for debugging are displayed in the Signals of Interest column (Figure 14-8).

Figure 14-8. The Failed Assertion Details for Verilog Design

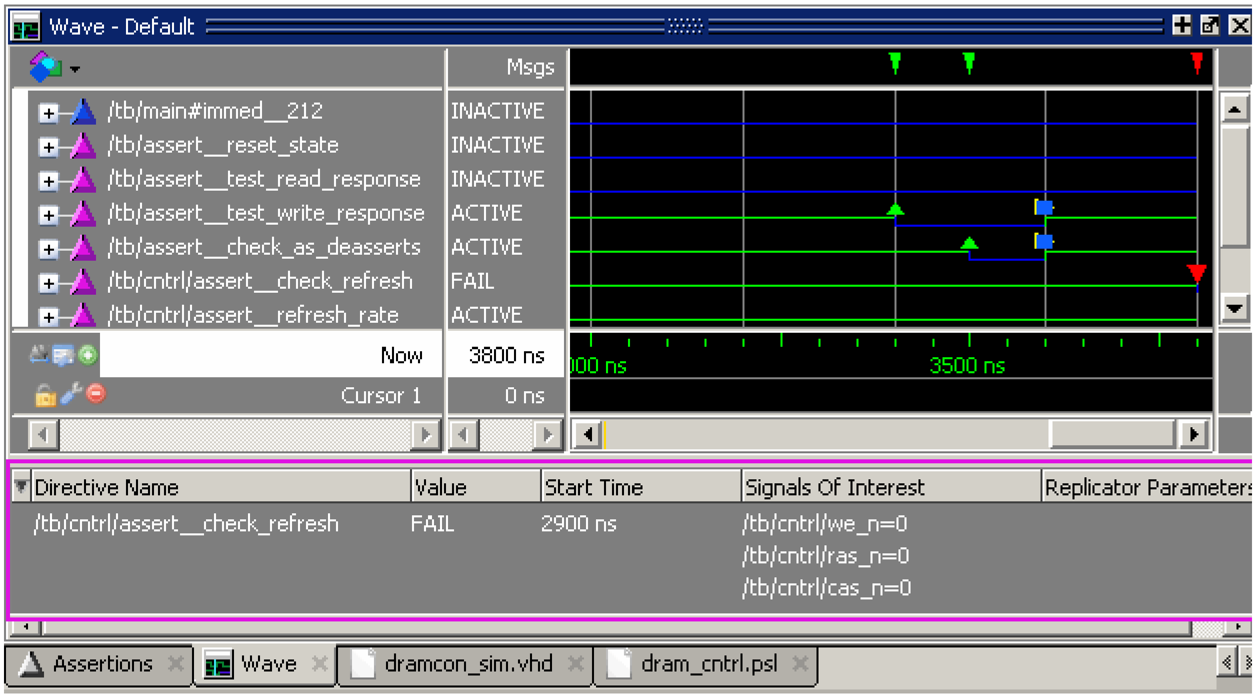


	Msgs
/tb/assert__test_read_response	INACTIVE
/tb/assert__test_write_response	ACTIVE
/tb/assert__check_as_deasserts	ACTIVE
/tb/cntrl/assert__check_refresh	FAIL
/tb/cntrl/assert__refresh_rate	ACTIVE
/tb/cntrl/assert__check_write	ACTIVE
/tb/cntrl/assert__check_read	INACTIVE
Now	3100 ns
Cursor 1	0 ns

Directive Name	Value	Start Time	Signals Of Interest	Replicator Parameter:
/tb/cntrl/assert__check_refresh	FAIL	2700 ns	/tb/cntrl/we_n=St0 /tb/cntrl/ras_n=St0 /tb/cntrl/cas_n=St0	

VHDL: The Assertion Debug pane of the Wave window shows that the *assert_check_refresh* assertion failed. The Start Time of the assertion is 2900 ns. And signals of interest for debugging are displayed in the Signals of Interest column Figure 14-9.

Figure 14-9. The Failed Assertion Detials for VHDL Design


Wave - Default
Msgs
/tb/main#immed__212 INACTIVE
/tb/assert__reset_state INACTIVE
/tb/assert__test_read_response INACTIVE
/tb/assert__test_write_response ACTIVE
/tb/assert__check_as_deasserts ACTIVE
/tb/cntrl/assert__check_refresh FAIL
/tb/cntrl/assert__refresh_rate ACTIVE
Now 3800 ns
Cursor 1 0 ns
000 ns
3500 ns
Assertions
Wave
dramcon_sim.vhd
dram_cntrl.psl

Directive Name	Value	Start Time	Signals Of Interest	Replicator Parameters
/tb/cntrl/assert__check_refresh	FAIL	2900 ns	/tb/cntrl/we_n=0 /tb/cntrl/ras_n=0 /tb/cntrl/cas_n=0	

The Signals of Interest column displays the signals responsible for the assertion failure.

Debugging the Assertion Failure

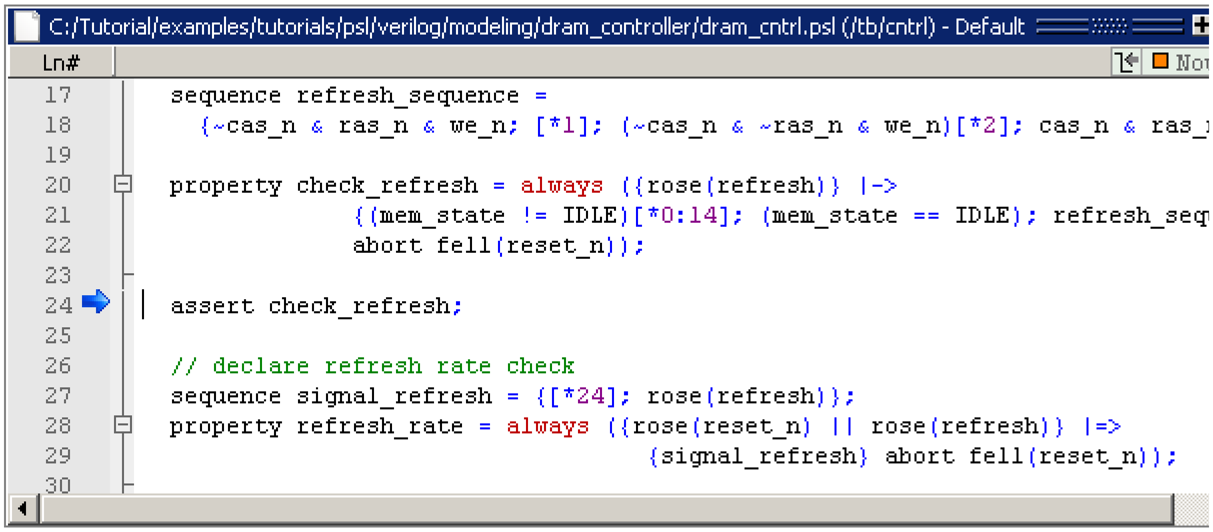
Now we can debug the assertion failure.

Procedure

1. View the source code (*dram_cntrl.psl*) of the failed assertion.

   The current line arrow points to the failed assertion on line 24 (Figure 14-10). 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 14-10. Source Code for Failed Assertion



C:/Tutorial/examples/tutorials/psl/verilog/modeling/dram_controller/dram_cntrl.psl (/tb/cntrl) - Default

```
Ln#
17    sequence refresh_sequence =
18      {~cas_n & ras_n & we_n; [*1]; (~cas_n & ~ras_n & we_n)[*2]; cas_n & ras_
19
20    property check_refresh = always ({rose(refresh)} |->
21                 {(mem_state != IDLE)[*0:14]; (mem_state == IDLE); refresh_seq
22                 abort fell(reset_n));
23
24    assert check_refresh;
25
26    // declare refresh rate check
27    sequence signal_refresh = {[*24]; rose(refresh)};
28    property refresh_rate = always ({rose(reset_n) || rose(refresh)} |=>
29                                    {signal_refresh} abort fell(reset_n));
30
```

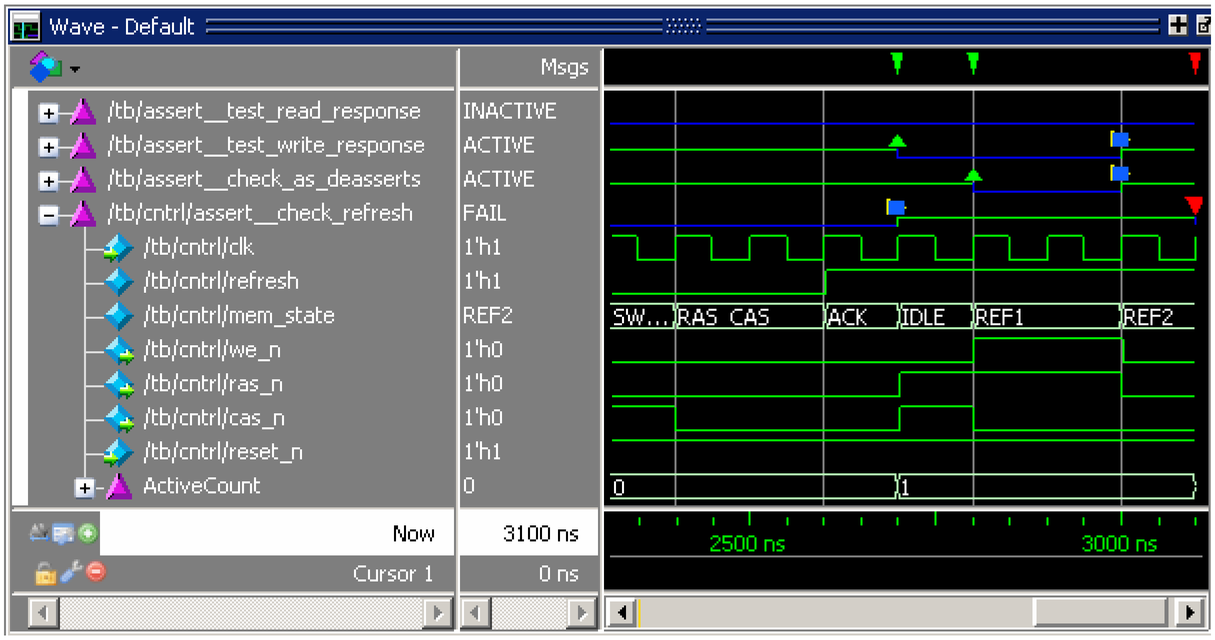
The *refresh_sequence* (second line of the property) is defined on line 18. The key part of the refresh protocol is that *we_n* must be held high (write enable not active) for the entire refresh cycle.

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 14-11).

Figure 14-11. Examining we_n With Respect to mem_state



Wave - Default

	Msgs
/tb/assert__test_read_response	INACTIVE
/tb/assert__test_write_response	ACTIVE
/tb/assert__check_as_deasserts	ACTIVE
/tb/cntrl/assert__check_refresh	FAIL
/tb/cntrl/clk	1'h1
/tb/cntrl/refresh	1'h1
/tb/cntrl/mem_state	REF2
/tb/cntrl/we_n	1'h0
/tb/cntrl/ras_n	1'h0
/tb/cntrl/cas_n	1'h0
/tb/cntrl/reset_n	1'h1
ActiveCount	0
Now	3100 ns
Cursor 1	0 ns

It is easy to see that *we_n* is high only during the *REF1* state. It is low during *REF2* of *mem_state*.

Let's examine *we_n* further.

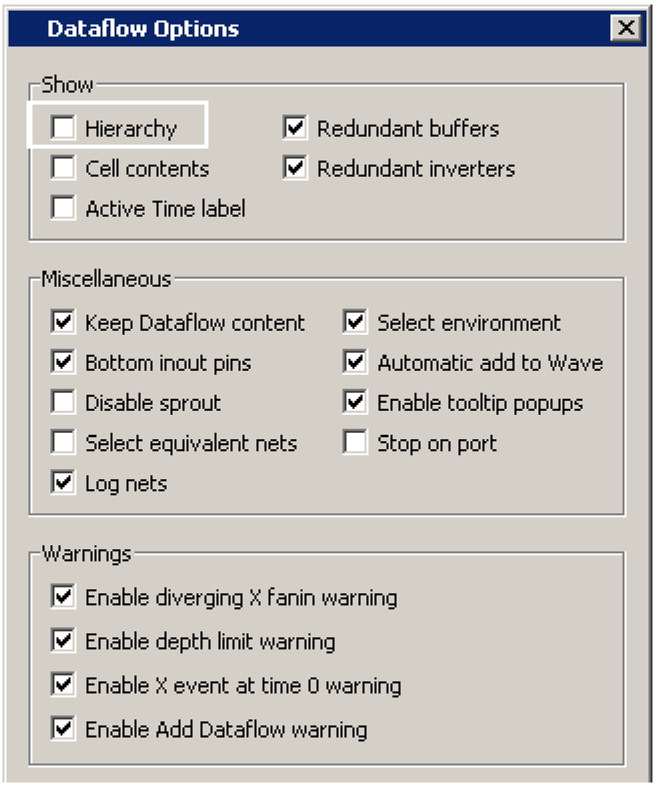
3. Examine *we_n* in the Dataflow and Source windows.

   a. Open the Dataflow window by selecting **View** > **Dataflow** from the Main menu, then select the Dataflow window to make sure it is active. A "Dataflow" menu should appear in the menu bar.

   b. Select **Dataflow** > **Dataflow Preferences** > **Options** from the menus to open the Dataflow Options dialog box. (If the Dataflow window is undocked, select **Tools** > **Options** from the Dataflow window menus.)

   c. Uncheck the **Show Hierarchy** selection as shown in Figure 14-12 and click **OK**.

Figure 14-12. Dataflow Options Dialog Box


Dataflow Options

Show
- [ ] Hierarchy
- [ ] Cell contents
- [ ] Active Time label
- [x] Redundant buffers
- [x] Redundant inverters

Miscellaneous
- [x] Keep Dataflow content
- [x] Bottom inout pins
- [ ] Disable sprout
- [ ] Select equivalent nets
- [x] Log nets
- [x] Select environment
- [x] Automatic add to Wave
- [x] Enable tooltip popups
- [ ] Stop on port

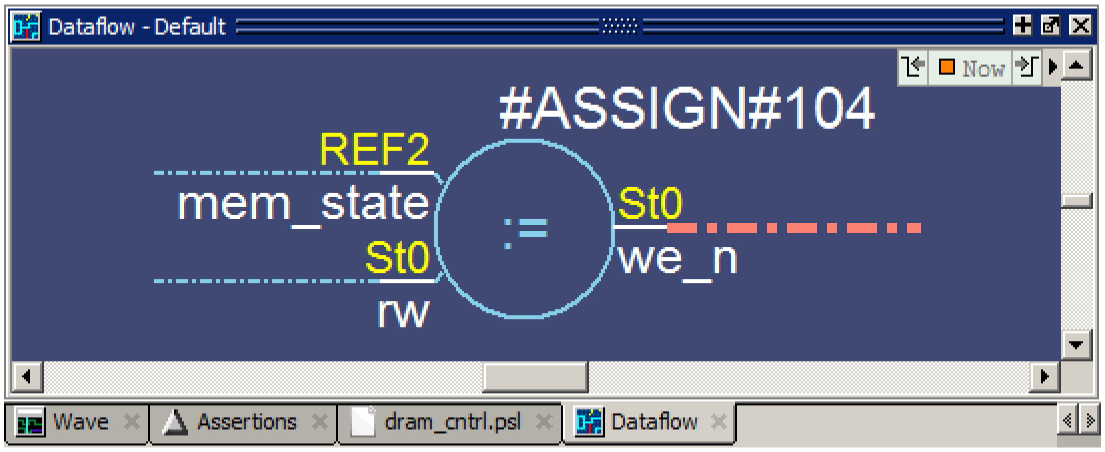
Warnings
- [x] Enable diverging X fanin warning
- [x] Enable depth limit warning
- [x] Enable X event at time 0 warning
- [x] Enable Add Dataflow warning

d. Select the write enable signal *we_n* in the Wave window.

e. Select **Add** > **To Dataflow** > **Selected Items** from the Main menus.

Verilog: The Dataflow window shows that *we_n* is driven by the *#ASSIGN#104* process, with inputs *rw* and *mem_state* (Figure 14-13). 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.

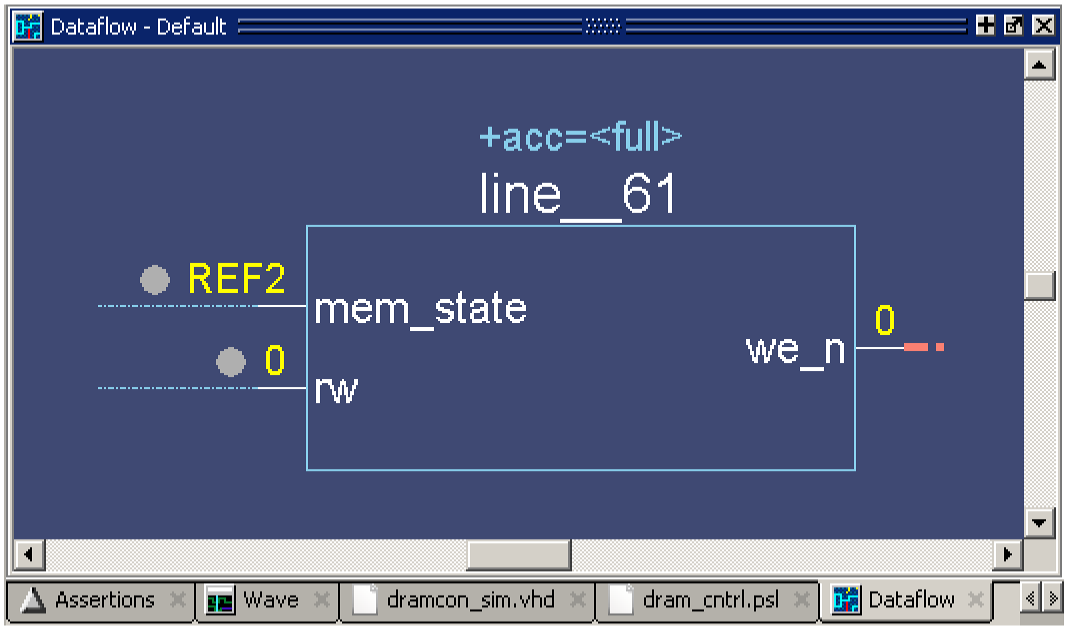
Figure 14-13. Viewing we_n in the Dataflow Window - Verilog



Note
If you see something other than the ‘St0’ value for the *we_n* signal, enter the **radix -symbolic** command at the VSIM> prompt.

VHDL: The Dataflow window shows that *we_n* is driven by the process at line 61, which has inputs *rw* and *mem_state* (Figure 14-14). 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* has a value of 0 when *mem_state* is REF2. As noted above, *we_n* should be 1. This is the reason for the assertion failure.

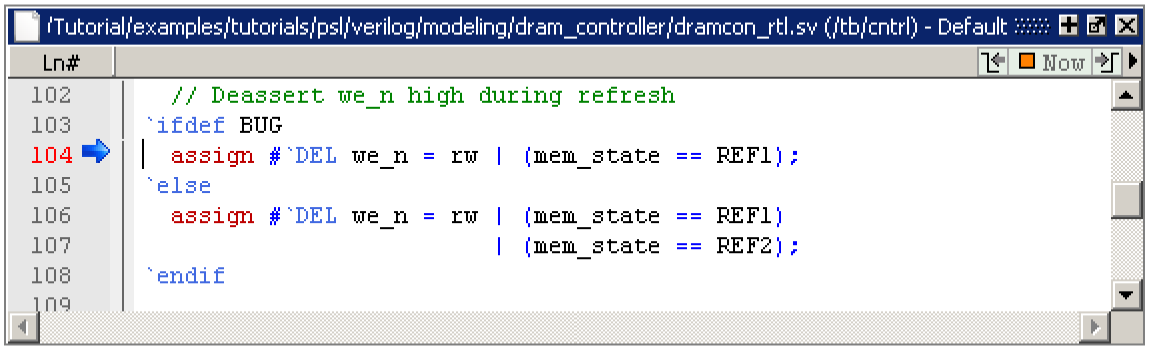
Figure 14-14. Viewing we_n in the Dataflow Window - VHDL



f. Double-click the *#ASSIGN#104* process that drives *we_n* (*line_61* in VHDL) 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 14-15). In this line you can see that the logic assigning *we_n* is wrong - it does not account for the *REF2* state.

Figure 14-15. Finding the Bug in the Source Code - Verilog

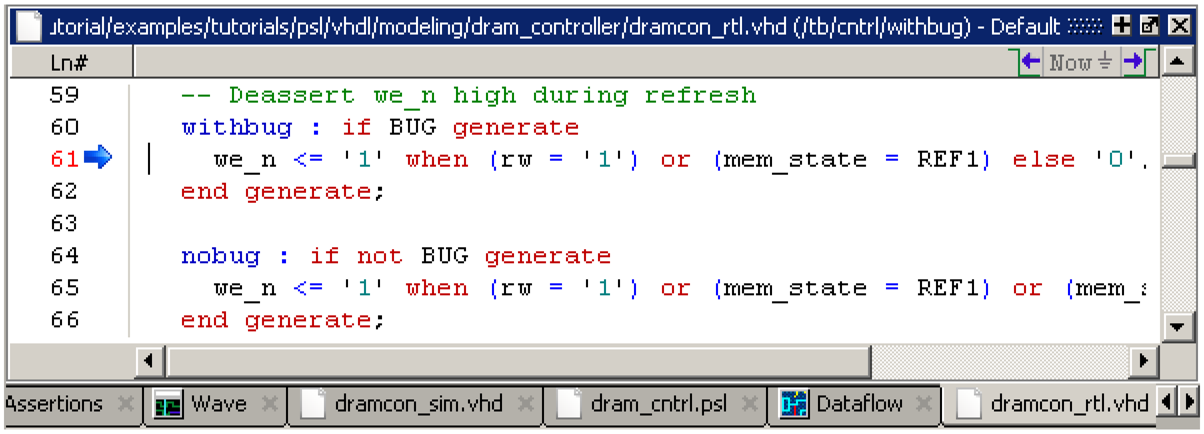

/Tutorial/examples/tutorials/psl/verilog/modeling/dram_controller/dramcon_rtl.sv (/tb/cntrl) - Default

```
102      // Deassert we_n high during refresh
103    `ifdef BUG
104      assign #`DEL we_n = rw | (mem_state == REF1);
105    `else
106      assign #`DEL we_n = rw | (mem_state == REF1)
107                             | (mem_state == REF2);
108    `endif
109
```

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 (Figure 14-16). 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 an incorrect assignment is used. The correct assignment is shown immediately below (in line 65) – *we_n* is held high through both states of the refresh cycle.

Figure 14-16. Finding the Bug in the Source Code - VHDL


.torial/examples/tutorials/psl/vhdl/modeling/dram_controller/dramcon_rtl.vhd (/tb/cntrl/withbug) - Default

```
59     -- Deassert we_n high during refresh
60     withbug : if BUG generate
61       we_n <= '1' when (rw = '1') or (mem_state = REF1) else '0',
62     end generate;
63
64     nobug : if not BUG generate
65       we_n <= '1' when (rw = '1') or (mem_state = REF1) or (mem_s
66     end generate;
```

Lesson Wrap-Up

This concludes this lesson.

1. To return the wildcard filter to its factory default settings, enter:

   set WildcardFilter "default"

2. To end the current simulation:

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

Chapter 15
SystemVerilog Assertions and Functional Coverage

This lesson provides a step-by-step introduction to functional hardware verification using SystemVerilog assertion and functional coverage capabilities.

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 test bench 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 Questa SIM.

For more information, refer to the User's Manual Chapter: Verification with Functional Coverage.

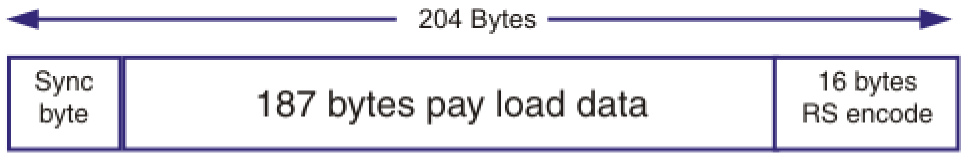
The files are located in the following directories:

Verilog – */<install_dir>/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.

Figure 15-1. Incoming Data

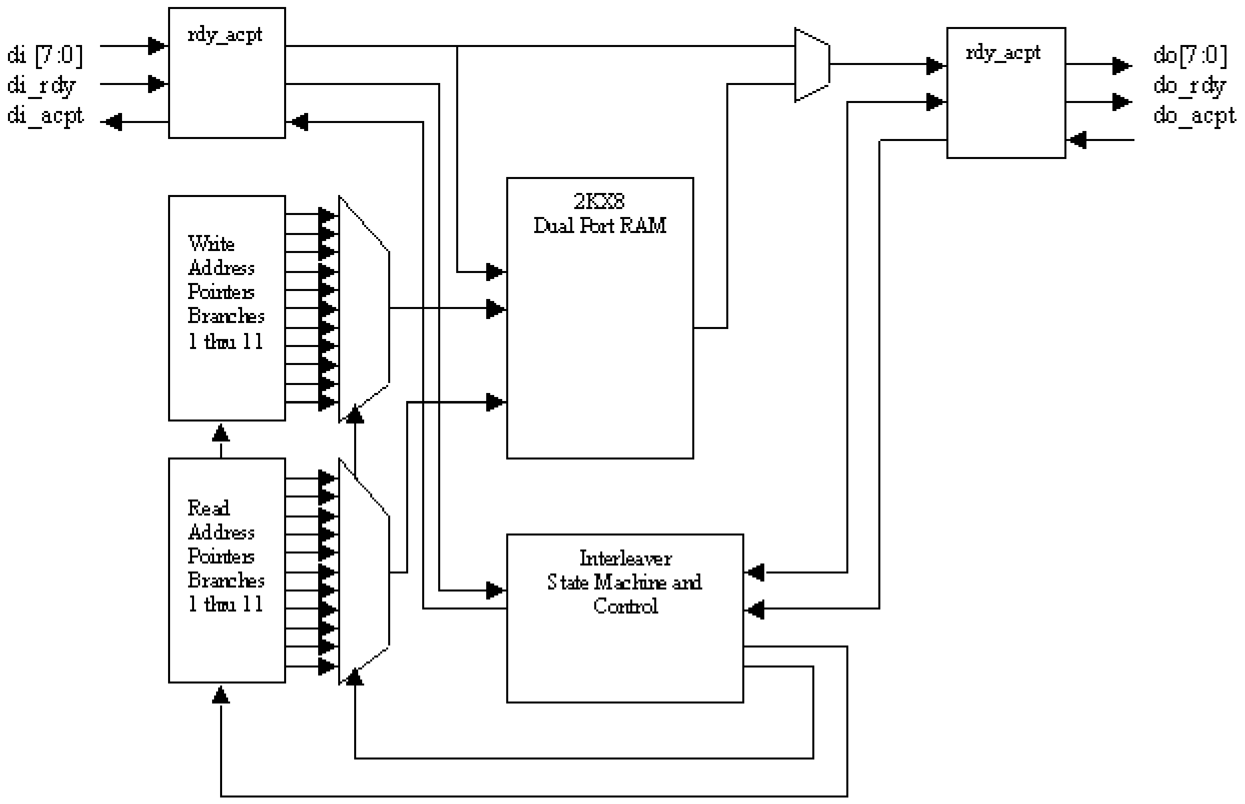


The interleaver has 12 levels numbered 0 to 11. Each level, except the first, can be conceptually thought of as a FIFO shift register. The depth of each register is 17 greater than the previous level. The first level (level 0) has a depth of zero (0); level 1 has a depth of 17; level 2, a depth of 34, and so on. Level 11 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 15-2.

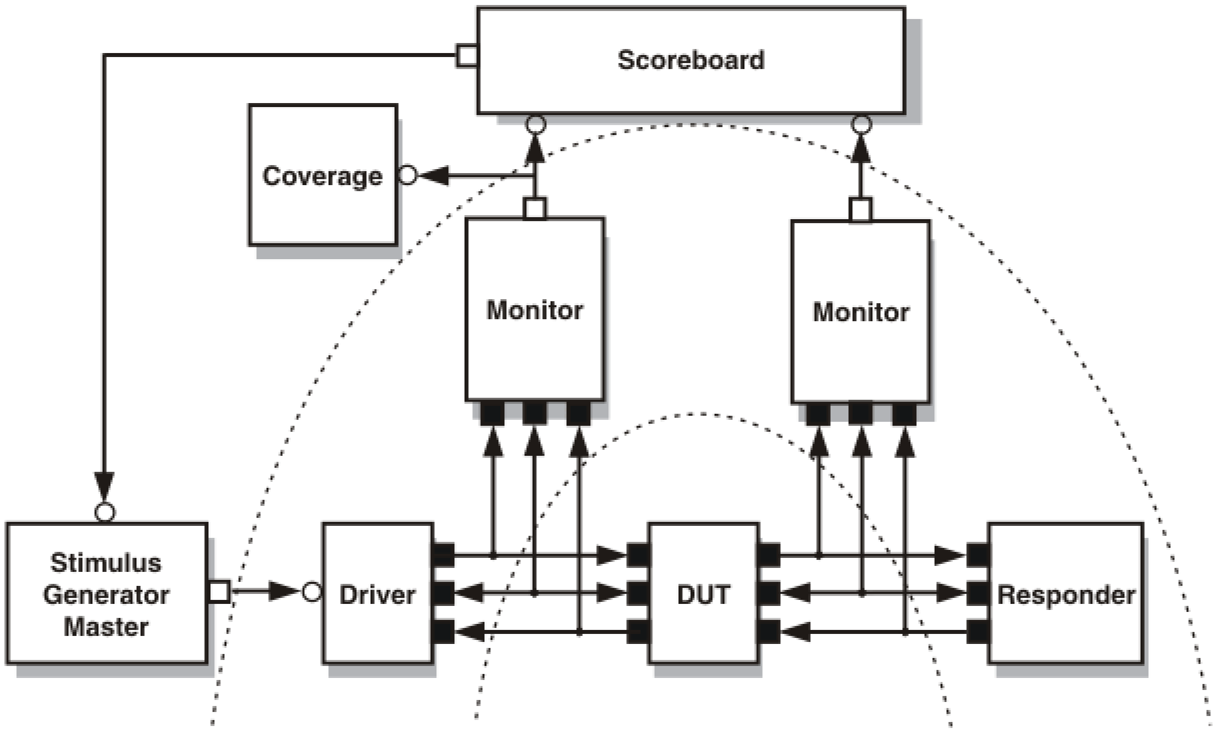
Figure 15-2. Block Diagram of the Inteleaver


di [7:0]
di_rdy
di_acpt
rdy_acpt
Write Address Pointers Branches 1 thru 11
Read Address Pointers Branches 1 thru 11
2KX8 Dual Port RAM
Interleaver State Machine and Control
rdy_acpt
do[7:0]
do_rdy
do_acpt

The Test Bench

The figure below shows how the test bench components are connected. The stimulus generator creates random data packets and sends them to the driver. Even though the test bench is module based, the stimulus generator still creates packets that are transaction based (SV class). This is the big advantage offered by the Advanced Verification Methodology (AVM) - it allows you to take advantage of transaction level modeling (TLM) techniques without having to convert your test bench to a complete object oriented programming environment.

Figure 15-3. Block Diagram of the Test Bench



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 test bench) provides the handshaking *ready/accept* signals needed for packet delivery.

Run the Simulation without Assertions

In order to demonstrate the advantage of using assertions for debugging your design, we'll start by simulating the interleaver without assertions.

Procedure

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 Questa SIM if necessary.

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      Upon opening Questa SIM for the first time, you will see the Welcome to Questa SIM dialog box. 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.

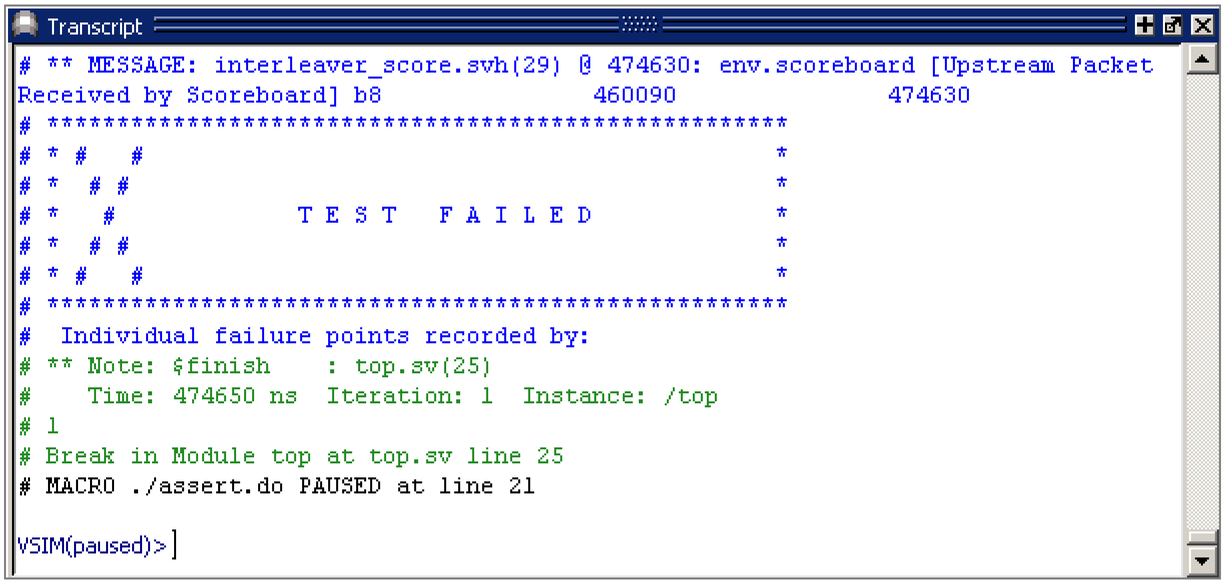
   a. Enter **do assert.do** at the Questa SIM> prompt.

      The *.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 box 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 window as shown in Figure 15-4. The summary information shows that 22 packets were correctly received by the scoreboard. This is a typical message from a self-checking test bench.

Figure 15-4. First Simulation Stops at Error



```
# ** MESSAGE: interleaver_score.svh(29) @ 474630: env.scoreboard [Upstream Packet
Received by Scoreboard] b8                 460090              474630
# *****************************************************
# * #   #                                            *
# *  # #                                             *
# *   #            T E S T   F A I L E D             *
# *  # #                                             *
# * #   #                                            *
# *****************************************************
#  Individual failure points recorded by:
# ** Note: $finish    : top.sv(25)
#    Time: 474650 ns  Iteration: 1  Instance: /top
# 1
# Break in Module top at top.sv line 25
# MACRO ./assert.do PAUSED at line 21

VSIM(paused)>
```

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

Now we'll run the simulation with assertions to demonstrate how assertions are used to speed debugging.

Procedure

1. Rerun the simulation with assertions.

   a. In the Transcript window, enter the **resume** command at the VSIM(paused)> prompt.

2. After the design loads, configure all assertions to "Break on Failure."

   a. The Assertions window should open. If it does not, select **View > Coverage > Assertions**.

   Notice that the assertions are enabled for both Pass and Failure for every assertion. 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 is 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. Click the Assertions tab or the header bar of the Assertions window to make it active. An “Assertions” menu selection will appear in the menu bar.

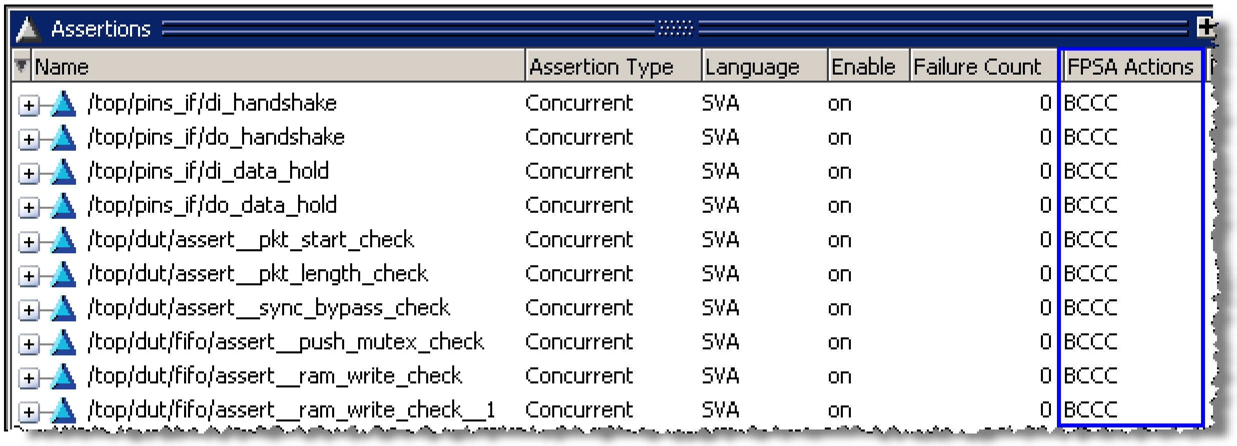
c. Make sure none of the assertions are selected (**Edit > Unselect All**).

d. Execute the following command:

assertion action -cond fail -exec break -r *

The FPSA Action column now shows "B" for “break” on any assertion failure (Figure 15-5). (FPSA indicates Failures, Passes, Starts, and Antecedents.) The column displays four letters. The first letter indicates the action for Failures, the second is the action for Passes, the third for Starts, and the fourth for Antecedents. (The Actions are Continue, Break, Exit, and TCL.) If the FPSA Action column is not displayed, you can view it by clicking the down arrow at the left end of the column header bar and selecting FPSA Action from the Configure Columns menu.

Figure 15-5. Assertions Set to Break on Failure



Name	Assertion Type	Language	Enable	Failure Count	FPSA Actions
/top/pins_if/di_handshake	Concurrent	SVA	on	0	BCCC
/top/pins_if/do_handshake	Concurrent	SVA	on	0	BCCC
/top/pins_if/di_data_hold	Concurrent	SVA	on	0	BCCC
/top/pins_if/do_data_hold	Concurrent	SVA	on	0	BCCC
/top/dut/assert__pkt_start_check	Concurrent	SVA	on	0	BCCC
/top/dut/assert__pkt_length_check	Concurrent	SVA	on	0	BCCC
/top/dut/assert__sync_bypass_check	Concurrent	SVA	on	0	BCCC
/top/dut/fifo/assert__push_mutex_check	Concurrent	SVA	on	0	BCCC
/top/dut/fifo/assert__ram_write_check	Concurrent	SVA	on	0	BCCC
/top/dut/fifo/assert__ram_write_check__1	Concurrent	SVA	on	0	BCCC

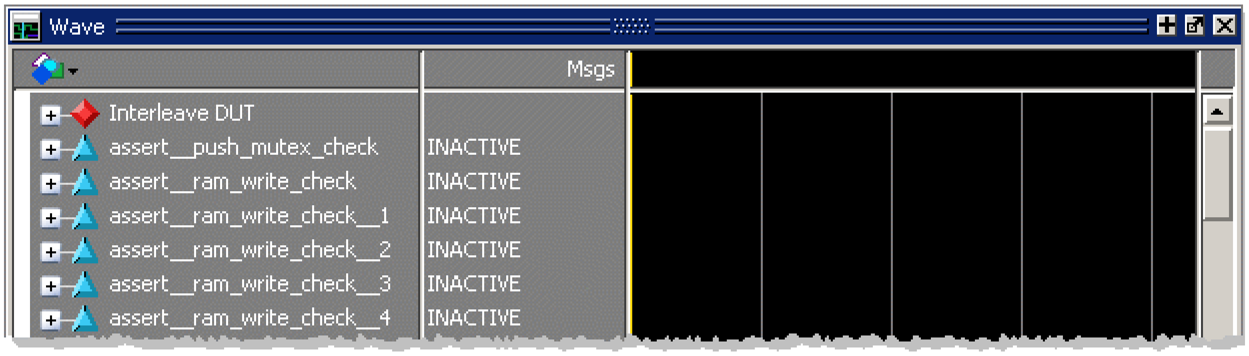
3. Add all assertions related to */top/dut/fifo* to the Wave window

a. In the Assertions window select */top/dut/fifo/assert_push_mutex_check*.

b. Press and hold the **Shift** key and select */top/dut/fifo/assert_ram_read_check_10*. All assertions related to */top/dut/fifo* should be selected (highlighted in blue).

c. Select **Add > To Wave > Selected Objects** from the Main menus. The selected assertions will appear in the Wave window, as in Figure 15-6.

Figure 15-6. Assertions in Wave Window



Interleave DUT
assert__push_mutex_check INACTIVE
assert__ram_write_check INACTIVE
assert__ram_write_check__1 INACTIVE
assert__ram_write_check__2 INACTIVE
assert__ram_write_check__3 INACTIVE
assert__ram_write_check__4 INACTIVE

Debugging with Assertions

Run the simulation and debug the assertion failure.

Procedure

1. Run the simulation with assertion failure tracking enabled.

    a. Enter **run -all** at the Questa SIM prompt of the Transcript window.

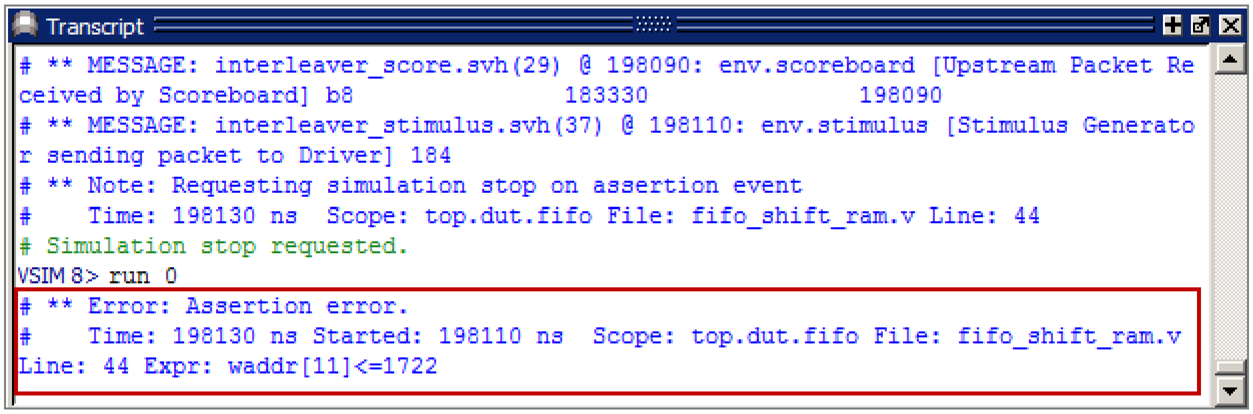
    b. When the simulator stops, enter **run 0**.

    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 window (Figure 15-7).

    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 15-7. Assertion Failure Message in the Transcript

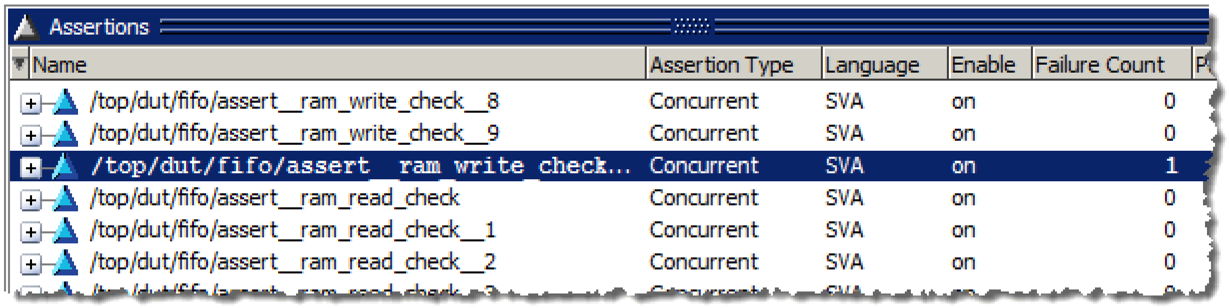


```
# ** MESSAGE: interleaver_score.svh(29) @ 198090: env.scoreboard [Upstream Packet Re
ceived by Scoreboard] b8                183330              198090
# ** MESSAGE: interleaver_stimulus.svh(37) @ 198110: env.stimulus [Stimulus Generato
r sending packet to Driver] 184
# ** Note: Requesting simulation stop on assertion event
#    Time: 198130 ns  Scope: top.dut.fifo File: fifo_shift_ram.v Line: 44
# Simulation stop requested.
VSIM 8> run 0
# ** Error: Assertion error.
#    Time: 198130 ns Started: 198110 ns  Scope: top.dut.fifo File: fifo_shift_ram.v
Line: 44 Expr: waddr[11]<=1722
```

3. View the assertion failure in the Assertions window.

   The failed assertion is highlighted and ’1’ is displayed in the Failure Count column for that assertion in the Verilog design (Figure 15-8).

Figure 15-8. Assertions Tab Shows Failure Count

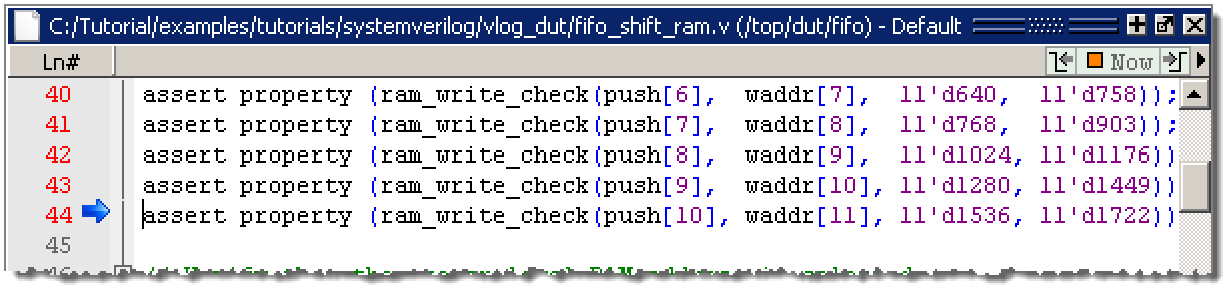


Name	Assertion Type	Language	Enable	Failure Count
/top/dut/fifo/assert__ram_write_check__8	Concurrent	SVA	on	0
/top/dut/fifo/assert__ram_write_check__9	Concurrent	SVA	on	0
/top/dut/fifo/assert__ram_write_check...	Concurrent	SVA	on	1
/top/dut/fifo/assert__ram_read_check	Concurrent	SVA	on	0
/top/dut/fifo/assert__ram_read_check__1	Concurrent	SVA	on	0
/top/dut/fifo/assert__ram_read_check__2	Concurrent	SVA	on	0

4. Examine the *fifo_shift_ram_props.v* source code view. The *fifo_shift_ram.v* tab should be open, as shown in (Figure 15-9).

   The simulation breaks on line 44 of the *fifo_shift_ram.v* module because the assertion on that line has failed. A blue arrow points to the failed assertion.

Figure 15-9. Simulation Stopped at Blue Pointer

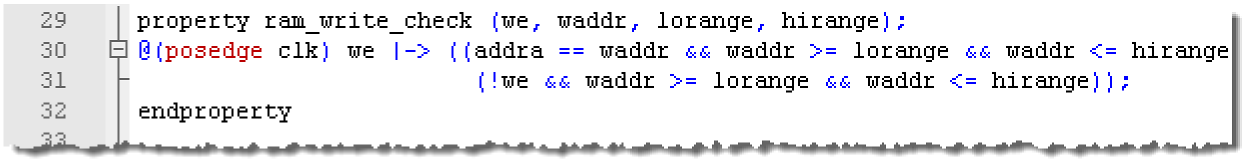

C:/Tutorial/examples/tutorials/systemverilog/vlog_dut/fifo_shift_ram.v (/top/dut/fifo) - Default
Ln#
Now

```
40   assert property (ram_write_check(push[6],  waddr[7],  11'd640,  11'd758));
41   assert property (ram_write_check(push[7],  waddr[8],  11'd768,  11'd903));
42   assert property (ram_write_check(push[8],  waddr[9],  11'd1024, 11'd1176))
43   assert property (ram_write_check(push[9],  waddr[10], 11'd1280, 11'd1449))
44   assert property (ram_write_check(push[10], waddr[11], 11'd1536, 11'd1722))
45
```

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.

Figure 15-10. Assertion Property Definition



```
property ram_write_check (we, waddr, lorange, hirange);
@(posedge clk) we |-> ((addra == waddr && waddr >= lorange && waddr <= hirange
                        (!we && waddr >= lorange && waddr <= hirange));
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:

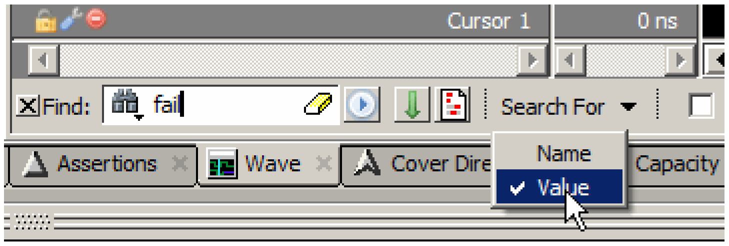
- *we* should be de-asserted,
- and, the next value of *waddr[11]* should still be within the range 1536 to 1722.

5. Click the **Wave** tab to search for and view the assertion failure in the Wave window.

a. **Edit** > **Find** to display the search bar in the Wave window.

b. In the search bar, select **Search For** > **Value** (Figure 15-11).

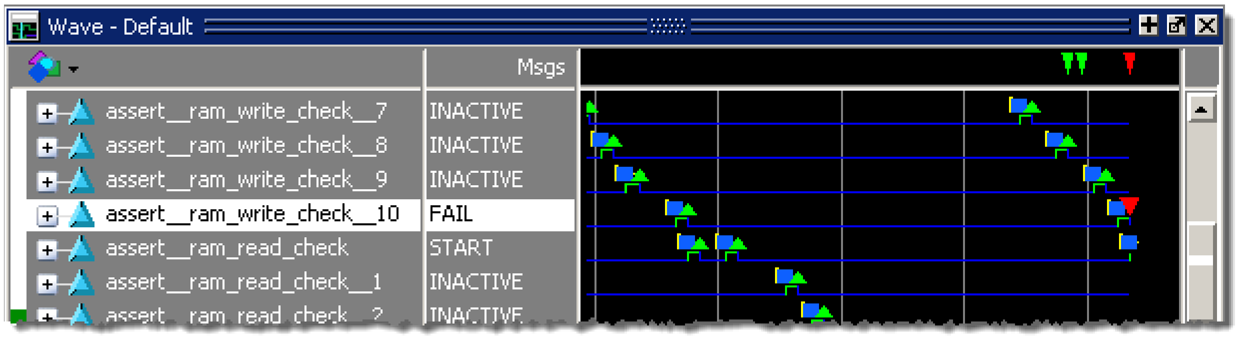
Figure 15-11. Search For Value



c. In the search bar text entry box enter: fail
The search begins as you type, and will highlight the "FAIL" value.

The inverted red triangle in the waveform view indicates an assertion failure (Figure 15-12).

Figure 15-12. Inverted Red Triangle Indicates Assertion Failure



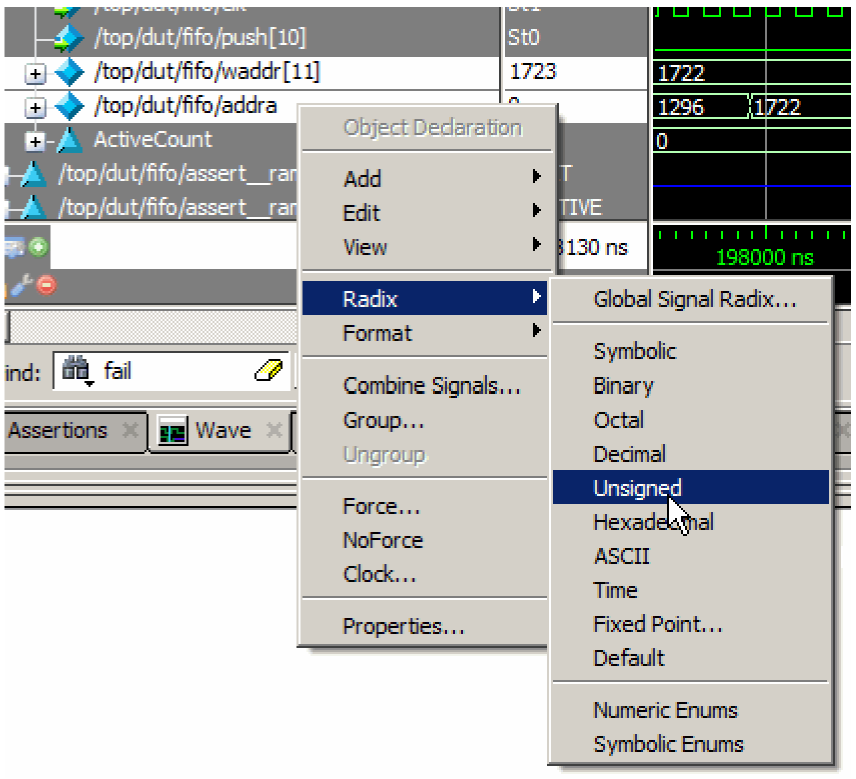
	Msgs
assert__ram_write_check__7	INACTIVE
assert__ram_write_check__8	INACTIVE
assert__ram_write_check__9	INACTIVE
assert__ram_write_check__10	FAIL
assert__ram_read_check	START
assert__ram_read_check__1	INACTIVE
assert__ram_read_check__2	INACTIVE

- 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).

d. Expand the *assert_ram_write_check_10* assertion (click the + sign next to it) in the Wave window and zoom in.

e. Change the radix of *addra* and *waddr* to “Unsigned” by selecting both signals, right-clicking the selected signals, then selecting **Radix** > **Unsigned** from the popup menu (Figure 15-13).

Figure 15-13. Setting the Radix

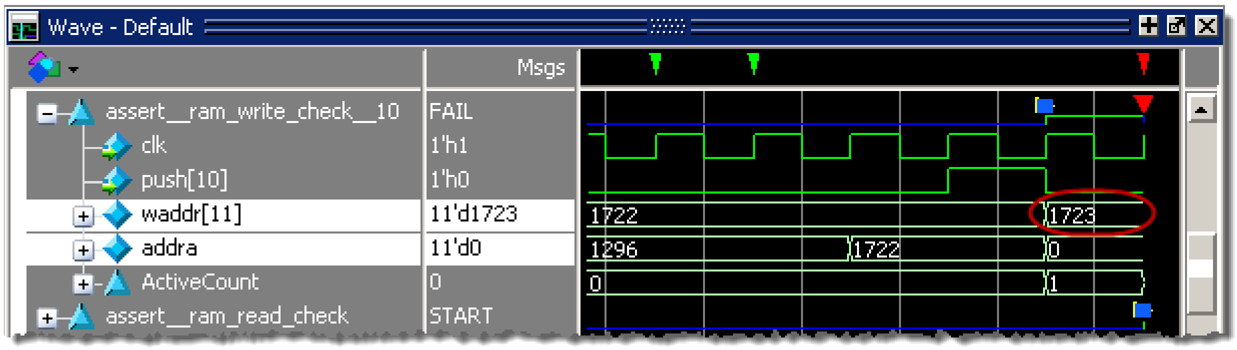


As you can see in Figure 15-14, the value of waddr[11] has incremented to 1723 which is out of the allowable address range. Remember, in the Transcript message for the assertion violation, the failing expression indicated that waddr[11] was out of range.

6. Examine the 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 15-14).

Figure 15-14. Diagnosing Assertion Failure in the Wave Window

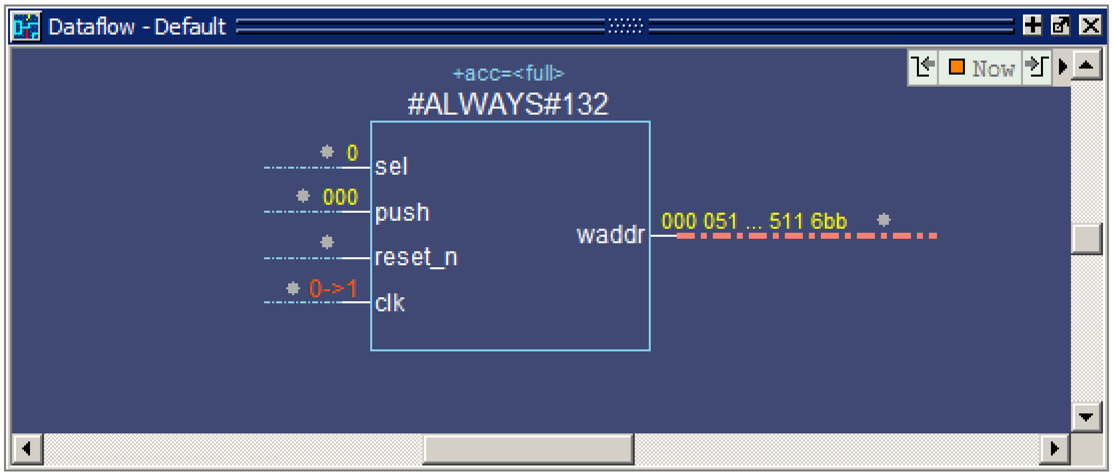


   b. Click the *waddr[11]* signal to select it, then select **Add** > **To Dataflow** > **Selected Items** from the menus.

This opens the selected signal in the Dataflow window.

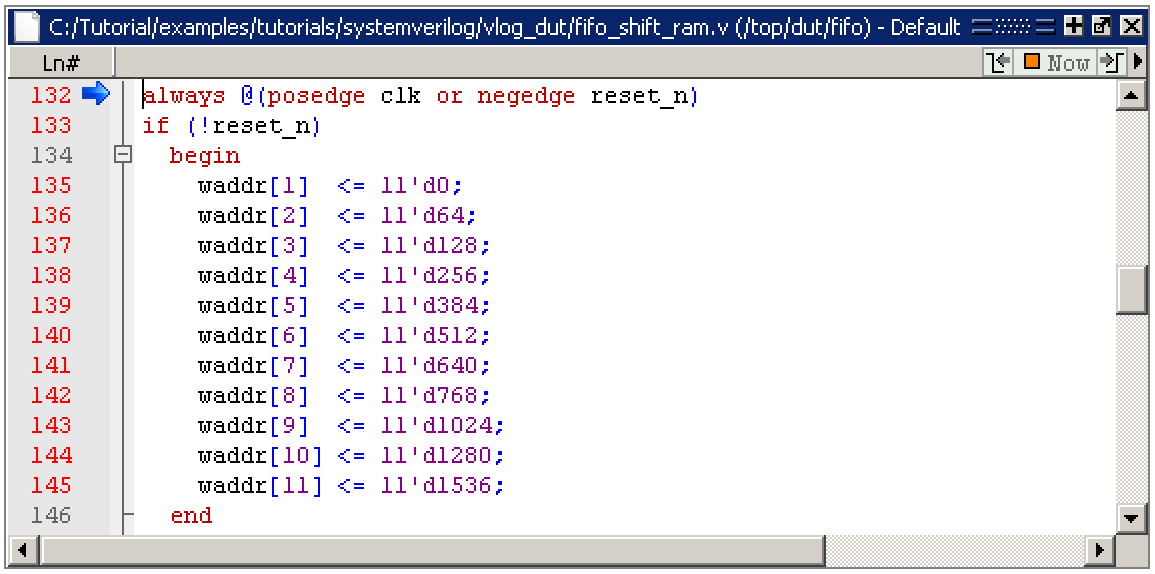
The *waddr[11]* signal will be highlighted, as shown in Figure 15-15, and the block shown is the ALWAYS procedure.

Figure 15-15. The waddr11 Signal in the Dataflow Window



c. Double-click 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 block (Figure 15-16).

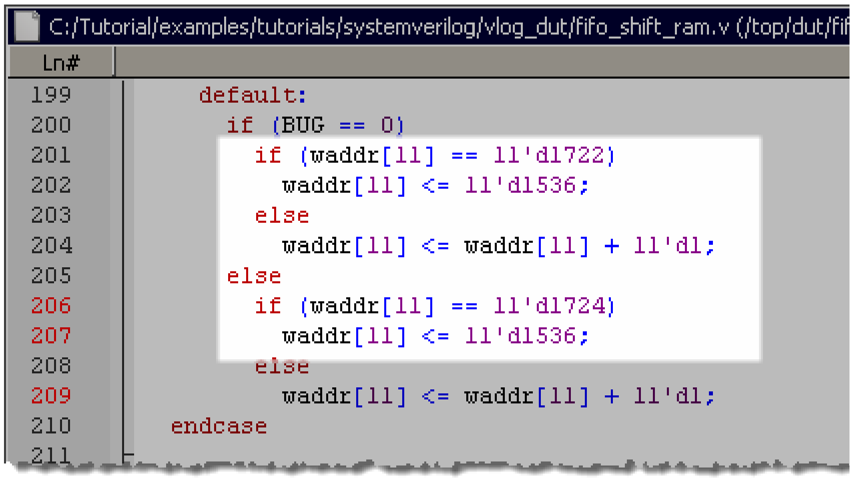
Figure 15-16. Source Code for the ALWAYS Block


C:/Tutorial/examples/tutorials/systemverilog/vlog_dut/fifo_shift_ram.v (/top/dut/fifo) - Default
Ln#
Now

```
always @(posedge clk or negedge reset_n)
if (!reset_n)
  begin
    waddr[1]  <= 11'd0;
    waddr[2]  <= 11'd64;
    waddr[3]  <= 11'd128;
    waddr[4]  <= 11'd256;
    waddr[5]  <= 11'd384;
    waddr[6]  <= 11'd512;
    waddr[7]  <= 11'd640;
    waddr[8]  <= 11'd768;
    waddr[9]  <= 11'd1024;
    waddr[10] <= 11'd1280;
    waddr[11] <= 11'd1536;
  end
```

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 11’1724 instead of 11’1722 (Figure 15-17). This is the cause of the error.

Figure 15-17. Source Code for waddr[11]



```
C:/Tutorial/examples/tutorials/systemverilog/vlog_dut/fifo_shift_ram.v (/top/dut/fi
Ln#
199       default:
200         if (BUG == 0)
201           if (waddr[11] == 11'd1722)
202             waddr[11] <= 11'd1536;
203           else
204             waddr[11] <= waddr[11] + 11'd1;
205         else
206           if (waddr[11] == 11'd1724)
207             waddr[11] <= 11'd1536;
208           else
209             waddr[11] <= waddr[11] + 11'd1;
210     endcase
211
```

7. Quit the simulation.

   a. Enter **quit -sim** at the **Questa SIM** prompt.

Exploring Functional Coverage

The functional coverage capabilities of SystemVerilog allow you to verify your designs on a functional level.

Procedure

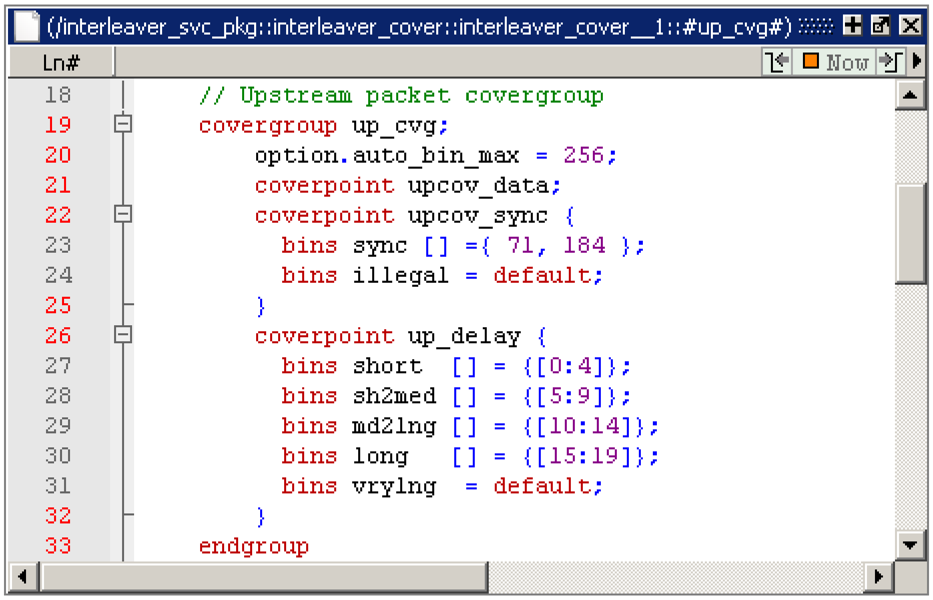
1. Load the interleaver once again.

   a. Enter **do fcov.do** at the Questa SIM> prompt.

Note
The *fcov.do* file includes the -cvg63 option to the vsim command. The -cvg63 option preserves data collection for individual instances, coverpoints, and crosses. Refer to SystemVerilog 2008 type_option.merge_instances for more information.

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 15-18 shows the source code of the *up_cvg* covergroup.

Figure 15-18. Covergroup Code


(/interleaver_svc_pkg::interleaver_cover::interleaver_cover__1::#up_cvg#)
Ln#
Now

```
    // Upstream packet covergroup
    covergroup up_cvg;
        option.auto_bin_max = 256;
        coverpoint upcov_data;
        coverpoint upcov_sync {
          bins sync [] ={ 71, 184 };
          bins illegal = default;
        }
        coverpoint up_delay {
          bins short  [] = {[0:4]};
          bins sh2med [] = {[5:9]};
          bins md2lng [] = {[10:14]};
          bins long   [] = {[15:19]};
          bins vrylng  = default;
        }
    endgroup
```

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.

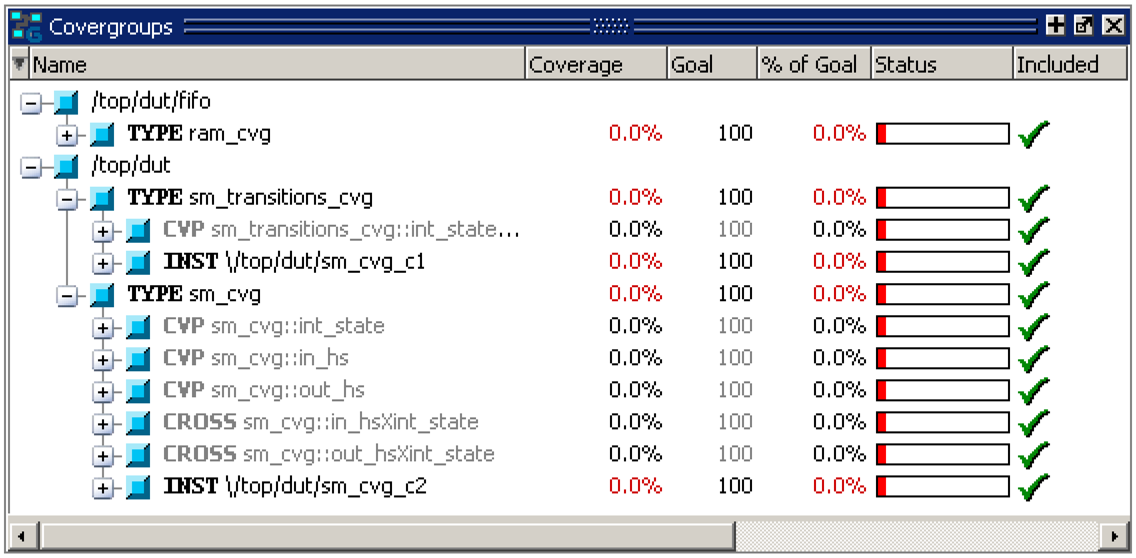
All other sync byte values are stored in the bin named *illegal* which should remain empty. The packet payload data transfer delays times are recorded since the driver randomly drives data to the interleaver. The packet payload data transfer delay bin names are self descriptive and create a separate bin for each delay value except for the *vrylng* (very long) bin which records any data transfer delay of 20 or more cycles.

2. **run 0**

   Covergroups are not displayed until the simulation is run. This step simply enables you to view covergroups in the Covergroups window.

3. In the Covergroups window (**View** > **Coverage** > **Covergroups**), 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 15-19. Covergroup Bins

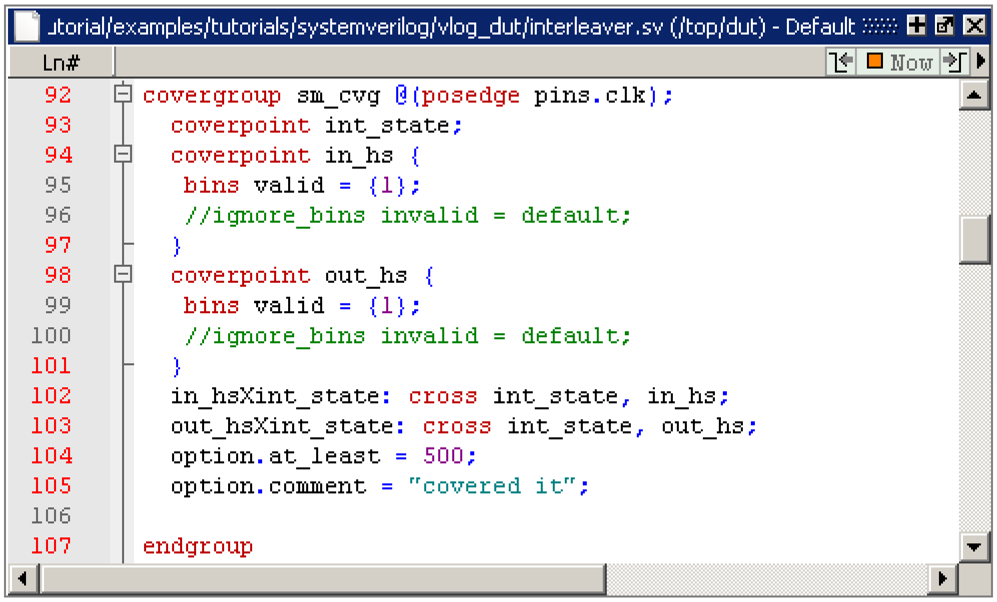


Name	Coverage	Goal	% of Goal	Status	Included
/top/dut/fifo					
TYPE ram_cvg	0.0%	100	0.0%	🟥	✓
/top/dut					
TYPE sm_transitions_cvg	0.0%	100	0.0%	🟥	✓
CVP sm_transitions_cvg::int_state...	0.0%	100	0.0%	🟥	✓
INST \/top/dut/sm_cvg_c1	0.0%	100	0.0%	🟥	✓
TYPE sm_cvg	0.0%	100	0.0%	🟥	✓
CVP sm_cvg::int_state	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%	🟥	✓
INST \/top/dut/sm_cvg_c2	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 15-20 shows the source code for the *sm_cvg* covergroup.

Figure 15-20. Covergroup sm_svg

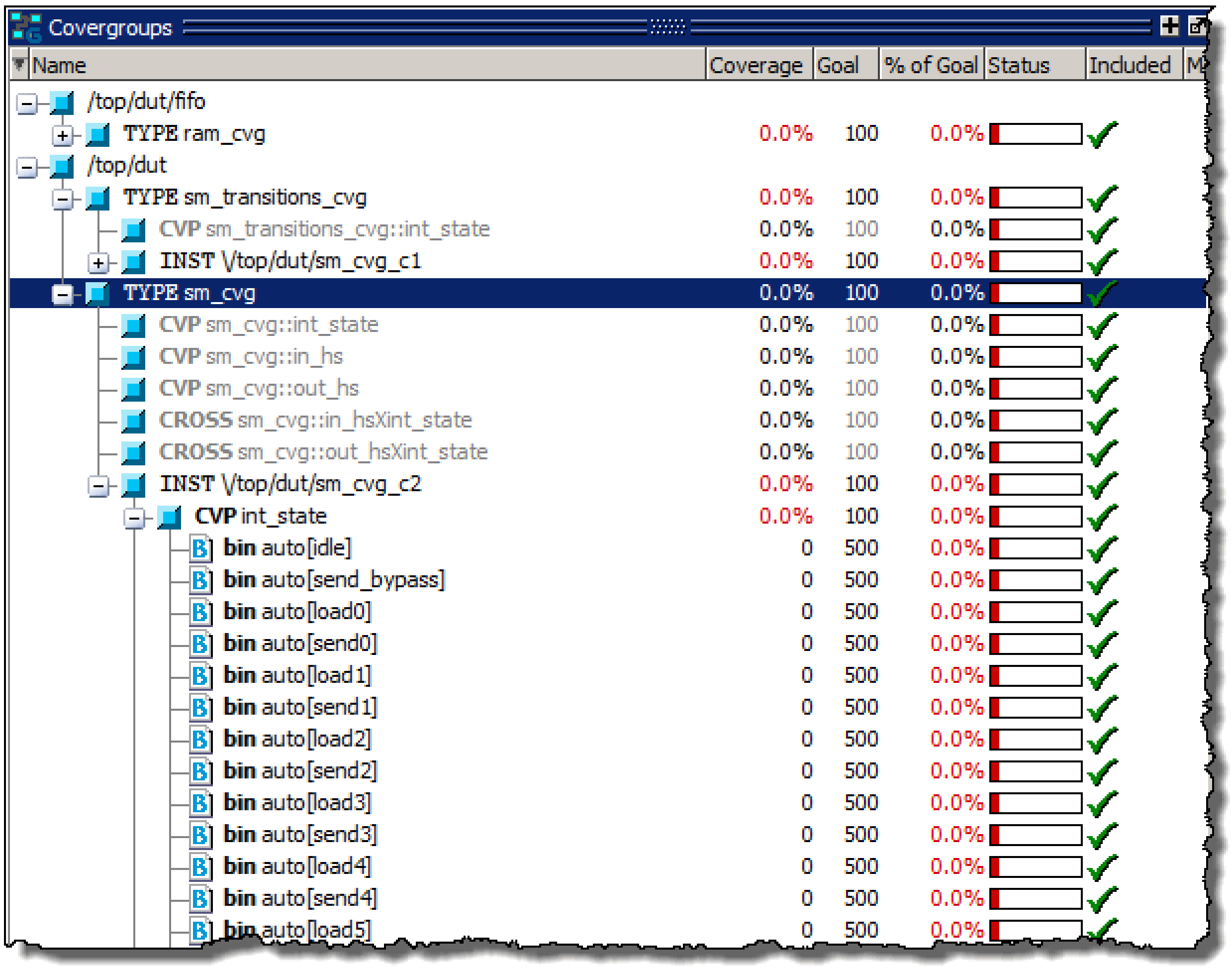

utorial/examples/tutorials/systemverilog/vlog_dut/interleaver.sv (/top/dut) - Default
Ln#
Now

```
covergroup sm_cvg @(posedge pins.clk);
  coverpoint int_state;
  coverpoint in_hs {
   bins valid = {1};
   //ignore_bins invalid = default;
  }
  coverpoint out_hs {
   bins valid = {1};
   //ignore_bins invalid = default;
  }
  in_hsXint_state: cross int_state, in_hs;
  out_hsXint_state: cross int_state, out_hs;
  option.at_least = 500;
  option.comment = "covered it";

endgroup
```

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 *oup_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 15-21 shows the *sm_cvg* covergroup with the *int_state* coverpoint expanded to show all bins. Notice the bin values show the enumerated state names.

Figure 15-21. Bins for the sm_cvg Covergroup



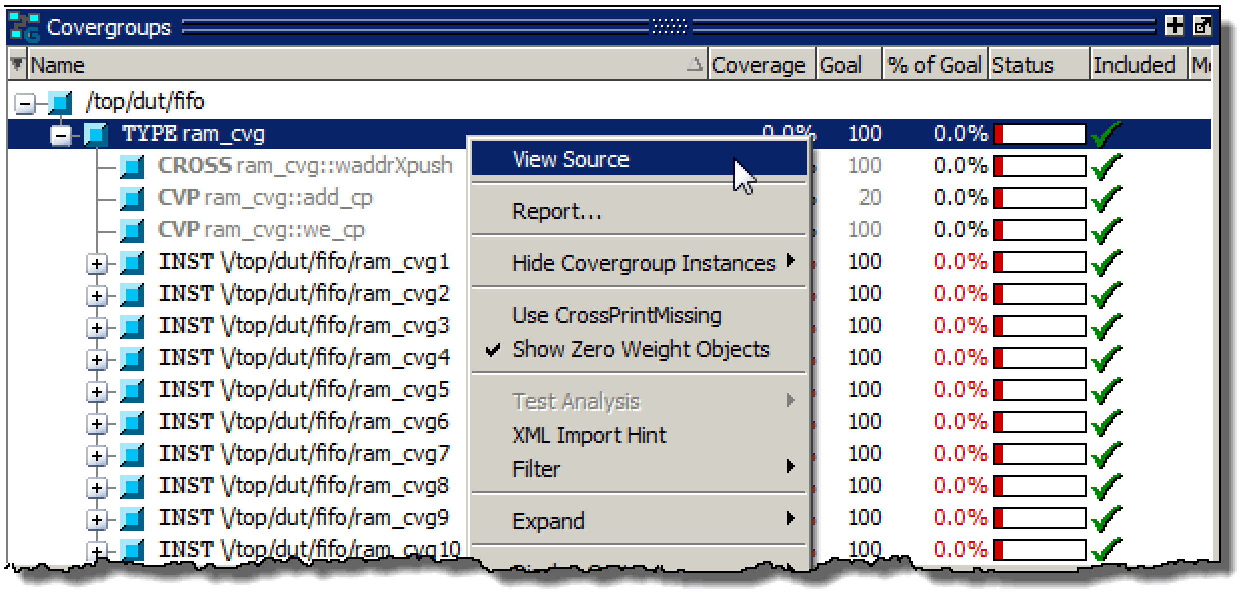
Covergroups

Name	Coverage	Goal	% of Goal	Status	Included	M
/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%		✓	
INST \/top/dut/sm_cvg_c1	0.0%	100	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%		✓	
INST \/top/dut/sm_cvg_c2	0.0%	100	0.0%		✓	
CVP 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%		✓	

4. 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 15-22).

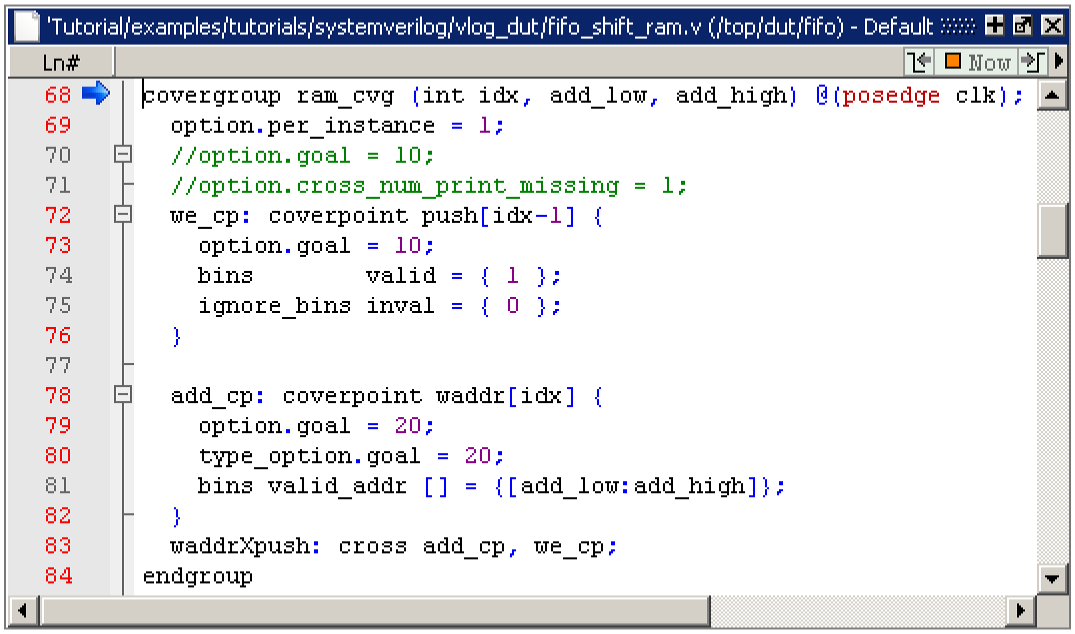
Figure 15-22. Viewing the Source Code for a Covergroup



Name	Coverage	Goal	% of Goal	Status	Included
/top/dut/fifo					
TYPE ram_cvg	0.0%	100	0.0%		✓
CROSS ram_cvg::waddrXpush		100	0.0%		✓
CVP ram_cvg::add_cp		20	0.0%		✓
CVP ram_cvg::we_cp		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg1		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg2		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg3		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg4		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg5		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg6		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg7		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg8		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg9		100	0.0%		✓
INST \/top/dut/fifo/ram_cvg10		100	0.0%		✓

The *fifo_shift_ram.v* source view will open to show the source code (Figure 15-23).

Figure 15-23. Source Code for ram_cvg Covergroup



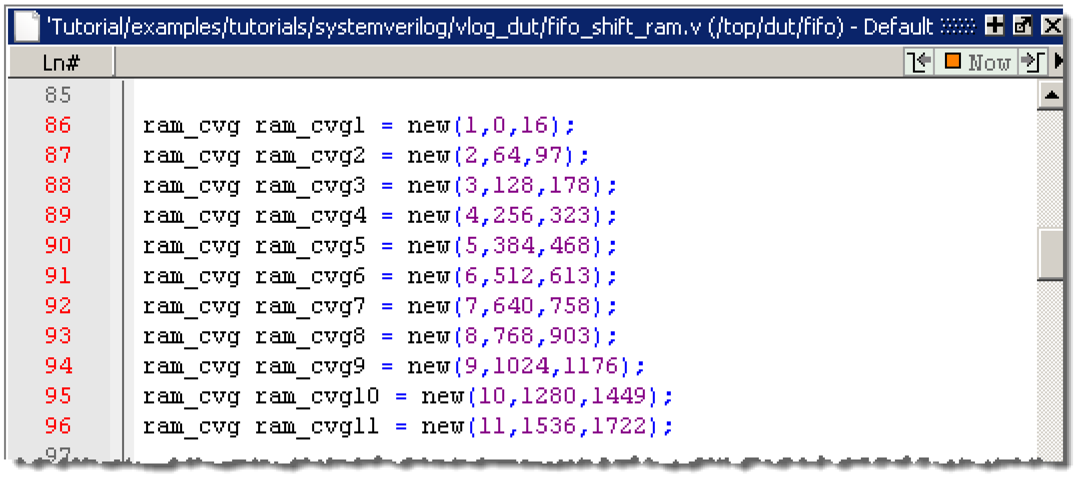
'Tutorial/examples/tutorials/systemverilog/vlog_dut/fifo_shift_ram.v (/top/dut/fifo) - Default

```
covergroup ram_cvg (int idx, add_low, add_high) @(posedge clk);
  option.per_instance = 1;
  //option.goal = 10;
  //option.cross_num_print_missing = 1;
  we_cp: coverpoint push[idx-1] {
    option.goal = 10;
    bins        valid = { 1 };
    ignore_bins inval = { 0 };
  }

  add_cp: coverpoint waddr[idx] {
    option.goal = 20;
    type_option.goal = 20;
    bins valid_addr [] = {[add_low:add_high]};
  }
  waddrXpush: cross add_cp, we_cp;
endgroup
```

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 15-24).

Figure 15-24. Covergroup Instances for ram_cvg


'Tutorial/examples/tutorials/systemverilog/vlog_dut/fifo_shift_ram.v (/top/dut/fifo) - Default
Ln#
Now

```
ram_cvg ram_cvg1 = new(1,0,16);
ram_cvg ram_cvg2 = new(2,64,97);
ram_cvg ram_cvg3 = new(3,128,178);
ram_cvg ram_cvg4 = new(4,256,323);
ram_cvg ram_cvg5 = new(5,384,468);
ram_cvg ram_cvg6 = new(6,512,613);
ram_cvg ram_cvg7 = new(7,640,758);
ram_cvg ram_cvg8 = new(8,768,903);
ram_cvg ram_cvg9 = new(9,1024,1176);
ram_cvg ram_cvg10 = new(10,1280,1449);
ram_cvg ram_cvg11 = new(11,1536,1722);
```

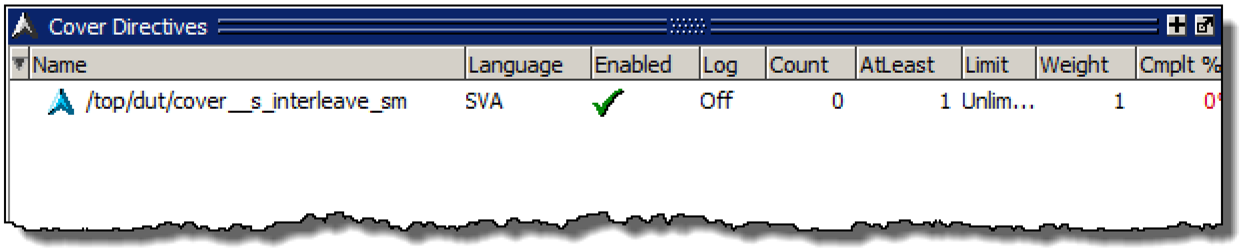
Since the covergroup contains the *option.per_instatnce = 1* statement (Figure 15-23), the simulator creates a separate covergroup for each instance which covers only the values passed to it in the constructor. The TYPE *ram_cvg* covergroup is the union of all the values of each individual covergroup instance.

5. Open the Cover Directives window and view the source code for the cover directive.

   a. If the Cover Directives window is not open, select **View** > **Coverage** > **Cover Directives**.

   The Cover Directives tab contains a single cover directive (Figure 15-25).

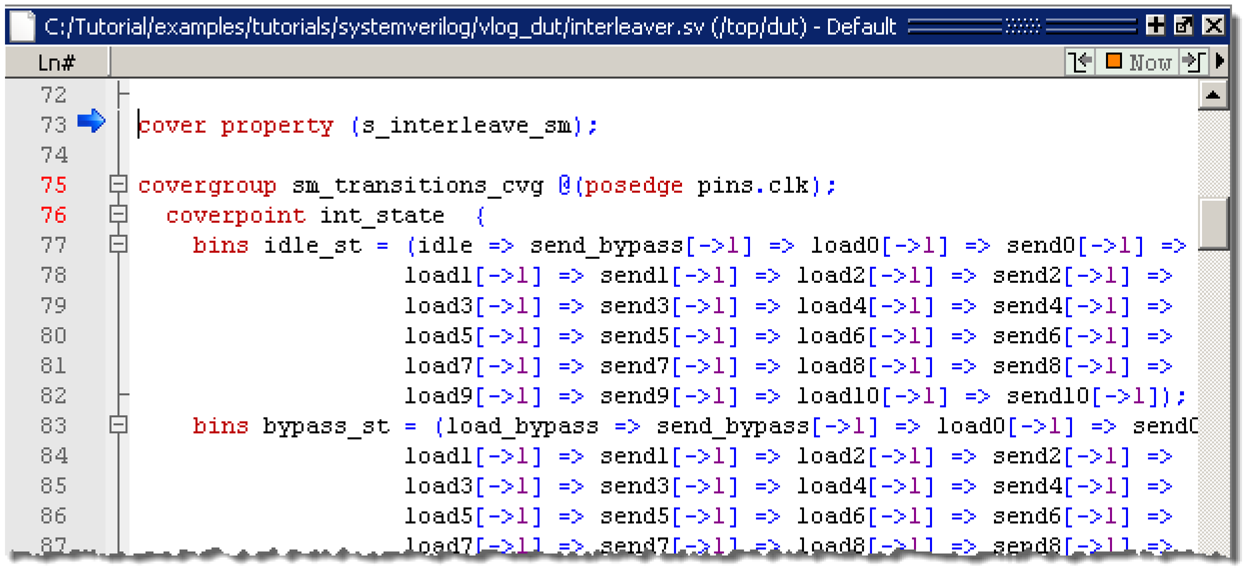
Figure 15-25. Cover Directive for the Interleaver Design



Name	Language	Enabled	Log	Count	AtLeast	Limit	Weight	Cmplt %
/top/dut/cover__s_interleave_sm	SVA	✓	Off	0	1	Unlim...	1	0

   b. Right-click the cover directive and select **View Source** from the popup menu. Figure 15-26 shows that this cover directive also tracks the interleaver state machine transitions.

Figure 15-26. Source Code for the Cover Directive


C:/Tutorial/examples/tutorials/systemverilog/vlog_dut/interleaver.sv (/top/dut) - Default
Ln#
Now

```
cover property (s_interleave_sm);

covergroup sm_transitions_cvg @(posedge pins.clk);
  coverpoint int_state  {
    bins idle_st = (idle => send_bypass[->1] => load0[->1] => send0[->1] =>
                    load1[->1] => send1[->1] => load2[->1] => send2[->1] =>
                    load3[->1] => send3[->1] => load4[->1] => send4[->1] =>
                    load5[->1] => send5[->1] => load6[->1] => send6[->1] =>
                    load7[->1] => send7[->1] => load8[->1] => send8[->1] =>
                    load9[->1] => send9[->1] => load10[->1] => send10[->1]);
    bins bypass_st = (load_bypass => send_bypass[->1] => load0[->1] => send0
                    load1[->1] => send1[->1] => load2[->1] => send2[->1] =>
                    load3[->1] => send3[->1] => load4[->1] => send4[->1] =>
                    load5[->1] => send5[->1] => load6[->1] => send6[->1] =>
                    load7[->1] => send7[->1] => load8[->1] => send8[->1] =>
```

SystemVerilog provides multiple ways to cover important items in a design. The advantage of using a cover directive is that the Questa SIM 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.

6. Add the cover directive to the Wave window twice.

   a. Return to the Cover Directives window and right-click the */top/dut/cover__s_interleave_sm* cover directive and select **Add Wave > Selected Functional Coverage**.

   b. Repeat.

   We'll explain why in a moment.

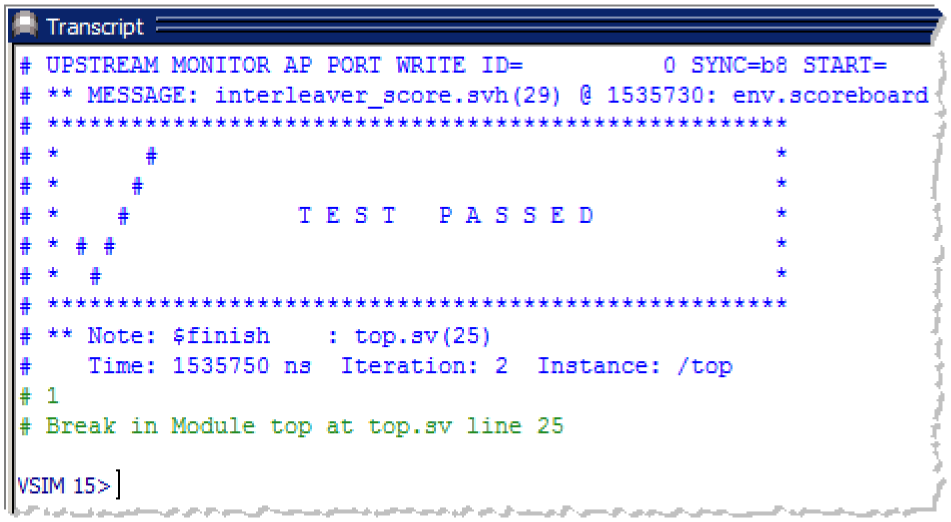
7. Run the simulation and view functional coverage information.

   a. Change your WildcardFilter to allow cover directives to be logged. The following command does not include the "Cover" argument, which is filtered by default.

   set WildcardFilter "Variable Constant Generic Parameter SpecParam Memory Assertion Endpoint CellInternal ImmediateAssert"

   b. 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 box that will ask, "Are you sure you want to finish?" Click **No**.) The Transcript window will display scoreboard information (Figure 15-27).

Figure 15-27. Scoreboard Information in the Transcript

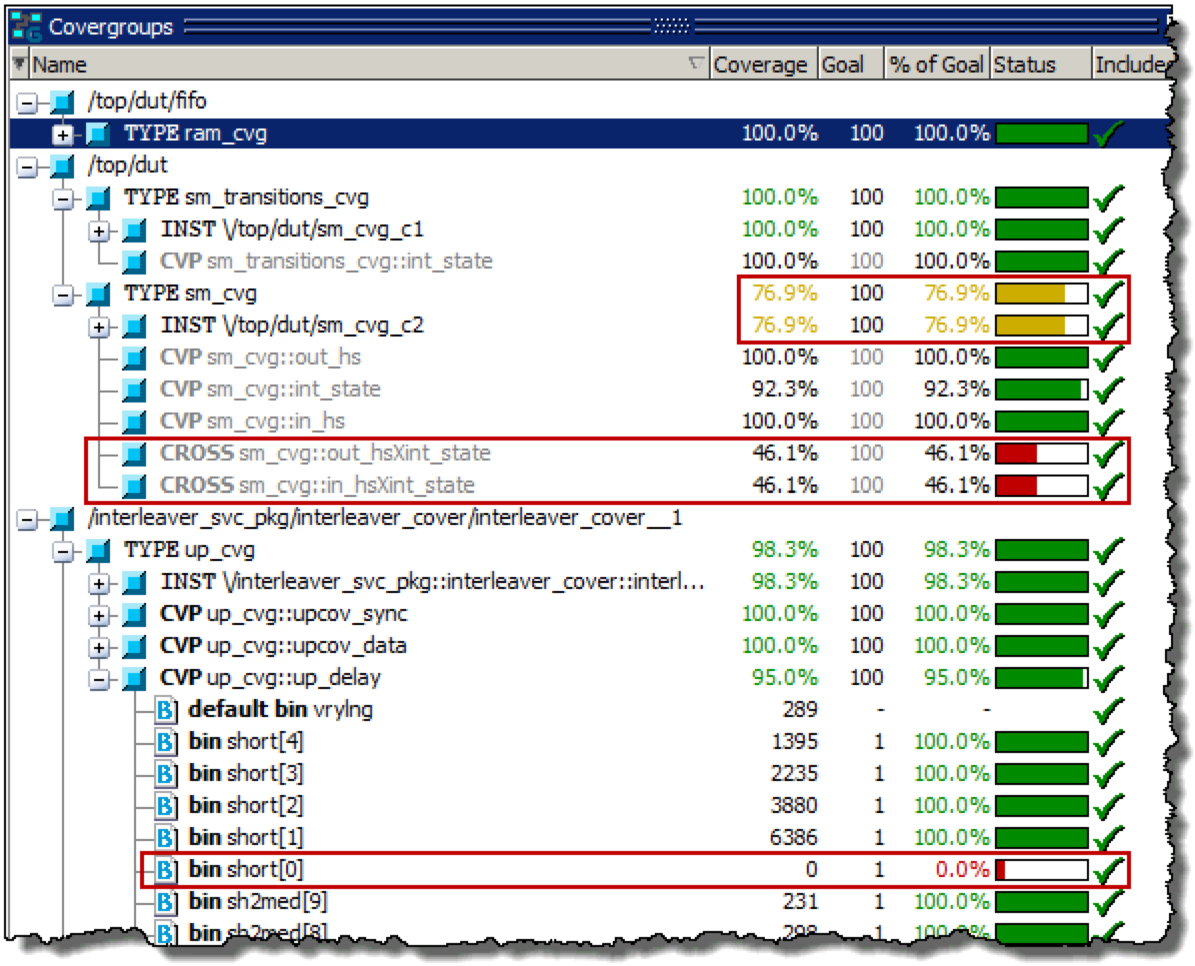


```
# UPSTREAM MONITOR AP PORT WRITE ID=          0 SYNC=b8 START=
# ** MESSAGE: interleaver_score.svh(29) @ 1535730: env.scoreboard
# *****************************************************
# *      #                                            *
# *     #                                             *
# *    #           T E S T   P A S S E D              *
# * # #                                               *
# *  #                                                *
# *****************************************************
# ** Note: $finish    : top.sv(25)
#    Time: 1535750 ns  Iteration: 2  Instance: /top
# 1
# Break in Module top at top.sv line 25

VSIM 15>
```

c. Expand the functional coverage information in the Covergroups window as shown in Figure 15-28. While our overall covergroup coverage is almost 95% in the nterleaver design (as shown in the status bar at the bottom right corner 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 coverage (76.9%) 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 15-28. Covergroup Coverage in the Cover Groups Window



Covergroups

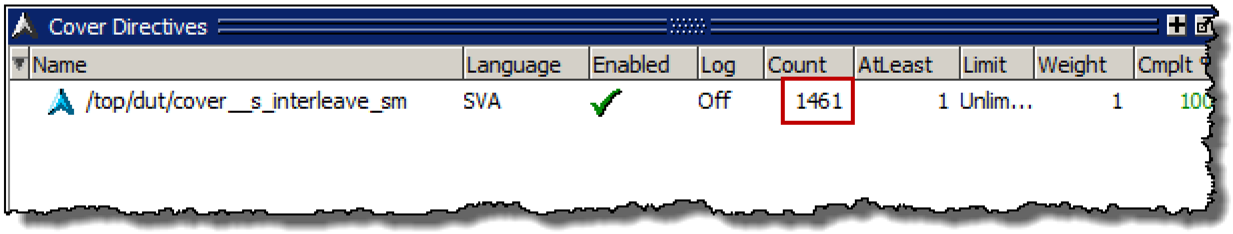
Name	Coverage	Goal	% of Goal	Status	Include
/top/dut/fifo					
TYPE ram_cvg	100.0%	100	100.0%	🟩	✓
/top/dut					
TYPE sm_transitions_cvg	100.0%	100	100.0%	🟩	✓
INST \/top/dut/sm_cvg_c1	100.0%	100	100.0%	🟩	✓
CVP sm_transitions_cvg::int_state	100.0%	100	100.0%	🟩	✓
TYPE sm_cvg	76.9%	100	76.9%	🟨	✓
INST \/top/dut/sm_cvg_c2	76.9%	100	76.9%	🟨	✓
CVP sm_cvg::out_hs	100.0%	100	100.0%	🟩	✓
CVP sm_cvg::int_state	92.3%	100	92.3%	🟩	✓
CVP sm_cvg::in_hs	100.0%	100	100.0%	🟩	✓
CROSS sm_cvg::out_hsXint_state	46.1%	100	46.1%	🟥	✓
CROSS sm_cvg::in_hsXint_state	46.1%	100	46.1%	🟥	✓
/interleaver_svc_pkg/interleaver_cover/interleaver_cover__1					
TYPE up_cvg	98.3%	100	98.3%	🟩	✓
INST \/interleaver_svc_pkg::interleaver_cover::interl...	98.3%	100	98.3%	🟩	✓
CVP up_cvg::upcov_sync	100.0%	100	100.0%	🟩	✓
CVP up_cvg::upcov_data	100.0%	100	100.0%	🟩	✓
CVP up_cvg::up_delay	95.0%	100	95.0%	🟩	✓
default bin vrylng	289	-	-		✓
bin short[4]	1395	1	100.0%	🟩	✓
bin short[3]	2235	1	100.0%	🟩	✓
bin short[2]	3880	1	100.0%	🟩	✓
bin short[1]	6386	1	100.0%	🟩	✓
bin short[0]	0	1	0.0%	🟥	✓
bin sh2med[9]	231	1	100.0%	🟩	✓
bin sh2med[8]	298	1	100.0%	🟩	✓

If you expand the *sm_transitions_cvg* covergroup you will see that it shows 1461 interleaver state transitions (86 when starting from the idle loop, and 1375 when starting from the bypass loop).

d. Open the Cover Directives tab.

The cover directive counts the same state transitions and, therefore, also indicates a count of 1461 transitions (Figure 15-29).

Figure 15-29. Cover Directive Counts State Transitions



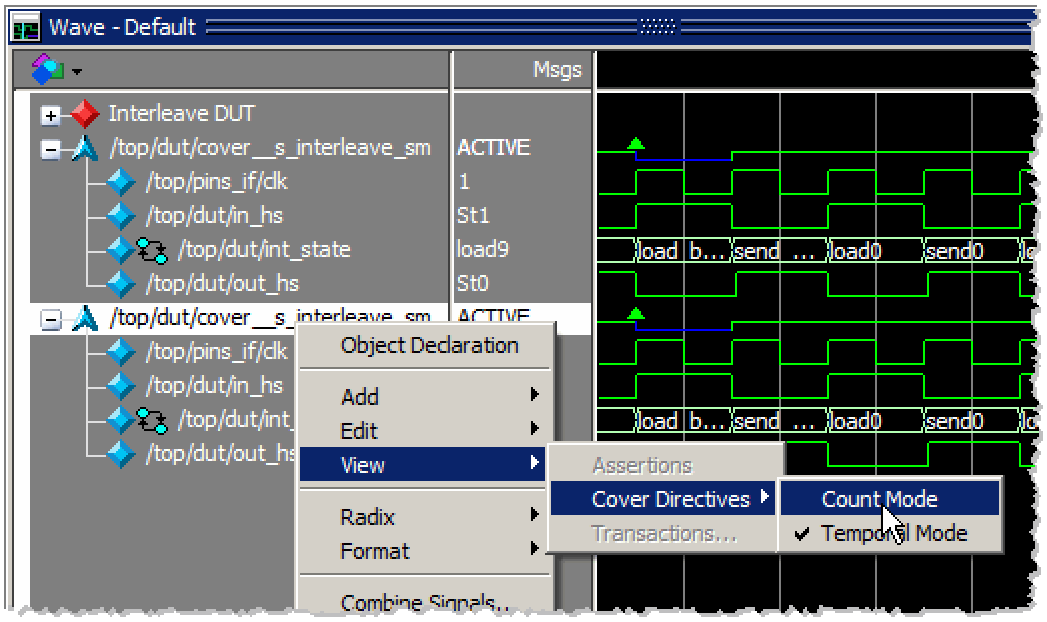
Cover Directives

Name	Language	Enabled	Log	Count	AtLeast	Limit	Weight	Cmplt
/top/dut/cover__s_interleave_sm	SVA	✓	Off	1461	1	Unlim...	1	100

8. 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 **View > Cover Directives > Count Mode** (Figure 15-30).

Figure 15-30. Changing the Cover Directive View to Count View



   b. View the Cover Directive in the Wave window.

   Figure 15-31 and Figure 15-32 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 15-31 there is 1240 ns between the start and end of the cover directive thread; in the next thread, shown in Figure 15-32, there is 780 ns.

Figure 15-31. First Temporal and Count Mode Views of Cover Directive

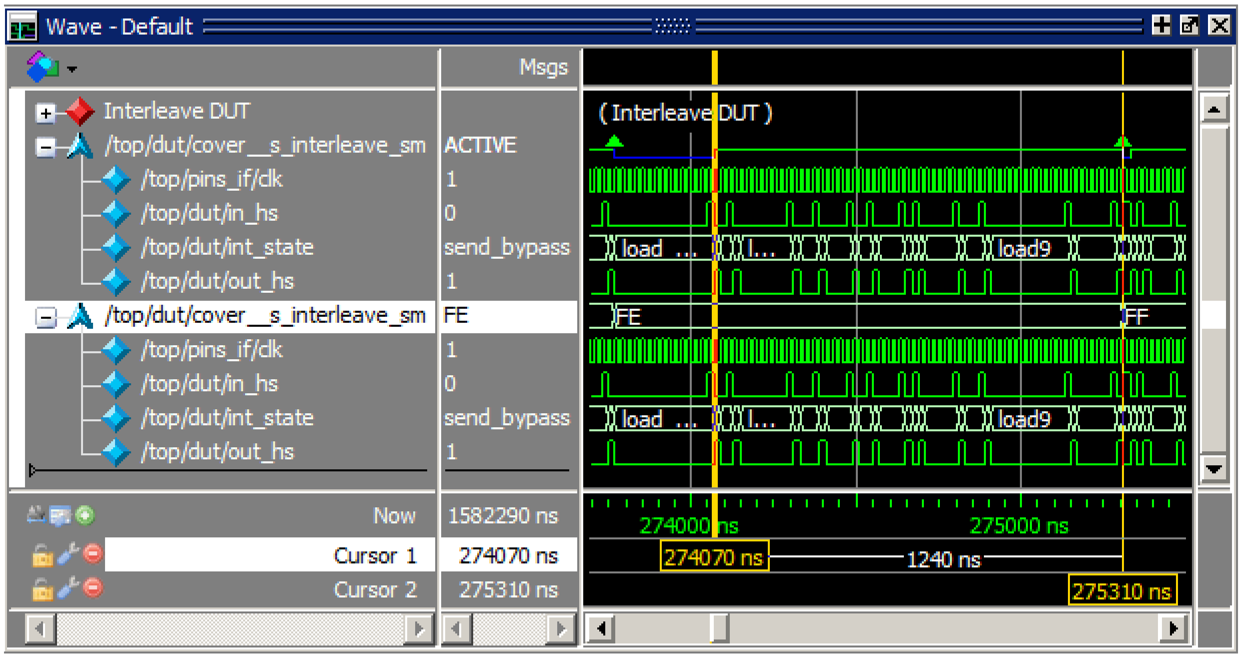
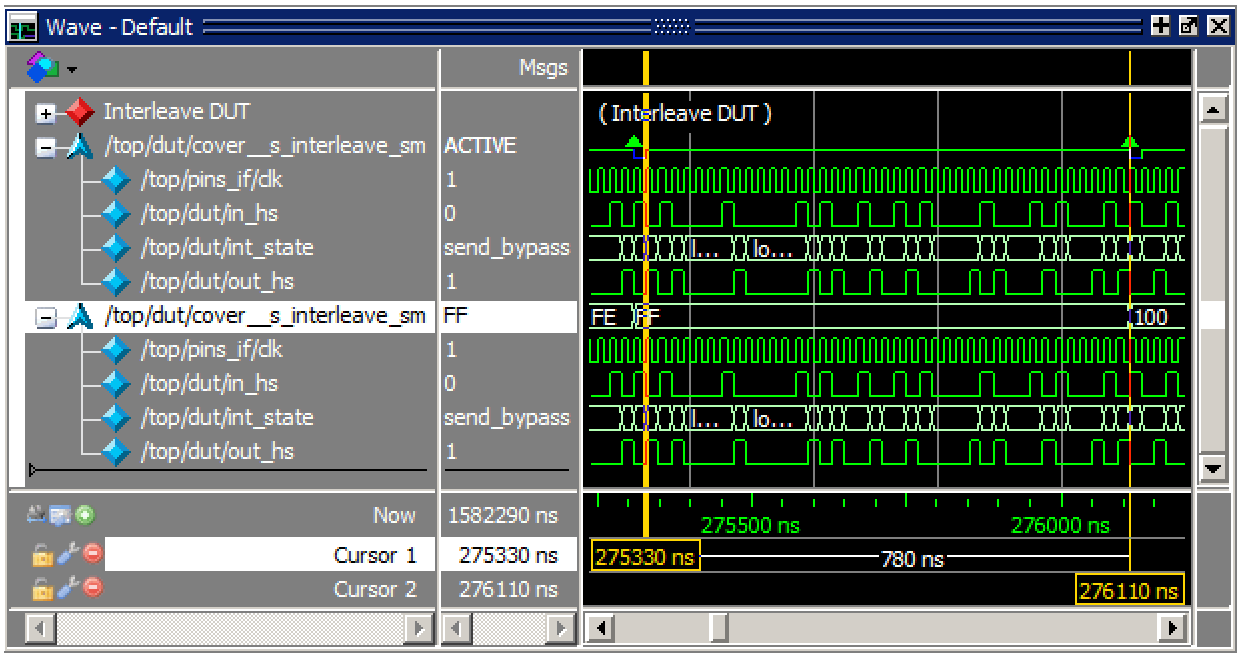


Figure 15-32. Second Temporal and Count Mode Views of Cover Directive



Creating Functional Coverage Reports

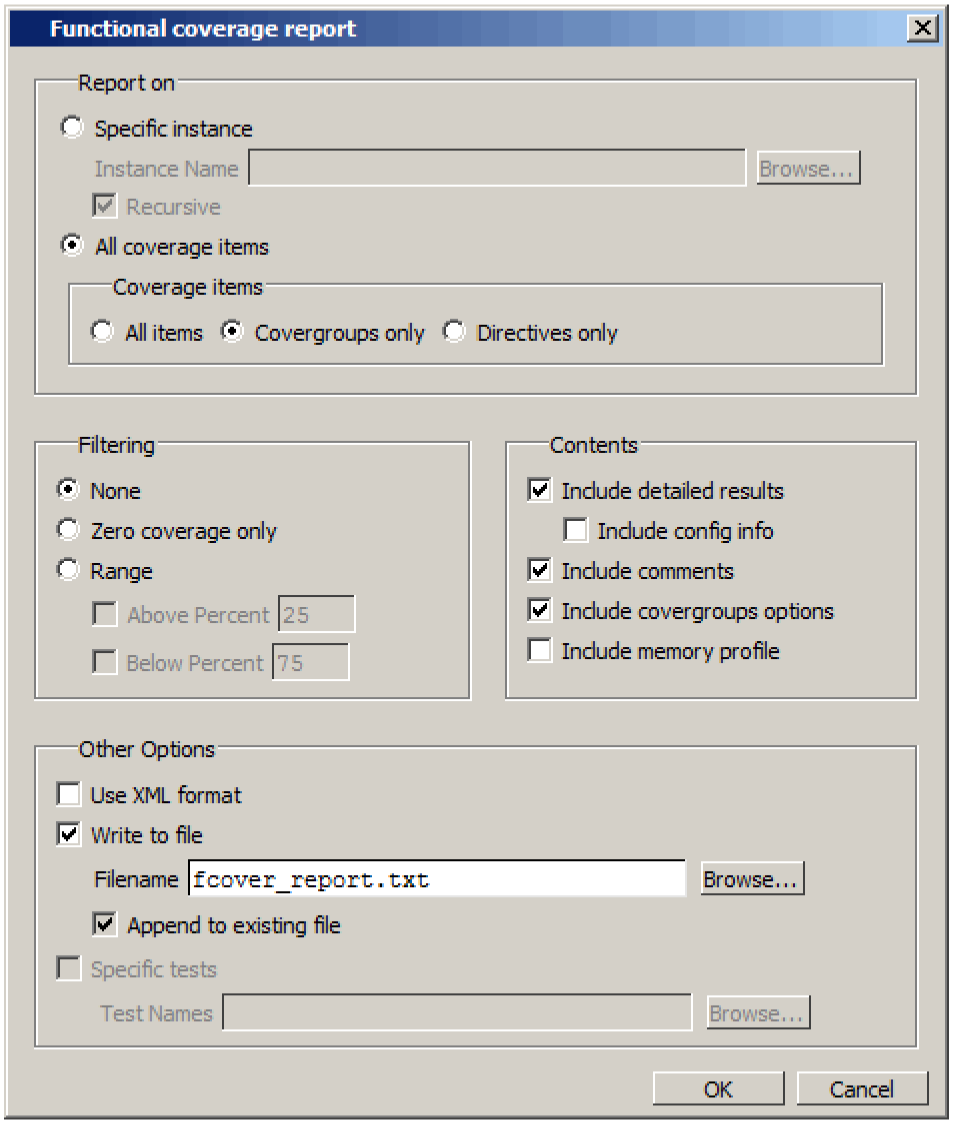
You can create functional coverage reports using dialog boxes accessible through the GUI or via commands entered at the command line prompt.

Procedure

1. Create a functional coverage report using the GUI.

   a. Right-click in the Cover Directives window and select **Report**. This opens the Functional coverage report dialog box (Figure 15-33).

Figure 15-33. Functional Coverage Report Dialog Box



Functional coverage report

Report on
- Specific instance
  - Instance Name:
  - [✓ Recursive]
- [✓ All coverage items]
  - Coverage items: All items [✓ Covergroups only] Directives only

Filtering
- [✓ None]
- Zero coverage only
- Range
  - Above Percent 25
  - Below Percent 75

Contents
- [✓ Include detailed results]
  - Include config info
- [✓ Include comments]
- [✓ Include covergroups options]
- Include memory profile

Other Options
- Use XML format
- [✓ Write to file]
  - Filename: fcover_report.txt
  - [✓ Append to existing file]
- Specific tests
  - Test Names:

OK Cancel

   b. With "All coverage items" selected, select **Covergroups only**.

   c. Select **Include covergroup options**.

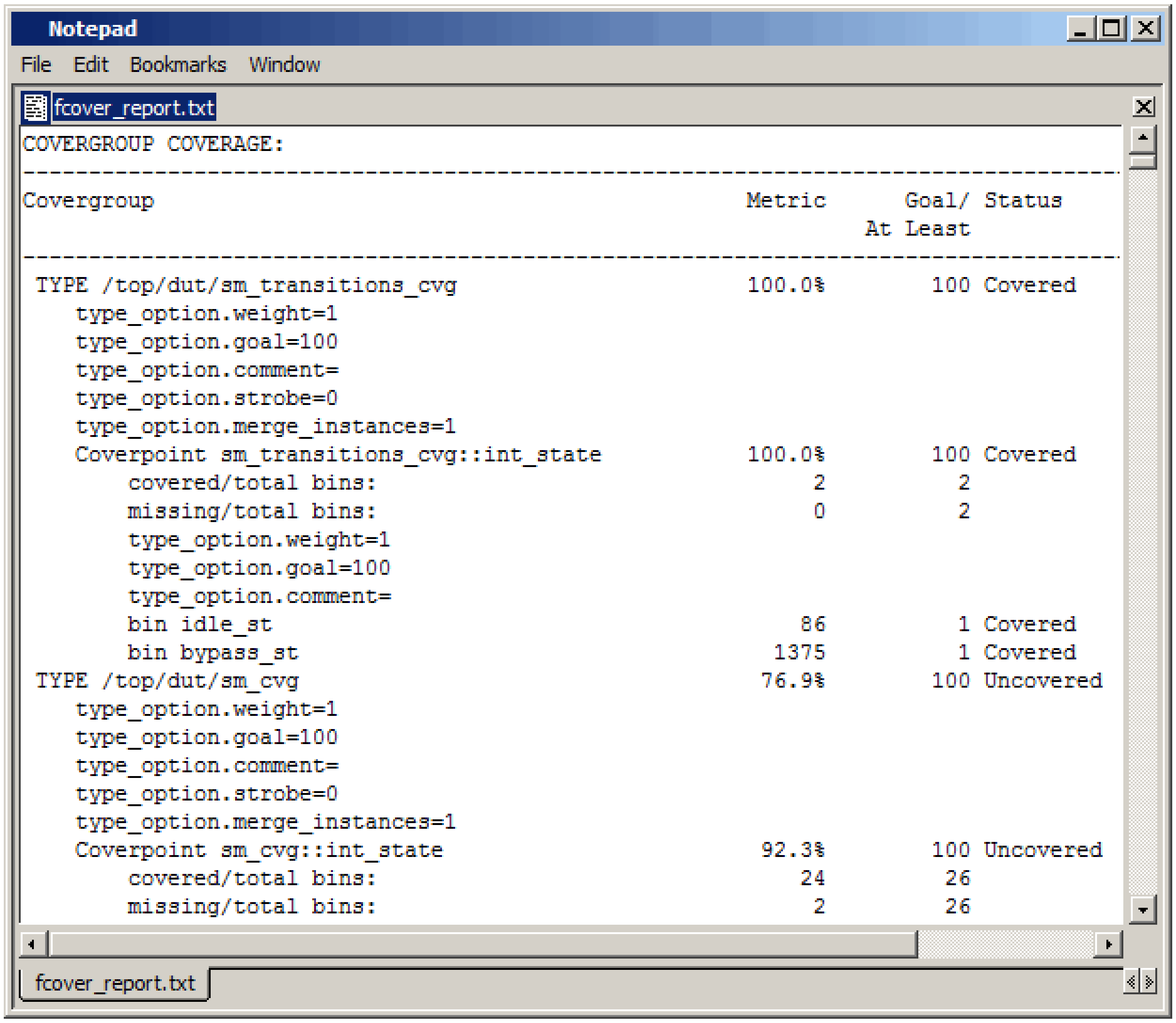
   d. Select **OK** to write the report to the file *fcover_report.txt*.

   The actions taken in the GUI are echoed in the transcript as follows:

```
coverage report -detail -cvg -comments -option
                -file fcover_report.txt -r /
```

The report will appear automatically in Questa SIM Notepad as shown in (Figure 15-34).

Figure 15-34. The Functional Coverage Report



```
COVERGROUP COVERAGE:
----------------------------------------------------------------------------------
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=
    type_option.strobe=0
    type_option.merge_instances=1
    Coverpoint sm_transitions_cvg::int_state           100.0%         100 Covered
        covered/total bins:                                 2           2
        missing/total bins:                                 0           2
        type_option.weight=1
        type_option.goal=100
        type_option.comment=
        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=
    type_option.strobe=0
    type_option.merge_instances=1
    Coverpoint sm_cvg::int_state                        92.3%         100 Uncovered
        covered/total bins:                                24          26
        missing/total bins:                                 2          26
```

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 Questa SIM.

Chapter 16
Using the SystemVerilog DPI

This lesson is designed to walk you through the basics of using the SystemVerilog Direct Programming Interface (DPI) with Questa SIM. 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. 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 the GUI 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, moving back and forth between the two languages.

Design Files for this Lesson

The Questa SIM installation comes with the design files you need to complete this lesson.

The files arelocated 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 Topics

User's Manual Appendix: Verilog Interfaces to C

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.

If you open the code for module *test.sv* in a text editor it should look like the code in Figure 16-1.

Figure 16-1. 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.

If you open the *foreign.c* source file in a text editor it should look like the code in Figure 16-2.

Figure 16-2. 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 16-3). 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 16-3. 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 16-4).

Figure 16-4. 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 16-5).

Figure 16-5. 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 16-6). 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 16-6. 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
```

Exploring the Makefile

A *Makefile* has been included with this lesson to help UNIX and Linux users compile and simulate the design, 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 16-7. 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$(QUESTA_HOME)/include -shared -g -o foreign.so foreign.c
 9
10 foreign_32:  foreign.c
11    gcc -I$(QUESTA_HOME)/include -shared -fPIC -m32 -g -o foreign.so
foreign.c
12
13 optimize:
14    vopt +acc test -o opt_test
15
16 foreign_windows: foreign.c
17    vsim -c opt_test -dpiexportobj exports
18    gcc -I$(QUESTA_HOME)/include -shared -g -o foreign.dll foreign.c
eports.obj -lmtipli -L$(QUESTA_HOME)/win32
19
20 sim:
21    vsim opt_test -sv_lib foreign
22
23 all:
24    worklib compile foreign optimize sim
25
26 all_windows:
27    worklib compile optimize foreign_windows sim
28
29 clean:
30    rm -rf work transcript vsim.wlf foreign.so foreign.dll exports.obj
31
```

The five targets in the *Makefile* are:

Line 1-2 – The vlib command creates the *work* library where the compiled files will be located.

Lines 4-5 – The vlog command invokes the vlog compiler on the *test.sv* source file.

Lines 7-11 and 16-18 – 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.

Lines 13-14 – 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 20 - 21 – 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.

Exploring the *windows.bat* File

A *windows.bat* file has been included for Windows users.

Figure 16-8. 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 %QUESTA_HOME%\include -shared -g -o foreign.dll foreign.c
exports.obj -lmtipli -L %QUESTA_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*.

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.

Compile and Load the Simulation

To start this lesson you must set your enivronment variable, then compile and load the design. In this case, compile and load is performed in a single step

Procedure

1. Create a new directory and copy into it all files from:
   <install_dir>/questasim/examples/tutorials/systemverilog/dpi_basic
2. Change directory to this new directory and make sure your QuestaSim environment is set up properly.
   a. Set the QUESTA_HOME environment variable to the Questa SIM installation directory.
   b. For Windows users, if you do not have the gcc-4.2.1-mingw32vc9 compiler installed, download it from SupportNet (http://supportnet.mentor.com/) and unzip it into the Questa SIM install tree. In addition, set your Path environment variable to C:\<install_directory>\gcc-4.2.1-mingw32vc9\bin.
3. **UNIX and Linux:** Use the **make** utility to compile and load the design into the simulator.

   Windows: Double-click the *windows.bat* file to compile and load the design, or use the Questa SIM icon to open the program, then enter the following commands in the Transcript window:

```
vlib work
vlog test.sv -dpiheader dpi_types.h foreign.c
vopt +acc test -o opt_test
vsim -i opt_test -do "add wave light; view source"
```

Run the Simulation

Now we can run the simulation to see what happens to the “light” object in the Objects window.

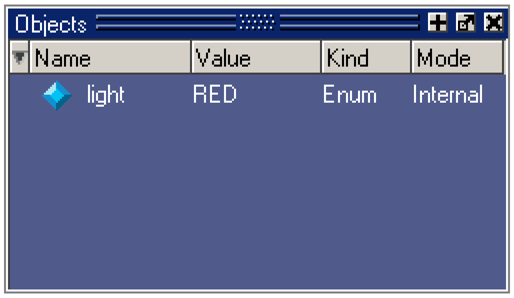
Procedure

1. **UNIX and Linux:** Drag and drop the *light* object into a Wave window.

   Windows: The *light* object has already been placed in the Wave window.

2. Once in simulation mode, 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 window in the Questa SIM graphic interface (Figure 16-9), 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 16-9. The light Signal in the Objects Window



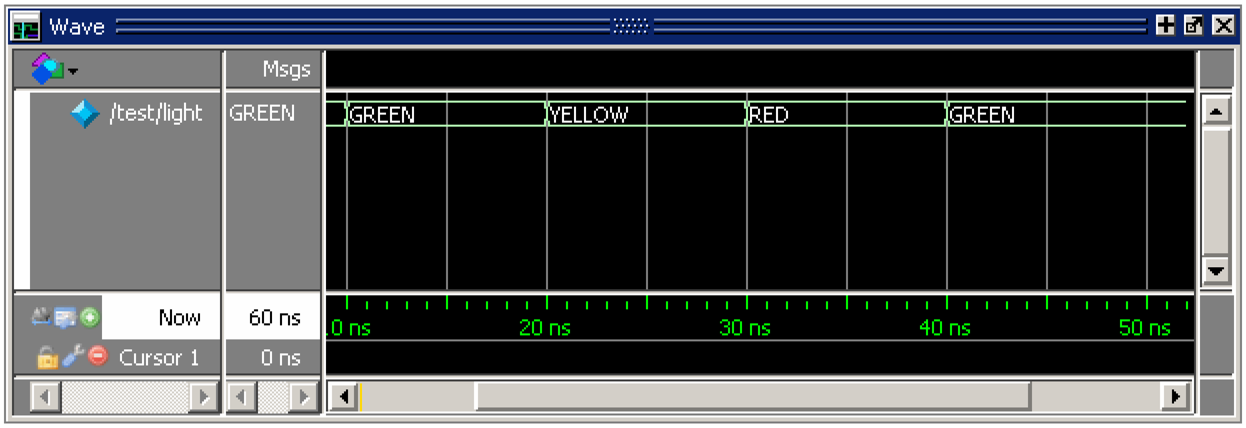
Name	Value	Kind	Mode
light	RED	Enum	Internal

3. 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 16-10).

Figure 16-10. The light Signal in the Wave Window



4. Restart the simulation.

   a. Click the Restart icon.

   b. In the Restart dialog box, click the **OK** button.

5. Run the simulation for 10 ns.

   a. Enter **run 10 ns** at the command line.

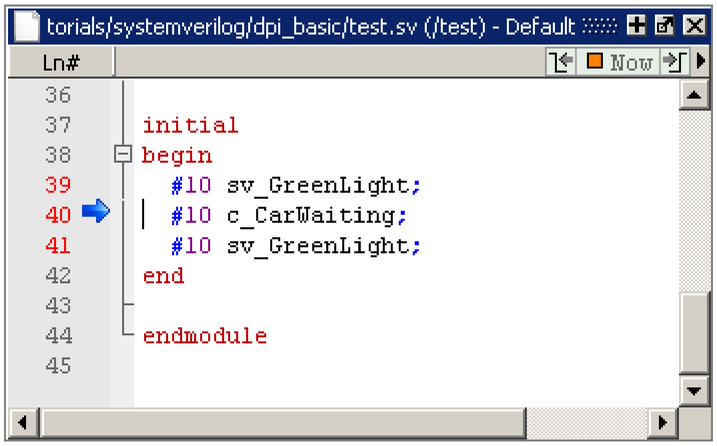
6. View the *test.sv* code in the Source window.

a. Select the **test.sv** tab.

7. Step through the code.

a. Click the Step Into icon and watch the blue arrow in the Source window move through the code for *test.sv* (Figure 16-11) 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 16-11. Source Code for test.sv



```
initial
begin
  #10 sv_GreenLight;
  #10 c_CarWaiting;
  #10 sv_GreenLight;
end

endmodule
```

Lesson Wrap-Up

This concludes this lesson on the basics of how DPI works in Questa SIM. 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 17
Using SystemVerilog DPI for Data Passing

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 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.

Prerequisites

1. Set the QUESTA_HOME environment variable to the Questa SIM installation directory.
2. For Windows users, if you do not have the gcc-4.2.1-mingw32vc9 compiler installed, download it from SupportNet (http://supportnet.mentor.com/) and unzip it into the Questa SIM install tree. In addition, set your Path environment variable to C:\<install_directory>\gcc-4.2.1-mingw32vc9\bin.

Related Topics

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.

If you open the code for the *foreign.c* file in a text editor it should look like the code in Figure 17-1.

Figure 17-1. 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.

Now let’s look at the SystemVerilog source code. If you open the *test.sv* source file in a text editor it should look like the code in Figure 17-2.

Figure 17-2. 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.

Explore the Makefile

A *Makefile* has been included with this lesson to help UNIX and Linux users compile and simulate the design, 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 17-3. 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$(QUESTA_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.

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 is included for Windows users.

Figure 17-4. 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 %QUESTA_HOME%\include -shared -g -o foreign.dll foreign.c -lmtipli -L
%QUESTA_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.

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.

Compile and Load the Simulation

We will compile the design and load the simulator using a makefile for Unix and Linux installations, and a *windows.bat* file for Windows installations.

Procedure

1. Create a new directory and copy into it all files from:
   <install_dir>/questasim/examples/tutorials/systemverilog/data_passing
2. Change directory to this new directory and make sure your QuestaSim environment is set up properly.
3. **UNIX and Linux:** Use the **make** utility to compile and load the design into the simulator.

   Windows: Double-click the *windows.bat* file, or use the Questa SIM icon to open the program, then enter the following commands in the Transcript window:

   ```
   vlib work
   vlog test.sv -dpiheader dpi_types.h foreign.c
   vopt +acc test -o opt_test
   vsim -i opt_test -do "view source"
   radix -symbolic
   ```

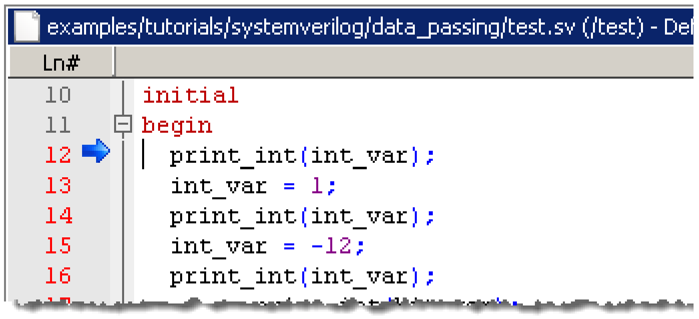
Run the Simulation

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.

Procedure

1. Right-click the *test* instance in the Structure (sim) window and select View Declaration from the popup menu that appears. This will open a Source window and display the *test.sv* source code.
2. 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 17-5) where we print out the value of *int_var* – which is defined as an *int* on line #6.

Figure 17-5. Line 12 of test.sv in the Source Window

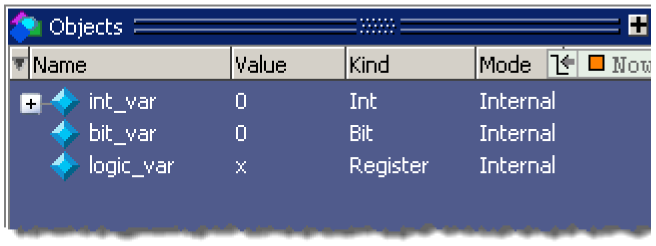


examples/tutorials/systemverilog/data_passing/test.sv (/test) - Def

```
Ln#
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);
```

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 17-6).

Figure 17-6. The Value of int_var is Currently 0



Objects

Name	Value	Kind	Mode
int_var	0	Int	Internal
bit_var	0	Bit	Internal
logic_var	x	Register	Internal

3. 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:

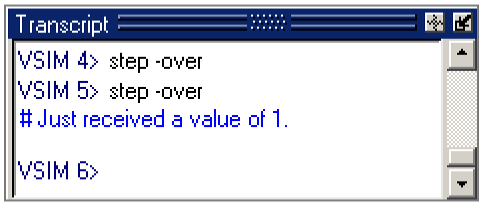
   ```
   Just received a value of 0.
   ```

   That's what we expect to happen. So far, so good.

4. 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.

5. Now do another Step Over and you should see a 1 being printed in the Transcript window this time (Figure 17-7).

Figure 17-7. The Value of int_var Printed to the Transcript Window



Transcript

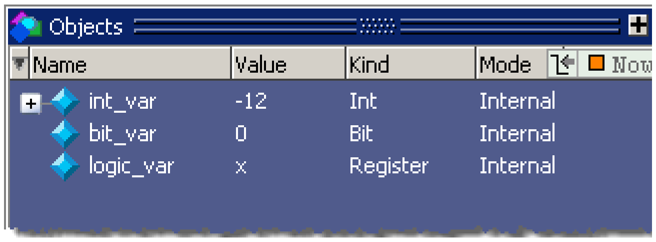
```
VSIM 4> step -over
VSIM 5> step -over
# Just received a value of 1.

VSIM 6>
```

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.

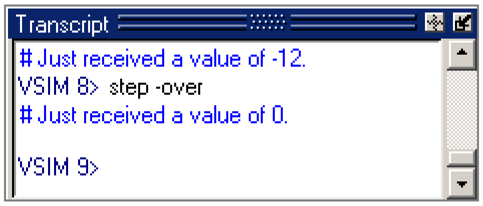
7. Click Step Over again and verify the results in the Objects window (Figure 17-8) and in the Transcript window (Figure 17-9).

Figure 17-8. 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 17-9. Transcript Shows the Value Returned for bit_var



```
# Just received a value of -12.
VSIM 8> step -over
# Just received a value of 0.

VSIM 9>
```

8. Click Step Over twice to set *bit_var* to a 1 and print to the transcript.

9. Click Step Over once 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 17-1, 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, Questa SIM 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 17-3) you will see an option in the vlog command called **-dpiheader** with an output file name as its argument. As the compiler 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 17-10).

Figure 17-10. The dpi_types.h File

```
/* MTI_DPI */

/*
 * Copyright 2004 Mentor Graphics Corporation.
 *
 * Note:
 *   This file is automatically generated.
 *   Please do not edit this file - you will lose your edits.
 *
 * Settings when this file was generated:
 *   PLATFORM = 'win32'
 *       Info = SE 6.1c 2005.11
*/
#ifndef INCLUDED_DPI_TYPES
#define INCLUDED_DPI_TYPES

#ifdef __cplusplus
extern "C" {
#endif

#include "svdpi.h"

DPI_DLLESPEC
void
print_int(
    int int_in);

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 Questa SIM installation directory (that's why we have "-I$(QUESTA_HOME)/include" on the command line for C compilation in the Makefile's "foreign" target – see Figure 17-3). This svLogic type is basically an unsigned char.

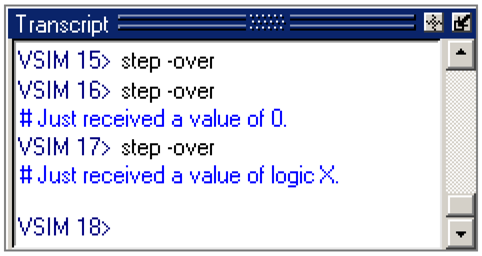
When you put *#include dpi_types.h* in your C source file, all these function prototypes and data types will be made available to you. In fact, we strongly recommend that you use this file when writing the C code that will interface with Verilog via DPI.

Look back at the *test.sv* file (Figure 17-2) 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 17-11). You can take a look at the *foreign.c* file to see how this was accomplished.

Figure 17-11. The Transcript Shows the Correct Value of logic X



```
VSIM 15> step -over
VSIM 16> step -over
# Just received a value of 0.
VSIM 17> step -over
# Just received a value of logic X.

VSIM 18>
```

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 18
Comparing Waveforms

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.

Design Files for this Lesson

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

The Questa SIM 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 Topics

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.

Procedure

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/compare* to the new directory.

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

2. Start Questa SIM and change to the exercise directory.

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

   a. Type **vsim** at a UNIX shell prompt or use the Questa SIM icon in Windows.

      If the Welcome to Questa SIM dialog box appears, click **Close**.

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

3. Execute the following commands:

   - Verilog

     ```
     vlib work
     vlog *.v
     vopt +acc test_sm -o opt_test_gold
     vsim -wlf gold.wlf opt_test_gold
     add wave *
     run 750 ns
     quit -sim
     ```

   - VHDL

     ```
     vlib work
     vcom -93 sm.vhd sm_seq.vhd sm_sram.vhd test_sm.vhd
     vopt +acc test_sm -o opt_test_gold
     vsim -wlf gold.wlf opt_test_gold
     add wave *
     run 750 ns
     quit -sim
     ```

   The **-wlf** switch is used with the vsim command to create the reference dataset called *gold.wlf*.

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 test bench to create differences between the two runs.

Procedure

Verilog

1. Edit the test bench.
   a. Select **File** > **Open** and open *test_sm.v*.
   b. Right-click in the Source window for the *test_sm.v* file and uncheck the **Read Only** selection in the popup menu.
   c. Scroll to line 122, which looks like this:
   ```
   @ (posedge clk) wt_wd('h10,'haa);
   ```
   d. Change the data pattern ’haa’ to ’hab’:
   ```
   @ (posedge clk) wt_wd('h10,'hab);
   ```
   e. Select **File** > **Save** to save the file.
2. Compile the revised file and rerun the simulation.
   ```
   vlog test_sm.v
   vopt +acc test_sm -o opt_test_sm
   vsim opt_test_sm
   add wave *
   run 750 ns
   ```

VHDL

1. Edit the test bench.
   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.
   - VHDL

```
vcom test_sm.vhd
vopt +acc test_sm -o opt_test_sm
vsim opt_test_sm
add wave *
run 750 ns
```

Comparing the Simulation Runs

Questa SIM includes a Comparison Wizard that walks you through the process. You can also configure the comparison manually with menu or command line commands.

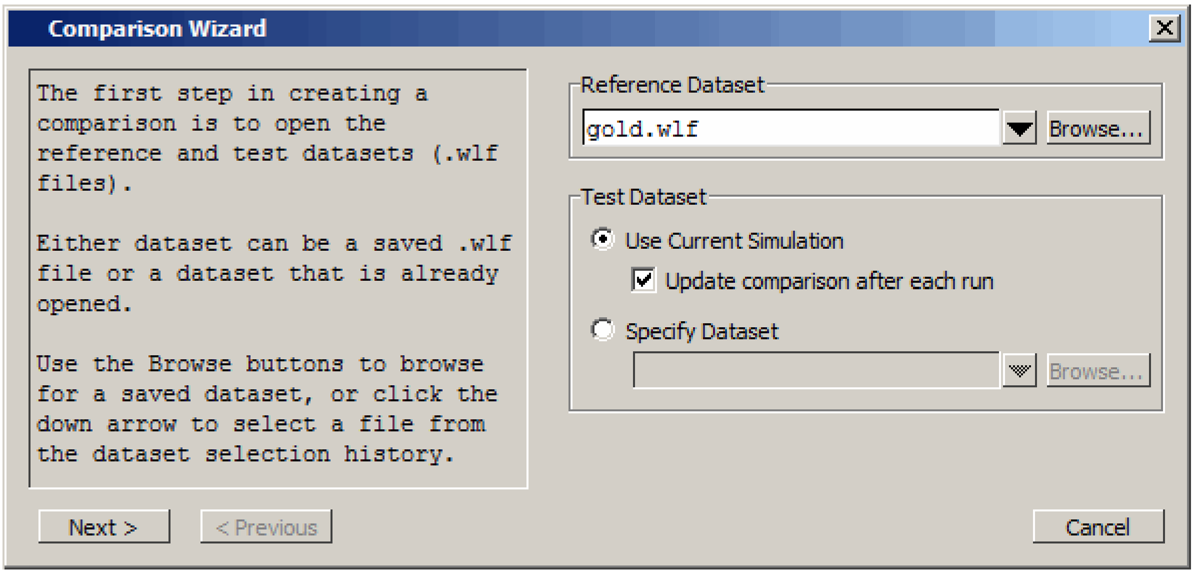
Procedure

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 18-1). Recall that *gold.wlf* is from the first simulation run.

Figure 18-1. First Dialog Box of the Waveform Comparison Wizard



```
The first step in creating a
comparison is to open the
reference and test datasets (.wlf
files).

Either dataset can be a saved .wlf
file or a dataset that is already
opened.

Use the Browse buttons to browse
for a saved dataset, or click the
down arrow to select a file from
the dataset selection history.
```

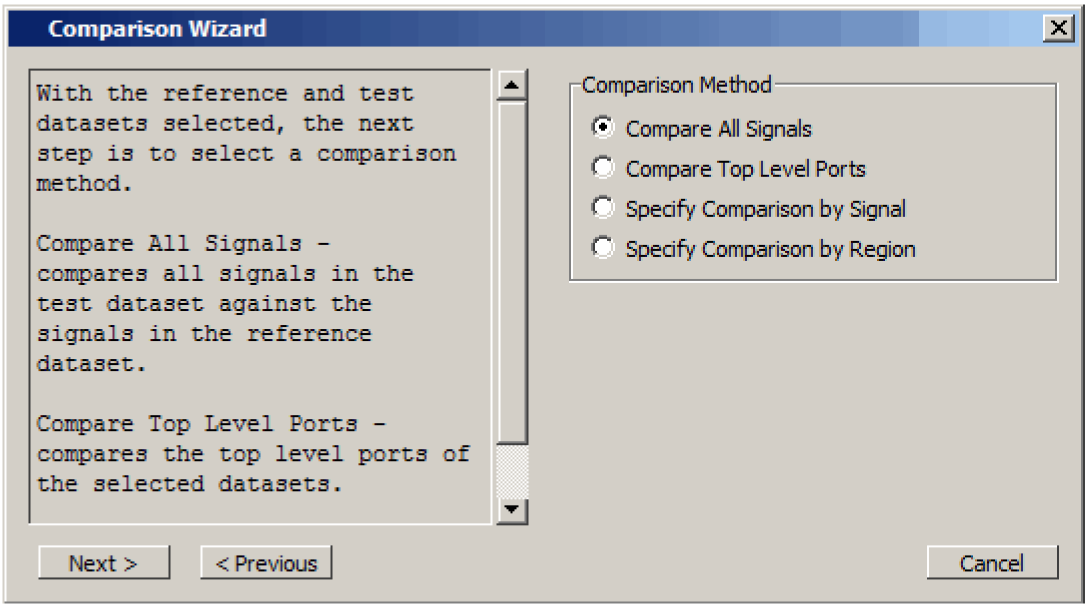
Reference Dataset: gold.wlf

Test Dataset: [● Use Current Simulation] ☑ Update comparison after each run; ○ Specify Dataset

   c. Leaving the test dataset set to **Use Current Simulation**, click **Next**.

   d. Select **Compare All Signals** in the second dialog box (Figure 18-2) and click **Next**.

Figure 18-2. Second Dialog Box of the Waveform Comparison Wizard


Comparison Wizard

```
With the reference and test
datasets selected, the next
step is to select a comparison
method.

Compare All Signals -
compares all signals in the
test dataset against the
signals in the reference
dataset.

Compare Top Level Ports -
compares the top level ports of
the selected datasets.
```

Comparison Method
- [✓ Compare All Signals]
- Compare Top Level Ports
- Specify Comparison by Signal
- Specify Comparison by Region

e. In the next three dialog boxes, click **Next**, **Compute Differences Now**, and **Finish**, respectively.

   Questa SIM performs the comparison and displays the compared signals in the Wave window.

Viewing Comparison Data

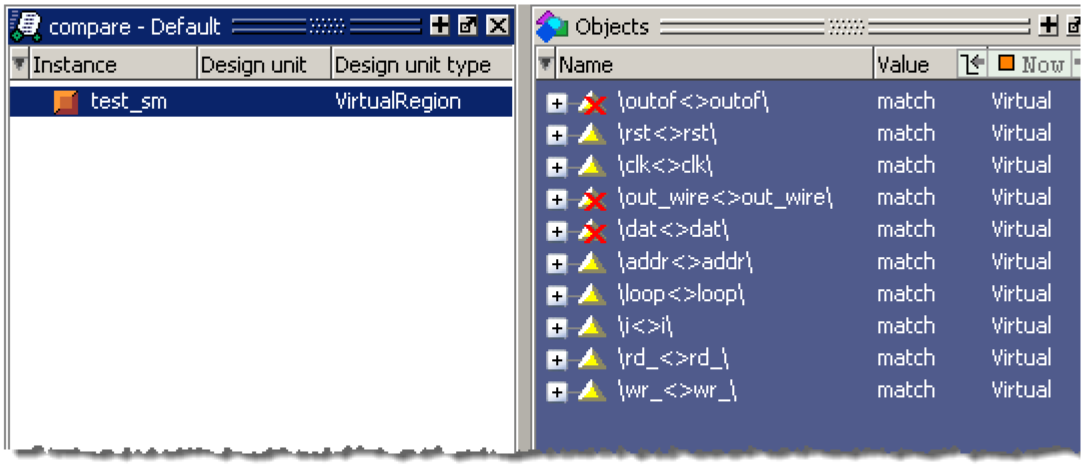
Comparison data is displayed in the Structure (compare), Transcript, Objects, Wave and List windows. Compare objects are denoted by a yellow triangle.

The Compare window shows the region that was compared.

The Transcript window shows the number of differences found between the reference and test datasets.

The Objects window shows comparison differences when you select the comparison object in the Structure (compare) window (Figure 18-3).

Figure 18-3. Comparison information in the compare and Objects windows



compare - Default

Instance	Design unit	Design unit type
test_sm		VirtualRegion

Objects

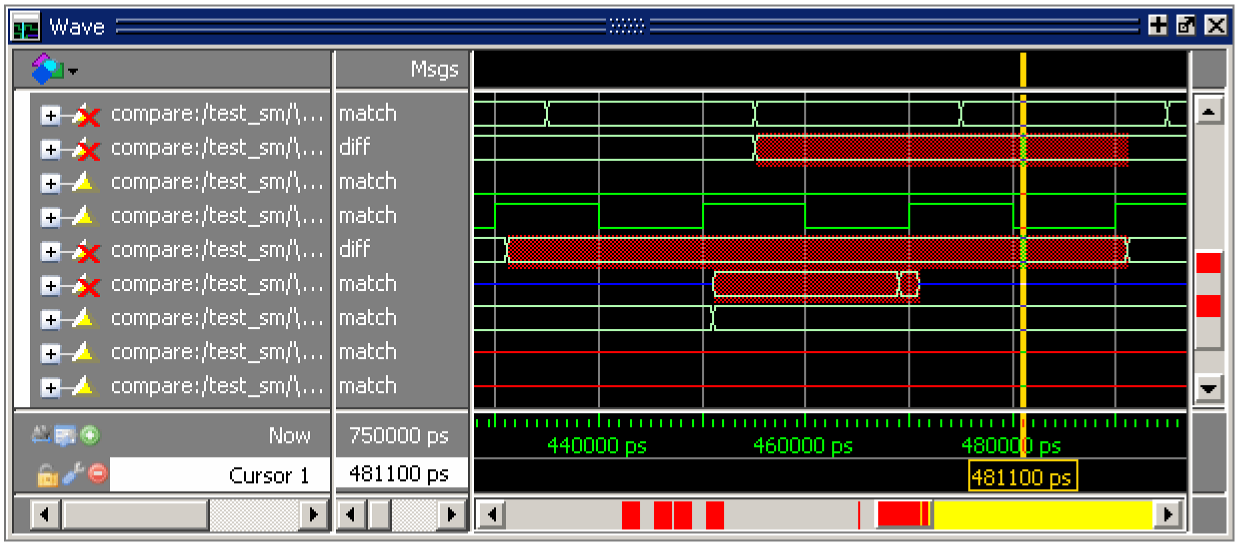
Name	Value	Now
\outof<>outof\	match	Virtual
\rst<>rst\	match	Virtual
\clk<>clk\	match	Virtual
\out_wire<>out_wire\	match	Virtual
\dat<>dat\	match	Virtual
\addr<>addr\	match	Virtual
\loop<>loop\	match	Virtual
\i<>i\	match	Virtual
\rd_<>rd_\	match	Virtual
\wr_<>wr_\	match	Virtual

Comparison Data in the Wave Window

The Wave window displays comparison information in a clear graphic format.

- timing differences are denoted by a red X's in the pathnames column (Figure 18-4),

Figure 18-4. Comparison objects in the Wave window



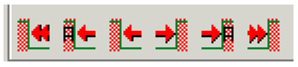
Wave

	Msgs
compare:/test_sm/\...	match
compare:/test_sm/\...	diff
compare:/test_sm/\...	match
compare:/test_sm/\...	match
compare:/test_sm/\...	diff
compare:/test_sm/\...	match
compare:/test_sm/\...	match
compare:/test_sm/\...	match
compare:/test_sm/\...	match
Now	750000 ps
Cursor 1	481100 ps

- 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 18-5).

Figure 18-5. The compare icons



From left to right, the buttons 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>.

Viewing Comparison Data in the List Window

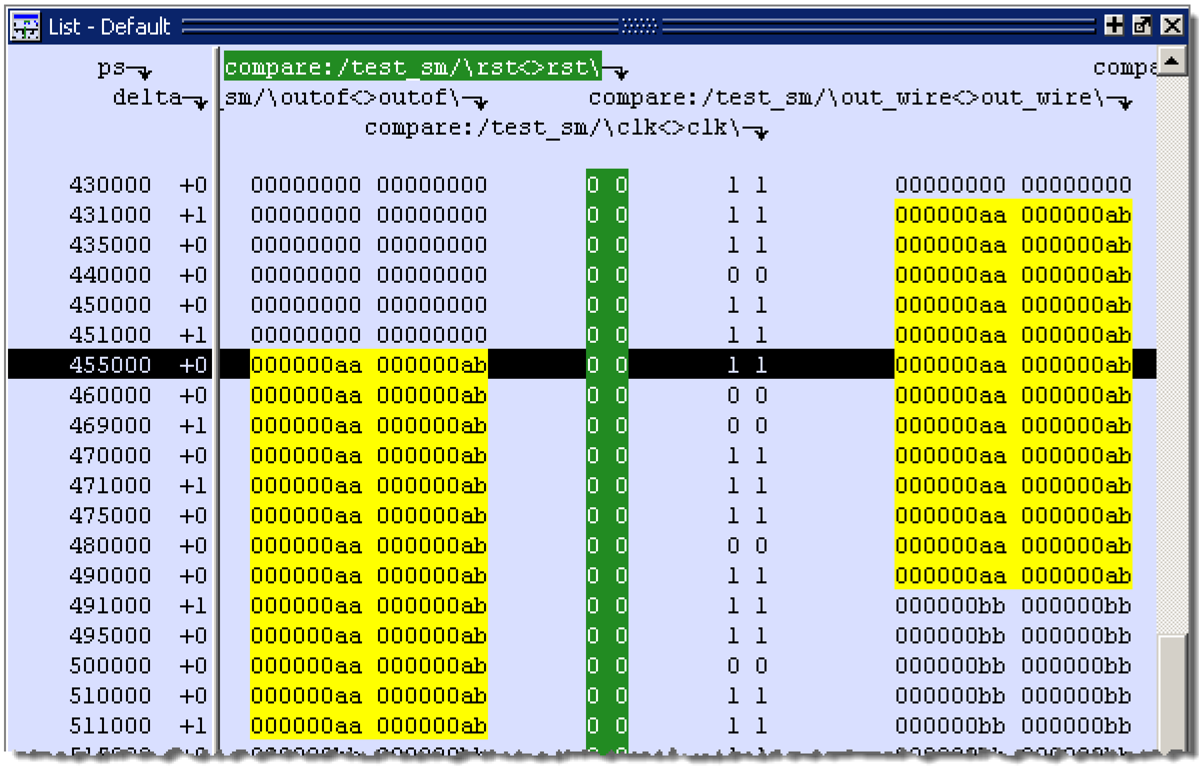
You can also view the results of your waveform comparison in the List window.

Procedure

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 18-6). Differences that have been annotated have red highlighting.

Figure 18-6. Compare differences in the List window



```
        ps      compare:/test_sm/\rst<>rst\                                    compa
     delta     sm/\outof<>outof\          compare:/test_sm/\out_wire<>out_wire\
                          compare:/test_sm/\clk<>clk\

 430000  +0   00000000 00000000      0 0      1 1      00000000 00000000
 431000  +1   00000000 00000000      0 0      1 1      000000aa 000000ab
 435000  +0   00000000 00000000      0 0      1 1      000000aa 000000ab
 440000  +0   00000000 00000000      0 0      0 0      000000aa 000000ab
 450000  +0   00000000 00000000      0 0      1 1      000000aa 000000ab
 451000  +1   00000000 00000000      0 0      1 1      000000aa 000000ab
 455000  +0   000000aa 000000ab      0 0      1 1      000000aa 000000ab
 460000  +0   000000aa 000000ab      0 0      0 0      000000aa 000000ab
 469000  +1   000000aa 000000ab      0 0      0 0      000000aa 000000ab
 470000  +0   000000aa 000000ab      0 0      1 1      000000aa 000000ab
 471000  +1   000000aa 000000ab      0 0      1 1      000000aa 000000ab
 475000  +0   000000aa 000000ab      0 0      1 1      000000aa 000000ab
 480000  +0   000000aa 000000ab      0 0      0 0      000000aa 000000ab
 490000  +0   000000aa 000000ab      0 0      1 1      000000aa 000000ab
 491000  +1   000000aa 000000ab      0 0      1 1      000000bb 000000bb
 495000  +0   000000aa 000000ab      0 0      1 1      000000bb 000000bb
 500000  +0   000000aa 000000ab      0 0      0 0      000000bb 000000bb
 510000  +0   000000aa 000000ab      0 0      1 1      000000bb 000000bb
 511000  +1   000000aa 000000ab      0 0      1 1      000000bb 000000bb
```

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 Questa SIM.

To save comparison data so it can be reloaded into Questa SIM, 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.

Procedure

1. Save the comparison data to a text file.

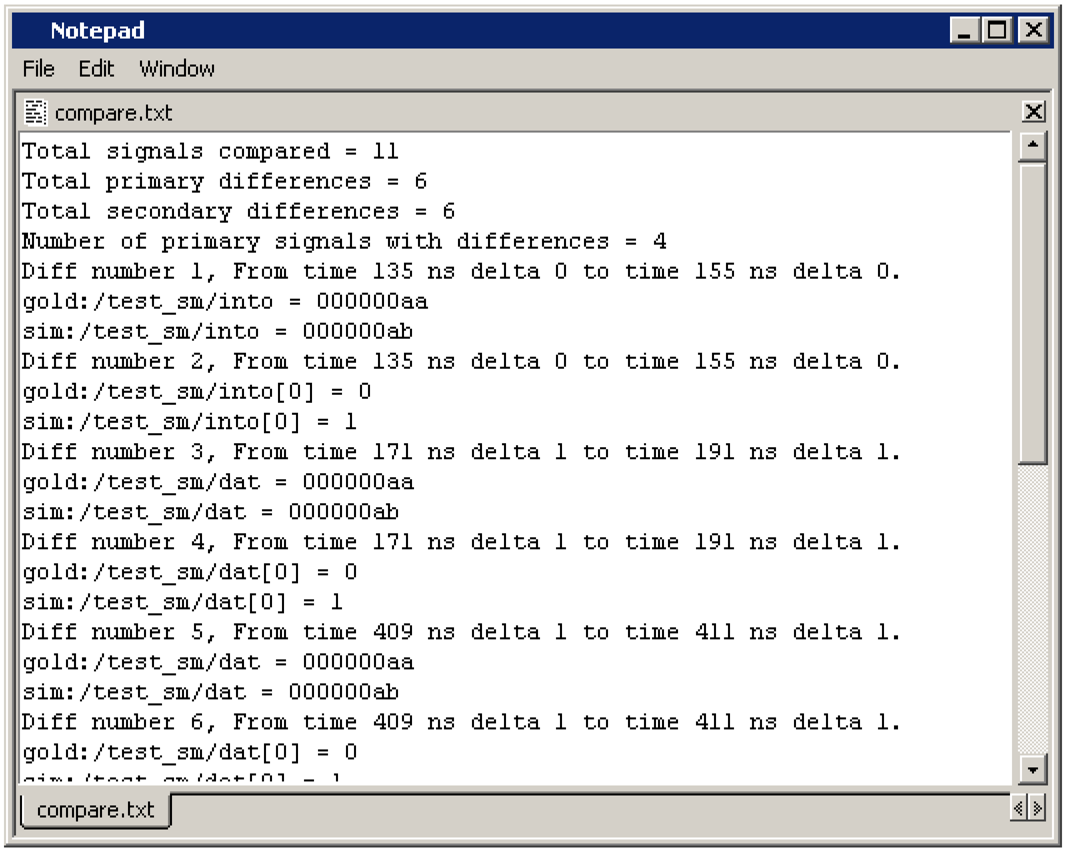
   a. In the Main window, select **Tools > Waveform Compare > Differences > Write Report**.

   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 18-7).

Figure 18-7. Coverage data saved to a text file



```
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 = 000000aa
sim:/test_sm/into = 000000ab
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 = 000000aa
sim:/test_sm/dat = 000000ab
Diff number 4, From time 171 ns delta 1 to time 191 ns delta 1.
gold:/test_sm/dat[0] = 0
sim:/test_sm/dat[0] = 1
Diff number 5, From time 409 ns delta 1 to time 411 ns delta 1.
gold:/test_sm/dat = 000000aa
sim:/test_sm/dat = 000000ab
Diff number 6, From time 409 ns delta 1 to time 411 ns delta 1.
gold:/test_sm/dat[0] = 0
```

d. Close Notepad when you have finished viewing the report.

2. Save the comparison data in files that can be reloaded into Questa SIM.

   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.

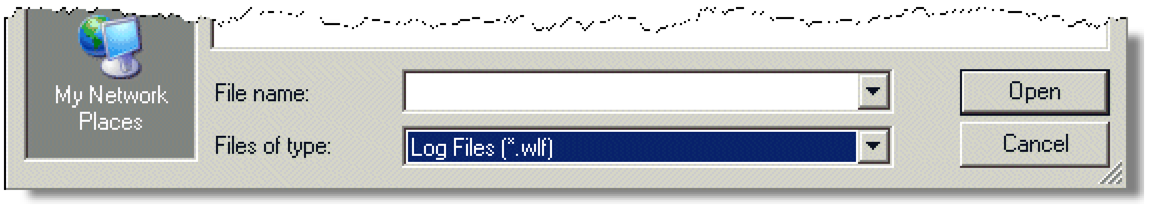
   e. Select **Tools** > **Waveform Compare** > **End Comparison**.

3. Reload the comparison data.

   a. With the Structure (sim) window active, select **File** > **Open**.

   b. Change the **Files of Type** to Log Files (*.wlf) (Figure 18-8).

Figure 18-8. Displaying Log Files in the Open Dialog Box



File name:

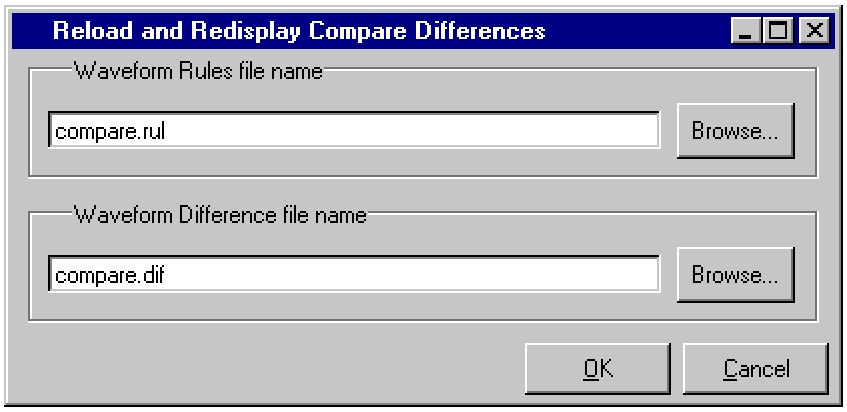
Files of type: Log Files (*.wlf)

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 box should already have the correct Waveform Rules and Waveform Difference files specified (Figure 18-9).

Figure 18-9. Reloading Saved Comparison Data



Waveform Rules file name: compare.rul

Waveform Difference file name: compare.dif

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 Questa SIM> prompt.

Chapter 19
Automating Simulation

Aside from executing a couple of pre-existing DO files, the previous lessons focused on using Questa SIM 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 Questa SIM 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*>/<*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.

Creating a Simple DO File

Creating a DO file is as simple as typing a set of commands in a text file. In this exercise, you will create a DO file that loads a design, adds signals to the Wave window, provides stimulus to those signals, and then advances the simulation. You can also create a DO file from a saved transcript file.

Refer to "Saving a Transcript File as a DO file."

Procedure

1. Change to the directory you created in the "Basic Simulation" lesson.
2. Create a DO file that will 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:

```
vsim testcounter_opt
```

```
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**.

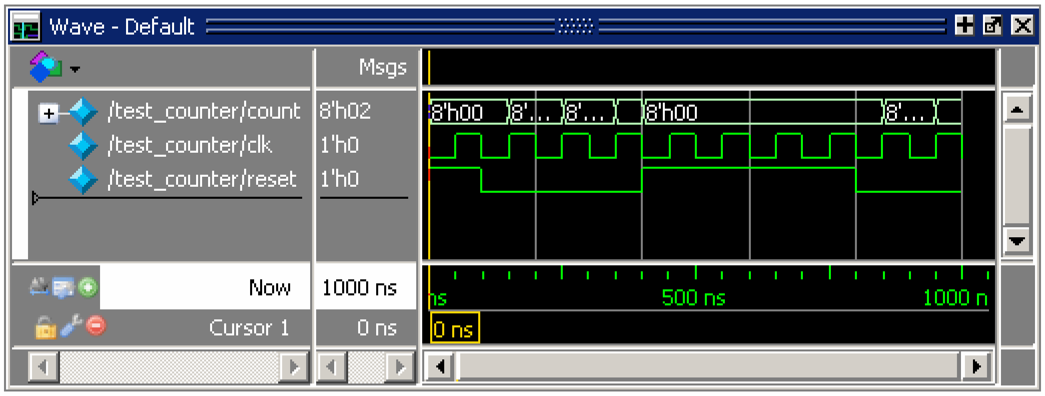
   b. Type **sim.do** in the File name: field and save it to the current directory.

4. Execute the DO file.

   a. Enter **do sim.do** at the VSIM> prompt.

   Questa SIM loads the design, executes the saved commands and draws the waves in the Wave window. (Figure 19-1)

Figure 19-1. Wave Window After Running the DO File



5. When you are done with this exercise, select **File** > **Quit** to quit Questa SIM.

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 Questa SIM 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 Questa SIM commands and specify that file when you invoke the simulator.

Procedure

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 *

   #change radix to symbolic
   radix -symbolic

   # 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. Optimize the counter design unit.

   a. Enter the following command at the DOS/UNIX prompt:

   vopt +acc counter -o counter_opt

5. Run the command line mode simulation.

a. Enter the following command at the DOS/UNIX prompt:

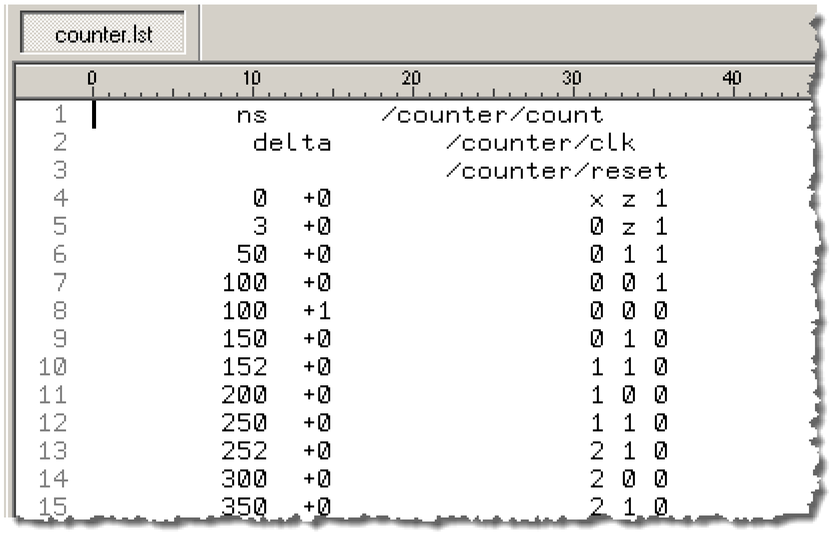
vsim -c -do sim.do counter_opt -wlf counter_opt.wlf

The **-c** argument instructs Questa SIM 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.

6. 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 Figure 19-2:

Figure 19-2. Output of the Counter



```
 1             ns        /counter/count
 2               delta         /counter/clk
 3                             /counter/reset
 4              0   +0                   x  z  1
 5              3   +0                   0  z  1
 6             50   +0                   0  1  1
 7            100   +0                   0  0  1
 8            100   +1                   0  0  0
 9            150   +0                   0  1  0
10            152   +0                   1  1  0
11            200   +0                   1  0  0
12            250   +0                   1  1  0
13            252   +0                   2  1  0
14            300   +0                   2  0  0
15            350   +0                   2  1  0
```

The output may appear slightly different if you used the VHDL version.

7. View the results in the GUI.

Since you saved the simulation results in *counter_opt.wlf*, you can view them in the GUI by invoking VSIM with the **-view** argument.

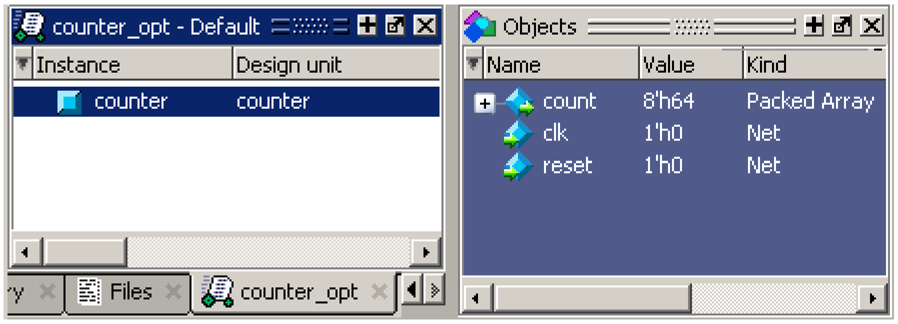
Note

Make sure your PATH environment variable is set with the current version of Questa SIM at the front of the string.

a. Type **vsim -view counter_opt.wlf** at the prompt.

The GUI opens and a dataset tab named "counter_opt" is displayed as in Figure 19-3. (Select **View** > **Objects** in the main menus if you do not see an Objects window.)

Figure 19-3. The counter_opt.wlf Dataset in the Main Window Workspace



b. Right-click the *counter* instance and select **Add Wave**.

The waveforms display in the Wave window.

8. When you finish viewing the results, select **File** > **Quit** to close Questa SIM.

Using Tcl with the Simulator

The DO files used in previous exercises contained only Questa SIM 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 called bookmarks.

Procedure

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 - 100] [expr $stime + 50]" 0
 add button "$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 100 time units to the current simulation time plus 50 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]== "8'h27"} {
      add_wave_zoom $now 1
   } elseif {[examine count]== "8'h47"} {
      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*" into the directory you created in the Basic Simulation lesson.

2. Load the *test_counter* design unit and make sure the radix is set to binary.

a. Start Questa SIM.

b. Select **File > Change Directory** and change to the directory you saved the DO file to above (the directory you created in the Basic Simulation lesson).

c. Type **radix -binary** at the ModelSim> prompt

d. Enter the following command at the QuestaSim> prompt:

vsim testcounter_opt

3. Execute the DO file and run the design.

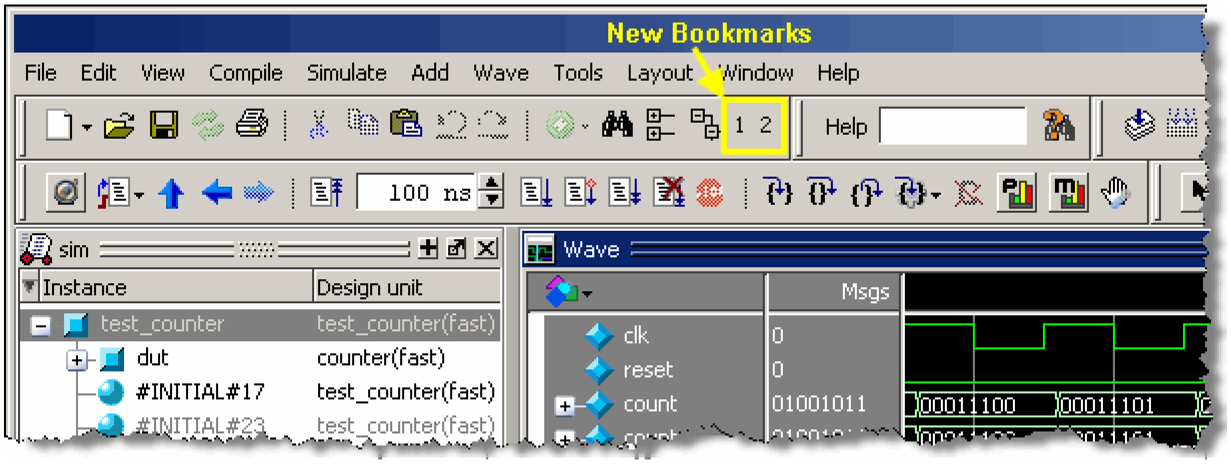
a. Type **do add_bkmrk.do** at the VSIM> prompt.

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 19-4).

Figure 19-4. Buttons Added to the Main Window Toolbar



c. Click the buttons and watch the Wave window zoom in and scroll to the time when *count* is the value specified in the DO file.

d. If the Wave window is docked in the Main window make it the active window (click anywhere in the Wave window), then select **Bookmarks** > **bk1**. If the window is undocked, select **Bookmarks** > **bk1** in the Wave window.

Watch the Wave window zoom in and scroll to the time when *count* is 8'h27. Try the **bk2** bookmark as well.

Lesson Wrap-Up

This concludes this lesson.

1. Select **File** > **Quit** to close Questa SIM.

Related Topics

User's Manual Chapter: Tcl and DO Files.

Practical Programming in Tcl and Tk, Brent B. Welch, Copyright 1997

Chapter 20
Getting Started With Power Aware

This tutorial for Questa Power Aware Simulation (Questa PASim) is designed to teach you about:

- The Unified Power Format (UPF) 2.0 Successive Refinement Methodology for creating a power intent specification. This UPF refinement capability is especially useful for the design of IP-based systems that may be targeted to multiple implementation technologies. It enables you to perform early power aware simulations when little or no design implementation details are known. As your design progresses towards the implementation phase, this refinement capability allows you to update the power intent specification of the design as more detailed system-level information becomes known. Implementation details regarding voltage values are added in an implementation specific UPF file which can then be used in the synthesis process.
- The Questa SIM power aware verification usage flow including the processing of power intent, enabling static and dynamic power aware assertion checks, generating power aware reports, and enabling the capture and tracking of power aware coverage data.
- The role of Liberty files in power aware simulations to help identify pre-existing power management cells in your design such that the simulator does not also insert the UPF specified cells, thus duplicating their functionality

Power Aware Tutorial Concepts

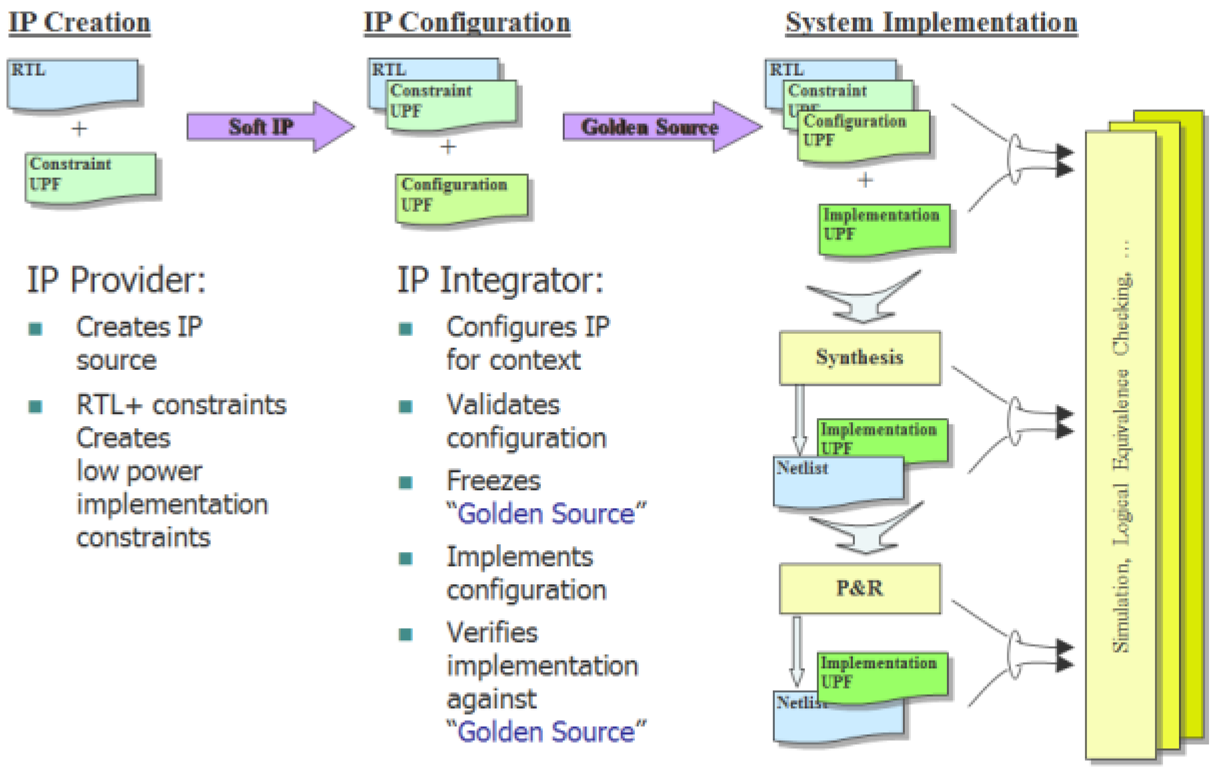
There are a few general concepts you should understand before starting on the interactive portion of the tutorial.

UPF Refinement Methodology

This tutorial utilizes the UPF 2.0 Successive Refinement Methodology for power intent specification.

This methodology (illustrated in Figure 20-1) is useful for the development of a power intent specification for IP-based systems and reflects the realities of SoC designs today. To illustrate this methodology, the tutorial involves running three different power aware simulations on the same RTL design. The second and third simulations each involve processing one additional UPF file to reflect the updated power intent information added from the previous simulation. Finally, the tutorial includes a fourth simulation to illustrate the use of Liberty files in Questa SIM.

Figure 20-1. UPF Refinement Methodology



Power Intent Processing

The UPF power intent process step occurs by executing the vopt command and loading both the top-level design and the UPF file, as specified by the -pa_upf option to the vopt command.

The power intent processing step results in the creation of the optimized simulation object that has been extended with some or all of the following power management features depending on the contents of the UPF file(s):

- Partitioning of the design elements into the desired power domains including the construction of the power distribution network. During the later phases of the Successive Refinement flow, this would include supply ports, supply nets, and power switches.
- Insertion of power architecture cells (retention registers, isolation cells, and level shifters) and the creation of the power control signals required to control their behavior.
- Instrumentation of the design with power-specific debug features to aid in the visualization of the UPF created objects and their power-related behavior (retention, corruption, and isolation clamping on power down with restoration on power up)
- Insertion of automatic assertions to check for power-related error conditions, such as the correct control signal sequencing.
- Instrumentation of the design to enable the capture, reporting, and tracking of power state and state transition coverage data.

Specifying Power Aware Options

You can customize many parts of your Questa PASim flow through the use of command line arguments to the vopt command.

Refer to the vopt command in the Command Reference Manual for a complete list, specifically those starting with -pa_. Many of these power aware arguments have additional values that can be specified by separating them via a + (plus sign) to enable or disable certain UPF capabilities and GUI-related debug capabilities.

A few additional power aware capabilities worth mentioning are those related to reading of Liberty files as well as the -pa_upfextensions command which enables the use of some non-standard UPF commands and options as well as several power aware-specific capabilities. Refer to the *Power Aware Simulation User's Manual* for Liberty file usage flows and options for enabling any necessary UPF extensions.

You can use Liberty files to enable additional power aware features, which include the ability to:

- Identify pre-existing power management cells that may be present in a design.
- Enable automatic connections of UPF supplies to PG pins on simulation models that require them in order to simulate properly.
- Extend a hard-macro's non-power aware behavior simulation model with power aware capabilities contained in the hard macro's Liberty file.

Design Files for This Lesson

The design for this example is a clock-driven memory interleaver with an associated test bench.

The directory structure is located in the *<install_dir>/examples/tutorials/pa_sim* directory, as shown below:

```
pa_sim
   |
   example_one
      |
      +-- Libraries           Verilog and SystemVerilog library source
      |     |
      |     +-- io
      |     |
      |     +-- isolation
      |     |
      |     +-- level_shift
      |     |
      |     +-- retention
      |     |
      |     +-- sram_256x16
      |     |
      |     +-- std_cell
      |
      +-- Questa              Simulation directory
      |     |
      |     +-- scripts       Compilation & simulation commands
      |
      +-- RTL                 Source files for RTL design
      |
      +-- TB                  Testbech for RTL/GLS designs
      |
      +-- GLS                 Source files for GLS design
      |
      +-- UPF                 UPF file for power intent
```

For this exercise, you run all simulations from the *example_one* directory.

Script Files

The *Questa/scripts/* directory contains the following do files for compiling and running all simulations:

- *compile_rtl.do* — Compile RTL source.
- *compile_gls.do* — Compile GLS source.
- *analyze_confi1.do* — Process UPF files for the first power aware RTL simulation.
- *analyze_confi2.do* — Process UPF files for the second power aware RTL simulation.
- *analyze_imp.do* — Process UPF files for the third power aware RTL simulation.
- *analyze_gls.do* — Process UPF and Liberty files for power aware gate-level simulation.
- *liberty_gls.do* — Read Liberty files and create Liberty attribute library.
- *doit_rtl.do* — Run the first two RTL power aware simulation.

- *doit_imp.do* — Runs the third RTL and GLS power aware simulations.

Preparation Tasks for RTL Simulations

There are two tasks you must perform before beginning the various simulation runs for this tutorial.

Creating a Working Location

Before you simulate the design for this example, you should make a copy of it in a working location.

Procedure

1. Create a new directory outside your installation directory for Questa SIM, and copy the design files for this example into it.
2. Invoke Questa SIM (if necessary).
   a. Type vsim at a UNIX shell prompt or double-click the Questa SIM icon in Windows.

      When you open Questa SIM for the first time, you will see the Welcome to Questa SIM dialog box. Click Close.

   b. Select **File** > **Change Directory** from the main menu, and navigate to

      ```
      <my_tutorial>/pa_sim/example_one
      ```

      where *my_tutorial* is the directory you created.

Compiling the Source Files of the Design

The compilation step processes the HDL design and generates code for simulation. This step is the same for both power aware and non-power aware simulation.

Procedure

- Enter the following command in the Transcript window to compile all RTL source files:

  ```
  do ./Questa/scripts/compile_rtl.do
  ```

  Note that this .do file runs the following Questa SIM commands:

  ```
  vlib work
  vlog -f ./Questa/scripts/compile_rtl.f
  ```

Also note that compiling the design files for power aware simulation requires no new or power aware specific switches.

Power Aware RTL Simulation 1

The first simulation involves the constraint (*mem_ctrl_const.upf*) and configuration (*mem_ctrl_config.upf*) UPF files.

Processing UPF Files (Simulation 1)

You can open the files (using the edit command in the Transcript window) to view the limited power intent information they contain. A third UPF file will be used to connect the power control registers of the test bench to the UPF-created power control logic ports of the design.

Procedure

- Analyze the power aware design by entering the following command in the Transcript window:

  ```
  do ./Questa/scripts/analyze_config1.do
  ```

  This script runs the vopt command with the following power aware arguments:

  ```
  vopt interleaver_tester +acc \
     -pa_upf ./UPF/mem_ctrl_config.upf \
     -pa_top "/interleaver_tester/dut" \
     -pa_genrpt=pa+de \
     -pa_checks=s+r+i \
     -pa_disable=defaultoff \
     -pa_enable=highlight+debug \
     -pa_tclfile connect_ports.upf \
     -o top_opt
  ```

  For this command, the -pa_upf and -pa_top arguments are the minimum required arguments for power intent processing. As always, vopt requires both the top-level design name (interleaver_tester) and the optimized object name (-o top_opt).

  - -pa_upf — Specifies the location of the UPF power intent file to be loaded. (required).
  - -pa_top — Identifies the power-managed DUT in the design hierarchy as vopt is invoked on the test bench. Note this option does not set the UPF scope (required).
  - -pa_genrpt=pa+de — Enables generation of the power-aware power architecture (*report.pa.txt*) and design element (*report.de.txt*) report files. You can view these reports in the current directory after invoking vsim by using the pa report command. (Optional).

- -pa_checks=s+r+i — Enable static Isolation (ISO) and Level-Shifter (LS) analysis during power-intent processing as well as all dynamic retention and isolation checks. The pa report command will also generate the *report.static.txt* file (Optional).
- -pa_disable=defaultoff — As per the UPF 2.0 LRM, supplies default to the OFF state. This option disables this behavior so all UPF supplies will default to FULL_ON state. (Optional).
- -pa_enable=highlight+debug — This switch enables both corruption and isolation highlighting in the wave window as well as enabling several GUI debug capabilities, including UPF object visualization (Optional).
- -pa_tclfile — Specifies an additional power aware related file to be read in by vopt. The file can contain additional UPF commands or any desired TCL commands to augment processing of the power intent information. The UPF scope for this file defaults to the top-level vopt design unit unless explicitly set.

During the vopt run you will see several vopt warnings (9885, 9833, and 9834) which can be ignored because the UPF files have purposely not defined any supply_nets. Note that the since this tutorial has enabled power aware static checks, you will also see the following warning message:

```
** Warning: (vopt-9855) [ UPF_ISO_STATIC_CHK ] Found Total 17 Not
analyzed isolation cells.
```

Static isolation analysis determines the need for an isolation cell when there is a power domain crossing and there is a power state in which the source domain is OFF and the sink domain is ON. The UPF files only define the power states for the PD_mem_ctrl domain so the vopt step is unable to determine the need for isolation in the design. This is due to the fact that the UPF files do not specify the operational voltage values for each power domain, therefore this step does not perform any static level shifter analysis.

Run Power Aware Simulation (Simulation 1)

This first power aware simulation using the Questa SIM GUI will acquaint you with several of its power aware and UPF debug features.

The test bench power cycles the PD_mem_ctrl domain three separate times. The first time the domain is power cycled, the power cycle control sequence is done correctly. The second power cycle does not isolate the PD_mem_ctrl domains outputs while the retention protocol is likewise not done correctly during the third time the domain is power cycled.

Procedure

- Enter the following command in the Transcript window to begin power aware simulation,

```
do ./Questa/scripts/doit_rtl.do
```

which runs the vsim command with the following arguments:

```
vsim top_opt \
   +nowarnTSCALE +nowarnTFMPC +notimingchecks \
   -pa -pa_highlight \
   -msgmode both \
   -displaymsgmode both \
   -coverage \
   -l rtl.log \
   -wlf rtl.wlf \
   -do ./scripts/sim.do
```

The -pa argument is required to invoke the simulator in power aware mode.

The -pa_highlight option enables corruption and isolation signal highlighting in the wave window. Note that the ability to show corruption and isolation highlighting must first be enabled during the vopt UPF processing step via the -pa_enable=highlight option.

The vsim command will issue an error if a power aware optimized design object is loaded and the -pa option is not used. Likewise, vsim will issue an error if a non-power aware optimized objects is loaded and the -pa option is used.

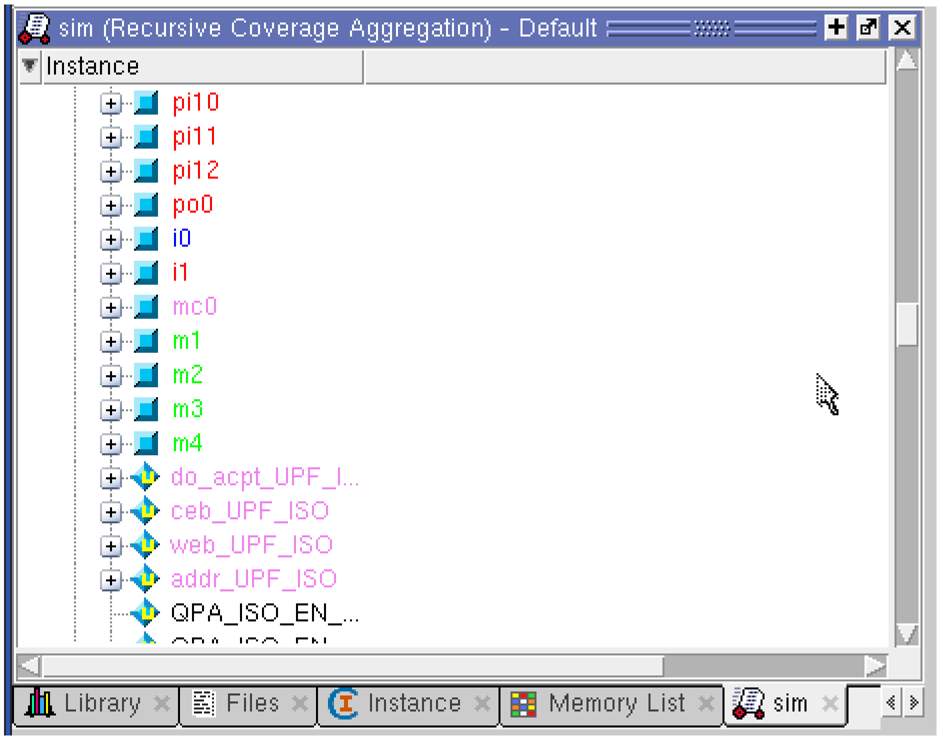
Exploring GUI Debug Features

There are several important simulation and UPF debug features available in the Questa SIM GUI.

Procedure

1. In the sim tab, expand the red-colored dut instance and scroll down to match the view as shown in Figure 20-2.

Figure 20-2. Power Domain Colorization in the Sim Tab



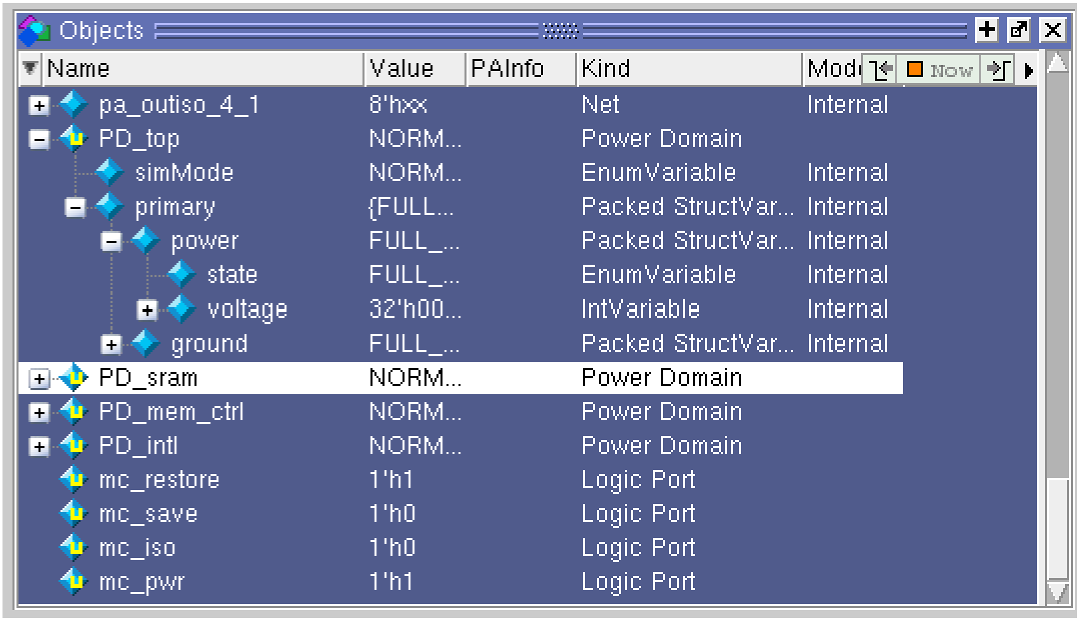
In the sim tab, design objects are colored according to the power domain they are placed in. The sim tab also contains a PD Name column which explicitly indicates the power domain that each instance belongs to. Any UPF inserted instances have a small yellow “u” annotated over the light-blue diamond shaped Verilog icon and these UPF inserted instances are colored based on the power domain that inferred their insertion.

2. Click on the red colored dut instance in the sim tab to select it, if it’s not already selected

3. Select the **View** > **Objects** menu item to open the Objects window

4. Scroll down to the bottom of the Objects window to see the UPF objects that were added to design as shown in Figure 20-3

Figure 20-3. UPF Created Objects



Name	Value	PAInfo	Kind	Mode
pa_outiso_4_1	8'hxx		Net	Internal
PD_top	NORM...		Power Domain	
simMode	NORM...		EnumVariable	Internal
primary	{FULL...		Packed StructVar...	Internal
power	FULL_...		Packed StructVar...	Internal
state	FULL_...		EnumVariable	Internal
voltage	32'h00...		IntVariable	Internal
ground	FULL_...		Packed StructVar...	Internal
PD_sram	NORM...		Power Domain	
PD_mem_ctrl	NORM...		Power Domain	
PD_intl	NORM...		Power Domain	
mc_restore	1'h1		Logic Port	
mc_save	1'h0		Logic Port	
mc_iso	1'h0		Logic Port	
mc_pwr	1'h1		Logic Port	

These UPF objects, which include power domains, supply_sets, and power control logic ports and nets, can be added to the wave window and their values viewed over the course of the simulation just like any other HDL objects.

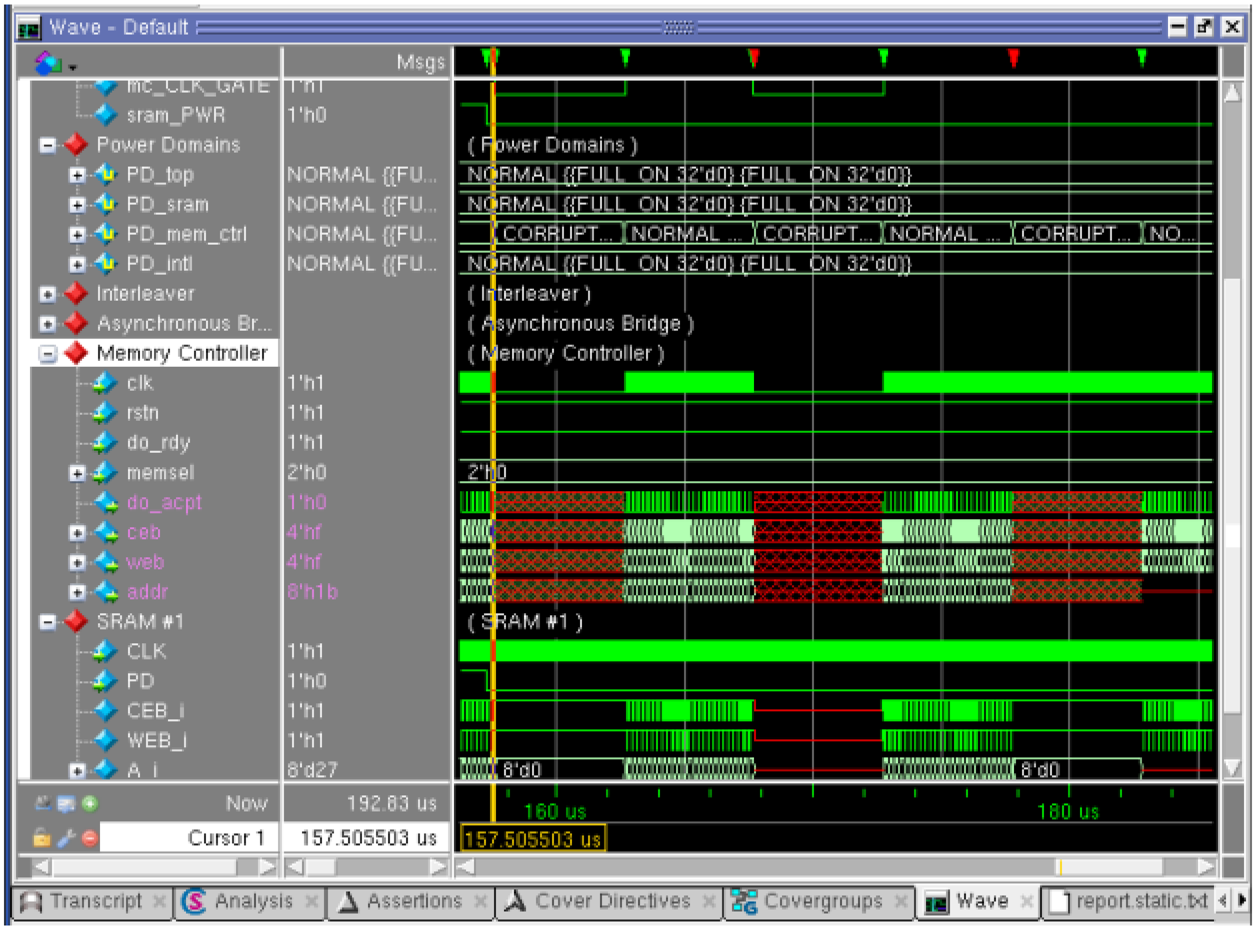
5. Click the Wave tab on the right side of the main window to view results of this simulation.

6. Select **Wave** > **Wave Preferences**, then the Display tab, then check PA waveform highlighting to enable further debug features in the Wave window.

7. In the Wave window, adjust the zoom level so that it shows the first three tests (about 155us to 185us), as shown in Figure 20-4.

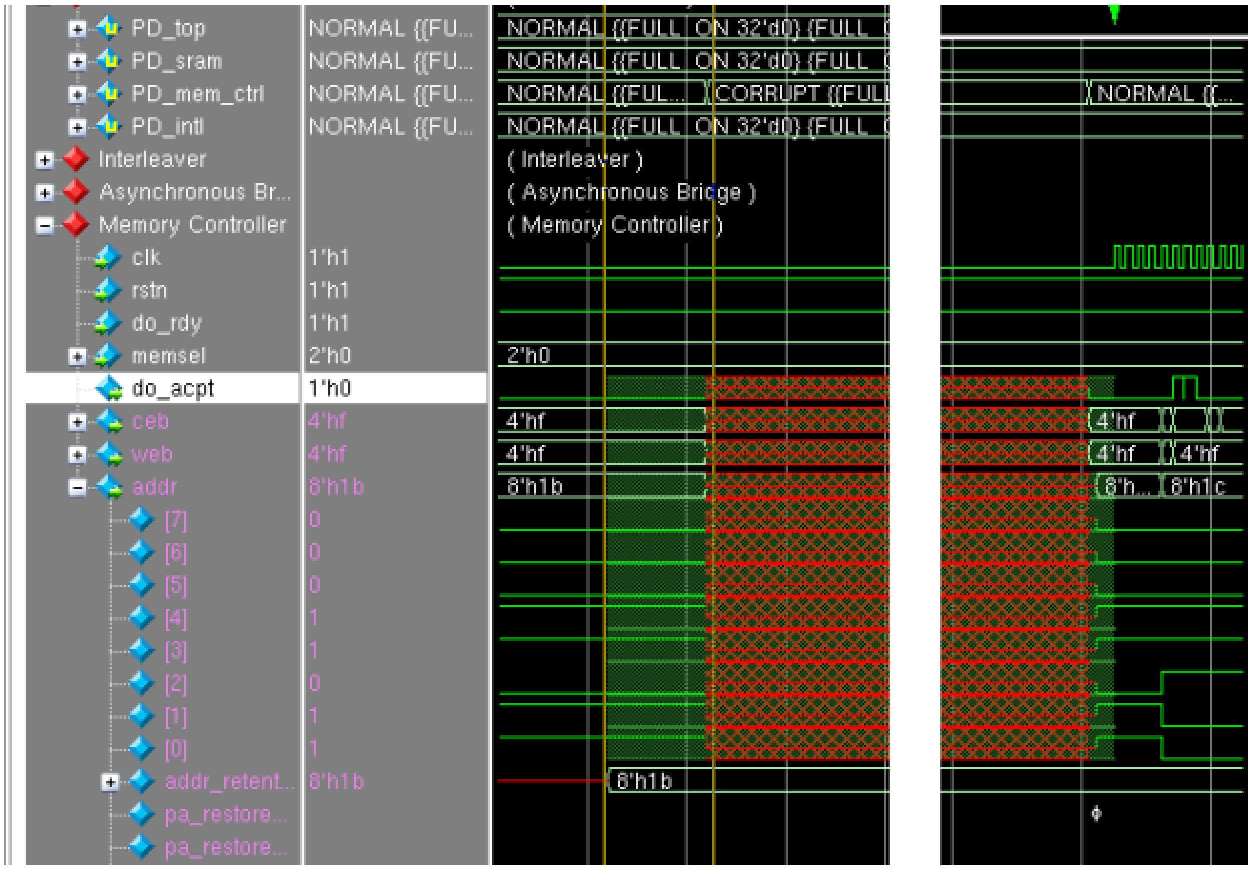
   The red cross-hatched areas indicate that the respective signals are corrupted because the PD_mem_ctrl domain's simstate is set to corrupt via the logic expression in the add_power_state command when the mc_PWR signal is LOW.

Figure 20-4. Wave Window Corruption Highlighting



8. Zoom in a little more to view the first power down cycle (around 157us) as shown in the left side of Figure 20-5. The power up of the domain (around 162.6us) is shown to the right side of the image. To see these areas you will need to zoom in separately on these areas.

Figure 20-5. Wave Window Isolation Highlighting and Retention Values



The light green shading indicates that the power domain ports are being isolated which corresponds to the time when the mc_ISO signal is asserted high. The actual value seen in the wave window depends on the location of the tool inserted isolation cells. Even though the mem_ctrl_config.upf file purposely doesn't specify the location for the PD_mem_ctrl isolation strategies, the tool picks a default location which defaults to the PD_mem_ctrl domains parent module which is the interleaver i0 instance. If you look at the downstream blocks from the memory controller's outputs, you can see the isolated values. At the inputs to the SRAMs, the address is clamped to a 0 while both the write and chip enable signals are clamped to 1 during the isolation period.

9. Expand the addr signal of the Memory Controller group to see the addr_retention_1 signal as shown in the figure.

   Each retention element in the wave window will have a <name>_retention_<n> element added to it that displays the retained register value. The register values need to be saved prior to power down and you can see that the addr signals 8'h1b value appears on the addr_retention_1 signal at the falling edge of the mc_SAVE signal. On the power up cycle, notice that the addr register value remains unknown after power is restored until the falling edge of the mc_RESTORE signal occurs at which time the retained 8'h1b value is again visible on the addr register.

Analyzing Power Aware Check Assertions

If you recall, the second time the PD_mem_ctrl domain is power cycled, the test bench doesn't enable isolation for the PD_mem_ctrl domain outputs. In the wave window, starting at about 167us, you will notice that the address, write enable, and chip enable inputs to the SRAM models (SRAM #1 signal group) are all unknown as a result of the unisolated outputs from the PD_mem_ctrl domain which have propagated to the SRAMs—this is a problem.

The third time the domain is power cycled the testbench doesn't follow the proper retention protocol by gating off the clock to the retention elements. Notice that after the third power cycle period, which ends at arund183us, the memory controller addr output remains unknown after power has be restored – this also is a problem.

While the testbench for the tutorial has been specifically coded to cause these power control sequence errors as a means to illustrate the capability of a few specific power aware assertion checks, the ability to have power aware simulation automatically generate over 20 various power specific checks provides an easy way to detect many types of subtle power related errors that may otherwise go undetected or require you to manually write assertions to provide them. If you recall, during the UPF processing step, all isolation and retention checks were enabled using the -pa_checks=i+r arguments respectively.

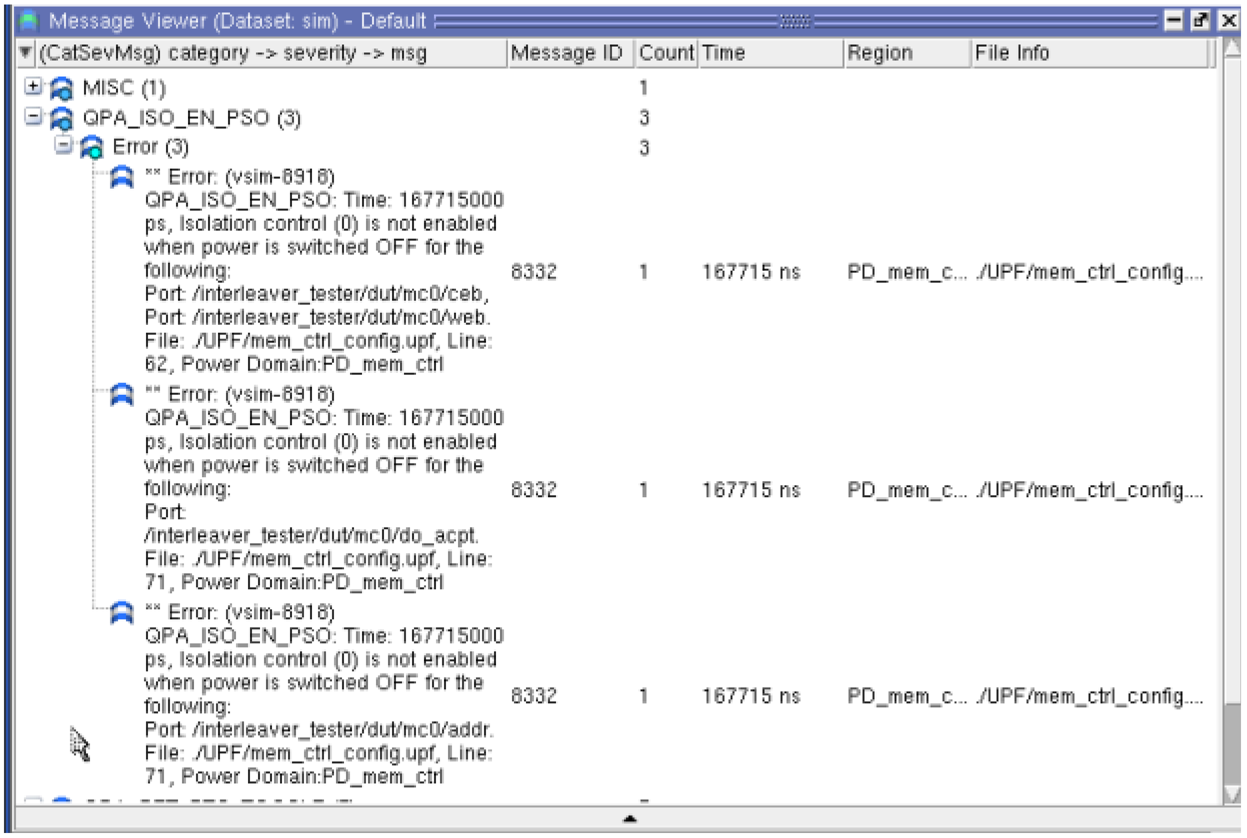
While all isolation and retention checks have been enabled in this tutorial, not all power aware assertion checks are applicable for every design. Questa PASim provides you with the ability to enable individual isolation/retention checks via specific i*xx* or r*xx* pa_checks options (such as ifc or rpo) as well as control any enabled power aware checks using the pa msg command.

Procedure

1. Select the **View** > **Message Viewer** > **sim** menu item to open the Message Viewer window.
2. Right- click in the Message Viewer window and select **Hierarchy Configuration** followed by **CatSevMsg** to change the default view.
3. Expand the QPA_ISO_EN_PSO category followed by the Errors severity to see the three error messages as shown in Figure 20-6. There is an error message for each un-isolated power domain ports

   Note that the default categorization of messages in the Message Viewer according to severity.

Figure 20-6. Message Viewer: Isolation Not Enabled Errors



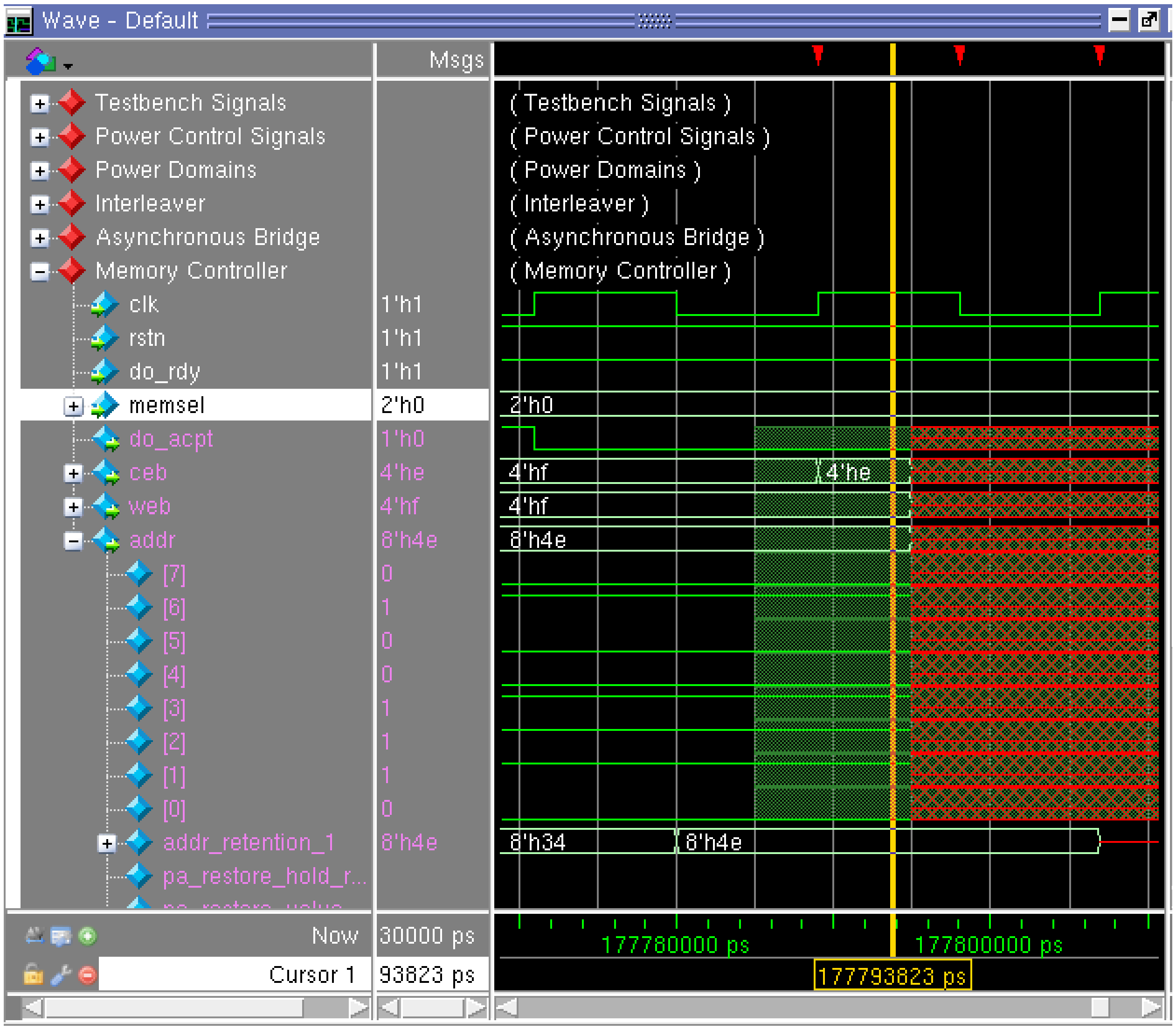
(CatSevMsg) category -> severity -> msg	Message ID	Count	Time	Region	File Info
MISC (1)		1			
QPA_ISO_EN_PSO (3)		3			
Error (3)		3			
** Error: (vsim-8918) QPA_ISO_EN_PSO: Time: 167715000 ps, Isolation control (0) is not enabled when power is switched OFF for the following: Port: /interleaver_tester/dut/mc0/ceb, Port: /interleaver_tester/dut/mc0/web. File: ./UPF/mem_ctrl_config.upf, Line: 62, Power Domain:PD_mem_ctrl	8332	1	167715 ns	PD_mem_c...	./UPF/mem_ctrl_config....
** Error: (vsim-8918) QPA_ISO_EN_PSO: Time: 167715000 ps, Isolation control (0) is not enabled when power is switched OFF for the following: Port: /interleaver_tester/dut/mc0/do_acpt. File: ./UPF/mem_ctrl_config.upf, Line: 71, Power Domain:PD_mem_ctrl	8332	1	167715 ns	PD_mem_c...	./UPF/mem_ctrl_config....
** Error: (vsim-8918) QPA_ISO_EN_PSO: Time: 167715000 ps, Isolation control (0) is not enabled when power is switched OFF for the following: Port: /interleaver_tester/dut/mc0/addr. File: ./UPF/mem_ctrl_config.upf, Line: 71, Power Domain:PD_mem_ctrl	8332	1	167715 ns	PD_mem_c...	./UPF/mem_ctrl_config....

4. Double-click on either the 167715ns value in the Time column or the UPF file name in the File Info column to open the wave-window with the active cursor placed at that time or open the UPF file showing the line where the isolation strategy is defined, respectively.

5. To debug why the memory controller's addr register value remains unknown after the third time the PD_mem_ctrl domain is power cycled for retention, double-click on the 177789ns time value in the Message Viewers Time column for the very first QSPA_RET_RTC_TOGGLE error message

6. In the wave window, zoom in around the cursor position and expand the addr signal if it's not expanded already. As shown in Figure 20-7, the retained 8'h4e addr value shown by the addr_retention_1 signal becomes unknown at the first rising-edge of the clk signal after the PD_mem_ctrl domain is corrupted, as indicated by the pa_store_x event. The retained value is lost because the retention condition, specified in the mem_ctrl_ret retention strategy, is violated, which can be seen by double clicking the mem_ctrl_config.upf file indicated in the File Info column of the Message Viewer

Figure 20-7. Retention Value Over-written



7. Close the simulation by typing the following command in the Transcript window

```
quit -sim
```

Power Aware RTL Simulation 2

The second simulation run involves processing the *rtl_top.upf* file in addition to the two UPF files used in the first power aware simulation.

Processing UPF Files (Simulation 2)

The new UPF file updates the power intent specification with the following information:

- Explicit supply_set information, including designation of which specific supply_set is the primary supply for each power domain. Also included is the designation of the isolation and retention supplies for the PD_mem_ctrl domain.
- An internal switch for the PD_mem_ctrl domain to reflect the fact that the power rail will be switched instead of the ground rail.
- Update to the PD_mem_ctrl OFF power state to reflect that fact that the power rail is switched.
- Location point of inferred isolation cells for PD_mem_ctrl isolation strategies.
- New power state information for each domain.

Procedure

1. Enter the following command in the Transcript window to perform the power intent processing step for the second simulation:

   ```
   do ./Questa/scripts/analyze_config2.do
   ```

   The only changes in this script is that the -pa_upf vopt option specifies the *rtl_top.upf* file and the -pa_coverage option has also been added.

   The -pa_coverage option instructs vopt to generate the power state coverage information that will be captured during the running of this second RTL simulation.

   Notice that since that the power state information for all power domains has been added, the static isolation results now show the following:

   ```
   # -- Analyzing UPF for Power Aware Static Checks...
   # -- Analysis of Power Aware Static Checks Complete.
   # -- Power Aware Static Checks status:
   # ** Note: (vopt-9857) [ UPF_ISO_STATIC_CHK ] Found Total 17 Valid
   isolation cells.
   ```

2. Run the pa report command in the Transcript window to view the detailed static isolation report.
3. In your text editor of choice, open the *report.static.txt* file and view the detailed static isolation report file.
4. Open both the *report.pa.txt* and *report.de.txt* files to see the contents of the power architecture report and design element report information, respectively.

Run Power Aware Simulation (Simulation 2)

You can now run the second power aware simulation with the same .do file used to run the first power aware Simulation.

In this simulation run you'll be introduced to power state FSM visualization capabilities. Power state information (state hits and state transitions) is displayed in the same manner as any RTL

FSM detected in the design. Note that the previous power aware debug features described in the first power aware simulation are available in this run as well

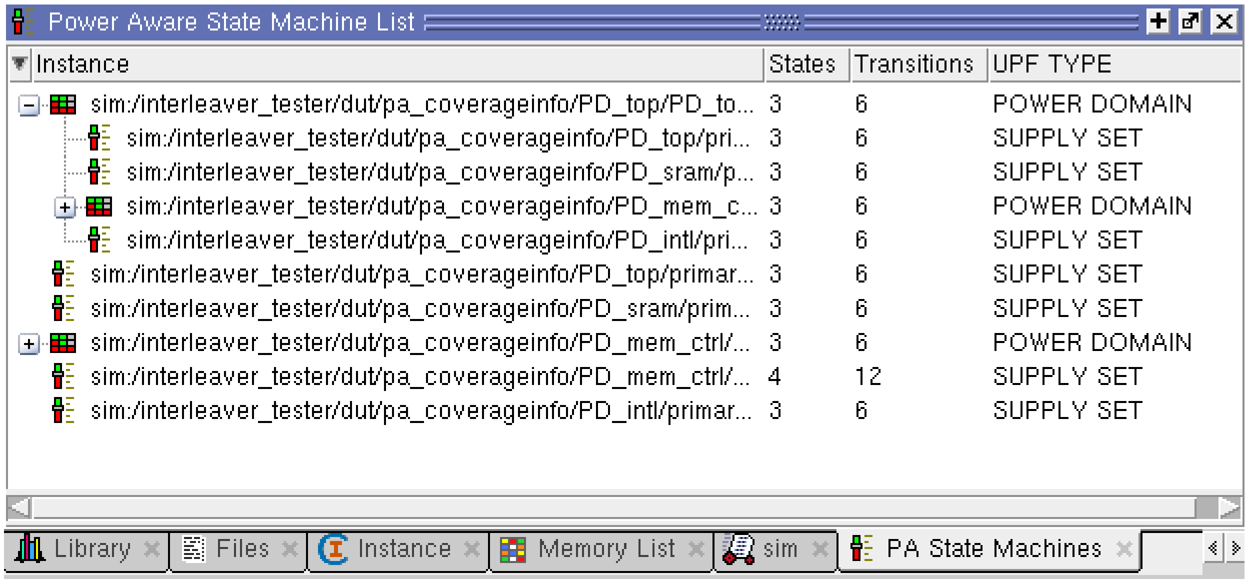
Procedure

1. Type the command below to invoke run the simulation:

```
do ./Questa/scripts/doit_rtl.do
```

2. Select the **View** > **PA State Machine List** menu item to see the defined power states as shown in Figure 20-8.

Figure 20-8. Power Aware State Machine List Window

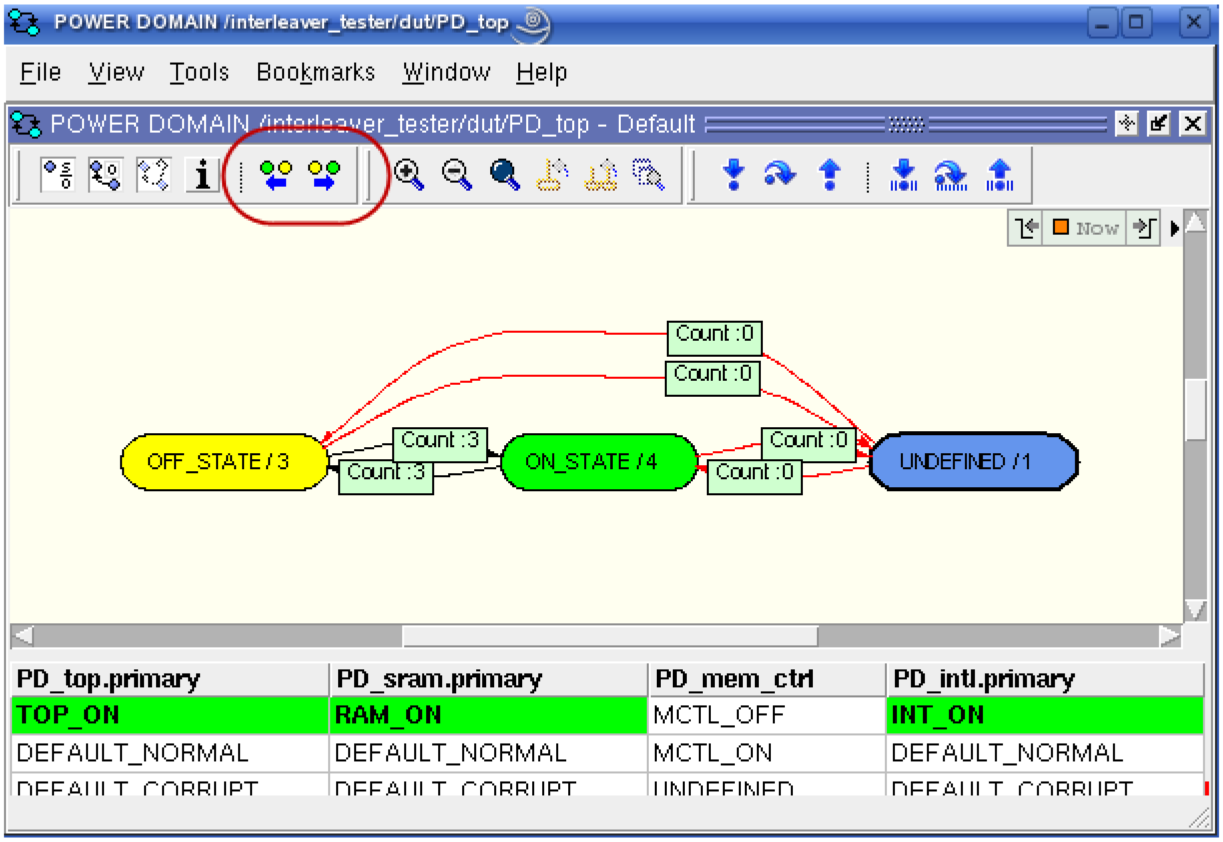


Instance	States	Transitions	UPF TYPE
sim:/interleaver_tester/dut/pa_coverageinfo/PD_top/PD_to...	3	6	POWER DOMAIN
sim:/interleaver_tester/dut/pa_coverageinfo/PD_top/pri...	3	6	SUPPLY SET
sim:/interleaver_tester/dut/pa_coverageinfo/PD_sram/p...	3	6	SUPPLY SET
sim:/interleaver_tester/dut/pa_coverageinfo/PD_mem_c...	3	6	POWER DOMAIN
sim:/interleaver_tester/dut/pa_coverageinfo/PD_intl/pri...	3	6	SUPPLY SET
sim:/interleaver_tester/dut/pa_coverageinfo/PD_top/primar...	3	6	SUPPLY SET
sim:/interleaver_tester/dut/pa_coverageinfo/PD_sram/prim...	3	6	SUPPLY SET
sim:/interleaver_tester/dut/pa_coverageinfo/PD_mem_ctrl/...	3	6	POWER DOMAIN
sim:/interleaver_tester/dut/pa_coverageinfo/PD_mem_ctrl/...	4	12	SUPPLY SET
sim:/interleaver_tester/dut/pa_coverageinfo/PD_intl/primar...	3	6	SUPPLY SET

Both the PD_intl/primary_obj and PD_mem_ctrl/PD_mem_ctrl_PS/PD_mem_ctrl_obj power sate objects appear twice in this pane because they are defined once as well as used in the power state definitions for the PD_top/PD_top_PS/PD_top_obj power state object.

3. Double-click the row for the PD_top Power Domain to open the power state diagram for that power state, as shown in Figure 20-9. You will want to undock the window on the Windows platform to have the toolbar within the window.

Figure 20-9. PD_top Domain Power State Diagram



PD_top.primary	PD_sram.primary	PD_mem_ctrl	PD_intl.primary
TOP_ON	RAM_ON	MCTL_OFF	INT_ON
DEFAULT_NORMAL	DEFAULT_NORMAL	MCTL_ON	DEFAULT_NORMAL
DEFAULT_CORRUPT	DEFAULT_CORRUPT	UNDEFINED	DEFAULT_CORRUPT

4. By default the power states diagram pane also lists the respective power states which comprise the displayed power state diagram in a tabular form. The current state is colored green, the previous state is colored yellow, and all other states are colored blue. The red circled previous/next state FSM toolbar icons can be used to transition the power state diagram to the respective power state if it has occurred during the simulation.

5. Double-click on the power state object names shown in the tabular list to display the power state diagram for that power state object.

6. This concludes the second power aware simulation. Exit the simulation by typing the command below in the transcript window

```
quit -sim
```

Power Aware RTL Simulation 3

The third simulation run involves processing the *rtl_top_imp.upf* file in addition to the three UPF files used in the second power aware simulation.

Processing UPF Files (Simulation 3)

The new UPF file updates the power intent specification with the following implementation power intent information:

- The top-level supply_ports and supply_nets.
- Updates the supply_set power and ground functions with the supply_net names that provide each function.
- Updates the power state information to provide the operation voltages for each domains primary supply.
- Defines the required level shifters for the design now that the voltage values for each power domain is known.
- Specifies the input and output supplies required to power the inferred level shifter cells from both the PD_mem_ctrl and PD_intl power domains level shifter strategies, respectively.

Procedure

1. To perform power intent processing step for the third simulation, enter the following in the Transcript window:

   ```
   do ./Questa/scripts/analyze_imp.do
   ```

   The changes in this script is that the -pa_upf option to vopt now specifies the *rtl_top_imp.upf* file, the -pa_disable=defaultoff option has been removed, a few more pa_checks options (ul+cp+p+ugc+upc) have been added, and the -pa_enable option has two additional options (sndebug+autotestplan).

   Note that because the operation voltage values of each domain has been specified, power, the static level shifting analysis is now able to be done:

   ```
   # -- Analyzing UPF for Power Aware Static Checks...
   # -- Analysis of Power Aware Static Checks Complete.
   # -- Power Aware Static Checks status:
   # ** Note: (vopt-9851) [ UPF_LS_STATIC_CHK ] Found Total 45 Valid
   level shifters.
   # ** Note: (vopt-9857) [ UPF_ISO_STATIC_CHK ] Found Total 17 Valid
   isolation cells.
   ```

2. To view the detailed new static analysis report, run the pa report command in the Transcript window.

3. Open the *report.static.txt* file to see the additional level shifter cell information for each power domain source-to-sink ports that requires it. The individual level shifted candidate ports for each domain are also now reported in the *report.pa.txt* file as well

Run Power Aware Simulation (Simulation 3)

In this simulation run you will be introduced to the Questa PASim supply network visualization feature and the power coverage test plan tracking.

Power state information (state hits and state transitions) is displayed in a same manner as any normal RTL FSM detected in the design. The fact that power state coverage is shown during the implementation phase of the Successive Refinement Methodology is purely arbitrary. Power state coverage can be enabled earlier in the flow if desired. Note that all the previous power aware debug features described in the first two power aware simulations are available in this run as well.

Procedure

1. Type the command below to invoke run the simulation:

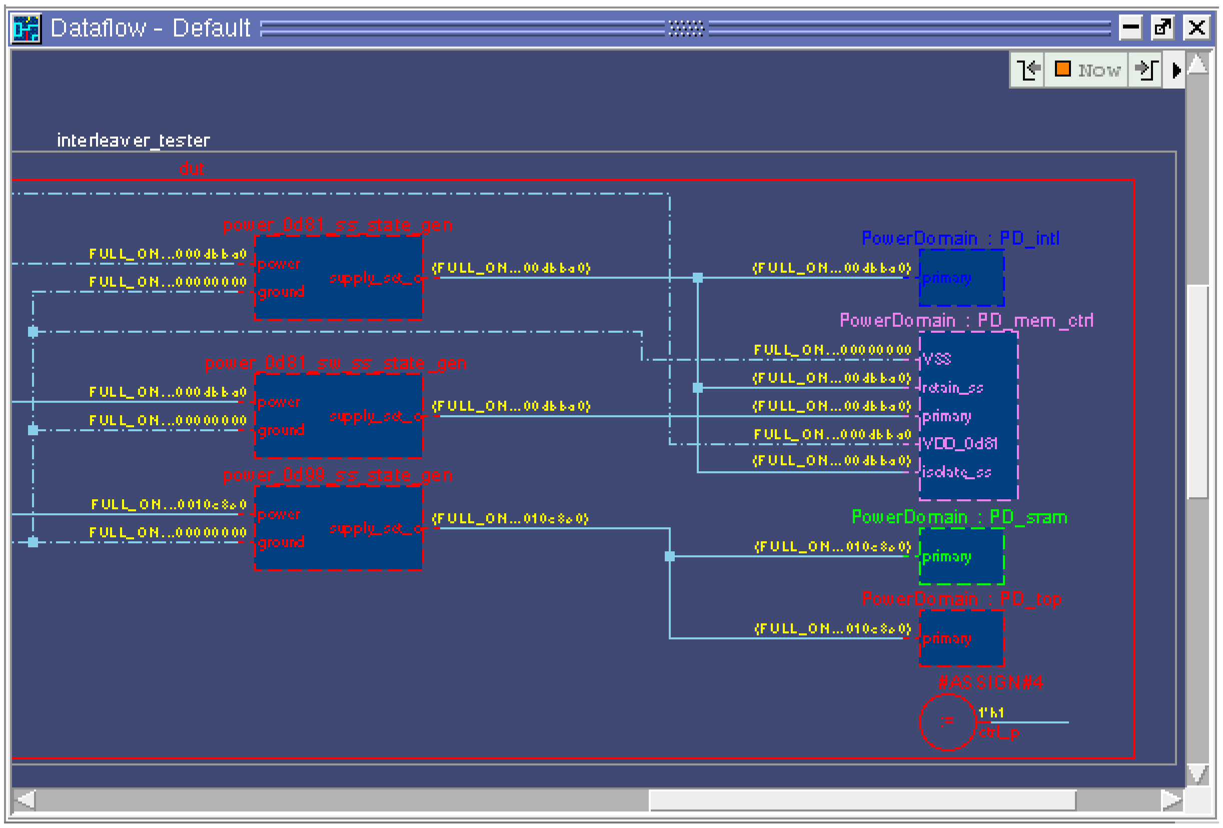
   ```
   do ./Questa/scripts/doit_imp.do
   ```

2. Since the voltage values have been specified in the UPF, the .do file contains a vsim plusarg argument (+IMPLEMENT) which is used to execute the imported package UPF defined supply_on functions. The test bench includes a supply_on function for each UPF defined supply_net which explicitly sets both the state and voltage values respectively (see lines 65-70 in TB/interleaver_tester.sv).

3. Select **View > UPF > Supply Network** to display the Dataflow window, which will show the interleaver_tester module box with the folded dut instance inside it.

4. To unfold the dut, right-click inside the dut instance box and then choose **Unfold Selected** from the popup menu. The Dataflow window will display the supply network connectivity, as shown in Figure 20-10.

Figure 20-10. Supply Network Connectivity Visualization



The Dataflow shows the supply network connectivity to four power domain objects on the far right side. These power domain objects can also be unfolded, if desired, to view the isolation, level_shifter, and retention strategies they might contain. Unfolding a power domain block does not show connectivity to the design elements that make up the domain.

5. In the transcript window, type the command below to generate the *testplan.ucdb* file

```
pa autotestplan -filename=testplan.ucdb
```

6. In the transcript window, type the command below to save the simulation coverage database:

```
coverage save pa_cov.ucdb
```

7. In the Transcript window, type the following to quit the sim but leave the GUI open:

```
quit -sim
```

8. Select the **View** > **Verification Management** > **Browser** menu item to open the verification management UCDB file browser.

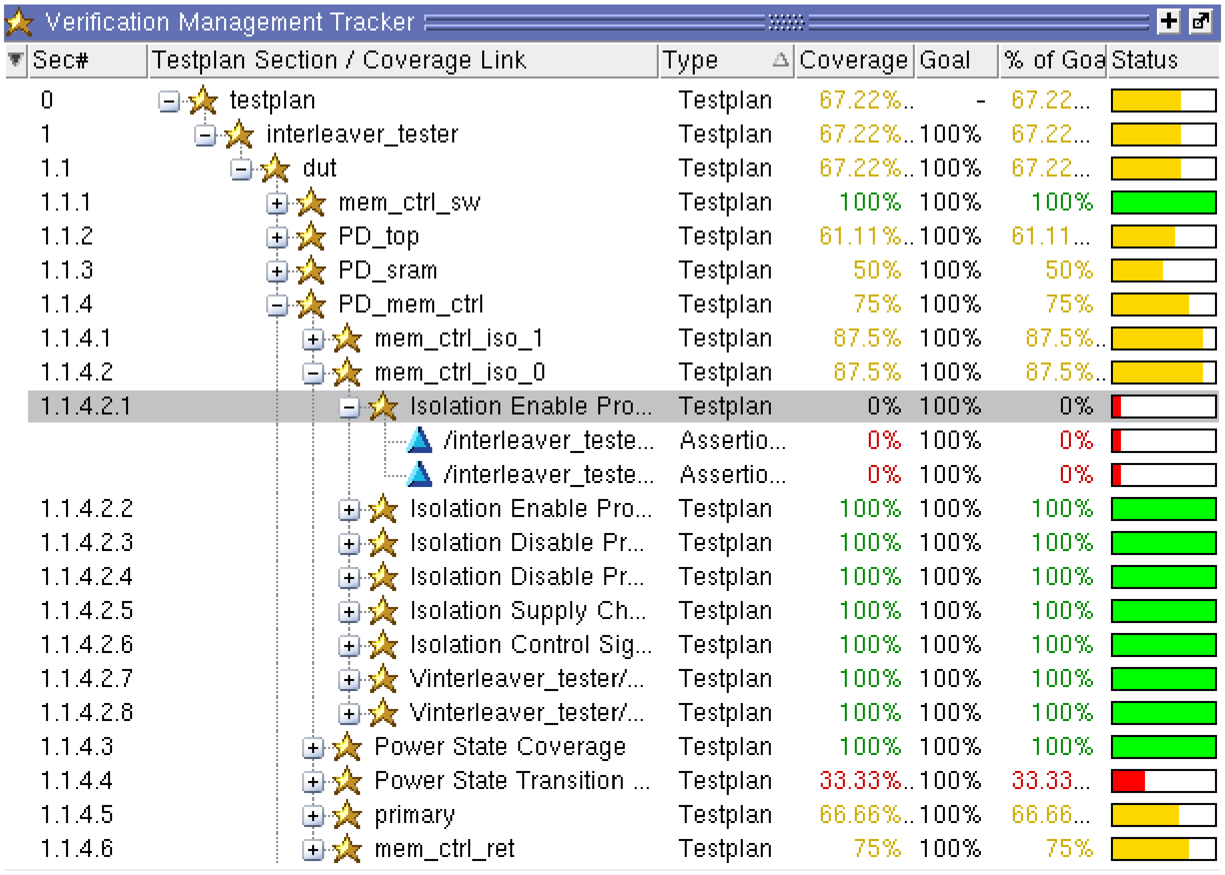
9. In the Verification Management Browser, right-click and select **Add File**. Select the two UCDB files and then click **Open** to add them to the browser.

10. Select both UCDB files in the browser, then right-click and select **Merge** from the pop-up menu. Click **OK** on the Merge Files window, accepting the default settings, which results in the merged UCDB file (*merge.ucdb*) appearing in the browser.

11. Double -click on the *merge.ucdb* file to load it in the GUI's coverage view mode and the Tracker pane will be opened to view the power aware coverage results

12. Select the Tracker window to make it active and expand the testplan sections down to 1.1.4.2.1 as shown in Figure 20-11

Figure 20-11. Verification Management Tracker Pane



Verification Management Tracker

Sec#	Testplan Section / Coverage Link	Type	Coverage	Goal	% of Goa	Status
0	testplan	Testplan	67.22%..	-	67.22...	🟨
1	interleaver_tester	Testplan	67.22%..	100%	67.22...	🟨
1.1	dut	Testplan	67.22%..	100%	67.22...	🟨
1.1.1	mem_ctrl_sw	Testplan	100%	100%	100%	🟩
1.1.2	PD_top	Testplan	61.11%..	100%	61.11...	🟨
1.1.3	PD_sram	Testplan	50%	100%	50%	🟨
1.1.4	PD_mem_ctrl	Testplan	75%	100%	75%	🟨
1.1.4.1	mem_ctrl_iso_1	Testplan	87.5%	100%	87.5%..	🟨
1.1.4.2	mem_ctrl_iso_0	Testplan	87.5%	100%	87.5%..	🟨
1.1.4.2.1	Isolation Enable Pro...	Testplan	0%	100%	0%	🟥
	/interleaver_teste...	Assertio...	0%	100%	0%	🟥
	/interleaver_teste...	Assertio...	0%	100%	0%	🟥
1.1.4.2.2	Isolation Enable Pro...	Testplan	100%	100%	100%	🟩
1.1.4.2.3	Isolation Disable Pr...	Testplan	100%	100%	100%	🟩
1.1.4.2.4	Isolation Disable Pr...	Testplan	100%	100%	100%	🟩
1.1.4.2.5	Isolation Supply Ch...	Testplan	100%	100%	100%	🟩
1.1.4.2.6	Isolation Control Sig...	Testplan	100%	100%	100%	🟩
1.1.4.2.7	Vinterleaver_tester/...	Testplan	100%	100%	100%	🟩
1.1.4.2.8	Vinterleaver_tester/...	Testplan	100%	100%	100%	🟩
1.1.4.3	Power State Coverage	Testplan	100%	100%	100%	🟩
1.1.4.4	Power State Transition ...	Testplan	33.33%..	100%	33.33...	🟥
1.1.4.5	primary	Testplan	66.66%..	100%	66.66...	🟨
1.1.4.6	mem_ctrl_ret	Testplan	75%	100%	75%	🟨

The auto generated power aware testplan includes not only the power state hits and state transitions but it also includes all the automated power aware assertion checks. Assertions are treated as coverage points if they pass at least once and never fail during a simulation.

As you can see from this example, the Isolation Enable Protocol Checks (1.1.4.2.1) have zero coverage as these are the checks that failed. The same is true for the Retention Condition Toggle checks (1.1.4.6.5, not shown).

13. In the transcript, type the command below to exit the coverage view mode.

```
dataset close merge
```

This completes the RTL portion of the power aware tutorial

Liberty Files and Power Aware Simulations

The role that Liberty files play in power aware simulations varies widely. As mentioned previously, one important use of Liberty files is to identify the existence of power management cells which are already present in a design. For designs containing pre-existing power management cells, it is typically required that they be detected in order to prevent the insertion of duplicate power management cells.

Questa PASim has several different ways to detect the presence of power management cells in a design. Liberty files contain attributes for cells that can be extracted and then used to identify the intended functionality of various cell types. This tutorial illustrates the use of Liberty (.lib) files to detect power management cells in a post synthesis gate-level netlist.

Note that Questa PASim does not support the reading of a design compiler generated Liberty file format (.db).

Creating a Liberty Attribute Library

You can specify a list of Liberty files as an argument to the vopt command with the -pa_libertyfiles= option to be processed at the same time as any UPF files.

Because Liberty files typically do not change and can be quite large in size, spending time to reprocessing the same Liberty files each time vopt is run can be avoided by creating a Liberty attribute library in a separate vopt command. You can then load the Liberty attribute library with vopt when processing the UPF file(s).

Procedure

Invoke Questa SIM (if necessary) by either typing vsim at a Linux shell prompt or by double-clicking on the Questa SIM icon in Windows.

1. Create the Liberty attribute library by executing the command below.

   ```
   do ./Questa/scripts/liberty_gls.do
   ```

   This script runs the vopt command with the following arguments:

   ```
   -pa_libertyfiles=
   ./Libraries/isolation/isolation.lib,./Libraries/retention/retention
   .lib,\
   ./Libraries/level_shift/level_shift.lib \
   -pa_dumplibertydb=qpa_liberty_lib \
   ```

   Note that the list of Liberty files must not contain any spaces. Also you can use any name for the Liberty attribute library created by the -pa_dumplibertydb= option

2. Compile the gate-level design into a new work library.

   ```
   do ./Questa/scripts/compile_gls.do
   ```

3. Process the UPF file for the GLS netlist.

   ```
   do ./Questa/scripts/analyze_gls.do
   ```

   This script contains many of the same power aware and UPF options as those used when analyzing the UPF for the RTL design. The two exceptions are the -pa_loadlibertydb option and the -pa_gls option. Obviously the -pa_loadlibertydb options loads the extracted Liberty attribute library while the -pa_gls option instructs Questa PASim not to insert any power management cells into the design. The -pa_gls option should not be used when the design has some but not all of the power management cells inserted.

4. Type pa report in the transcript window to generate the reports.

5. Open the *report.pa.txt* file. In this report file you will see the list of -instance annotated isolation cells as well as “Signals with Retention cells” and “Signals with Level Shifter Cells”. All these cell instances have been detected via the attributes extracted from the Liberty files into the qpa_liberty_library.

6. Open the *interp_gls.upf* file from the current directory. This file was written as a result of the save_upf command at the bottom of the *UPF/compile.upf* file. Search for the word “instance” in the file and you will see that Questa PASim has interpreted the original UPF file to include the -instance annotation for all ISO/RET/LS strategies in the UPF file.

   It is important to note that Questa PASim assumes that all the -instance annotated power management cells identified in a design contain all the necessary functionality required to simulate properly. Unfortunately, this assumption does not hold true for all power management cells, especially retention cells. If a retention cell does not contain the necessary internal retention logic, then Questa PASim will only add it if the retention cell detection capability is first disabled by using the vopt option.

   ```
   -pa_disable=detectret
   ```

7. In this gate-level design, the DFF_RET cells do not contain the retention behavior required to simulate properly. Execute the command below to reprocess the UPF file with the retention detection capability disabled.

   ```
   do ./Questa/scripts/analyze_gls_wret.do
   ```

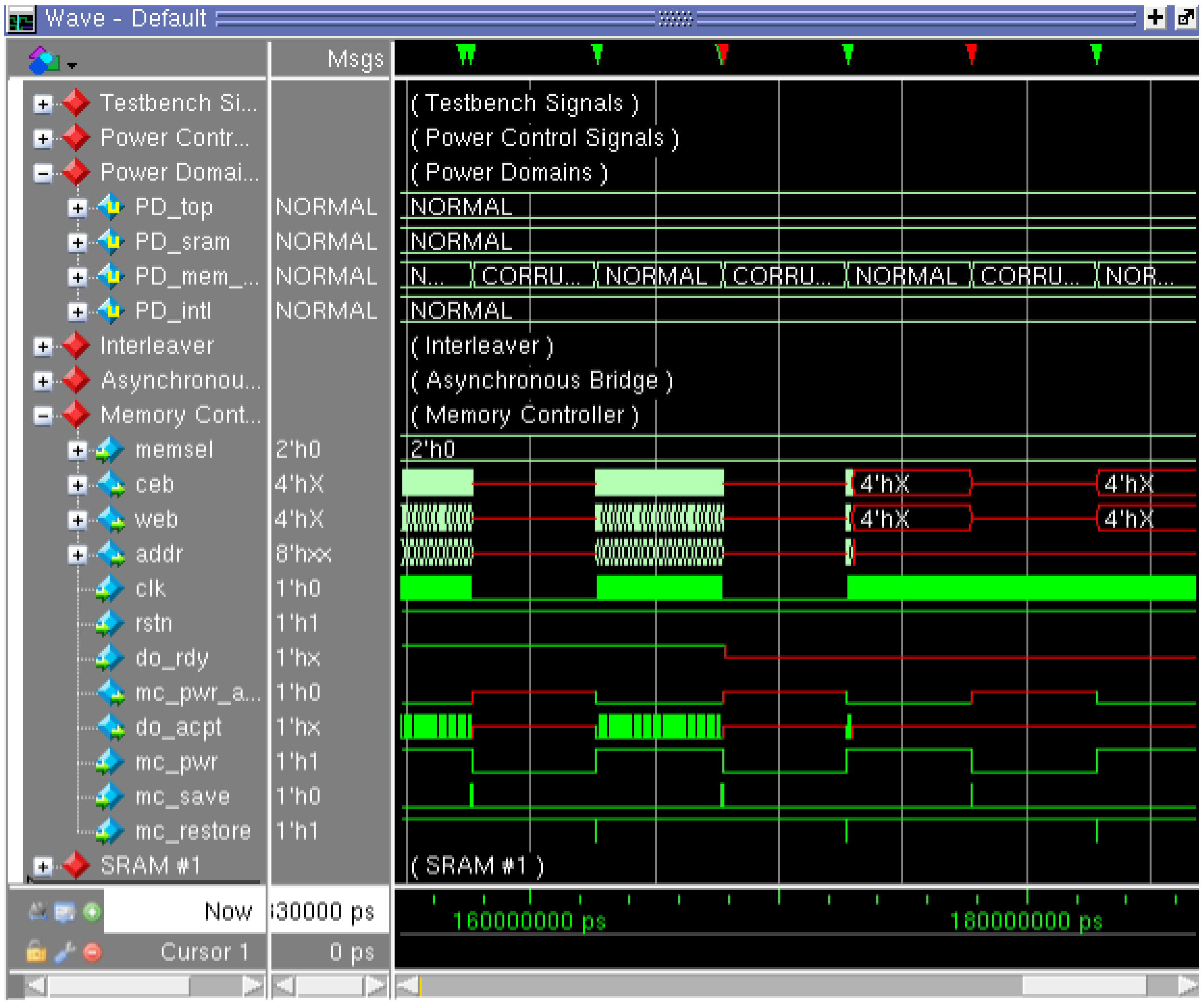
8. Run the gate-level power aware simulation typing the command below in the transcript window

   ```
   do ./Questa/scripts/doit_imp.do
   ```

9. Look in the Wave window and you will see a view as shown in Figure 20-12. While the same three Memory controller power cycles occur, notice that when the powered up the second time, the memory controller addr, ceb, and web ports quickly go from their

restored value to unknown in just a few clock cycles. The unknown values on these ports are a direct result of not asserting the isolation enable, allowing the X value on the do_acpt port to propagate to async_bridge instance in the PD_intl domain which in-turn caused the do_rdy input from the async_bridge to be unknown after power up. The do_rdy X value feeds directly into present state registers that in turn, caused the unknown values on these three ports.

Figure 20-12. Gate-level Power Aware Simulation



Wave - Default

	Msgs
Testbench Si...	
Power Contr...	
Power Domai...	
PD_top	NORMAL
PD_sram	NORMAL
PD_mem_...	NORMAL
PD_intl	NORMAL
Interleaver	
Asynchronou...	
Memory Cont...	
memsel	2'h0
ceb	4'hX
web	4'hX
addr	8'hxx
clk	1'h0
rstn	1'h1
do_rdy	1'hx
mc_pwr_a...	1'h0
do_acpt	1'hx
mc_pwr	1'h1
mc_save	1'h0
mc_restore	1'h1
SRAM #1	

Now 130000 ps
Cursor 1 0 ps

If you compare the screen shot with the same one for the Figure 20-4, you will notice that the corruption highlighting is also not present on the memory controller ceb, web, and addr output ports. In a gate-level power aware simulation corruption highlighting is only visible when looking directly at the output port of a cell.

1. In the sim tab, expand the dut/mc0 instance and then select the U3 instance. Press ctrl+w to add the cell's ports to the bottom of the wave window. The corruption highlighting will be visible on the Z output port of the cell.

2. This concludes the power aware GLS simulation and the tutorial. Close the GUI by typing the command below in the transcript.

   ```
   quit -f
   ```

Index

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S 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 (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer 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 Customer does not agree to these terms and conditions, promptly return or, in the case of Software 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.

1. **ORDERS, FEES AND PAYMENT.**

1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.

1.2. Amounts invoiced will be paid, 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. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, 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 Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; 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. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. 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. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable

files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer’s compiled computer program, provided that Customer distributes these files only in conjunction with Customer’s compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics’ real-time operating systems or other embedded software products into Customer’s products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

4.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively “Beta Code”), which may not be used without Mentor Graphics’ explicit authorization. Upon Mentor Graphics’ authorization, Mentor Graphics grants to Customer 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. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.

4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer’s use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer’s evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees 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 Customer’s 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 Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

5.1. Customer 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. Customer 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. Customer shall not make Products available in any form to any person other than Customer’s employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively “Files”), including without limitation files containing Standard Verification Rule Format (“SVRF”) and Tcl Verification Format (“TVF”) which are Mentor Graphics’ trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.

5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer’s employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise (“Attempted Transfer”), without Mentor Graphics’ prior written consent and payment of Mentor Graphics’ then-current applicable relocation and/or transfer fees. 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/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer’s permitted successors in interest and assigns.

5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics’ then current End-User Support Terms located at http://supportnet.mentor.com/supportterms.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer’s requirements or that operation of Products 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. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **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 RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE 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 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). EXCEPT TO THE EXTENT 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 ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **INDEMNIFICATION.** CUSTOMER AGREES 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 THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **INFRINGEMENT.**

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

11.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

11.4. THIS SECTION 11 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

12. **TERMINATION AND EFFECT OF TERMINATION.**

12.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

12.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

13. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.

14. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

15. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

16. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all 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. The provisions of this Section 16 shall survive the termination of this Agreement.

17. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is 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 the courts 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 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. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

18. **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.

19. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 140201, Part No. 258976