ModelSim®

SE

User’s Manual

Version 5.8c

Published: 5/Mar/04

The world’s most popular HDL simulator

Model Technology
8005 Boeckman Road, Bldg. E4
Wilsonville, OR 97070 USA

phone: (503) 685-0820
fax: (503) 685-0910
e-mail: support@model.com, sales@model.com
home page: http://www.model.com
support page: http://www.model.com/support

Table of Contents

1 - Introduction

Chapter contents

This documentation was written for ModelSim SE for UNIX and Microsoft Windows.

ModelSim graphic interface

While your operating system interface provides the window-management frame, ModelSim controls all internal-window features including menus, buttons, and scroll bars. The resulting simulator interface remains consistent within these operating systems:

- SPARCstation with OpenWindows, OSF/Motif, or CDE
- IBM RISC System/6000 with OSF/Motif
- Hewlett-Packard HP 9000 Series 700 with HP VUE, OSF/Motif, or CDE
- Redhat Linux with KDE or GNOME
- Microsoft Windows 98/Me/NT/2000/XP

Because ModelSim's graphic interface is based on Tcl/TK, you also have the tools to build your own simulation environment. Preference variables and configuration commands (see "Preference variables located in INI files" (UM-617) for details) give you control over the use and placement of windows, menus, menu options, and buttons. See "Tcl and macros (DO files)" (UM-591) for more information on Tcl.

For an in-depth look at ModelSim's graphic interface, see *Chapter 10 - Graphic interface*.

ModelSim modes of operation

Many users run ModelSim interactively–pushing buttons and/or pulling down menus in a series of windows in the GUI (graphic user interface). But there are really three modes of ModelSim operation, the characteristics of which are outlined in the following table.:

ModelSim use mode	Characteristics	How ModelSim is invoked
GUI	interactive; has graphical windows, push-buttons, menus, and a command line in the transcript. Default mode.	via a desktop icon or from the OS command shell prompt. Example: `OS> vsim`
Command-line	interactive command line; no GUI.	with **-c** argument at the OS command prompt. Example: `OS> vsim -c`
Batch	non-interactive batch script; no windows or interactive command line.	at OS command shell prompt using "here document" technique or redirection of standard input. Example: `C:\modeltech> vsim vfiles.v <infile >outfile`

The ModelSim User's Manual focuses primarily on the GUI mode of operation. However, this section provides an introduction to the Command-line and Batch modes.

Command-line mode

In command-line mode ModelSim executes any startup command specified by the Startup (UM-625) variable in the *modelsim.ini* file. If **vsim** (CR-357) is invoked with the **-do <"command_string">** option, a DO file (macro) is called. A DO file executed in this manner will override any startup command in the *modelsim.ini* file.

During simulation a transcript file is created containing any messages to stdout. A transcript file created in command-line mode may be used as a DO file if you invoke the **transcript on** command (CR-278) after the design loads (see the example below). The **transcript on** command writes all of the commands you invoke to the transcript file. For example, the following series of commands results in a transcript file that can be used for command input if *top* is re-simulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages... then execute:

```
transcript on
force clk 1 50, 0 100 -repeat 100
run 500
run @5000
quit -f
```

Rename transcript files that you intend to use as DO files. They will be overwritten the next time you run **vsim** if you don't rename them. Also, simulator messages are already commented out, but any messages generated from your design (and subsequently written to the transcript file) will cause the simulator to pause. A transcript file that contains only valid simulator commands will work fine; comment out anything else with a "#".

Stand-alone tools pick up project settings in command-line mode if they are invoked in the project's root directory. If invoked outside the project directory, stand-alone tools pick up project settings only if you set the **MODELSIM** environment variable to the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

Batch mode

Batch mode is an operational mode that provides neither an interactive command line nor interactive windows. In a UNIX environment, **vsim** can be invoked in batch mode by redirecting standard input using the “here-document” technique. In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are re-directed from and to files.

Here is an example of the "here-document" technique:

```
vsim top <<!
log -r *
run 100
do test.do
quit -f
!
```

Here is an example of a batch mode simulation using redirection of std input and output:

```
c:\modeltech\vsim counter < yourfile > outfile
```

where "yourfile" is a script containing various ModelSim commands.

Standards supported

ModelSim VHDL implements the VHDL language as defined by IEEE Standards 1076-1987, 1076-1993, and 1076-2002. ModelSim also supports the 1164-1993 *Standard Multivalue Logic System for VHDL Interoperability*, and the 1076.2-1996 *Standard VHDL Mathematical Packages* standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specs.

ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364-1995 and 1364-2001. ModelSim Verilog also supports a partial implementation of System Verilog 3.1, Accellera's Extensions to Verilog® (see */<install_dir>/modeltech/docs/ technotes/svlog.note* for implementation details). The Open Verilog International *Verilog LRM version 2.0* is also applicable to a large extent. Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim PE and SE users.

In addition, all products support SDF 1.0 through 3.0, VITAL 2.2b, VITAL'95 – IEEE 1076.4-1995, and VITAL 2000 – IEEE 1076.4-2000.

ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.0.1 reference simulator.

Assumptions

We assume that you are familiar with the use of your operating system and its graphic interface.

We also assume that you have a working knowledge of VHDL, Verilog, and/or SystemC. Although ModelSim is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Finally, we assume that you have worked the appropriate lessons in the *ModelSim Tutorial* and are familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* is available from the ModelSim **Help** menu. The *ModelSim Tutorial* is also available from the Support page of our web site: www.model.com.

Sections in this document

In addition to this introduction, you will find the following major sections in this document:

What is an "Item"

Because ModelSim works with VHDL, Verilog, and System C, an “item” refers to any valid design element in VHDL, Verilog, or SystemC. The word "item" is used whenever a specific language reference is not needed. Depending on the context, “item” can refer to any of the following:

VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, alias, or variable
Verilog	function, module instantiation, named fork, named begin, net, task, register, or variable
SystemC	module instantiation, named fork, named begin, net, task, register, or variable

Text conventions

Text conventions used in this manual include:

italic text	provides emphasis and sets off filenames, pathnames, and design unit names
bold text	indicates commands, command options, menu choices, package and library logical names, as well as variables, dialog box selections, and language keywords
`monospace type`	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: **File > Quit**
path separators	examples will show either UNIX or Windows path separators - use separators appropriate for your operating system when trying the examples
UPPER CASE	denotes file types used by ModelSim (e.g., DO, WLF, INI, MPF, PDF, etc.)

Where to find our documentation

ModelSim documentation is available from our website at www.model.com/support or in the following formats and locations:

Document	Format	How to get it
ModelSim SE Installation & Licensing Guide	paper	shipped with Model*Sim*
	PDF	select **Main window** > **Help** > **SE Documentation**; also available from the Support page of our web site: www.model.com
ModelSim SE Quick Guide (command and feature quick-reference)	paper	shipped with Model*Sim*
	PDF	select **Main window** > **Help** > **SE Documentation**, also available from the Support page of our web site: www.model.com
ModelSim SE Tutorial	PDF, HTML	select **Main window** > **Help** > **SE Documentation**; also available from the Support page of our web site: www.model.com
ModelSim SE User's Manual	PDF, HTML	select **Main window** > **Help** > **SE Documentation**
ModelSim SE Command Reference	PDF, HTML	select **Main window** > **Help** > **SE Documentation**
Foreign Language Interface Reference	PDF, HTML	select **Main window** > **Help** > **SE Documentation**
Std_DevelopersKit User's Manual	PDF	www.model.com/support/documentation/BOOK/sdk_um.pdf The Standard Developer's Kit is for use with Mentor Graphics QuickHDL.
Command Help	ASCII	type `help [command name]` at the prompt in the Main window
Error message help	ASCII	type `verror <msgNum>` at the Main window or shell prompt
Tcl Man Pages (Tcl manual)	HTML	select **Main window** > **Help** > **Tcl Man Pages**, or find *contents.htm* in *\modeltech\docs\tcl_help_html*
Technotes	HTML	select Technotes dropdown on www.model.com/support

Download a free PDF reader with Search

Model Technology's PDF documentation requires an Adobe Acrobat Reader for viewing. The Reader may be installed from the ModelSim CD. It is also available without cost from Adobe at www.adobe.com. Be sure to download the Acrobat Reader with Search to take advantage of the index file supplied with our documentation; the index makes searching for keywords much faster.

Technical support and updates

Support

Model Technology online and email technical support options, maintenance renewal, and links to international support contacts:
www.model.com/support/default.asp

Mentor Graphics support:
www.mentor.com/supportnet

Updates

Access to the most current version of ModelSim:
www.model.com/downloads/default.asp

Latest version email

Place your name on our list for email notification of news and updates:
www.model.com/products/informant.asp

2 - Projects

Chapter contents

This chapter discusses ModelSim projects. Projects simplify the process of compiling and simulating a design and are a great tool for getting started with ModelSim.

Introduction

What are projects?

Projects are collection entities for HDL/SystemC designs under specification or test. At a minimum, projects have a root directory, a work library, and "metadata" which are stored in a *.mpf* file located in a project's root directory. The metadata include compiler switch settings, compile order, and file mappings. Projects may also include:

- HDL and SystemC source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see "Creating a Simulation Configuration" (UM-44))
- Folders (see "Organizing projects with folders" (UM-46))

▲ **Important:** Project metadata are updated and stored *only* for actions taken within the project itself. For example, if you have a file in a project, and you compile that file from the command line rather than using the project menu commands, the project will not update to reflect any new compile settings.

What are the benefits of projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don’t need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project. Compile order is maintained for HDL-only designs.
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to HDL/SystemC source files
- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload the initial settings from the project *.mpf* file every time the project is opened

Project conversion between versions

Projects are generally not backwards compatible for either number or letter releases. When you open a project created in an earlier version (e.g, you're using 5.6 and you open a project created in 5.5), you'll see a message warning that the project will be converted to the newer version. You have the option of continuing with the conversion or cancelling the operation.

As stated in the warning message, a backup of the original project is created before the conversion occurs. The backup file is named *<project name>.mpf.bak* and is created in the same directory in which the original project is located.

Note: Due to the significant changes, projects created in versions prior to 5.5 cannot be converted automatically. If you created a project in an earlier version, you will need to recreate it in versions later than 5.5. With the new interface even the most complex project should take less than 15 minutes to recreate. Follow the instructions in the ensuing pages to recreate your project.

Getting started with projects

This section describes the four basic steps to working with a project.

Step 1 — Creating a new project (UM-34)

This creates a .mpf file and a working library.

Step 2 — Adding items to the project (UM-35)

Projects can reference or include HDL/SystemC source files, folders for organization, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

Step 3 — Compiling the files (UM-38)

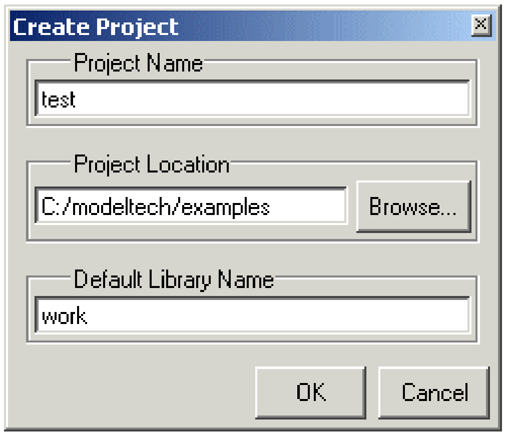
This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.

Step 4 — Simulating a design (UM-39)

This specifies the design unit you want to simulate and opens a structure tab in the Main window workspace.

Step 1 — Creating a new project

Select **File** > **New** > **Project** (Main window) to create a new project. This opens the **Create Project** dialog.



Project Name: test

Project Location: C:/modeltech/examples

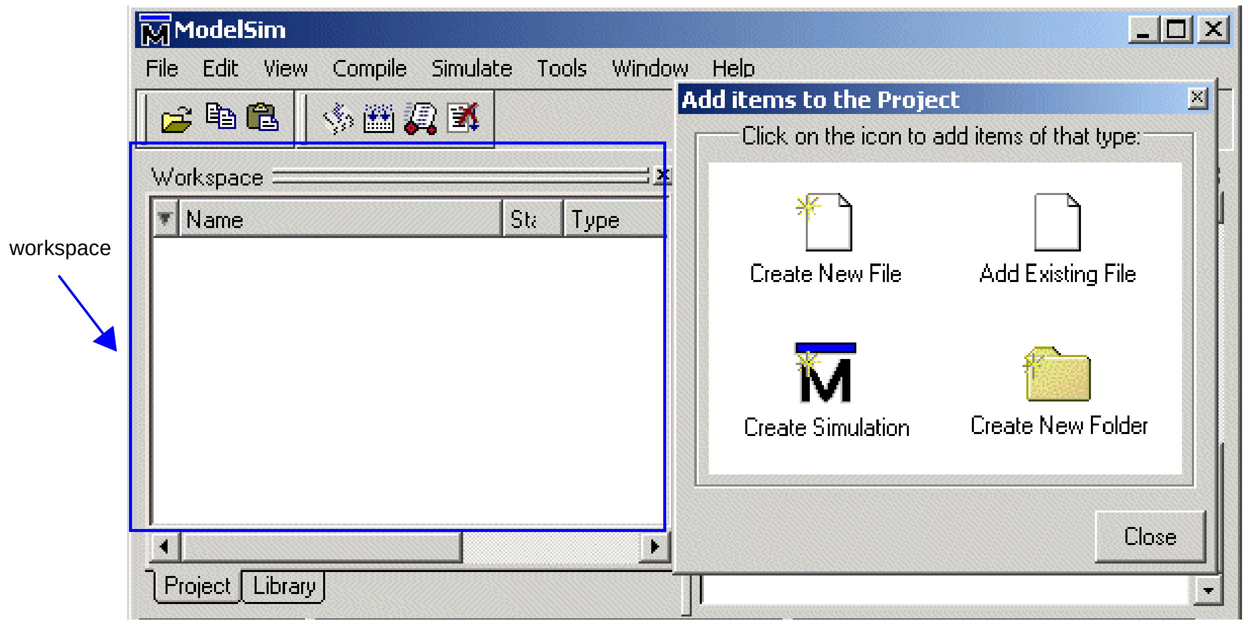
Default Library Name: work

The dialog includes these options:

- **Project Name**
  The name of the new project.
- **Project Location**
  The directory in which the *.mpf* file will be created.

- **Default Library Name**
  The name of the working library. See "Working library versus resource libraries" (UM-54) for more details on work libraries. You can generally leave the **Default Library Name** set to "work." The name you specify will be used to create a working library subdirectory within the Project Location.

After selecting OK, you will see a blank Project tab in the workspace area of the Main window and the **Add Items to the Project** dialog.



The name of the current project is shown at the bottom left corner of the Main window.

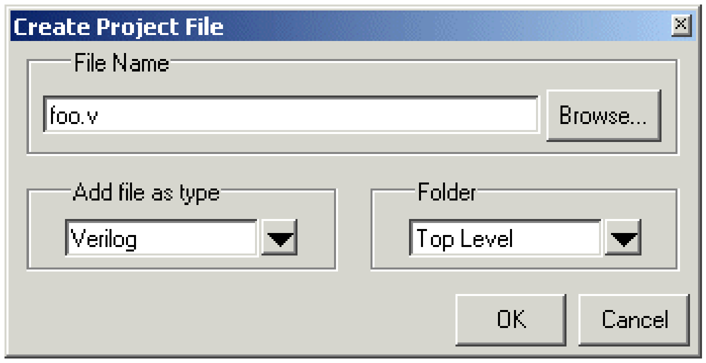
Step 2 — Adding items to the project

The **Add Items to the Project** dialog includes these options:

- **Create New File**
  Create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source window. See below for details.
- **Add Existing File**
  Add an existing file. See below for details.
- **Create Simulation**
  Create a Simulation Configuration that specifies source files and simulator options. See "Creating a Simulation Configuration" (UM-44) for details.
- **Create New Folder**
  Create an organization folder. See "Organizing projects with folders" (UM-46) for details.

Create New File

The **Create New File** command lets you create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source window. You can also access this command by selecting **File > Add to Project > New File** (Main window) or right-clicking (2nd button in Windows; 3rd button in UNIX) in the Project tab and selecting **Add to Project > New File**.


Create Project File
File Name: foo.v
Add file as type: Verilog
Folder: Top Level

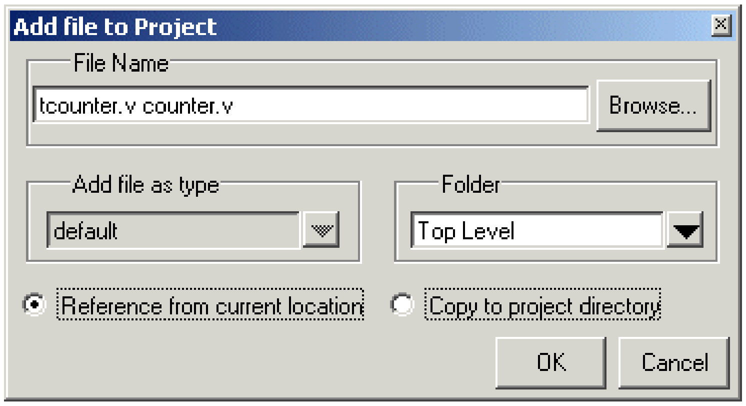
The **Create Project File** dialog includes these options:

- **File Name**
  The name of the new file.
- **Add file as type**
  The type of the new file. Select VHDL, Verilog, SystemC, TCL, or text.
- **Folder**
  The organization folder in which you want the new file placed. You must first create folders in order to access them here. See "Organizing projects with folders" (UM-46) for details.

When you select OK, the file is listed in the Project tab of the Main window workspace.

Add Existing File

You can also access this command by selecting **File > Add to Project > Existing File** (Main window) or by right-clicking (2nd button in Windows; 3rd button in UNIX) in the Project tab and selecting **Add to Project > Existing File**.



File Name: tcounter.v counter.v

Add file as type: default

Folder: Top Level

[✓ Reference from current location] Copy to project directory

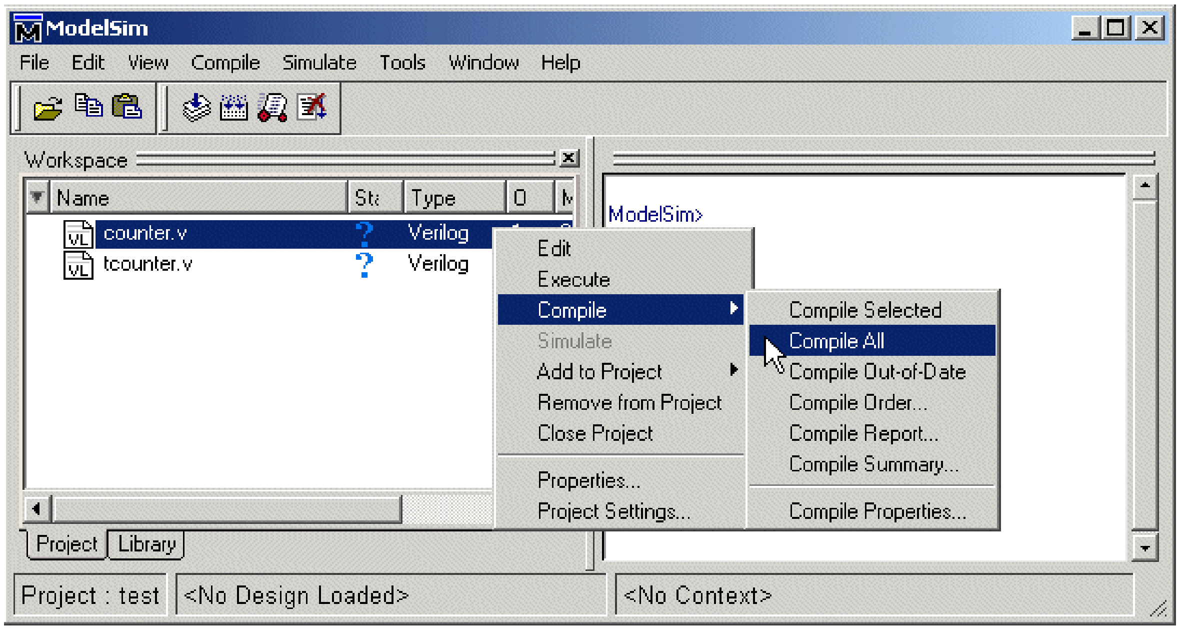
The **Add file to Project** dialog includes these options:

- **File Name**
  The name of the file to add. You can add multiple files at one time.
- **Add file as type**
  The type of the file. "Default" assigns type based on the file extension (e.g., *.v* is type Verilog).
- **Folder**
  The organization folder in which you want the file placed. You must first create folders in order to access them here. See "Organizing projects with folders" (UM-46) for details.
- **Reference from current location/Copy to project directory**
  Choose whether to reference the file from its current location or to copy it into the project directory.

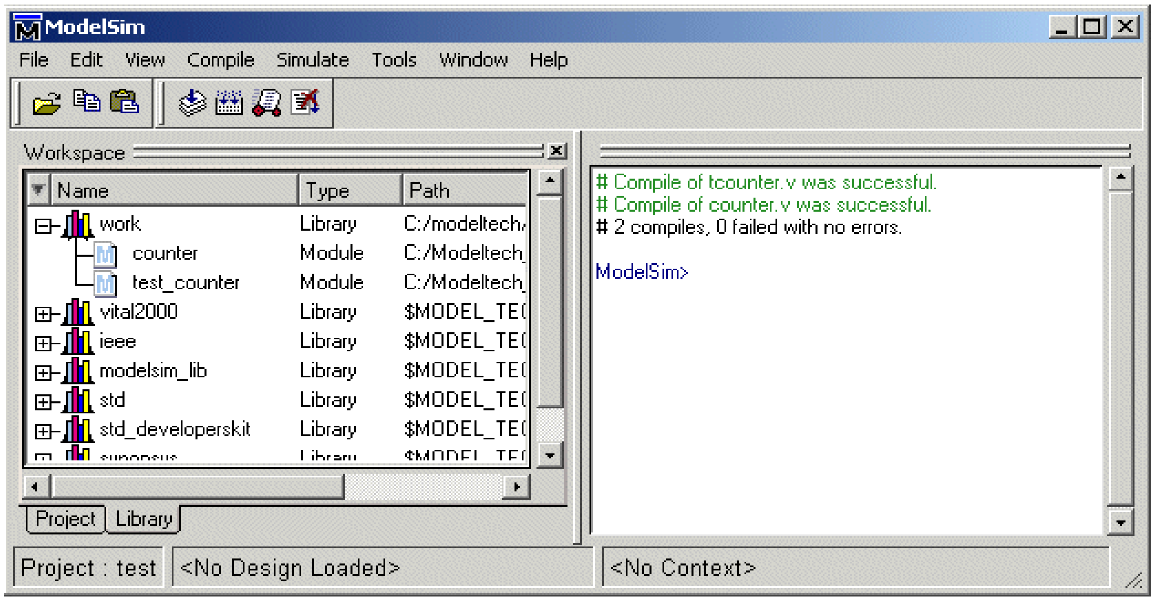
When you select OK, the file(s) is listed in the Project tab of the Main window workspace.

Step 3 — Compiling the files

The question marks next to the files in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile > Compile All** (Main window) or right click in the Project tab and select **Compile > Compile All**.



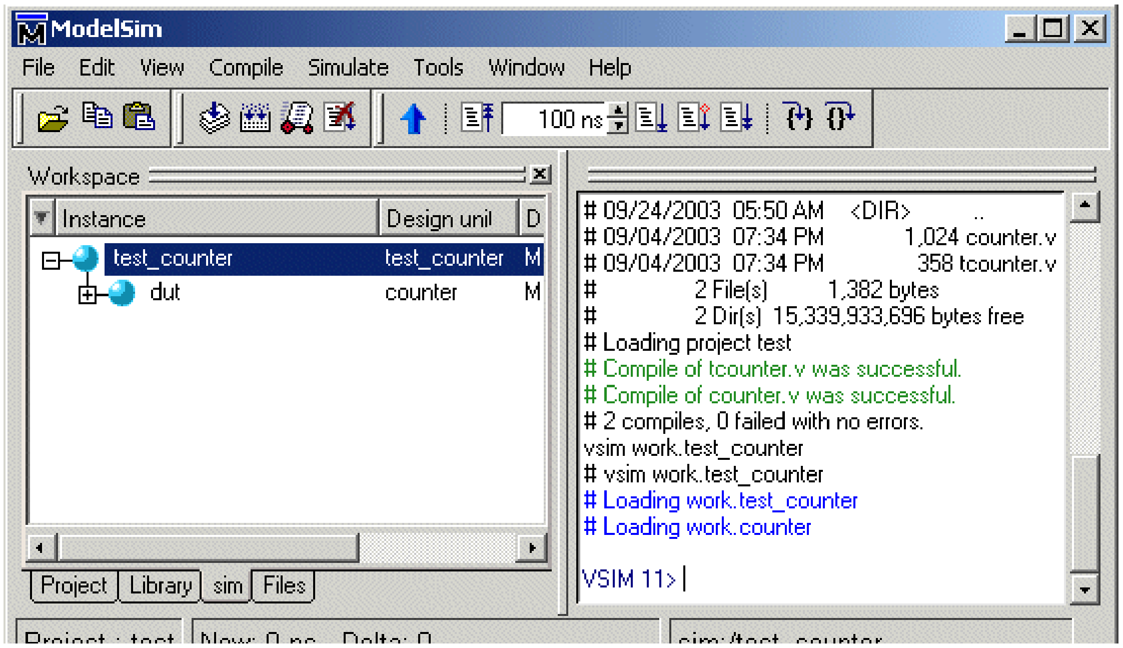
Once compilation is finished, click the Library tab, expand library *work* by clicking the "+", and you will see the compiled design units.



```
# Compile of tcounter.v was successful.
# Compile of counter.v was successful.
# 2 compiles, 0 failed with no errors.

ModelSim>
```

Step 4 — Simulating a design

To simulate one of the designs, either double-click the name or right-click the name and select Simulate. A new tab named *sim* appears that shows the structure of the active simulation.



```
# 09/24/2003  05:50 AM    <DIR>          ..
# 09/04/2003  07:34 PM             1,024 counter.v
# 09/04/2003  07:34 PM               358 tcounter.v
#                2 File(s)          1,382 bytes
#                2 Dir(s)  15,339,933,696 bytes free
# Loading project test
# Compile of tcounter.v was successful.
# Compile of counter.v was successful.
# 2 compiles, 0 failed with no errors.
vsim work.test_counter
# vsim work.test_counter
# Loading work.test_counter
# Loading work.counter

VSIM 11> |
```

At this point you are ready to run the simulation and analyze your results. You often do this by adding signals to the Wave window and running the simulation for a given period of time. See the *ModelSim Tutorial* for examples.

Other basic project operations

Open an existing project

If you previously exited ModelSim with a project open, ModelSim automatically will open that same project upon startup. You can open a different project by selecting **File** > **Open** > **Project** (Main window).

Close a project

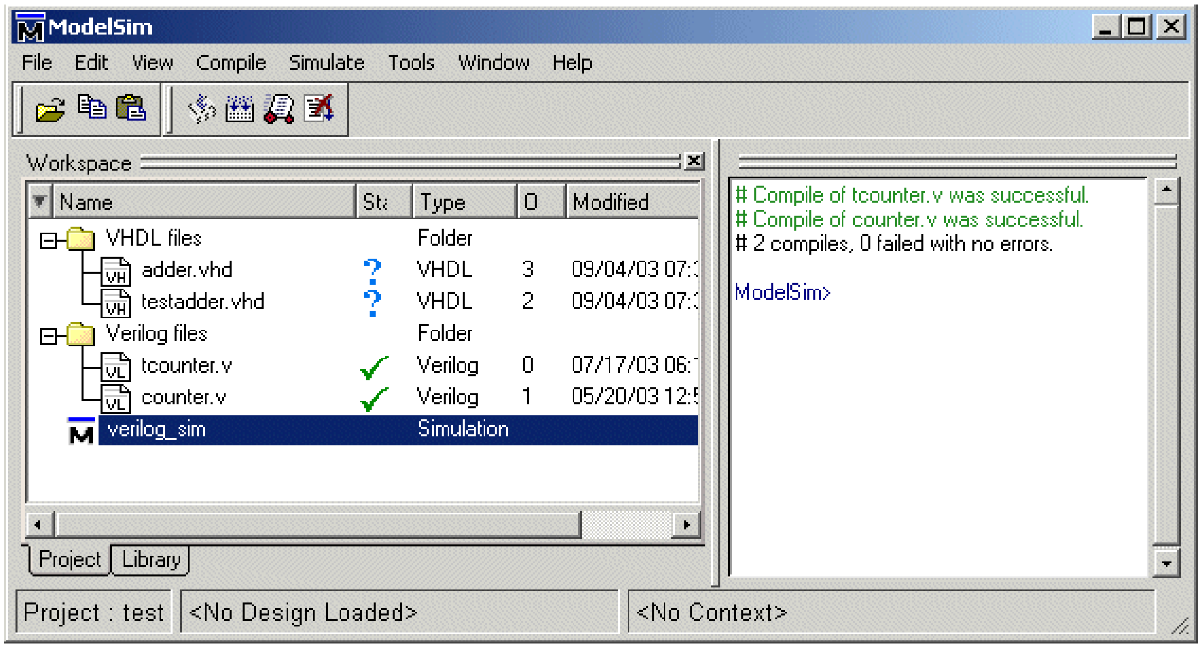
Select **File** > **Close** > **Project** (Main window) or right-click in the Project tab and select **Close Project**. This closes the Project tab but leaves the Library tab open in the workspace. Note that you cannot close a project while a simulation is in progress.

Delete a project

Select **File** > **Delete** > **Project** (Main window). You cannot delete a project while it is open.

The Project tab

The Project tab contains information about the items in your project. By default the tab is divided into five columns.



Name	St	Type	O	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	09/04/03 07:
testadder.vhd	?	VHDL	2	09/04/03 07:
Verilog files		Folder		
tcounter.v	✓	Verilog	0	07/17/03 06:
counter.v	✓	Verilog	1	05/20/03 12:
verilog_sim		Simulation		

```
# Compile of tcounter.v was successful.
# Compile of counter.v was successful.
# 2 compiles, 0 failed with no errors.

ModelSim>
```

Project : test <No Design Loaded> <No Context>

Name – The name of a file or object.

Status – Identifies whether a source file has been successfully compiled. Applies only to VHDL or Verilog files. A question mark means the file hasn’t been compiled or the source file has changed since the last successful compile; an X means the compile failed; a check mark means the compile succeeded; a checkmark with a yellow triangle behind it means the file compiled but there were warnings generated.

Type – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.

Order – The order in which the file will be compiled when you execute a Compile All command.

Modified – The date and time of the last modification to the file.

You can hide or show columns by right-clicking on a column title and selecting or deselecting entries.

Sorting the list

You can sort the list by any of the five columns. Click on a column heading to sort by that column; click the heading again to invert the sort order. An arrow in the column heading indicates which field the list is sorted by and whether the sort order is descending (down arrow) or ascending (up arrow).

Project tab context menu

Like the other workspace tabs, the Project tab has a context menu that you access by clicking your right mouse button (2nd button in Windows; 3rd button in UNIX) anywhere in the tab. The context menu has the following commands:

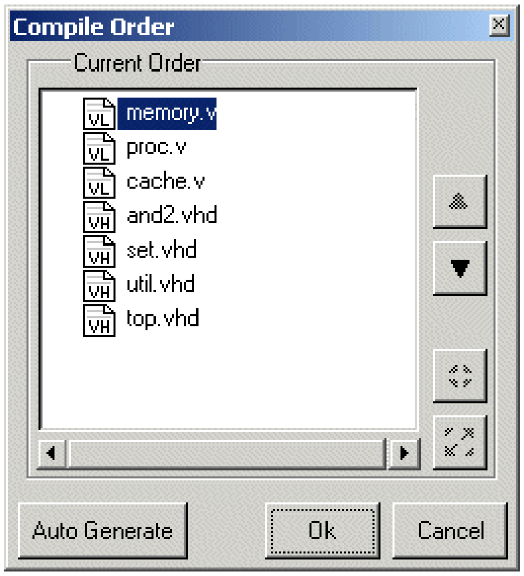
Edit	open the selected file in an editor
Execute	execute the selected Verilog, VHDL, WLF, or DO file
Compile	provides these options: Compile Selected – compile the selected file(s); note that if you select a folder and select **Compile Selected**, it will compile all files in the folder and any sub-folders Compile All – compile all source files included in the project Compile Out-of-Date – compile source files that have been modified since the last compile Compile Order – set compile order for all files in the project; see "Changing compile order" (UM-42) for more details. Compile Order is not supported for SystemC files. Compile Report – show the compilation history of the selected file Compile Summary – show the compilation history of the entire project Compile Properties – view/change project compiler settings for the selected source file(s)
Simulate	load the design unit(s) and associated simulation options from the selected Simulation Configuration; see "Creating a Simulation Configuration" (UM-44) for more details
Add to Project	provides these options: New File – add a new file to the project Existing File – add an existing file to the project Simulation Configuration – create a new Simulation Configuration; see "Creating a Simulation Configuration" (UM-44) for more details Folder – add an organization folder to the project; see "Organizing projects with folders" (UM-46) for more details
Remove from Project	remove the selected item from the project
Close Project	close the active project
Properties	view/change compiler settings for the selected source file(s)
Project Settings	change settings for the project; see "Project settings" (UM-49)

Changing compile order

The Compile Order dialog box is functional for HDL-only designs. When you compile all files in a project, ModelSim by default compiles the files in the order in which they were added to the project. You have two alternatives for changing the default compile order: 1) select and compile each file individually; 2) specify a custom compile order.

To specify a custom compile order, follow these steps:

1 Select **Compile** > **Compile Order** (Main window) or select it from the context menu in the Project tab.



2 Drag the files into the correct order or use the up and down arrow buttons. Note that you can select multiple files and drag them simultaneously.

Auto-generating compile order

Auto Generate is supported for HDL-only designs. The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct 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.

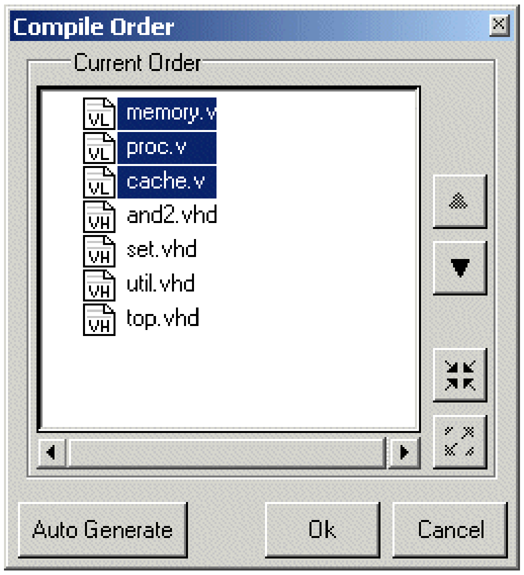
Files can be displayed in the Project tab in alphabetical or compile order (using the **Sort by Alphabetical Order** or **Sort by Compile Order** commands on the context menu). Keep in mind that the order you see in the Project tab is not necessarily the order in which the files will be compiled.

Grouping files

You can group two or more files in the Compile Order dialog so they are sent to the compiler at the same time. For example, you might have one file with a bunch of Verilog define statements and a second file that is a Verilog module. You would want to compile these two files together.

To group files, follow these steps:

1 Select the files you want to group.



2 Click the Group button.

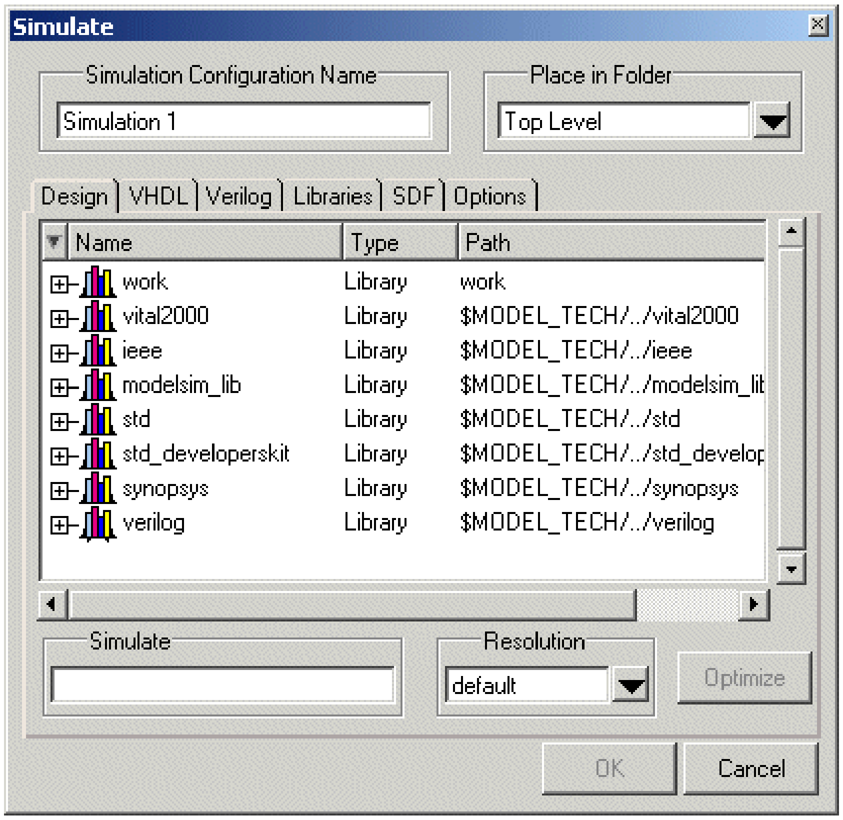
To ungroup files, select the group and click the Ungroup button.

Creating a Simulation Configuration

A Simulation Configuration associates a design unit(s) and its simulation options. For example, say you routinely load a particular design and you have to specify the simulator resolution, generics, and SDF timing files. Ordinarily you would have to specify those options each time you load the design. With a Simulation Configuration, you would specify the design and those options and then save the configuration with a name (e.g., *top_config*). The name is then listed in the Project tab and you can double-click it to load the design along with its options.

To create a Simulation Configuration, follow these steps:

1 Select **File > Add to Project > Simulation Configuration** (Main window) or select it from the context menu in the Project tab.



Simulation Configuration Name: Simulation 1

Place in Folder: Top Level

Name	Type	Path
work	Library	work
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lit
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_develop
synopsys	Library	$MODEL_TECH/../synopsys
verilog	Library	$MODEL_TECH/../verilog

Resolution: default

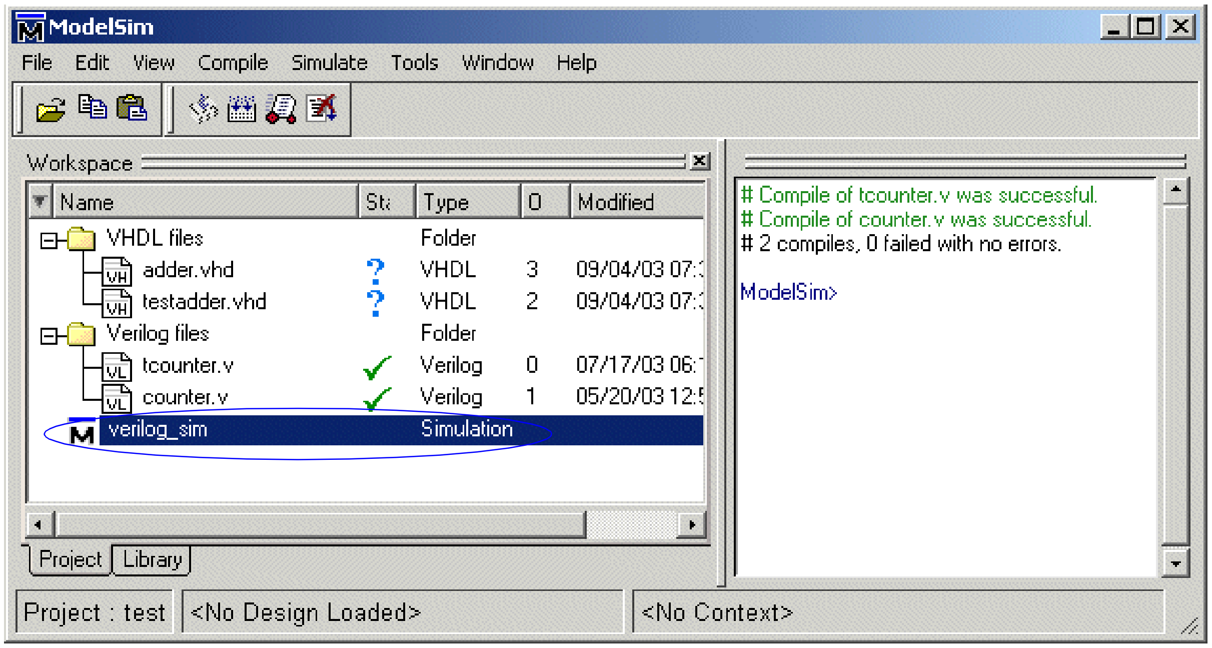
2 Specify a name in the **Simulation Configuration Name** field.

3 Specify the folder in which you want to place the configuration (see "Organizing projects with folders" (UM-46)).

4 Select one or more design unit(s). Use the Control and/or Shift keys to select more than one design unit. The design unit names appear in the **Simulate** field when you select them.

5 Use the other tabs in the dialog to specify any required simulation options. All of the options in this dialog are described under "Simulating with the graphic interface" (UM-377).

Click OK and the simulation configuration is added to the Project tab.



Name	St	Type	O	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	09/04/03 07:
testadder.vhd	?	VHDL	2	09/04/03 07:
Verilog files		Folder		
tcounter.v	✓	Verilog	0	07/17/03 06:
counter.v	✓	Verilog	1	05/20/03 12:
verilog_sim		Simulation		

```
# Compile of tcounter.v was successful.
# Compile of counter.v was successful.
# 2 compiles, 0 failed with no errors.

ModelSim>
```

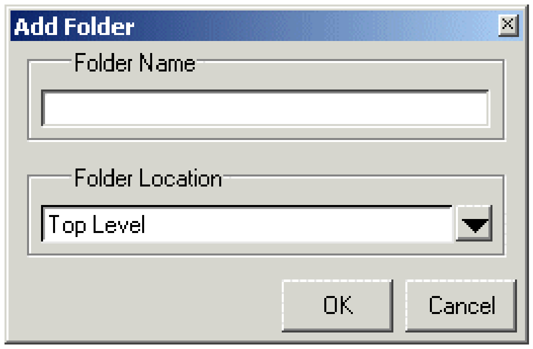
Double-click the Simulation Configuration item to load it.

Organizing projects with folders

The more files you add to a project, the harder it can be to locate the item you need. You can add "folders" to the project to organize your files. These folders are akin to directories in that you can have multiple levels of folders and sub-folders. However, no actual directories are created via the file system–the folders are present only within the project file.

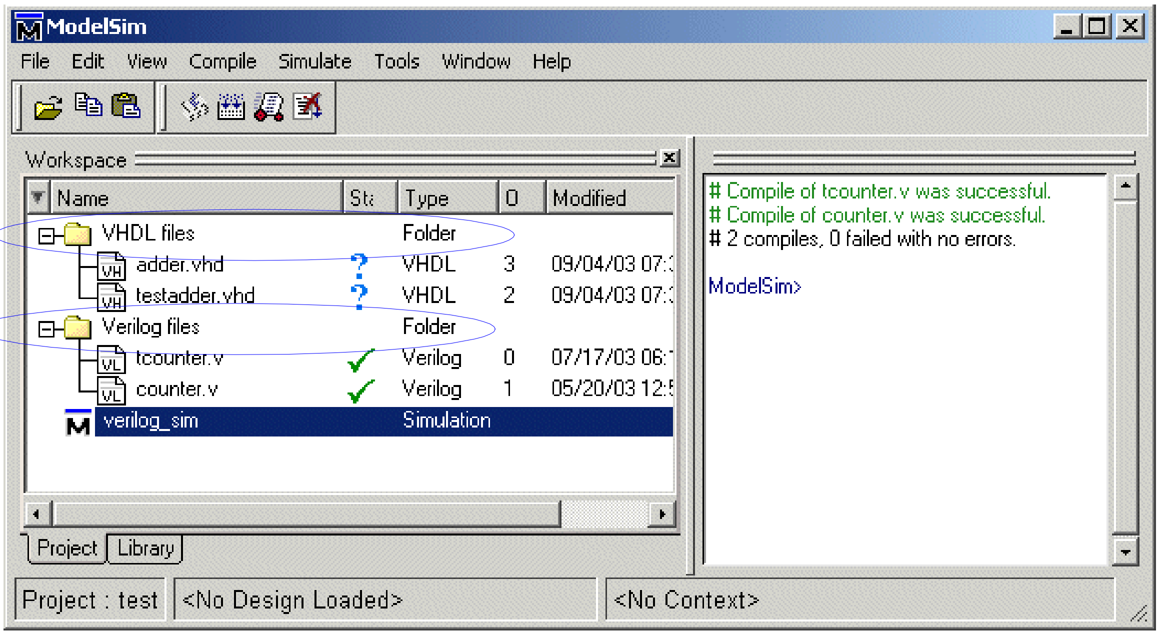
Adding a folder

To add a folder to your project, select **File > Add to Project > Folder** or right-click in the Project tab and select **Add to Project > Folder**.



Folder Location: Top Level

Specify the Folder Name, the location for the folder, and click OK. The folder will be displayed in the Project tab.

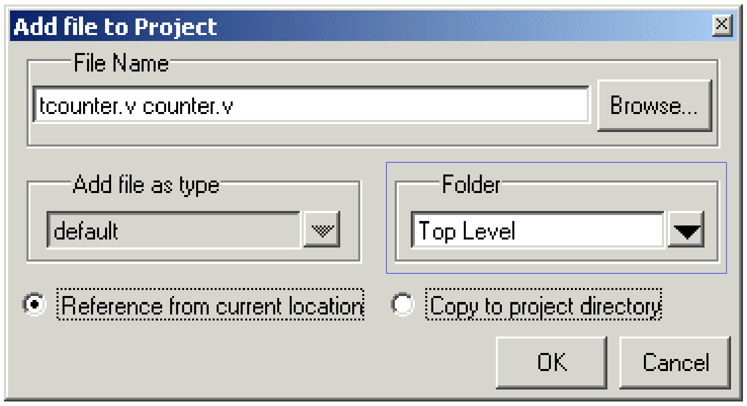


Name	St	Type	O	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	09/04/03 07::
testadder.vhd	?	VHDL	2	09/04/03 07::
Verilog files		Folder		
tcounter.v	✓	Verilog	0	07/17/03 06::
counter.v	✓	Verilog	1	05/20/03 12:5
verilog_sim		Simulation		

```
# Compile of tcounter.v was successful.
# Compile of counter.v was successful.
# 2 compiles, 0 failed with no errors.

ModelSim>
```

Project : test | <No Design Loaded> | <No Context>

You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.



Add file to Project

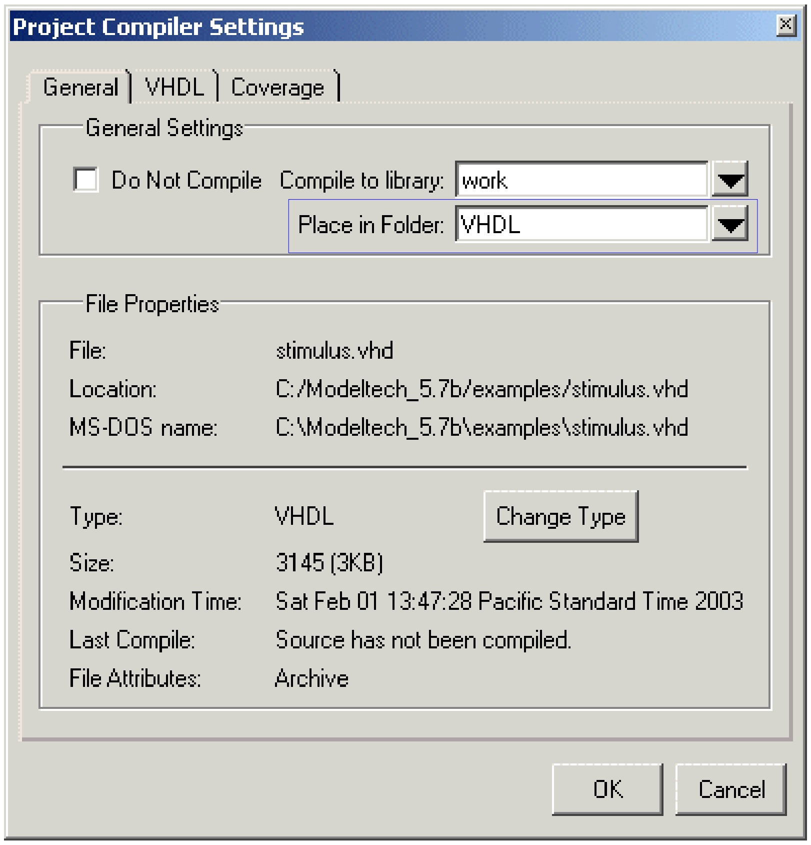
File Name: tcounter.v counter.v

Add file as type: default

Folder: Top Level

[✓ Reference from current location] Copy to project directory

If you want to move a file into a folder later on, you can do so using the Properties dialog for the file (right-click on the file and select Properties from the context menu).



Project Compiler Settings

General Settings

Do Not Compile: ☐

Compile to library: work

Place in Folder: VHDL

File Properties

File:	stimulus.vhd
Location:	C:/Modeltech_5.7b/examples/stimulus.vhd
MS-DOS name:	C:\Modeltech_5.7b\examples\stimulus.vhd
Type:	VHDL
Size:	3145 (3KB)
Modification Time:	Sat Feb 01 13:47:28 Pacific Standard Time 2003
Last Compile:	Source has not been compiled.
File Attributes:	Archive

On Windows platforms, you can also just drag-and-drop a file into a folder.

Specifying file properties and project settings

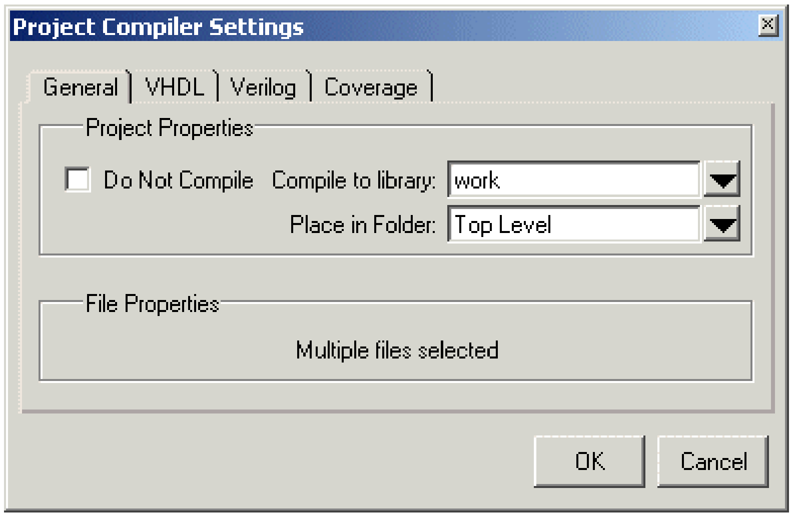
You can set two types of properties in a project: file properties and project settings. File properties affect individual files; project settings affect the entire project.

File compilation properties

The VHDL and Verilog compilers (**vcom** and **vlog**, respectively) have numerous options that affect how a design is compiled and subsequently simulated. You can customize the settings on individual files or a group of files.

Important: Any changes you make to the compile properties outside of the project, whether from the command line, the GUI, or the *modelsim.ini* file, *will not* affect the properties of files already in the project.

To customize specific files, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting dialog varies depending on the number and type of files you have selected. If you select a single VHDL, Verilog file, you'll see the General tab and the VHDL or Verilog tab, respectively. If you select a SystemC file, you will see only the General tab. On the General tab, you'll see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you'll see all four tabs but no file information on the General tab.



Do Not Compile [ ]   Compile to library: work
Place in Folder: Top Level

File Properties
Multiple files selected

The General tab includes these options:

- **Do Not Compile**
  Determines whether the file is excluded from the compile.
- **Compile to library**
  Specifies to which library you want to compile the file; defaults to the working library.
- **Place in Folder**
  Specifies the folder in which to place the selected file(s). See "Organizing projects with folders" (UM-46) for details on folders.

- **File Properties**
  A variety of information about the selected file (e.g, type, size, path). Displays only if a single file is selected in the Project tab.

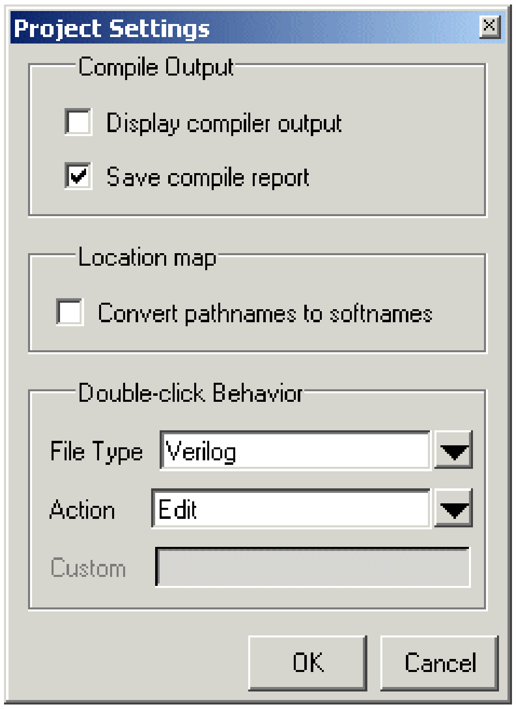
The definitions of the options on the VHDL and Verilog tabs can be found in the section "Setting default compile options" (UM-370). The definitions for the options on the Coverage tab can be found in the section "Enabling Code Coverage" (UM-423).

When setting options on a group of files, keep in mind the following:

- If two or more files have different settings for the same option, the checkbox in the dialog will be "grayed out." If you change the option, you cannot change it back to a "multi- state setting" without cancelling out of the dialog. Once you click OK, ModelSim will set the option the same for all selected files.
- If you select a combination of VHDL and Verilog files, the options you set on the VHDL and Verilog tabs apply only to those file types.

Project settings

To modify project settings, right-click anywhere within the Project tab and select **Project Settings**.


Project Settings

Compile Output
☐ Display compiler output
☑ Save compile report

Location map
☐ Convert pathnames to softnames

Double-click Behavior
File Type: Verilog
Action: Edit
Custom:

The **Project Settings** dialog includes these options:

- **Display compiler output**
  Prints verbose compile output to the Transcript. By default verbose output is produced in the Compile Report only.
- **Save compile report**
  Saves verbose compile output to disk. You can access the report by right-clicking a file and selecting **Compile > Compile Report**.

- **Location map**
  Specifies whether physical paths for the project files should be saved as soft paths if they are present in the location map. See "Referencing source files with location maps" (UM-66) for more details on using location maps.
- **Double-click Behavior**
  Specifies the action to take when you double-click a type of file. If you select Custom, you can specify a Tcl command in the text box below the file type.

  You can use *%f* for filename substitution. For example, if you wanted double click on a Tcl file to open the file with Notepad, you would insert the following in the text box:

  ```
  notepad %f
  ```

  ModelSim will substitute the *%f* with the filename that was clicked on, then execute the string.

Accessing projects from the command line

Generally, projects are used from within the ModelSim GUI. However, standalone tools will use the project file if they are invoked in the project's root directory. If you want to invoke outside the project directory, set the **MODELSIM** environment variable with the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

You can also use the **project** command (CR-227) from the command line to perform common operations on projects.

3 - Design libraries

Chapter contents

VHDL contains *libraries*, which are objects that contain compiled design units; libraries are given names so they may be referenced. Verilog designs simulated within ModelSim are compiled into libraries as well.

Design library overview

A *design library* is a directory or archive that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, and configurations; Verilog modules and UDPs (user-defined primitives); and SystemC modules. The design units are classified as follows:

- **Primary design units**
  Consist of entities, package declarations, configuration declarations, modules, UDPs, and SystemC modules. Primary design units within a given library must have unique names.
- **Secondary design units**
  Consist of architecture bodies, package bodies, and optimized Verilog modules. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities or modules.

Design unit information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

Working library versus resource libraries

Design libraries can be used 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).

Only one library can be the working library. In contrast any number of libraries can be resource libraries during a compilation. 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 (see "Specifying the resource libraries" (UM-61)).

A common example of using both a working library and a resource library is one where your gate-level design and testbench are compiled into the working library, and the design references gate-level models in a separate resource library.

The library named **work** has special attributes within ModelSim; it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units (i.e., it doesn't need to be mapped). In other words the **work** library is the default *working* library.

Archives

By default, design libraries are stored in a directory structure with a sub-directory for each design unit in the library. Alternatively, you can configure a design library to use archives. In this case each design unit is stored in its own archive file. To create an archive, use the **-archive** argument to the **vlib** command (CR-344).

Generally you would do this only in the rare case that you hit the reference count limit on I-nodes due to the ".." entries in the lower-level directories (the maximum number of sub-directories on UNIX and Linux is 65533). An example of an error message that is produced when this limit is hit is:

```
mkdir: cannot create directory `65534': Too many links
```

Archives may also have limited value to customers seeking disk space savings.

Note that GMAKE won’t work with these archives on the IBM platform.

Working with design libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within ModelSim, design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception; extended identifiers are not supported for library names.

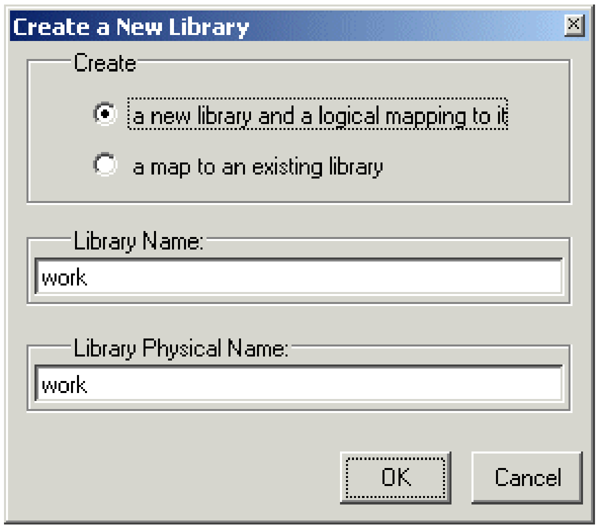
Creating a library

When you create a project (see "Getting started with projects" (UM-34)), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface.

From the ModelSim prompt or a UNIX/DOS prompt, use this **vlib** command (CR-344):

```
vlib <directory_pathname>
```

To create a new library with the ModelSim graphic interface, select **File > New > Library** (Main window).


Create a New Library

Create

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

a map to an existing library

Library Name: work

Library Physical Name: work

The **Create a New Library** dialog box includes these options:

- **Create a new library and a logical mapping to it**
  Type the new library name into the **Library Name** field. This creates a library sub-directory in your current working directory, initially mapped to itself. Once created, the mapped library is easily remapped to a different library.
- **Create a map to an existing library**
  Type the new library name into the **Library Name** field, then type into the **Library Maps to** field or **Browse** to select a library name for the mapping.
- **Library Name**
  Type the logical name of the new library into this field.

- **Library Physical Name**
  Type the physical name of the new library into this field. ModelSim will create a directory with this name.
- **Library Maps to**
  Type or **Browse** for a mapping for the specified library. This field is visible and can be changed only when the **Create a map to an existing library** option is selected.

When you click **OK**, ModelSim creates the specified library directory and writes a specially-formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a ModelSim library.

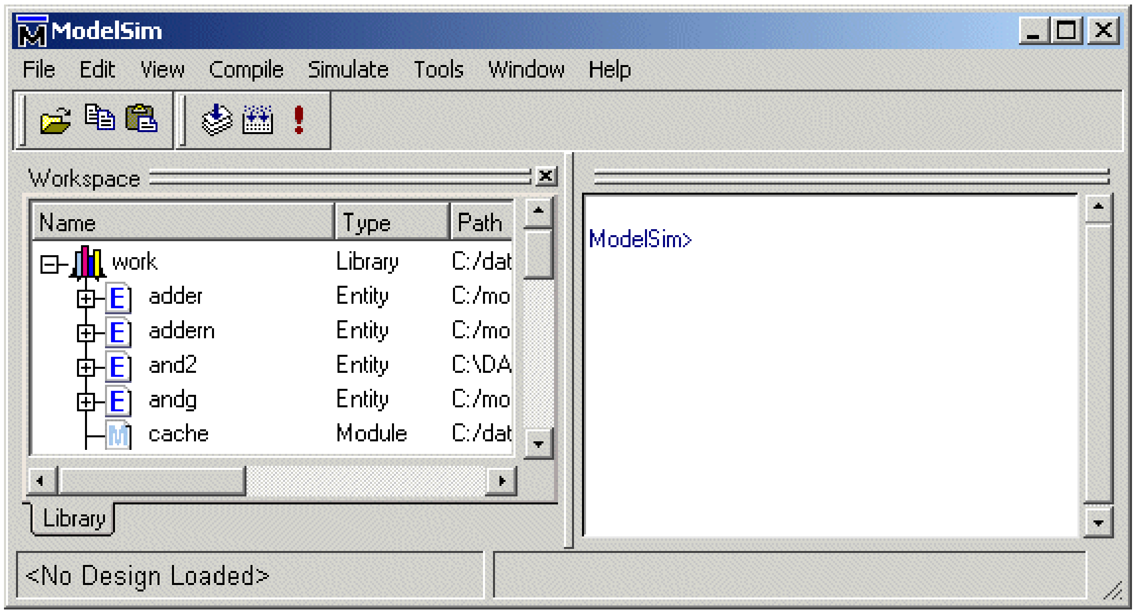
The new map entry is written to the *modelsim.ini* file in the [Library] section. See "[Library] library path variables" (UM-617) for more information.

Note: Remember that a design library is a special kind of directory; the only way to create a library is to use the ModelSim GUI or the **vlib** command (CR-344). Do not create libraries using UNIX or Windows commands.

Managing library contents

Library contents can be viewed, deleted, recompiled, edited and so on using either the graphic interface or command line.

The Library tab in the Main window workspace provides access to design units (configurations, modules, packages, entities, architectures, and SystemC modules) in a library. The listing is organized hierarchically, and the unit types are identified both by icon (entity (E), module (M), and so forth) and the Type column.



Name	Type	Path
work	Library	C:/dat
adder	Entity	C:/mo
addern	Entity	C:/mo
and2	Entity	C:\DA
andg	Entity	C:/mo
cache	Module	C:/dat

<No Design Loaded>

The Library tab has a context menu that you access by clicking your right mouse button (Windows—2nd button, UNIX—3rd button) in the Library tab.

The context menu includes the following commands:

- **Simulate**
  Loads the selected design unit and opens structure and Files tabs in the workspace. Related command line command is **vsim** (CR-357).
- **Edit**
  Opens the selected design unit in the Source window, or if a library is selected, opens the Edit Library Mapping dialog (see "Library mappings with the GUI" (UM-59)).
- **Refresh**
  Rebuilds the library image of the selected library without using source code. Related command line command is **vcom** (CR-303) or **vlog** (CR-345) with the **-refresh** argument.
- **Recompile**
  Recompiles the selected design unit. Related command line command is **vcom** (CR-303) or **vlog** (CR-345).
- **Optimize**
  Optimizes a Verilog design unit. Related command line command is **vlog** (CR-345) with the **+opt** argument. See "Compiling with +opt" (UM-128) for further details.
- **Update**
  Updates the display of available libraries and design units.
- **Delete**
  Deletes the selected design unit. Related command line command is **vdel** (CR-315).

  Deleting a package, configuration, or entity will remove the design unit from the library. If you delete an entity that has one or more architectures or a Verilog module that has one or more optimized versions, the entity and all its associated architectures or the module and all its optimized versions will be deleted.

  You can also delete an architecture without deleting its associated entity. Expand the entity, right-click the desired architecture name, and select Delete. You are prompted for confirmation before any design unit is actually deleted.
- **New**
  Create a new library.
- **Properties**
  Displays various properties (e.g., Name, Type, Source, etc.) of the selected design unit or library.

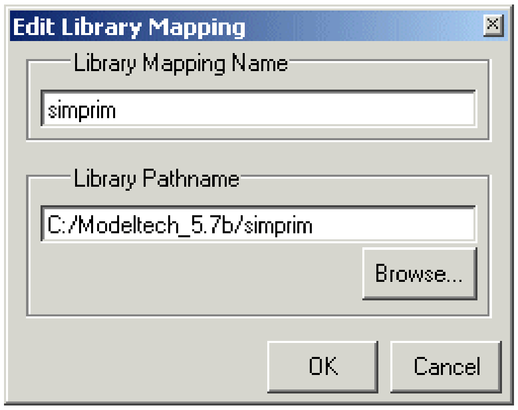
Assigning a logical name to a design library

VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the GUI, a command, or a project to assign a logical name to a design library.

Library mappings with the GUI

To associate a logical name with a library, select the library in the workspace, right-click and select **Edit** from the context menu. This brings up a dialog box that allows you to edit the mapping.



Library Mapping Name: simprim

Library Pathname: C:/Modeltech_5.7b/simprim

The dialog box includes these options:

- **Library Mapping Name**
  The logical name of the library.
- **Library Pathname**
  The pathname to the library.

Library mapping from the command line

You can issue a command to set the mapping between a logical library name and a directory; its form is:

```
vmap <logical_name> <directory_pathname>
```

You may invoke this command from either a UNIX/DOS prompt or from the command line within ModelSim.

The **vmap** (CR-356) command adds the mapping to the library section of the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, use a text editor and add a line under the [Library] section heading using the syntax:

```
<logical_name> = <directory_pathname>
```

More than one logical name can be mapped to a single directory. For example, suppose the *modelsim.ini* file in the current working directory contains following lines:

```
[Library]
work = /usr/rick/design
my_asic = /usr/rick/design
```

This would allow you to use either the logical name **work** or **my_asic** in a **library** or **use** clause to refer to the same design library.

Unix symbolic links

You can also create a UNIX symbolic link to the library using the host platform command:

```
ln -s <directory_pathname> <logical_name>
```

The **vmap** command (CR-356) can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

```
vmap <logical_name>
```

Library search rules

The system searches for the mapping of a logical name in the following order:

- First the system looks for a *modelsim.ini* file.
- If the system doesn't find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, the system searches the current working directory for a subdirectory that matches the logical name.

An error is generated by the compiler if you specify a logical name that does not resolve to an existing directory.

Moving a library

Individual design units in a design library cannot be moved. An *entire* design library can be moved, however, by using standard operating system commands for moving a directory or an archive.

Setting up libraries for group use

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search the library section of the initialization file specified by the "others" clause. For example:

```
[library]
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

Specifying the resource libraries

Verilog resource libraries

ModelSim supports and encourages separate compilation of distinct portions of a Verilog design. The **vlog** (CR-345) compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs that are referenced by the simulator as it loads the design. See "Library usage" (UM-111).

▲ **Important:** Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to **vlog** (CR-345).

VHDL resource libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The **vcom** command (CR-303) adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, you can use **vcom** **-work** and specify the name of the desired target library.

Default binding rules for VHDL resource libraries

A common question related to VHDL resource libraries is how ModelSim handles default binding for components. ModelSim addresses default binding at compile time. When looking for an entity to bind with, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE 1076-1987 contained a flaw that made it almost impossible for an entity to be directly visible if it had the same name as the component. In short, if a component was declared in an architecture, any like-named entity above that declaration would be hidden because component/entity names cannot be overloaded. As a result we implemented the following rules for determining default binding:

- If a directly visible entity has the same name as the component, use it.
- If the component is declared in a package, search the library that contained the package for an entity with the same name.

If neither of these methods is successful, ModelSim will also do the following:

- Search the work library.
- Search all other libraries that are currently visible by means of the **library** clause.

Note that these second two searches are an extension to the 1076 standard.

Predefined libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual, Std 1076*. See also, "Using the TextIO package" (UM-86).

A VHDL **use** clause can be specified to select particular declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source.

By default, every VHDL design unit is assumed to contain the following declarations:

```
LIBRARY std, work;
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, add the suffix *.all* to the library/package name. For example, the **use** clause above specifies that all declarations in the package *standard*, in the design library named *std*, are to be visible to the VHDL design unit immediately following the **use** clause. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

Alternate IEEE libraries supplied

The installation directory may contain two or more versions of the IEEE library:

- *ieeepure*
  Contains only IEEE approved packages (accelerated for ModelSim).
- *ieee*
  Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including math_complex, math_real, numeric_bit, numeric_std, std_logic_1164, std_logic_misc, std_logic_textio, std_logic_arith, std_logic_signed, std_logic_unsigned, vital_primitives, and vital_timing.

You can select which library to use by changing the mapping in the *modelsim.ini* file. The *modelsim.ini* file in the installation directory defaults to the *ieee* library.

Rebuilding supplied libraries

Resource libraries are supplied precompiled in the *modeltech* installation directory. If you need to rebuild these libraries, the sources are provided in the *vhdl_src* directory; a macro file is also provided for Windows platforms (*rebldlibs.do*). To rebuild the libraries, invoke the DO file from within ModelSim with this command:

```
do rbldlibs.do
```

Make sure your current directory is the *modeltech* install directory before you run this file.

▶ **Note:** Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using **vcom** (CR-303) with the **-93** option.

Shell scripts are provided for UNIX (*rebuild_libs.csh* and *rebuild_libs.sh*). To rebuild the libraries, execute one of the *rebuild_libs* scripts while in the *modeltech* directory.

Regenerating your design libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (see "Managing library contents" (UM-57)), or by using the **-refresh** argument to **vcom** (CR-303) and **vlog** (CR-345).

From the command line, you would use vcom with the **-refresh** option to update VHDL design units in a library, and vlog with the **-refresh** option to update Verilog design units. By default, the work library is updated; use **-work <library>** to update a different library. For example, if you have a library named *mylib* that contains both VHDL and Verilog design units:

```
vcom -work mylib -refresh
vlog -work mylib -refresh
```

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of ModelSim (4.6 and later only). In general, this works for moving forwards or backwards on a release. Moving backwards on a release may not work if the models used compiler switches or directives that do not exist in the older release.

▶ **Note:** You don't need to regenerate the *std*, *ieee*, *vital22b*, and *verilog* libraries. Also, you cannot use the **-refresh** option to update libraries that were built before the 4.6 release.

Maintaining 32-bit and 64-bit versions in the same library

It is possible with ModelSim to maintain 32-bit and 64-bit versions of a design in the same library. To do this, compile the design with the 32-bit version and "refresh" the design with the 64-bit version. For example:

Using the 32-bit version of ModelSim:

```
vcom file1.vhd -work asic_lib
vcom file2.vhd -work asic_lib
```

Next, using the 64-bit version of ModelSim:

```
vcom -work asic_lib -refresh
```

This allows you to use either version without having to do a refresh.

Do not compile the design with one version, and then recompile it with the other. If you do this, ModelSim will remove the first module, because it could be "stale."

Protecting source code and using -nodebug

The **-nodebug** argument for both **vcom** (CR-303) and **vlog** (CR-345) hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

▶ **Note: -nodebug** encrypts entire files. The Verilog **`protect** compiler directive allows you to encrypt regions within a file. See "ModelSim compiler directives" (UM-152) for details.

When you compile with **-nodebug**, all source text, identifiers, and line number information are stripped from the resulting compiled object, so ModelSim cannot locate or display any information of the model except for the external pins. Specifically, the Source window will not display the design units' source code, the Structure window will not display the internal structure, the Signals window will not display internal signals, the Process window will not display internal processes, and the Variables window will not display internal variables. In addition, none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands.

You can access the design units comprising your model via the library, and you may invoke **vsim** (CR-357) directly on any of these design units and see the ports. To restrict even this access in the lower levels of your design, you can use the following **-nodebug** options when you compile:

Command and switch	Result
vcom -nodebug=ports	makes the ports of a VHDL design unit invisible
vlog -nodebug=ports	makes the ports of a Verilog design unit invisible
vlog -nodebug=pli	prevents the use of PLI functions to interrogate the module for information
vlog -nodebug=ports+pli	combines the functions of -nodebug=ports and -nodebug=pli

Don't use the **=ports** option on a design without hierarchy, or on the top level of a hierarchical design. If you do, no ports will be visible for simulation. Rather, compile all lower portions of the design with **-nodebug=ports** first, then compile the top level with **-nodebug** alone.

Design units or modules compiled with **-nodebug** can only instantiate design units or modules that are also compiled **-nodebug**.

Referencing source files with location maps

Pathnames to source files are recorded in libraries by storing the working directory from which the compile is invoked and the pathname to the file as specified in the invocation of the compiler. The pathname may be either a complete pathname or a relative pathname.

ModelSim tools that reference source files from the library locate a source file as follows:

- If the pathname stored in the library is complete, then this is the path used to reference the file.
- If the pathname is relative, then the tool looks for the file relative to the current working directory. If this file does not exist, then the path relative to the working directory stored in the library is used.

This method of referencing source files generally works fine if the libraries are created and used on a single system. However, when multiple systems access a library across a network, the physical pathnames are not always the same and the source file reference rules do not always work.

Using location mapping

Location maps are used to replace prefixes of physical pathnames in the library with environment variables. The location map defines a mapping between physical pathname prefixes and environment variables.

ModelSim tools open the location map file on invocation if the MGC_LOCATION_MAP (UM-613) environment variable is set. If MGC_LOCATION_MAP is not set, ModelSim will look for a file named *"mgc_location_map"* in the following locations, in order:

- the current directory
- your home directory
- the directory containing the ModelSim binaries
- the ModelSim installation directory

Use these two steps to map your files:

1 Set the environment variable MGC_LOCATION_MAP to the path to your location map file.

2 Specify the mappings from physical pathnames to logical pathnames:

```
$SRC
/home/vhdl/src
/usr/vhdl/src

$IEEE
/usr/modeltech/ieee
```

Pathname syntax

The logical pathnames must begin with *$* and the physical pathnames must begin with */*. The logical pathname is followed by one or more equivalent physical pathnames. Physical pathnames are equivalent if they refer to the same physical directory (they just have different pathnames on different systems).

How location mapping works

When a pathname is stored, an attempt is made to map the physical pathname to a path relative to a logical pathname. This is done by searching the location map file for the first physical pathname that is a prefix to the pathname in question. The logical pathname is then substituted for the prefix. For example, "/usr/vhdl/src/test.vhd" is mapped to "$SRC/test.vhd". If a mapping can be made to a logical pathname, then this is the pathname that is saved. The path to a source file entry for a design unit in a library is a good example of a typical mapping.

For mapping from a logical pathname back to the physical pathname, ModelSim expects an environment variable to be set for each logical pathname (with the same name). ModelSim reads the location map file when a tool is invoked. If the environment variables corresponding to logical pathnames have not been set in your shell, ModelSim sets the variables to the first physical pathname following the logical pathname in the location map. For example, if you don't set the SRC environment variable, ModelSim will automatically set it to "/home/vhdl/src".

Mapping with Tcl variables

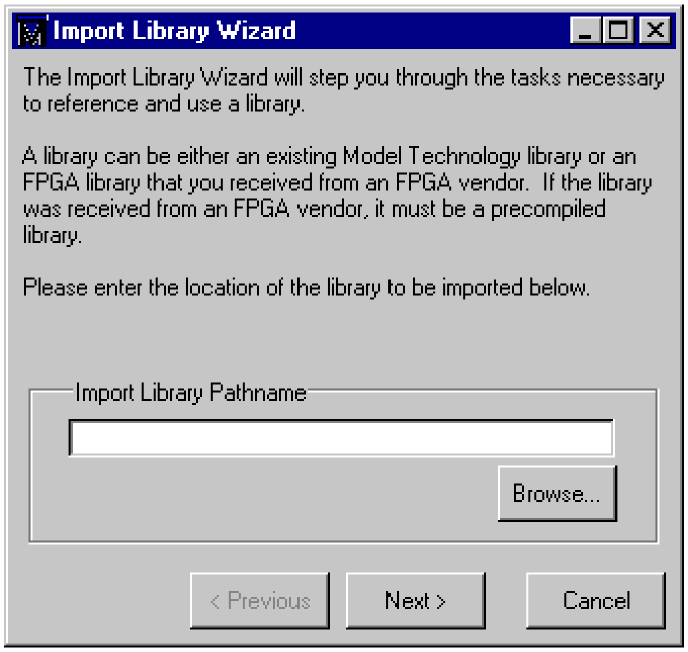
Two Tcl variables may also be used to specify alternative source-file paths; SourceDir and SourceMap. See "Preference variables located in Tcl files" (UM-631) for more information on Tcl preference variables.

Importing FPGA libraries

ModelSim includes an import wizard for referencing and using vendor FPGA libraries. The wizard scans for and enforces dependencies in the libraries and determines the correct mappings and target directories.

Important: The FPGA libraries you import must be pre-compiled. Most FPGA vendors supply pre-compiled libraries configured for use with ModelSim.

To import an FPGA library, select **File > Import > Library** (Main window).



The Import Library Wizard will step you through the tasks necessary to reference and use a library.

A library can be either an existing Model Technology library or an FPGA library that you received from an FPGA vendor. If the library was received from an FPGA vendor, it must be a precompiled library.

Please enter the location of the library to be imported below.

Follow the instructions in the wizard to complete the import.

Protecting source code using -nodebug

The **-nodebug** argument for both **vcom** (CR-303) and **vlog** (CR-345) hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

▶ **Note: -nodebug** encrypts entire files. The Verilog **`protect** compiler directive allows you to encrypt regions within a file. See "ModelSim compiler directives" (UM-152) for details.

When you compile with **-nodebug**, all source text, identifiers, and line number information are stripped from the resulting compiled object, so ModelSim cannot locate or display any information of the model except for the external pins. Specifically, this means that:

- the Source window will not display the design units' source code
- the Structure window will not display the internal structure
- the Signals window will not display internal signals
- the Process window will not display internal processes
- the Variables window will not display internal variables
- none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands

You can access the design units comprising your model via the library, and you may invoke **vsim** (CR-357) directly on any of these design units and see the ports. To restrict even this access in the lower levels of your design, you can use the following **-nodebug** options when you compile:

Command and switch	Result
vcom -nodebug=ports	makes the ports of a VHDL design unit invisible
vlog -nodebug=ports	makes the ports of a Verilog design unit invisible
vlog -nodebug=pli	prevents the use of PLI functions to interrogate the module for information
vlog -nodebug=ports+pli	combines the functions of -nodebug=ports and -nodebug=pli

Don't use the **=ports** option on a design without hierarchy, or on the top level of a hierarchical design. If you do, no ports will be visible for simulation. Rather, compile all lower portions of the design with **-nodebug=ports** first, then compile the top level with **-nodebug** alone.

Design units or modules compiled with **-nodebug** can only instantiate design units or modules that are also compiled **-nodebug**.

4 - VHDL simulation

Chapter contents

This chapter provides an overview of compilation and simulation for VHDL; using the TextIO package with ModelSim*;* ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling; and documentation on ModelSim's special built-in utilities package.

The TextIO package is defined within the *VHDL Language Reference Manual, IEEE Std 1076*; it allows human-readable text input from a declared source within a VHDL file during simulation.

Compiling VHDL designs

Creating a design library

Before you can compile your design, you must create a library in which to store the compilation results. Use **vlib** (CR-344) to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX, MS Windows, or DOS commands – always use the **vlib** command (CR-344).

See "Design libraries" (UM-53) for additional information on working with libraries.

Invoking the VHDL compiler

ModelSim compiles one or more VHDL design units with a single invocation of **vcom** (CR-303), the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with 1076 -1987, 1076 -1993, and 1076-2002 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The **vcom** (CR-303) command compiles using 1076 -2002 rules by default; use the **-87** or **-93** argument to **vcom** (CR-303) to compile units written with version 1076-1987 or 1076 -1993, respectively. You can also change the default by modifying the VHDL93 variable in the *modelsim.ini* file (see "Preference variables located in INI files" (UM-617) for more information).

Dependency checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. **vcom** (CR-303) determines whether or not the compilation results have changed. For example, if you keep an entity and its architectures in the same source file and you modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and you will not have to recompile design units that depend on the entity.

Range and index checking

A range check verifies that a scalar value defined with a range subtype is always assigned a value within its range. An index check verifies that whenever an array subscript expression is evaluated, the subscript will be within the array's range.

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the **vcom** (CR-303) command. Or, you can use the **NoRangeCheck** and **NoIndexCheck** variables in the *modelsim.ini* file to specify whether or not they are performed. See "Preference variables located in INI files" (UM-617).

Range checks in ModelSim are slightly more restrictive than those specified by the VHDL LRM. ModelSim requires any assignment to a signal to also be in range whereas the LRM requires only that range checks be done whenever a signal is updated. Most assignments to signals update the signal anyway, and the more restrictive requirement allows ModelSim to generate better error messages.

Differences between language versions

There are three versions of the IEEE VHDL 1076 standard: VHDL-1987, VHDL-1993, and VHDL-2002. The default language version for ModelSim is VHDL-2002. If your code was written according to the ’87 or ’93 version, you may need to update your code or instruct ModelSim to use the earlier versions’ rules.

To select a specific language version, do one of the following:

- Select the appropriate version from the compiler options menu in the GUI
- Invoke **vcom** (CR-303) using the argument -87, -93, or -2002
- Set the VHDL93 variable in the [vcom] section of the *modelsim.ini* file. Appropriate values for VHDL93 are:

  - 0, 87, or 1987 for VHDL-1987

  - 1, 93, or 1993 for VHDL-1993

  - 2, 02, or 2002 for VHDL-2002

The following is a list of language incompatibilites that may cause problems when compiling a design.

- The only major problem between VHDL-93 and VHDL-2002 is the addition of the keyword "PROTECTED". VHDL-93 programs which use this as an identifier should choose a different name.

All other incompatibilities are between VHDL-87 and VHDL-93.

- VITAL and SDF

  It is important to use the correct language version for VITAL. VITAL2000 must be compiled with VHDL-93 or VHDL-2002. VITAL95 must be compiled with VHDL-87. A typical error message that indicates the need to compile under language version VHDL-87 is:

  ```
  "VITALPathDelay DefaultDelay parameter must be locally static"
  ```

- Purity of NOW

  In VHDL-93 the function "now" is impure. Consequently, any function that invokes "now" must also be declared to be impure. Such calls to "now" occur in VITAL. A typical error message:

  ```
  "Cannot call impure function 'now' from inside pure function '<name>'"
  ```

- Files

  File syntax and usage changed between VHDL-87 and VHDL-93. In many cases vcom issues a warning and continues:

  ```
  "Using 1076-1987 syntax for file declaration."
  ```

  In addition, when files are passed as parameters, the following warning message is produced:

  ```
  "Subprogram parameter name is declared using VHDL 1987 syntax."
  ```

  This message often involves calls to endfile(<name>) where <name> is a file parameter.

- Files and packages

  Each package header and body should be compiled with the same language version. Common problems in this area involve files as parameters and the size of type CHARACTER. For example, consider a package header and body with a procedure that has a file parameter:

  ```
  procedure proc1 ( out_file : out std.textio.text) ...
  ```

  If you compile the package header with VHDL-87 and the body with VHDL-93 or VHDL-2002, you will get an error message such as:

  ```
  "** Error: mixed_package_b.vhd(4): Parameter kinds do not conform between
  declarations in package header and body: 'out_file'."
  ```

- Direction of concatenation

  To solve some technical problems, the rules for direction and bounds of concatenation were changed from VHDL-87 to VHDL-93. You won't see any difference in simple variable/signal assignments such as:

  ```
  v1 := a & b;
  ```

  But if you (1) have a function that takes an unconstrained array as a parameter, (2) pass a concatenation expression as a formal argument to this parameter, and (3) the body of the function makes assumptions about the direction or bounds of the parameter, then you will get unexpected results. This may be a problem in environments that assume all arrays have "downto" direction.

- xnor

  "xnor" is a reserved word in VHDL-93. If you declare an xnor function in VHDL-87 (without quotes) and compile it under VHDL-2002, you will get an error message like the following:

  ```
  ** Error: xnor.vhd(3): near "xnor": expecting: STRING IDENTIFIER
  ```

- 'FOREIGN attribute

  In VHDL-93 package STANDARD declares an attribute 'FOREIGN. If you declare your own attribute with that name in another package, then ModelSim issues a warning such as the following:

  ```
  -- Compiling package foopack

  ** Warning: foreign.vhd(9): (vcom-1140) VHDL-1993 added a definition of the
  attribute foreign to package std.standard. The attribute is also defined in
  package 'standard'. Using the definition from package 'standard'.
  ```

- Size of CHARACTER type

  In VHDL-87 type CHARACTER has 128 values; in VHDL-93 it has 256 values. Code which depends on this size will behave incorrectly. This situation occurs most commonly in test suites that check VHDL functionality. It's unlikely to occur in practical designs. A typical instance is the replacement of warning message:

  ```
  "range nul downto del is null"
  ```

  by

  ```
  "range nul downto 'ÿ' is null" -- range is nul downto y(umlaut)
  ```

- bit string literals

  In VHDL-87 bit string literals are of type bit_vector. In VHDL-93 they can also be of type STRING or STD_LOGIC_VECTOR. This implies that some expressions that are unambiguous in VHDL-87 now become ambiguous is VHDL-93. A typical error message is:

  ```
  ** Error: bit_string_literal.vhd(5): Subprogram '=' is ambiguous. Suitable
  definitions exist in packages 'std_logic_1164' and 'standard'.
  ```

- In VHDL-87 when using individual subelement association in an association list, associating individual subelements with NULL is discouraged. In VHDL-93 such association is forbidden. A typical message is:

  ```
  "Formal '<name>' must not be associated with OPEN when subelements are
  associated individually."
  ```

Simulating VHDL designs

After compiling the design units, you can simulate your designs with **vsim** (CR-357). This section discusses simulation from the UNIX or Windows/DOS command line. You can also use a project to simulate (see "Getting started with projects" (UM-34)) or the **Simulate** dialog box (see "Simulating with the graphic interface" (UM-377)).

For VHDL invoke **vsim** (CR-357) with the name of the configuration, or entity/architecture pair. Note that if you specify a configuration you may not specify an architecture.

This example invokes **vsim** (CR-357) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

vsim (CR-357) is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking **vsim** with the **-sdfmin**, **-sdftyp**, or **-sdfmax** option. Using the SDF file *f1.sdf* in the current work directory, the following invocation of **vsim** annotates maximum timing values for the design unit *my_asic*:

```
vsim -sdfmax /my_asic=f1.sdf my_asic
```

By default, the timing checks within VITAL models are enabled. They can be disabled with the **+notimingchecks** option. For example:

```
vsim +notimingchecks topmod
```

Simulator resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The default resolution limit is set to the value specified by the **Resolution** (UM-624) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-238) with the **simulator state** option.

Overriding the resolution

You can override ModelSim's default resolution by specifying the **-t** option on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps topmod
```

Clearly you need to be careful when doing this type of operation. If the resolution set by **-t** is larger than a delay value in your design, the delay values in that design unit are rounded to the closest multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

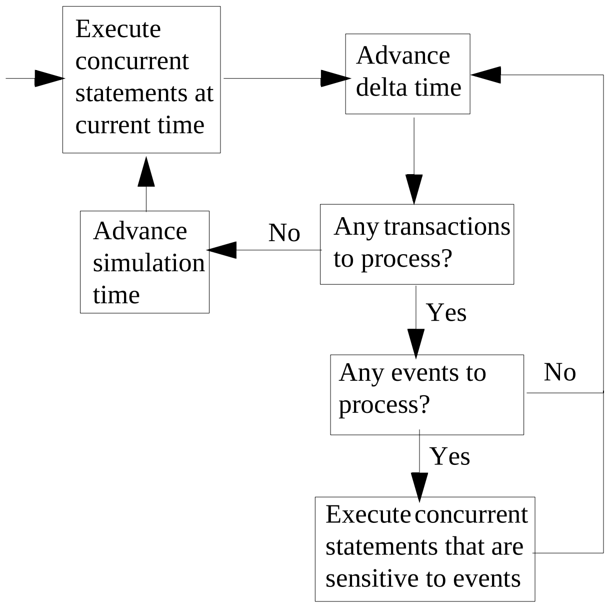
Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Delta delays

Event-based simulators such as ModelSim may process many events at a given simulation time. Multiple signals may need updating, statements that are sensitive to these signals must be executed, and any new events that result from these statements must then be queued and executed as well. The steps taken to evaluate the design without advancing simulation time are referred to as "delta times" or just "deltas."

The diagram below represents the process for VHDL designs. This process continues until the end of simulation time.


Execute concurrent statements at current time
Advance delta time
Any transactions to process?
No
Advance simulation time
Yes
Any events to process?
No
Yes
Execute concurrent statements that are sensitive to events

This mechanism in event-based simulators may cause unexpected results. Consider the following code snippet:

```
      clk2 <= clk;

      process (rst, clk)
      begin
        if(rst = '0')then
          s0 <= '0';
        elsif(clk'event and clk='1') then
          s0 <= inp;
        end if;
      end process;

    process (rst, clk2)
      begin
        if(rst = '0')then
          s1 <= '0';
        elsif(clk2'event and clk2='1') then
          s1 <= s0;
        end if;
      end process;
```

In this example you have two synchronous processes, one triggered with *clk* and the other with *clk2*. To your surprise, the signals change in the *clk2* process on the same edge as they are set in the *clk* process. As a result, the value of *inp* appears at *s1* rather than *s0*.

Here is what's happing. During simulation an event on *clk* occurs (from the testbench). From this event ModelSim performs the "clk2 <= clk" assignment and the process which is sensitive to *clk*. Before advancing the simulation time, ModelSim finds that the process sensitive to *clk2* can also be run. Since there are no delays present, the effect is that the value of *inp* appears at *s1* in the same simulation cycle.

In order to get the expected results, you must do one of the following:

- Insert a delay at every output
- Make certain to use the same clock
- Insert a delta delay

To insert a delta delay, you would modify the code like this:

```
process (rst, clk)
  begin
    if(rst = '0')then
      s0 <= '0';
    elsif(clk'event and clk='1') then
      s0 <= inp;
      s0_delayed <= s0;
    end if;
  end process;

 process (rst, clk2)
  begin
    if(rst = '0')then
      s1 <= '0';
    elsif(clk2'event and clk2='1') then
      s1 <= s0_delayed;
    end if;
  end process;
```

The best way to debug delta delay problems is observe your signals in the List window. There you can see how values change at each delta time.

Detecting infinite zero-delay loops

If a large number of deltas occur without advancing time, it is usually a symptom of an infinite zero-delay loop in the design. In order to detect the presence of these loops, ModelSim defines a limit, the "iteration limit", on the number of successive deltas that can occur. When ModelSim reaches the iteration limit, it issues a warning message.

The iteration limit default value is 5000. If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate > Simulation Options** menu, by modifying the *modelsim.ini* file, or by setting a Tcl variable called IterationLimit (UM-624). See "Preference variables located in INI files" (UM-617) for more information on modifying the *modelsim.ini* file.

If the problem persists, look for zero-delay loops. Run the simulation and look at the source code when the error occurs. Use the step button to step through the code and see which signals or variables are continuously oscillating. Two common causes are a loop that has no exit, or a series of gates with zero delay where the outputs are connected back to the inputs.

Simulating with an elaboration file

Overview

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

Starting with ModelSim version 5.6, you can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an elaboration file?

In many cases design loading time is not that important. For example, if you're doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you're comparing like to like.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Elaboration file flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

1. If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use $sdf_annotate system tasks. Note that use of $sdf_annotate causes timing to be applied after elaboration.

2. Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see "Modifying stimulus" (UM-82) below).

3. Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an elaboration file

Elaboration file creation is performed with the same **vsim** settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time, as detailed below.

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to **vsim** (CR-357).

The **-elab_cont** argument is used to create the elaboration file then continue with the simulation after the elaboration file is created. You can use the **-c** switch with **-elab_cont** to continue the simulation in command-line mode.

Important: Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

Loading an elaboration file

To load an elaboration file, use the **-load_elab <filename>** argument to **vsim** (CR-357). By default the elaboration file will load in command-line mode or interactive mode depending on the argument (-c or -i) used during elaboration file creation. If no argument was used during creation, the **-load_elab** argument will default to the interactive mode.

The **vsim** arguments listed below can be used with **-load_elab** to affect the simulation.

```
+<plus_args>
-c or -i
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

All other **vsim** arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

Important: The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Modifying stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the change command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The **<HDLfilename>** file name, if it appears in the design as a file name (for example, a VHDL FILE object as well as some Verilog sysfuncs that take file names), is substituted with the **<NEWfilename>** file name. This mapping occurs before environment variable expansion and can't be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

Using with the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard tf routines. The sizetf, misctf and checktf calls that occur during elaboration are played back at **-load_elab** to ensure the PLI model is in the correct simulation state. Registered user tf routines called from the Verilog HDL will not occur until **-load_elab** is complete and PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the FLI Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab_defer_fli** argument. When used in tandem with **-elab**, **-elab_defer_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, mti_IsRestore(), ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab_defer_fli**.

Syntax

See the **vsim** command (CR-357) for details on **-elab**, **-elab_cont**, **-elab_defer_fli**, **-compress_elab**, **-filemap_elab**, and **-load_elab**.

Example

Upon first simulating the design, use **vsim -elab <filename> <library_name.design_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load_elab <filename>**.

To change the stimulus without recoding, recompiling, and reloading the entire design, Modelsim allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling, and reloading the entire design is as follows:

vsim -load_elab <filename> -filemap_elab vectors=alt_vectors

Checkpointing and restoring simulations

The **checkpoint** (CR-99) and **restore** (CR-242) commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current **vsim** session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous **vsim** session (i.e., after quitting ModelSim)	vsim -restore <filename>

Checkpoint file contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the list and wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **$fopen** system task
- state of foreign architectures
- state of PLI/VPI code

Checkpoint exclusions

You *cannot* checkpoint/restore the following:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the *FLI Reference Manual* or *Chapter 6 - Verilog PLI / VPI* for more information.

Controlling checkpoint file compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

The difference between checkpoint/restore and restart

The **restart** (CR-240) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog $fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart,** however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

Using macros with restart and checkpoint/restore

The **restart** (CR-240) command resets and restarts the simulation kernel, and zeros out any user-defined commands, but it does not touch the state of the macro interpreter. This lets you do **restart** commands within macros.

The pause mode indicates that a macro has been interrupted. That condition will not be affected by a restart, and if the restart is done with an interrupted macro, the macro will still be interrupted after the restart.

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a **checkpoint** (CR-99) and later in the same session doing a **restore** (CR-242) of the earlier checkpoint. The **restore** does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

Using the TextIO package

To access the routines in TextIO, include the following statement in your VHDL source code:

```
USE std.textio.all;
```

A simple example using the package TextIO is:

```
USE std.textio.all;
ENTITY simple_textio IS
END;

ARCHITECTURE simple_behavior OF simple_textio IS
BEGIN
    PROCESS
        VARIABLE i: INTEGER:= 42;
        VARIABLE LLL: LINE;
    BEGIN
        WRITE (LLL, i);
        WRITELINE (OUTPUT, LLL);
        WAIT;
    END PROCESS;
END simple_behavior;
```

Syntax for file declaration

The VHDL’87 syntax for a file declaration is:

```
file identifier : subtype_indication is [ mode ] file_logical_name ;
```

where "file_logical_name" must be a string expression.

In newer versions of the 1076 spec, syntax for a file declaration is:

```
file identifier_list : subtype_indication [ file_open_information ] ;
```

where "`file_open_information`" is:

```
[open file_open_kind_expression] is file_logical_name
```

You can specify a full or relative path as the file_logical_name; for example (VHDL’87):

```
file filename : TEXT is in "/usr/rick/myfile";
```

Normally if a file is declared within an architecture, process, or package, the file is opened when you start the simulator and is closed when you exit from it. If a file is declared in a subprogram, the file is opened when the subprogram is called and closed when execution RETURNs from the subprogram. Alternatively, the opening of files can be delayed until the first read or write by setting the **DelayFileOpen** variable in the *modelsim.ini* file. Also, the number of concurrently open files can be controlled by the **ConcurrentFileLimit** variable. These variables help you manage a large number of files during simulation. See *Appendix A - ModelSim variables* for more details.

Using STD_INPUT and STD_OUTPUT within ModelSim

The standard VHDL’87 TextIO package contains the following file declarations:

```
file input: TEXT is in "STD_INPUT";
file output: TEXT is out "STD_OUTPUT";
```

Updated versions of the TextIO package contain these file declarations:

```
file input: TEXT open read_mode is "STD_INPUT";
file output: TEXT open write_mode is "STD_OUTPUT";
```

STD_INPUT is a file_logical_name that refers to characters that are entered interactively from the keyboard, and STD_OUTPUT refers to text that is displayed on the screen.

In ModelSim, reading from the STD_INPUT file allows you to enter text into the current buffer from a prompt in the Main window. The lines written to the STD_OUTPUT file appear in the Main window transcript.

TextIO implementation issues

Writing strings and aggregates

A common error in VHDL source code occurs when a call to a WRITE procedure does not specify whether the argument is of type STRING or BIT_VECTOR. For example, the VHDL procedure:

```
WRITE (L, "hello");
```

will cause the following error:

```
ERROR: Subprogram "WRITE" is ambiguous.
```

In the TextIO package, the WRITE procedure is overloaded for the types STRING and BIT_VECTOR. These lines are reproduced here:

```
procedure WRITE(L: inout LINE; VALUE: in BIT_VECTOR;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE(L: inout LINE; VALUE: in STRING;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

The error occurs because the argument "hello" could be interpreted as a string or a bit vector, but the compiler is not allowed to determine the argument type until it knows which function is being called.

The following procedure call also generates an error:

```
WRITE (L, "010101");
```

This call is even more ambiguous, because the compiler could not determine, even if allowed to, whether the argument "010101" should be interpreted as a string or a bit vector.

There are two possible solutions to this problem:

- Use a qualified expression to specify the type, as in:

```
WRITE (L, string'("hello"));
```

- Call a procedure that is not overloaded, as in:

```
WRITE_STRING (L, "hello");
```

The WRITE_STRING procedure simply defines the value to be a STRING and calls the WRITE procedure, but it serves as a shell around the WRITE procedure that solves the overloading problem. For further details, refer to the WRITE_STRING procedure in the io_utils package, which is located in the file *<install_dir>/modeltech/examples/ io_utils.vhd*.

Reading and writing hexadecimal numbers

The reading and writing of hexadecimal numbers is not specified in standard VHDL. The Issues Screening and Analysis Committee of the VHDL Analysis and Standardization Group (ISAC-VASG) has specified that the TextIO package reads and writes only decimal numbers.

To expand this functionality, ModelSim supplies hexadecimal routines in the package io_utils, which is located in the file *<install_dir>/modeltech/examples/io_utils.vhd*. To use these routines, compile the io_utils package and then include the following use clauses in your VHDL source code:

```
use std.textio.all;
use work.io_utils.all;
```

Dangling pointers

Dangling pointers are easily created when using the TextIO package, because WRITELINE de-allocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

Bad VHDL (because L1 and L2 both point to the same buffer):

```
READLINE (infile, L1);    -- Read and allocate buffer
L2 := L1;                 -- Copy pointers
WRITELINE (outfile, L1);  -- Deallocate buffer
```

Good VHDL (because L1 and L2 point to different buffers):

```
READLINE (infile, L1);      -- Read and allocate buffer
L2 := new string'(L1.all);  -- Copy contents
WRITELINE (outfile, L1);    -- Deallocate buffer
```

The ENDLINE function

The ENDLINE function described in the *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987* contains invalid VHDL syntax and cannot be implemented in VHDL. This is because access types must be passed as variables, but functions only allow constant parameters.

Based on an ISAC-VASG recommendation the ENDLINE function has been removed from the TextIO package. The following test may be substituted for this function:

```
(L = NULL) OR (L'LENGTH = 0)
```

The ENDFILE function

In the *VHDL Language Reference Manuals*, the ENDFILE function is listed as:

```
-- function ENDFILE (L: in TEXT) return BOOLEAN;
```

As you can see, this function is commented out of the standard TextIO package. This is because the ENDFILE function is implicitly declared, so it can be used with files of any type, not just files of type TEXT.

Using alternative input/output files

You can use the TextIO package to read and write to your own files. To do this, just declare an input or output file of type TEXT. For example, for an input file:

The VHDL’87 declaration is:

```
file myinput : TEXT is in "pathname.dat";
```

The VHDL’93 declaration is:

```
file myinput : TEXT open read_mode is "pathname.dat";
```

Then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

Flushing the TEXTIO buffer

Flushing of the TEXTIO buffer is controlled by the UnbufferedOutput (UM-625) variable in the *modelsim.ini* file.

Providing stimulus

You can stimulate and test a design by reading vectors from a file, using them to drive values onto signals, and testing the results. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

<install_dir>/modeltech/examples/stimulus.vhd

VITAL specification and source code

VITAL ASIC Modeling Specification

The IEEE 1076.4 VITAL ASIC Modeling Specification is available from the Institute of Electrical and Electronics Engineers, Inc.:

IEEE Customer Service
445 Hoes Lane
Piscataway, NJ 08854-1331

Tel: (732) 981-0060
Fax: (732) 981-1721
home page: http://www.ieee.org

VITAL source code

The source code for VITAL packages is provided in the */<install_dir>/modeltech/ vhdl_src/vital2.2b, /vital95, or /vital2000* directories.

VITAL packages

VITAL 1995 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 2000 accelerated packages are pre-compiled into the **vital2000** library. If you need to use the newer library, you'll need to add a **use** clause to your VHDL code to access the VITAL 2000 packages. For example:

```
LIBRARY vital2000;
USE vital2000.all
```

ModelSim VITAL compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. ModelSim is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, ModelSim accelerates the VITAL_Timing, VITAL_Primitives, and VITAL_memory packages. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

VITAL compliance checking

Compliance checking is important in enabling VITAL acceleration; to qualify for global acceleration, an architecture must be VITAL-level-one compliant. **vcom** (CR-303) automatically checks for VITAL 2000 compliance on all entities with the VITAL_Level0 attribute set, and all architectures with the VITAL_Level0 or VITAL_Level1 attribute set.

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking **vcom** (CR-303) with the option **-novitalcheck**. You can turn off compliance checking for VITAL 1995 and VITAL 2000 as well, but we strongly suggest that you leave checking on to ensure optimal simulation.

VITAL compliance warnings

The following LRM errors are printed as warnings (if they were considered errors they would prevent VITAL level 1 acceleration); they do not affect how the architecture behaves.

- Starting index constraint to DataIn and PreviousDataIn parameters to VITALStateTable do not match (1076.4 section 6.4.3.2.2)
- Size of PreviousDataIn parameter is larger than the size of the DataIn parameter to VITALStateTable (1076.4 section 6.4.3.2.2)
- Signal q_w is read by the VITAL process but is NOT in the sensitivity list (1076.4 section 6.4.3)

The first two warnings are minor cases where the body of the VITAL 1995 LRM is slightly stricter than the package portion of the LRM. Since either interpretation will provide the same simulation results, we chose to make these two cases warnings.

The last warning is a relaxation of the restriction on reading an internal signal that is not in the sensitivity list. This is relaxed only for the CheckEnabled parameters of the timing checks, and only if they are not read elsewhere.

You can control the visibility of VITAL compliance-check warnings in your **vcom** (CR-303) transcript. They can be suppressed by using the **vcom -nowarn** switch as in **vcom -nowarn 6**. The 6 comes from the warning level printed as part of the warning, i.e., ** WARNING: [6]. You can also add the following line to your *modelsim.ini* file in the [vcom] VHDL compiler control variables (UM-619) section.

```
[vcom]
Show_VitalChecksWarnings = 0
```

Compiling and simulating with accelerated VITAL packages

vcom (CR-303) automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

- **VITAL Level-0 optimization**
  This is a function-by-function optimization. It applies to all level-0 architectures, and any level-1 architectures that failed level-1 optimization.
- **VITAL Level-1 optimization**
  Performs global optimization on a VITAL 3.0 level-1 architecture that passes the VITAL compliance checker. This is the default behavior. Note that your models will run faster but at the cost of not being able to see the internal workings of the models.

Compiler options for VITAL optimization

Several **vcom** (CR-303) options control and provide feedback on VITAL optimization:

`-novital`

Causes **vcom** to use VHDL code for VITAL procedures rather than the accelerated and optimized timing and primitive packages. Allows breakpoints to be set in the VITAL behavior process and permits single stepping through the VITAL procedures to debug your model. Also, all of the VITAL data can be viewed in the Variables or Signals windows.

`-O0 | -O4`

Lowers the optimization to a minimum with **-O0** (capital oh zero). Optional. Use this to work around bugs, increase your debugging visibility on a specific cell, or when you want to place breakpoints on source lines that have been optimized out.

Enable optimizations with **-O4** (default).

`-debugVA`

Prints a confirmation if a VITAL cell was optimized, or an explanation of why it was not, during VITAL level-1 acceleration.

ModelSim VITAL built-ins will be updated in step with new releases of the VITAL packages.

Util package

The util package, included in ModelSim versions 5.5 and later, serves as a container for various VHDL utilities. The package is part of the modelsim_lib library which is located in the modeltech tree and is mapped in the default *modelsim.ini* file.

To access the utilities in the package, you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

get_resolution

get_resolution returns the current simulator resolution as a real number. For example, 1 femtosecond corresponds to 1e-15.

Syntax

```
resval := get_resolution;
```

Returns

Name	Type	Description
resval	real	The simulator resolution represented as a real

Arguments

None

Related functions

to_real() (UM-96)

to_time() (UM-97)

Example

If the simulator resolution is set to 10ps, and you invoke the command:

```
resval := get_resolution;
```

the value returned to resval would be 1e-11.

init_signal_driver()

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

See init_signal_driver (UM-525) in *Chapter 16 - Signal Spy* for complete details.

init_signal_spy()

The init_signal_spy() utility mirrors the value of a VHDL signal or Verilog register/net onto an existing VHDL signal or Verilog register. This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

See init_signal_spy (UM-528) in *Chapter 16 - Signal Spy* for complete details.

signal_force()

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_force works the same as the **force** command (CR-176) with the exception that you cannot issue a repeating force.

See signal_force (UM-530) in *Chapter 16 - Signal Spy* for complete details.

signal_release()

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register or net. This allows you to release signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_release works the same as the **noforce** command (CR-204).

See signal_release (UM-532) in *Chapter 16 - Signal Spy* for complete details.

to_real()

to_real() converts the physical type time value into a real value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 1900 fs to a real and the simulator resolution was ps, then the real value would be 2.0 (i.e., 2 ps).

Syntax

```
realval := to_real(timeval);
```

Returns

Name	Type	Description
realval	real	The time value represented as a real with respect to the simulator resolution

Arguments

Name	Type	Description
timeval	time	The value of the physical type time

Related functions

get_resolution (UM-94)

to_time() (UM-97)

Example

If the simulator resolution is set to ps, and you enter the following function:

```
realval := to_real(12.99 ns);
```

then the value returned to realval would be 12990.0. If you wanted the returned value to be in units of nanoseconds (ns) instead, you would use the **get_resolution** (UM-94) function to recalculate the value:

```
realval := 1e+9 * (to_real(12.99 ns)) * get_resolution();
```

If you wanted the returned value to be in units of femtoseconds (fs), you would enter the function this way:

```
realval := 1e+15 * (to_real(12.99 ns)) * get_resolution();
```

to_time()

to_time() converts a real value into a time value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 5.9 to a time and the simulator resolution was ps, then the time value would be 6 ps.

Syntax

```
timeval := to_time(realval);
```

Returns

Name	Type	Description
timeval	time	The real value represented as a physical type time with respect to the simulator resolution

Arguments

Name	Type	Description
realval	real	The value of the type real

Related functions

get_resolution (UM-94)

to_real() (UM-96)

Example

If the simulator resolution is set to ps, and you enter the following function:

```
timeval := to_time(72.49);
```

then the value returned to timeval would be 72 ps.

Foreign language interface

Foreign language interface (FLI) routines are C programming language functions that provide procedural access to information within Model Technology's HDL simulator, vsim. A user-written application can use these functions to traverse the hierarchy of an HDL design, get information about and set the values of VHDL objects in the design, get information about a simulation, and control (to some extent) a simulation run.

ModelSim’s FLI interface is described in detail in the *ModelSim FLI Reference*. This document is available from the **Help** menu within ModelSim or in the docs directory of a ModelSim installation.

Modeling memory

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Or, you may get very long load, elaboration, or run times.

These problems are usually explained by the fact that signals consume a substantial amount of memory (many dozens of bytes per bit), all of which needs to be loaded or initialized before your simulation starts.

Modeling memory with variables instead provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

In the example below, we illustrate three alternative architectures for entity "memory". Architecture "style_87_bad" uses a vhdl signal to store the ram data. Architecture "style_87" uses variables in the "memory" process, and architecture "style_93" uses variables in the architecture.

For large memories, architecture "style_87_bad" runs many times longer than the other two, and uses much more memory. This style should be avoided.

Both architectures "style_87" and "style_93" work with equal efficiently. You'll find some additional flexibility with the VHDL 1993 style, however, because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

```
library ieee;
use ieee.std_logic_1164.all;
use work.conversions.all;

entity memory is
    generic(add_bits : integer := 12;
            data_bits : integer := 32);
    port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
        data_in : in std_ulogic_vector(data_bits-1 downto 0);
        data_out : out std_ulogic_vector(data_bits-1 downto 0);
        cs, mwrite : in std_ulogic;
        do_init : in std_ulogic);
    subtype word is std_ulogic_vector(data_bits-1 downto 0);
    constant nwords : integer := 2 ** add_bits;
    type ram_type is array(0 to nwords-1) of word;
end;

architecture style_93 of memory is
        ------------------------------
        shared variable ram : ram_type;
        ------------------------------
begin
memory:
```

```
process (cs)
    variable address : natural;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                 ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process memory;
-- illustrates a second process using the shared variable
initialize:
process (do_init)
    variable address : natural;
    begin
        if rising_edge(do_init) then
            for address in 0 to nwords-1 loop
                ram(address) := data_in;
            end loop;
        end if;
    end process initialize;
end architecture style_93;

architecture style_87 of memory is
begin
memory:
process (cs)
    -----------------------
    variable ram : ram_type;
    -----------------------
    variable address : natural;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                 ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process;
end style_87;

architecture bad_style_87 of memory is
    ----------------------
    signal ram : ram_type;
    ----------------------
begin
memory:
process (cs)
    variable address : natural := 0;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) <= data_in;
                data_out <= data_in;
            else
                data_out <= ram(address);
            end if;
        end if;
    end process;
end bad_style_87;
```

```
------------------------------------------------------------
------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package conversions is
    function sulv_to_natural(x : std_ulogic_vector) return
                natural;
    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector;
end conversions;

package body conversions is

    function sulv_to_natural(x : std_ulogic_vector) return
                natural is
        variable n : natural := 0;
        variable failure : boolean := false;
    begin
        assert (x'high - x'low + 1) <= 31
            report "Range of sulv_to_natural argument exceeds
                natural range"
            severity error;
        for i in x'range loop
            n := n * 2;
            case x(i) is
                when '1' | 'H' => n := n + 1;
                when '0' | 'L' => null;
                when others    => failure := true;
            end case;
        end loop;
        assert not failure
            report "sulv_to_natural cannot convert indefinite
                std_ulogic_vector"
            severity error;

        if failure then
            return 0;
        else
            return n;
        end if;
    end sulv_to_natural;

    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector is
        variable x : std_ulogic_vector(bits-1 downto 0) :=
                (others => '0');
        variable tempn : natural := n;
    begin
        for i in x'reverse_range loop
            if (tempn mod 2) = 1 then
                x(i) := '1';
            end if;
            tempn := tempn / 2;
        end loop;
        return x;
    end natural_to_sulv;

end conversions;
```

Affecting performance by cancelling scheduled events

Performance will suffer if events are scheduled far into the future but then cancelled before they take effect. This situation will act like a memory leak and slow down simulation.

In VHDL this situation can occur several ways. The most common are waits with time-out clauses and projected waveforms in signal assignments.

The following code shows a wait with a time-out:

```
signals synch : bit := '0';
...
p: process
begin
    wait for 10 ms until synch = 1;
end process;

synch <= not synch after 10 ns;
```

At time 0, process *p* makes an event for time 10ms. When *synch* goes to 1 at 10 ns, the event at 10 ms is marked as cancelled but not deleted, and a new event is scheduled at 10ms + 10ns. The cancelled events are not reclaimed until time 10ms is reached and the cancelled event is processed. As a result there will be 500000 (10ms/20ns) cancelled but undeleted events. Once 10ms is reached, memory will no longer increase because the simulator will be reclaiming events as fast as they are added.

For projected waveforms the following would behave the same way:

```
signals synch : bit := '0';
...
p: process(synch)
begin
   output <= '0', '1' after 10ms;
end process;

synch <= not synch after 10 ns;
```

Converting an integer into a bit_vector

The following code demonstrates how to convert an integer into a bit_vector.

```
library ieee;
use ieee.numeric_bit.ALL;

entity test is
end test;

architecture only of test is
  signal s1 : bit_vector(7 downto 0);
  signal int : integer := 45;
begin
  p:process
  begin
    wait for 10 ns;
    s1 <= bit_vector(to_signed(int,8));
  end process p;
end only;
```

5 - Verilog simulation

Chapter contents

Introduction

This chapter describes how to compile and simulate Verilog designs with ModelSim Verilog. ModelSim Verilog implements the Verilog language as defined by the IEEE Standards 1364-1995 and 1364-2001. We recommend that you obtain these specifications for reference.

In addition to the functionality described in the IEEE Std 1364, ModelSim Verilog includes the following features:

- Standard Delay Format (SDF) annotator compatible with many ASIC and FPGA vendors' Verilog libraries
- Value Change Dump (VCD) file extensions for ASIC vendor test tools
- Dynamic loading of PLI/VPI applications (see *Chapter 6 - Verilog PLI / VPI*)
- Compilation into retargetable, executable code
- Incremental design compilation
- Extensive support for mixing VHDL and Verilog in the same design (including SDF annotation)
- Graphic Interface that is common with ModelSim VHDL
- Extensions to provide compatibility with Verilog-XL

The following functionality is partially implemented in ModelSim Verilog:

- Verilog Procedural Interface (VPI) (see */<install_dir>/modeltech/docs/technotes/Verilog_VPI.note* for details)
- System Verilog 3.1, Accellera's Extensions to Verilog® (see */<install_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details)

Many of the examples in this chapter are shown from the command line. For compiling and simulating within a project or ModelSim's GUI see:

- Getting started with projects (UM-34)
- Compiling with the graphic interface (UM-368)
- Simulating with the graphic interface (UM-377)

Compilation

Before you can simulate a Verilog design, you must first create a library and compile the Verilog source code into that library. This section provides detailed information on compiling Verilog designs. For information on creating a design library, see *Chapter 3 - Design libraries*.

The ModelSim Verilog compiler, **vlog**, compiles Verilog source code into retargetable, executable code, meaning that the library format is compatible across all supported platforms and that you can simulate your design on any platform without having to recompile your design specifically for that platform. As you compile your design, the resulting object code for modules and UDPs is generated into a library. By default, the compiler places results into the work library. You can specify an alternate library with the **-work** argument. The following is a simple example of how to create a work library, compile a design, and simulate it:

Contents of top.v:

```
module top;
    initial $display("Hello world");
endmodule
```

Create the work library:

```
% vlib work
```

Compile the design:

```
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

View the contents of the work library (optional):

```
% vdir
MODULE top
```

Simulate the design:

```
% vsim -c top
# Loading work.top
VSIM 1> run -all
# Hello world
VSIM 2> quit
```

In this example, the simulator was run without the graphic interface by specifying the **-c** argument. After the design was loaded, the simulator command **run -all** was entered, meaning to simulate until there are no more simulator events. Finally, the quit command was entered to exit the simulator. By default, a log of the simulation is written to the *transcript* file in the current directory.

Incremental compilation

By default, ModelSim Verilog supports incremental compilation of designs, thus saving compilation time when you modify your design. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler (although, you may wish to do so to optimize performance; see "Compiling for faster performance" (UM-127)).

You are not required to compile your design in any particular order because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator. Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

The following example shows how a hierarchical design can be compiled in top-down order:

Contents of top.v:

```
module top;
    or2 or2_i (n1, a, b);
    and2 and2_i (n2, n1, c);
endmodule
```

Contents of and2.v:

```
module and2(y, a, b);
    output y;
    input a, b;
    and(y, a, b);
endmodule
```

Contents of or2.v:

```
module or2(y, a, b);
    output y;
    input a, b;
    or(y, a, b);
endmodule
```

Compile the design in top down order (assumes work library already exists):

```
% vlog top.v
-- Compiling module top

Top level modules:
    top
% vlog and2.v
-- Compiling module and2

Top level modules:
    and2
% vlog or2.v
-- Compiling module or2

Top level modules:
    or2
```

Note that the compiler lists each module as a top level module, although, ultimately, only *top* is a top-level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top level module. This is just an informative message and can be ignored during incremental compilation. The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top-level module names for the simulator. For example,

```
% vlog top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
    top
```

The most efficient method of incremental compilation is to manually compile only the modules that have changed. This is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to always compile your entire design in one invocation of the compiler. If you specify the **-incr** argument, the compiler will automatically determine which modules have changed and generate code only for those modules. This is not as efficient as manual incremental compilation because the compiler must scan all of the source code to determine which modules must be compiled.

The following is an example of how to compile a design with automatic incremental compilation:

```
% vlog -incr top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
    top
```

Now, suppose that you modify the functionality of the *or2* module:

```
% vlog -incr top.v and2.v or2.v
-- Skipping module top
-- Skipping module and2
-- Compiling module or2

Top level modules:
    top
```

The compiler informs you that it skipped the modules *top* and *and2*, and compiled *or2*.

Automatic incremental compilation is intelligent about when to compile a module. For example, changing a comment in your source code does not result in a recompile; however, changing the compiler command line arguments results in a recompile of all modules.

▶ **Note:** Changes to your source code that do not change functionality but that do affect source code line numbers (such as adding a comment line) *will* cause all affected modules to be recompiled. This happens because debug information must be kept current so that ModelSim can trace back to the correct areas of the source code.

Library usage

All modules and UDPs in a Verilog design must be compiled into one or more libraries. One library is usually sufficient for a simple design, but you may want to organize your modules into various libraries for a complex design. If your design uses different modules having the same name, then you are required to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how you may organize your ASIC cells into one library and the rest of your design into another:

```
% vlib work
% vlib asiclib
% vlog -work asiclib and2.v or2.v
-- Compiling module and2
-- Compiling module or2

Top level modules:
    and2
    or2
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

Note that the first compilation uses the **-work asiclib** argument to instruct the compiler to place the results in the **asiclib** library rather than the default **work** library.

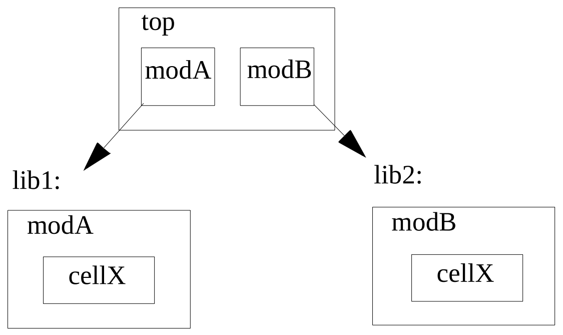
Library search rules

Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top-level modules are loaded from the library named **work** unless you prefix the modules with the **<library>.** option. All other Verilog instantiations are resolved in the following order:

- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the "Verilog-XL `uselib compiler directive" (UM-114).
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

Handling sub-modules with common names

The work library is not necessarily a library named **work**—rather, the **work** library refers to the library containing the module that instantiates the module or UDP that is currently being searched for. This definition is useful if you have hierarchical modules organized into separate libraries, and you have commonly-named sub-modules in the libraries that have different definitions. This may happen if you are using vendor-supplied libraries. For example, say you have the following:



cellX in *lib1* is defined differently than *cellX* in *lib2*. In this situation, you would specify **-L work** first in the search library arguments: **-L work -L lib1 -L lib2**. If you just specify **-L libA -L libB**, instantiations of *cellX* from *modB* resolve to the *modA* version of *cellX*.

Verilog-XL compatible compiler arguments

The compiler arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the **vlog** command (CR-345) for a description of each argument.

```
+define+<macro_name>[=<macro_text>]
+delay_mode_distributed
+delay_mode_path
+delay_mode_unit
+delay_mode_zero
-f <filename>
+incdir+<directory>
+mindelays
+maxdelays
+nowarn<mnemonic>
+typdelays
-u
```

Arguments supporting source libraries

The compiler arguments listed below support source libraries in the same manner as Verilog-XL. See the **vlog** command (CR-345) for a description of each argument.

Note that these source libraries are very different from the libraries that the ModelSim compiler uses to store compilation results. You may find it convenient to use these arguments if you are porting a design to ModelSim or if you are familiar with these arguments and prefer to use them.

Source libraries are searched after the source files on the command line are compiled. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules compiled from source libraries may in turn have additional unresolved references that cause the source libraries to be searched again. This process is repeated until all references are resolved or until no new unresolved references are found. Source libraries are searched in the order they appear on the command line.

```
-v <filename>
-y <directory>
+libext+<suffix>
+librescan
+nolibcell
-R [<simargs>]
```

Verilog-XL `uselib compiler directive

The **`uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler arguments. It has the advantage that a design may reference different modules having the same name. You compile designs that contain **`uselib** directive statements using the **-compile_uselibs** argument (described below) to **vlog** (CR-345).

The syntax for the **`uselib** directive is:

```
`uselib <library_reference>...
```

where <library_reference> is:

```
dir=<library_directory> | file=<library_file> | libext=<file_extension> |
lib=<library_name>
```

The library references are equivalent to command line arguments as follows:

```
dir=<library_directory> -y <library_directory>
file=<library_file> -v <library_file>
libext=<file_extension> +libext+<file_extension>
```

For example, the following directive

```
`uselib dir=/h/vendorA libext=.v
```

is equivalent to the following command line arguments:

```
-y /h/vendorA +libext+.v
```

Since the **`uselib** directives are embedded in the Verilog source code, there is more flexibility in defining the source libraries for the instantiations in the design. The appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved, completely overriding any previous **`uselib** directives.

-compile_uselibs argument

Use the **-compile_uselibs** argument to **vlog** (CR-345) to reference **`uselib** directives. The argument finds the source files referenced in the directive, compiles them into automatically created object libraries, and updates the *modelsim.ini* file with the logical mappings to the libraries.

When using **-compile_uselibs**, ModelSim determines into which directory to compile the object libraries by choosing, in order, from the following three values:

- The directory name specified by the **-compile_uselibs** argument. For example, `-compile_uselibs=./mydir`
- The directory specified by the MTI_USELIB_DIR environment variable (see "Environment variables" (UM-613))
- A directory named *mti_uselibs* that is created in the current working directory

Note: In ModelSim versions prior to 5.5, the library files referenced by the **`uselib** directive were not automatically compiled by ModelSim Verilog. To maintain backwards compatibility, this is still the default behavior when **-compile_uselibs** is not used. See www.model.com/support/documentation/BOOK/pre55_uselib.pdf for a description of the pre-5.5 implementation.

The following code fragment and compiler invocation show how two different modules that have the same name can be instantiated within the same design:

```
module top;
  `uselib dir=/h/vendorA libext=.v
  NAND2 u1(n1, n2, n3);
  `uselib dir=/h/vendorB libext=.v
  NAND2 u2(n4, n5, n6);
endmodule
```

This allows the NAND2 module to have different definitions in the vendorA and vendorB libraries.

`uselib is persistent

As mentioned above, the appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved. This may result in unexpected consequences. For example, consider the following compile command:

```
vlog -compile_uselibs dut.v srtr.v
```

Assume that *dut.v* contains a **`uselib** directive. Since *srtr.v* is compiled after *dut.v*, the **`uselib** directive is still in effect. When *srtr* is loaded it is using the **`uselib** directive from *dut.v* to decide where to locate modules. If this is not what you intend, then you need to put an empty **`uselib** at the end of *dut.v* to "close" the previous **`uselib** statement.

Verilog configurations

The Verilog 2001 spec added configurations. Configurations specify how a design is "assembled" during the elaboration phase of simulation. Configurations actually consist of two pieces: the library mapping and the configuration itself. The library mapping is used at compile time to determine into which libraries the source files are to be compiled. Here is an example of a simple library map file:

```
library work    ../top.v;
library rtlLib  lrm_ex_top.v;
library gateLib lrm_ex_adder.vg;
library aLib    lrm_ex_adder.v;
```

Here is an example of a library map file that uses **-incdir**:

```
library lib1 src_dir/*.v -incdir ../include_dir2, ../, my_incdir;
```

The name of the library map file is arbitrary. You specify the library map file using the **-libmap** argument to the **vlog** command (CR-345). Alternatively, you can specify the file name as the first item on the **vlog** command line, and the compiler will read it as a library map file.

The library map file must be compiled along with the Verilog source files. Multiple map files are allowed but each must be preceded by the **-libmap** argument.

The library map file and the configuration can exist in the same or different files. If they are separate, only the map file needs the **-libmap** argument. The configuration is treated as any other Verilog source file.

Simulation

The ModelSim simulator can load and simulate both Verilog and VHDL designs, providing a uniform graphic interface and simulation control commands for debugging and analyzing your designs. The graphic interface and simulator commands are described elsewhere in this manual, while this section focuses specifically on Verilog simulation.

Invoking the simulator

A Verilog design is ready for simulation after it has been compiled into one or more libraries. The simulator may then be invoked with the names of the top-level modules (many designs contain only one top level module). For example, if your top level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default all modules and UDPs are loaded from the library named **work**. Modules and UDPs from other libraries can be specified using the **-L** or **-Lf** arguments to **vsim** (see "Library usage" (UM-111) for details).

On successful loading of the design, the simulation time is set to zero, and you must enter a **run** command to begin simulation. Commonly, you enter **run -all** to run until there are no more simulation events or until **$finish** is executed in the Verilog code. You can also run for specific time periods (e.g., `run 100 ns`). Enter the **quit** command to exit the simulator.

Simulator resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The resolution limit defaults to the smallest time precision found among all of the **`timescale** compiler directives in the design. Here is an example of a **`timescale** directive:

```
`timescale 1 ns / 100 ps
```

The first number is the time units and the second number is the time precision. The directive above causes time values to be read as ns and to be rounded to the nearest 100 ps.

Modules without timescale directives

You may encounter unexpected behavior if your design contains some modules with timescale directives and others without. The time units for modules without a timescale directive default to the simulator resolution. For example, say you have the two modules shown in the table below:

Module 1	**Module 2**
`` `timescale 1 ns / 10 ps ``	`module mod2 (set);`
`module mod1 (set);`	`  output set;`
	`  reg set;`
`  output set;`	`  parameter d = 1.55;`
`  reg set;`	
`  parameter d = 1.55;`	`  initial`
	`  begin`
`  initial`	`    set = 1'bz;`
`  begin`	`    #d set = 1'b0;`
`    set = 1'bz;`	`    #d set = 1'b1;`
`    #d set = 1'b0;`	`  end`
`    #d set = 1'b1;`	
`  end`	`endmodule`
`endmodule`	

If you invoke **vsim** as `vsim mod2 mod1` then Module 1 sets the simulator resolution to 10 ps. Module 2 has no timescale directive, so the time units default to the simulator resolution, in this case 10 ps. If you watched */mod1/set* and */mod2/set* in the Wave window, you'd see that in Module 1 it transitions every 1.55 ns as expected (because of the 1 ns time unit in the timescale directive). However, in Module 2, *set* transitions every 20 ps. That's because the delay of 1.55 in Module 2 is read as 15.5 ps and is rounded up to 20 ps.

In such cases ModelSim will issue the following warning message during elaboration:

```
** Warning: (vsim-3010) [TSCALE] - Module 'mod1' has a `timescale directive
in effect, but previous modules do not.
```

If you invoke **vsim** as `vsim mod1 mod2`, the simulation results would be the same but ModelSim would produce a different warning message:

```
** Warning: (vsim-3009) [TSCALE] - Module 'mod2' does not have a `timescale
directive in effect, but previous modules do.
```

These warnings should ALWAYS be investigated.

If the design contains no `timescale directives, then the resolution limit and time units default to the value specified by the **Resolution** (UM-624) variable in the modelsim.ini file. (The variable is set to 1 ns by default.)

Multiple timescale directives

As alluded to above, your design can have multiple timescale directives. The timescale directive takes effect where it appears in a source file and applies to all source files which follow in the same **vlog** (CR-345) command. Separately compiled modules can also have different timescales. The simulator determines the smallest timescale of all the modules in a design and uses that as the simulator resolution.

`timescale, -t, and rounding

The optional **vsim** argument **-t** sets the simulator resolution limit for the overall simulation. If the resolution set by **-t** is larger than the precision set in a module, the time values in that module are rounded up. If the resolution set by **-t** is smaller than the precision of the module, the precision of that module remains whatever is specified by the `timescale directive. Consider the following code:

```
`timescale 1 ns / 100 ps

module foo;

  initial
    #12.536 $display
```

The list below shows three possibilities for **-t** and how the delays in the module would be handled in each case:

- **-t** not set

  The delay will be rounded to 12.5 as directed by the module's ‘timescale directive.

- **-t** is set to 1 fs

  The delay will be rounded to 12.5. Again, the module's precision is determined by the ‘timescale directive. ModelSim does not override the module's precision.

- **-t** is set to 1 ns

  The delay will be rounded to 12. The module's precision is determined by the **-t** setting. ModelSim has no choice but to round the module's time values because the entire simulation is operating at 1 ns.

Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Event ordering in Verilog designs

Event-based simulators such as ModelSim may process multiple events at a given simulation time. The Verilog language is defined such that you cannot explicitly control the order in which simultaneous events are processed. Unfortunately, some designs rely on a particular event order, and these designs may behave differently than you expect.

Event queues

Section 5 of the IEEE Std 1364-1995 LRM defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

- active events
- inactive events
- non-blocking assignment update events
- monitor events
- future events
  - inactive events
  - non-blocking assignment update events

The LRM dictates that events are processed as follows – 1) all active events are processed; 2) the inactive events are moved to the active event queue and then processed; 3) the non-blocking events are moved to the active event queue and then processed; 4) the monitor events are moved to the active queue and then processed; 5) simulation advances to the next time where there is an inactive event or a non-blocking assignment update event.

Within the active event queue, the events can be processed in any order, and new active events can be added to the queue in any order. In other words, you *cannot* control event order within the active queue. The example below illustrates potential ramifications of this situation.

Say you have these four statements:

1 always@(q) p = q;

2 always @(q) p2 = not q;

3 always @(p or p2) clk = p and p2;

4 always @(posedge clk)

and current values as follows: q = 0, p = 0, p2=1

The tables below show two of the many valid evaluations of these statements. Evaluation events are denoted by a number where the number is the statement to be evaluated. Update events are denoted *<name>(old->new)* where *<name>* indicates the reg being updated and *new* is the updated value.

Table 1: Evaluation 1

Event being processed	Active event queue
	q(0 → 1)
q(0 → 1)	1, 2
1	p(0 → 1), 2
p(0 → 1)	3, 2
3	clk(0 → 1), 2
clk(0 → 1)	4, 2
4	2
2	p2(1 → 0)
p2(1 → 0)	3
3	clk(1 → 0)
clk(1 → 0)	<empty>

Table 2: Evaluation 2

Event being processed	Active event queue
	q(0 → 1)
q(0 → 1)	1, 2
1	p(0 → 1), 2
2	p2(1 → 0), p(0 → 1)
p(0 → 1)	3, p2(1 → 0)
p2(1 → 0)	3
3	<empty> (clk doesn’t change)

Again, both evaluations are valid. However, in Evaluation 1, *clk* has a glitch on it; in Evaluation 2, *clk* doesn’t. This indicates that the design has a zero-delay race condition on *clk*.

'Controlling' event queues with blocking/non-blocking assignments

The only control you have over event order is to assign an event to a particular queue. You do this via blocking or non-blocking assignments.

Blocking assignments

Blocking assignments place an event in the active, inactive, or future queues depending on what type of delay they have:

- a blocking assignment without a delay goes in the active queue
- a blocking assignment with an explicit delay of 0 goes in the inactive queue
- a blocking assignment with a non-zero delay goes in the future queue

Non-blocking assignments

A non-blocking assignment goes into either the non-blocking assignment update event queue or the future non-blocking assignment update event queue. (Non-blocking assignments with no delays and those with explicit zero delays are treated the same.)

Non-blocking assignments should be used only for outputs of flip-flops. This insures that all outputs of flip-flops do not change until after all flip-flops have been evaluated. Attempting to use non-blocking assignments in combinational logic paths to remove race conditions may only cause more problems. (In the preceding example, changing all statements to non-blocking assignments would not remove the race condition.) This includes using non-blocking assignments in the generation of gated clocks.

The following is an example of how to properly use non-blocking assignments.

```
gen1: always @(master)
  clk1 = master;

gen2: always @(clk1)
  clk2 = clk1;

f1 : always @(posedge clk1)
  begin
    q1 <= d1;
  end

f2:   always @(posedge clk2)
  begin
    q2 <= q1;
  end
```

If written this way, a value on *d1* always takes two clock cycles to get from *d1* to *q2*.
If you change *clk1 = master* and *clk2 = clk1* to non-blocking assignments or *q2 <= q1* and *q1 <= d1* to blocking assignments, then *d1* may get to *q2* is less than two clock cycles.

Debugging event order issues

Since many models have been developed on Verilog-XL, ModelSim tries to duplicate Verilog-XL event ordering to ease the porting of those models to ModelSim. However, ModelSim does not match Verilog-XL event ordering in all cases, and if a model ported to ModelSim does not behave as expected, then you should suspect that there are event order dependencies.

ModelSim helps you track down event order dependencies with the following compiler arguments: **-compat**, **-hazards**, and **-keep_delta**.

See the **vlog** command (CR-345) for descriptions of **-compat** and **-keep_delta**.

Hazard detection

The **-hazard** argument to **vsim** (CR-357) detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. **vsim** detects the following kinds of hazards:

- WRITE/WRITE:
  Two processes writing to the same variable at the same time.
- READ/WRITE:
  One process reading a variable at the same time it is being written to by another process. ModelSim calls this a READ/WRITE hazard if it executed the read first.
- WRITE/READ:
  Same as a READ/WRITE hazard except that ModelSim executed the write first.

vsim issues an error message when it detects a hazard. The message pinpoints the variable and the two processes involved. You can have the simulator break on the statement where the hazard is detected by setting the **break on assertion** level to **Error**.

To enable hazard detection you must invoke **vlog** (CR-345) with the **-hazards** argument when you compile your source code and you must also invoke **vsim** with the **-hazards** argument when you simulate.

Important: Enabling **-hazards** implicitly enables the **-compat** argument. As a result, using this argument may affect your simulation results.

Limitations of hazard detection

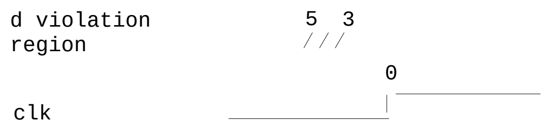
- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.
- A WRITE/READ or READ/WRITE hazard is flagged even if the write does not modify the variable's value.
- Glitches on nets caused by non-guaranteed event ordering are not detected.

Negative timing check limits

Verilog supports negative limit values in the $setuphold and $recrem system tasks. These tasks have optional delayed versions of input signals to insure proper evaluation of models with negative timing check limits. Delay values for these delayed nets are determined by the simulator so that valid data is available for evaluation before a clocking signal.

Example

```
$setuphold(posedge clk, negedge d, 5, -3, Notifier,,, clk_dly, d_dly);
```



ModelSim calculates the delay for signal *d_dly* as 4 time units instead of 3. It does this to prevent *d_dly* and *clk_dly* from occurring simultaneously when a violation isn't reported.

ModelSim accepts negative limit checks by default, unlike current versions of Verilog-XL. To match Verilog-XL default behavior (i.e., zeroing all negative timing check limits), use the **+no_neg_tcheck** argument to **vsim** (CR-357).

Negative timing constraint algorithm

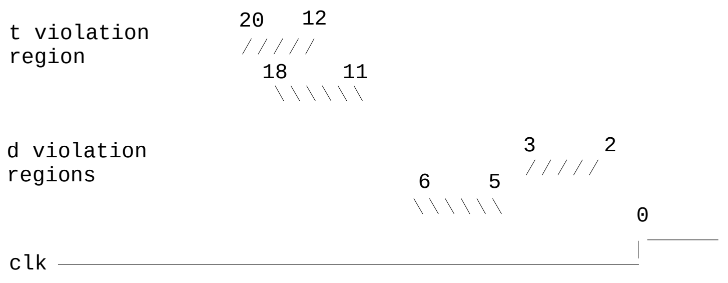
The algorithm ModelSim uses to calculate delays for delayed nets isn't described in IEEE Std 1364. Rather, ModelSim matches Verilog-XL behavior. The algorithm attempts to find a set of delays so the data net is valid when the clock net transitions and the timing checks are satisfied. The algorithm is iterative because a set of delays can be selected that satisfies all timing checks for a pair of inputs but then causes mis-ordering of another pair (where both pairs of inputs share a common input). When a set of delays that satisfies all timing checks is found, the delays are said to converge.

When none of the delay sets cause convergence, the algorithm pessimistically changes the timing check limits to force convergence. Basically the algorithm zeroes the smallest negative $setup/$recovery limit. If a negative $setup/$recovery doesn't exist, then the algorithm zeros the smallest negative $hold/$removal limit. After zeroing a negative limit, the delay calculation procedure is repeated. If the delays don't converge, the algorithm zeros another negative limit, repeating the process until convergence is found.

A simple example will help clarify the algorithm. Assume you have the following timing checks:

```
$setuphold(posedge clk, posedge d, 3, -2 , NOTIFIER,,, clk_dly, d_dly);
$setuphold(posedge clk, negedge d, 6, -5 , NOTIFIER,,, clk_dly, d_dly);
$setuphold(posedge clk, posedge t, 20, -12 , NOTIFIER,,, clk_dly, t_dly);
$setuphold(posedge clk, negedge t, 18, -11 , NOTIFIER,,, clk_dly, t_dly);
```

The violation regions for t and d in this example are:



Note that the delays between *clk/clk_dly*, *t/t_dly*, and *d/d_dly* are not edge sensitive, and they must be the same for both rising and falling transitions of *clk*, *t*, and *d*. A *d* => *d_dly* delay of 5 will satisfy the negedge case (transitions of *d* from 5 to 0 before *clk* won't be latched), but valid transitions of posedge *d*, in the region of 5 to 3 before *clk*, won't latch correctly. Therefore, to find convergence, the algorithm starts zeroing negative **$hold** limits (-12, then -11, and then -5). The check limits on *t* are zeroed first because of their magnitude.

ModelSim will display messages when limits are zeroed if you use the **+ntc_warn** argument. Even if you don't set **+ntc_warn**, ModelSim displays a summary of any zeroed limits.

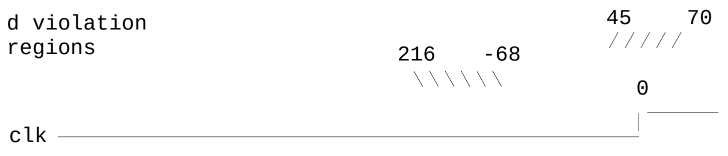
Extending check limits without zeroing

If zeroing limits is too pessimistic for your design, you can use the **vsim** (CR-357) arguments **-extend_tcheck_data_limit** and **-extend_tcheck_ref_limit** instead. These arguments cause a one-time extension of qualifying data or reference limits in an attempt to provide a solution prior to any limit zeroing. A limit qualifies if it bounds a violation region which does not overlap a related violation region.

An example will help illustrate. Assume you have the following timing checks:

```
$setuphold( posedge clk, posedge d,   45,  70, notifier,,,dclk,dd);
$setuphold( posedge clk, negedge d,  216, -68, notifier,,,dclk,dd);
```

The violation regions for d in this example are:



The delay net delay analysis in this case does not provide a solution. The required negative hold delay of 68 between *d* and *dd* could cause a non-violating posedge *d* transition to be delayed on *dd* so that it could arrive after *dclk* for functional evaluation. By default the -68 hold limit is set pessimistically to 0 to insure the correct functional evaluation.

Alternatively, you could use **-extend_tcheck_data_limit** to overlap the regions. In this example we must specify the percentage by which to "decrease" the negative hold limit in order to overlap the positive setup limit. In other words, you must extend the 216, -68 region to 216, -44. You would calculate the percentage as follows:

1 Calculate the size of the negative edge violation region:

216 - 68 = 148

2 Calculate the gap between the negative hold limit and the positive setup limit and add one timing unit to allow for overlap:

68 - 45 = 23 + 1 = 24

3 Divide the gap size by the violation region size:

24 / 148 = .16

Hence, you would set **-extend_tcheck_data_limit** to 16.

Note: ModelSim will extend the limit only as far as is needed to derive a solution. So if you used 100 in the previous example, it would still only extend the limit 16 percent. Indeed, in some cases it may be easiest to select a large percentage number and not worry about an exact calculation.

Using delayed inputs for timing checks

By default ModelSim performs timing checks on inputs specified in the timing check. If you want timing checks performed on the delayed inputs, use the **+delayed_timing_checks** argument to vsim.

Consider and example. This timing check:

```
$setuphold(posedge clk, posedge t, 20, -12, NOTIFIER,,, clk_dly, t_dly);
```

reports a timing violation when posedge *t* occurs in the violation region:

```
              20      -12
t             /////////
                              0
                               ___________
clk          ________________|
```

With the +delayed_timing_checks argument, the violation region between the delayed inputs is:

```
                      7        1
t_dly                 /////////
                              0
                               ___________
clk_dly      ________________|
```

Although the check is performed on the delayed inputs, the timing check violation message is adjusted to reference the undelayed inputs. Only the report time of the violation message is noticeably different between the delayed and undelayed timing checks.

By far the greatest difference between these modes is evident when there are conditions on a delayed check event because the condition is not implicitly delayed. Also, timing checks specified without explicit delayed signals are delayed, if necessary, when they reference an input that is delayed for a negative timing check limit.

Verilog-XL compatible simulator arguments

The simulator arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the **vsim** command (CR-357) for a description of each argument.

```
+alt_path_delays
-l <filename>
+maxdelays
+mindelays
+multisource_int_delays
+no_cancelled_e_msg
+no_neg_tchk
+no_notifier
+no_path_edge
+no_pulse_msg
+no_show_cancelled_e
+nosdfwarn
+nowarn<mnemonic>
+ntc_warn
+pulse_e/<percent>
+pulse_e_style_ondetect
+pulse_e_style_onevent
+pulse_int_e/<percent>
+pulse_int_r/<percent>
+pulse_r/<percent>
+sdf_nocheck_celltype
+sdf_verbose
+show_cancelled_e
+transport_int_delays
+transport_path_delays
+typdelays
```

Compiling for faster performance

This section describes how to use the **-fast** compiler argument to analyze and optimize an entire design for improved simulation performance. This argument improves performance for RTL, behavioral, and gate-level designs (See below for important information specific to gate-level designs.).

ModelSim's default mode of compilation defers module instantiations, parameter propagation, and hierarchical reference resolution until the time that a design is loaded by the simulator (see "Incremental compilation" (UM-109)). This has the advantage that a design does not have to be compiled all at once, allowing independent compilation of modules without requiring knowledge of the context in which they are used.

Compiling modules independently provides flexibility to the user, but results in less efficient simulation performance in many cases. For example, the compiler must generate code for a module containing parameters as though the parameters are variables that will receive their final values when the design is loaded by the simulator. If the compiler is allowed to analyze the entire design at once, then it can determine the final values of parameters and treat them as constants in expressions, thus generating more efficient code. This is just one example of many other optimizations that require analysis of the entire design.

Compiling with -fast

The **-fast** compiler argument allows the compiler to propagate parameters and perform global optimizations. A requirement of using the **-fast** argument is that you must compile the source code for your entire design in a single invocation of the compiler. The following is an example invocation of the compiler and its resulting messages:

```
% vlog -fast cpu_rtl.v
-- Compiling module fp_unit
-- Compiling module mult_56
-- Compiling module testbench
-- Compiling module cpu
-- Compiling module i_unit
-- Compiling module mem_mux
-- Compiling module memory32
-- Compiling module op_unit

Top level modules:
    testbench

Analyzing design...
Optimizing 8 modules of which 6 are inlined:
-- Inlining module i_unit(fast)
-- Inlining module mem_mux(fast)
-- Inlining module op_unit(fast)
-- Inlining module memory32(fast)
-- Inlining module mult_56(fast)
-- Inlining module fp_unit(fast)
-- Optimizing module cpu(fast)
-- Optimizing module testbench(fast)
```

The "Analyzing design..." message indicates that the compiler is building the design hierarchy, propagating parameters, and analyzing design object usage. This information is then used in the final step of generating module code optimized for the specific design. Note that some modules are inlined into their parent modules.

Once the design is compiled, it can be simulated in the usual way:

```
% vsim -c testbench
# Loading work.testbench(fast)
# Loading work.cpu(fast)
VSIM 1> run -all
VSIM 2> quit
```

As the simulator loads the design, it issues messages indicating that the optimized modules are being loaded. There are no messages for loading the inlined modules because their code is inlined into their parent modules.

Incremental compiles with -fast

You can compile a design incrementally by using the **-incr** argument in tandem with **-fast**. By using **-incr**, only changed modules are recompiled. This may decrease compilation time significantly for large designs. Note, however, that if you change any other compiler options, all modules are recompiled regardless if you use **-incr**.

Compiling with +opt

The **+opt** compiler argument may be used instead of **-fast** when it is undesirable to compile the entire design in a single invocation of the compiler (when using a Makefile, for example, that only compiles files that have been modified). After compiling the design without **-fast**, the design may then be optimized using **+opt**.

The optimizations performed by **+opt** are identical to those performed by **-fast**. The only difference between the two arguments is that **+opt** does not need to compile the source code; **+opt** loads the design units from the libraries and regenerates optimized code for them. If the design units reside in multiple libraries, then it may be necessary to use the **-L** and **-Lf** arguments to specify the search libraries.

Any options that are appropriate for **-fast** are appropriate for **+opt**. Specifically, you can also use the **+acc** option to enable PLI access.

See the **vlog** command (CR-345) for syntax.

Compiling mixed designs with -fast

A Verilog design compiled with **-fast** or optimized with **+opt** allows instantiation of VHDL components underneath the Verilog. The VHDL design units must be compiled into a library before optimizing the Verilog design that references them. The Verilog compiler issues a warning message to emphasize that the VHDL instantiations are not optimized. For best performance with **-fast** and **+opt**, instantiate Verilog modules when possible.

A Verilog module compiled with **-fast** can be instantiated from VHDL as long as the VHDL does not need to modify the parameters of the module.

Compiling gate-level designs with -fast

Gate-level designs often have large netlists that are slow to compile with **-fast**. In most cases we recommend the following flow for optimizing gate-level designs:

- Compile the cell library using **-fast** and the **-forcecode** argument. The **-forcecode** argument ensures that code is generated for inlined modules.
- Compile the device under test and testbench *without* **-fast**.

One case where you wouldn't follow this flow is when the testbench has hierarchical references into the cell library. Optimizing the library alone would result in unresolved references. In such a case, you'll have to compile the library, design, and testbench with **-fast** in one invocation of the compiler. The hierarchical reference cells are then not optimized.

Note too that as of ModelSim version 5.5b, several new switches to vlog can be used to further increase optimizations on gate-level designs. The **+nocheck** arguments are described in the Command Reference under the **vlog** command (CR-345).

You can use the **write cell_report** command (CR-388) and the **-debugCellOpt** argument to the **vlog** command (CR-345) to obtain information about which cells have and have not been optimized. **write cell_report** produces a text file that lists all modules. Modules with "(cell)" following their names are optimized cells. For example,

```
Module: top
Architecture: fast

Module: bottom (cell)
Architecture: fast
```

In this case, both top and bottom were compiled with **-fast**, but top was not optimized and bottom was.

The **-debugCellOpt** argument is used with **-fast** when compiling the cell library. Using this argument results in Main window transcript output that identifies why certain cells were not optimized.

Note: ModelSim versions 5.6 and later recognize a module as a gate if the module contains a non-empty specify block. Earlier versions identified gate cells using the `celldefine directive.

Referencing the optimized design

The compiler automatically assigns a secondary name to distinguish the design-specific optimized code from the unoptimized code that may coexist in the same library. The default secondary name for optimized code is "fast", and the default secondary name for unoptimized code is "verilog". You may specify an alternate name (other than "fast") for optimized code using the **-fast=<name>** option. For example, to assign the secondary name "opt1" to your optimized code, you would enter the following:

```
% vlog -fast=opt1 cpu_rtl.v
```

If you have multiple designs that use common modules compiled into the same library, then you need to assign a different secondary name for each design so that the optimized code for a module used in one design context is not overwritten with the optimized code for the same module used in another context. This is true even if the designs are small variations of each other, such as different testbenches. For example, suppose you have two testbenches that instantiate and test the same design. You might assign different secondary names as follows:

```
% vlog -fast=t1 testbench1.v design.v
-- Compiling module testbench1
-- Compiling module design

Top level modules:
    testbench1

Analyzing design...
Optimizing 2 modules of which 0 are inlined:
-- Optimizing module design(t1)
-- Optimizing module testbench1(t1)

% vlog -fast=t2 testbed2.v design.v
-- Compiling module testbench2
-- Compiling module design

Top level modules:
    testbench2

Analyzing design...
Optimizing 2 modules of which 0 are inlined:
-- Optimizing module design(t2)
-- Optimizing module testbench2(t2)
```

All of the modules within *design.v* compiled for *testbench1* are identified by *t1* within the library, whereas for *testbench2* they are identified by *t2*. When the simulator loads *testbench1*, the instantiations from *testbench1* reference the *t1* versions of the code.

Likewise, the instantiations from *testbench2* reference the *t2* versions. Therefore, you only need to invoke the simulator on the desired top-level module and the correct versions of code for the lower level instances are automatically loaded.

The only time that you need to specify a secondary name to the simulator is when you have multiple secondary names associated with a top-level module. If you omit the secondary name, then, by default, the simulator loads the most recently generated code (optimized or unoptimized) for the top-level module. You may explicitly specify a secondary name to load specific optimized code (specify "verilog" to load the unoptimized code). For example, suppose you have a top-level testbench that works in conjunction with each of several other top-level modules that only contain defparams that configure the design. In this case, you need to compile the entire design for each combination, using a different secondary name for each. For example,

```
% vlog -fast=c1 testbench.v design.v config1.v
-- Compiling module testbench
-- Compiling module design
-- Compiling module config1

Top level modules:
    testbench
    config1

Analyzing design...
Optimizing 3 modules of which 0 are inlined:
-- Optimizing module design(c1)
-- Optimizing module testbench(c1)
-- Optimizing module config1(c1)

% vlog -fast=c2 testbench.v design.v config2.v
-- Compiling module testbench
-- Compiling module design
-- Compiling module config2

Top level modules:
    testbench
    config2

Analyzing design...
Optimizing 3 modules of which 0 are inlined:
-- Optimizing module design(c2)
-- Optimizing module testbench(c2)
-- Optimizing module config2(c2)
```

Since the module "testbench" has two secondary names, you must specify which one you want when you invoke the simulator. For example,

```
% vsim 'testbench(c1)' config1
```

Note that it is not necessary to specify the secondary name for config1, because it has only one secondary name. If you omit the secondary name, the simulator defaults to loading the secondary name specified in the most recent compilation of the module.

If you prefer to use the **Simulate** dialog box to select top-level modules, then those modules compiled with **-fast** can be expanded to view their secondary names. Click on the one you wish to simulate.

To view the library contents via the GUI, expand the library in the Library tab (Main window) to see the modules and their associated secondary names. From the command line, execute the **vdir** command (CR-316) on a specific module. For example,

```
VSIM 1> vdir design
# MODULE design
#     Optimized Module t1
#     Optimized Module t2
```

Note: In some cases, an optimized module will have "__<n>" appended to its secondary name. This happens when multiple instantiations of a module require different versions of optimized code (for example, when the parameters of each instance are set to different values).

Enabling design object visibility with the +acc option

Some of the optimizations performed by the **-fast** argument impact design visibility to both the user interface and the PLI routines. Many of the nets, ports, and registers are unavailable by name in user interface commands and in the various graphic interface windows. In addition, many of these objects do not have PLI Access handles, potentially affecting the operation of PLI applications. However, a handle is guaranteed to exist for any object that is an argument to a system task or function.

In the early stages of design, you may choose to compile without the **-fast** argument so as to retain full debug capabilities. Alternatively, you may use one or more **+acc** options in conjunction with **-fast** to enable access to specific design objects. However, keep in mind that enabling design object access may reduce simulation performance.

The syntax for the **+acc** option is as follows:

```
+acc[=<spec>][+<module>[.]]
```

<spec> is one or more of the following characters:

<spec>	Meaning
b	Enable access to individual bits of vector nets. This is necessary for PLI applications that require handles to individual bits of vector nets. Also, some user interface commands require this access if you need to operate on net bits.
c	Enable access to library cells. By default any Verilog module that contains a non-empty specify block may be optimized, and debug and PLI access may be limited. This option keeps module cell visibility.
l	Enable line number directives and process names for line debugging, profiling, and code coverage.
n	Enable access to nets.
p	Enable access to ports. This disables the module inlining optimization, and should be used for PLI applications that require access to port handles, or for debugging (see below).
r	Enable access to registers (including memories, integer, time, and real types).
s	Enable system tasks.
t	Enable access to tasks and functions.

If **<spec>** is omitted, then access is enabled for all objects.

<module> is a module name, optionally followed by "." to indicate all children of the module. Multiple modules are allowed, each separated by a "+". If no modules are specified, then all modules are affected. We strongly recommend specifying modules when using +acc. Doing so will lessen the impact on performance.

If your design uses PLI applications that look for object handles in the design hierarchy, then it is likely that you will need to use the **+acc** option. For example, the built-in

$dumpvars system task is an internal PLI application that requires handles to nets and registers so that it can call the PLI routine **acc_vcl_add()** to monitor changes and dump the values to a VCD file. This requires that access is enabled for the nets and registers on which it operates. Suppose you want to dump all nets and registers in the entire design, and that you have the following $dumpvars call in your testbench (no arguments to $dumpvars means to dump everything in the entire design):

```
initial $dumpvars;
```

Then you need to compile your design as follows to enable net and register access for all modules in the design:

```
% vlog -fast +acc=rn testbench.v design.v
```

As another example, suppose you only need to dump nets and registers of a particular instance in the design (the first argument of **1** means to dump just the variables in the instance specified by the second argument):

```
initial $dumpvars(1, testbench.u1);
```

Then you need to compile your design as follows (assuming *testbench.u1* refers to the module *design*):

```
% vlog -fast +acc=rn+design testbench.v design.v
```

Finally, suppose you need to dump everything in the children instances of *testbench.u1* (the first argument of **0** means to also include all children of the instance):

```
initial $dumpvars(0, testbench.u1);
```

Then you need to compile your design as follows:

```
% vlog -fast +acc=rn+design. testbench.v design.v
```

To gain maximum performance, it may be necessary to enable the minimum required access within the design.

Using pre-compiled libraries

When using the **-fast** argument, if the source code is unavailable for any of the modules referenced in a design, then you must search libraries for the precompiled modules using the **-L** or **-Lf** argument to **vlog** (CR-345). The compiler optimizes pre-compiled modules the same as if the source code is available. The optimized code for a pre-compiled module is written to the same library in which the module is found.

The compiler automatically searches libraries specified in the **`uselib** directive (see Verilog-XL `uselib compiler directive (UM-114)). If your design exclusively uses **`uselib** directives to reference modules in other libraries, then you don't need to specify library search arguments to the compiler.

▶ **Note:** If you use **-L** or **-Lf** with the compiler, you must also you use them with **vsim** (CR-357) when you simulate the design.

Event order and optimized designs

As mentioned earlier in the chapter, the Verilog language does not require that the simulator execute simultaneous events in any particular order. Optimizations performed by **-fast** may expose event order dependencies that cause a design to behave differently than when compiled without **-fast**. Event order dependencies are considered errors and should be corrected (see "Event ordering in Verilog designs" (UM-119) for details). Alternatively, you may use the **-keep_delta** argument (see **vlog** (CR-345)) to disable most **-fast** optimizations that potentially reorder events. Keep in mind this may reduce performance.

Timing checks in optimized designs

Timing checks are performed whether you compile the design with or without **-fast**. In general you'll see the same results in either case. However, in a cell where there are both interconnect delays and conditional timing checks, you might see different timing check results.

Without **-fast** the conditional checks are evaluated with non-delayed values, complying with the original IEEE Std 1364-1995 specification. With **-fast** the conditional checks will be evaluated with delayed values, complying with the new IEEE Std 1364-2001 specification.

Using -fast on cells with internal delay

Cells with internal delays normally are not optimized by **-fast**. However, if you compile with the **+delay_mode_path** switch (which is what we usually suggest), all internal delays are set to zero automatically and only path delays are used. This allows ModelSim to optimize the cell.

If a cell relies on internal delays to function correctly, you cannot optimize that cell.

Simulating with an elaboration file

Overview

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

Starting with ModelSim version 5.6, you can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an elaboration file?

In many cases design loading time is not that important. For example, if you're doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you're comparing like to like.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Elaboration file flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

1. If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use $sdf_annotate system tasks. Note that use of $sdf_annotate causes timing to be applied after elaboration.

2. Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see "Modifying stimulus" (UM-138) below).

3. Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an elaboration file

Elaboration file creation is performed with the same **vsim** settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time, as detailed below.

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to **vsim** (CR-357).

The **-elab_cont** argument is used to create the elaboration file then continue with the simulation after the elaboration file is created. You can use the **-c** switch with **-elab_cont** to continue the simulation in command-line mode.

▲ **Important:** Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

Loading an elaboration file

To load an elaboration file, use the **-load_elab <filename>** argument to **vsim** (CR-357). By default the elaboration file will load in command-line mode or interactive mode depending on the argument (-c or -i) used during elaboration file creation. If no argument was used during creation, the **-load_elab** argument will default to the interactive mode.

The **vsim** arguments listed below can be used with **-load_elab** to affect the simulation.

```
+<plus_args>
-c or -i
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

All other **vsim** arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

▲ **Important:** The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Modifying stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the change command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The **<HDLfilename>** file name, if it appears in the design as a file name (for example, a VHDL FILE object as well as some Verilog sysfuncs that take file names), is substituted with the **<NEWfilename>** file name. This mapping occurs before environment variable expansion and can’t be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

Using with the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard tf routines. The sizetf, misctf and checktf calls that occur during elaboration are played back at **-load_elab** to ensure the PLI model is in the correct simulation state. Registered user tf routines called from the Verilog HDL will not occur until **-load_elab** is complete and PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the FLI Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab_defer_fli** argument. When used in tandem with **-elab**, **-elab_defer_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, mti_IsRestore(), ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab_defer_fli**.

Syntax

See the **vsim** command (CR-357) for details on **-elab**, **-elab_cont**, **-elab_defer_fli**, **-compress_elab**, **-filemap_elab**, and **-load_elab**.

Example

Upon first simulating the design, use **vsim -elab <filename> <library_name.design_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load_elab <filename>**.

To change the stimulus without recoding, recompiling, and reloading the entire design, Modelsim allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling, and reloading the entire design is as follows:

vsim -load_elab <filename> -filemap_elab vectors=alt_vectors

Checkpointing and restoring simulations

The **checkpoint** (CR-99) and **restore** (CR-242) commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current **vsim** session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous **vsim** session (i.e., after quitting ModelSim)	vsim -restore <filename>

Checkpoint file contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the list and wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **$fopen** system task
- state of foreign architectures
- state of PLI/VPI code

Checkpoint exclusions

You *cannot* checkpoint/restore the following:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the *FLI Reference Manual* or *Chapter 6 - Verilog PLI / VPI* for more information.

Controlling checkpoint file compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

The difference between checkpoint/restore and restart

The **restart** (CR-240) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog $fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart,** however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

Using macros with restart and checkpoint/restore

The **restart** (CR-240) command resets and restarts the simulation kernel, and zeros out any user-defined commands, but it does not touch the state of the macro interpreter. This lets you do **restart** commands within macros.

The pause mode indicates that a macro has been interrupted. That condition will not be affected by a restart, and if the restart is done with an interrupted macro, the macro will still be interrupted after the restart.

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a **checkpoint** (CR-99) and later in the same session doing a **restore** (CR-242) of the earlier checkpoint. The **restore** does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

Cell libraries

Model Technology passed the ASIC Council's Verilog test suite and achieved the "Library Tested and Approved" designation from Si2 Labs. This test suite is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors' Verilog cell libraries are compatible with ModelSim Verilog.

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 13 in the IEEE Std 1364-1995 for details on specify blocks, and section 14.5 for details on timing constraints. ModelSim Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

SDF timing annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files. See *Chapter 17 - Standard Delay Format (SDF) Timing Annotation* for details.

Delay modes

Verilog models may contain both distributed delays and path delays. The delays on primitives, UDPs, and continuous assignments are the distributed delays, whereas the port-to-port delays specified in specify blocks are the path delays. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with the distributed delays set to zero. For example,

```
module and2(y, a, b);
    input a, b;
    output y;

    and(y, a, b);

    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule
```

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364). This is the default behavior, but you can specify alternate delay modes with compiler directives and arguments. These arguments and directives are compatible with Verilog-XL. Compiler delay mode arguments take precedence over delay mode directives in the source code.

Distributed delay mode

In distributed delay mode the specify path delays are ignored in favor of the distributed delays. Select this delay mode with the **+delay_mode_distributed** compiler argument or the **`delay_mode_distributed** compiler directive.

Path delay mode

In path delay mode the distributed delays are set to zero in any module that contains a path delay. Select this delay mode with the **+delay_mode_path** compiler argument or the **`delay_mode_path** compiler directive.

Note that this mode allows modules with non-zero delay to be optimized with -fast. See "Using -fast on cells with internal delay" (UM-135) for further details.

Unit delay mode

In unit delay mode the distributed delays are set to one unit of simulation resolution (determined by the minimum time_precision argument in all ‘timescale directives in your design or the value specified with the -t argument to vsim), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler argument or the **`delay_mode_unit** compiler directive.

Zero delay mode

In zero delay mode the distributed delays are set to zero, and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_zero** compiler argument or the **`delay_mode_zero** compiler directive.

System tasks

The IEEE Std 1364 defines many system tasks as part of the Verilog language, and ModelSim Verilog supports all of these along with several non-standard Verilog-XL system tasks. The system tasks listed in this chapter are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI) or Verilog Procedural Interface (VPI). If the simulator issues warnings regarding undefined system tasks, then it is likely that these system tasks are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 system tasks

The following system tasks are described in detail in the IEEE Std 1364.

Timescale tasks	**Simulator control tasks**	**Simulation time functions**	**Command line input**
$printtimescale	$finish	$realtime	$test$plusargs
$timeformat	$stop	$stime	$value$plusargs
		$time	

Probabilistic distribution functions	**Conversion functions**	**Stochastic analysis tasks**	**Timing check tasks**
$dist_chi_square	$bitstoreal	$q_add	$hold
$dist_erlang	$itor	$q_exam	$nochange
$dist_exponential	$realtobits	$q_full	$period
$dist_normal	$rtoi	$q_initialize	$recovery
$dist_poisson	$signed	$q_remove	$setup
$dist_t	$unsigned		$setuphold
$dist_uniform			$skew
$random			$width[a]
			$removal
			$recrem

a.Verilog-XL ignores the threshold argument even though it is part of the Verilog spec. ModelSim does not ignore this argument. Be careful that you don’t set the threshhold argument greater-than-or-equal to the limit argument as that essentially disables the $width check. Note too that you cannot override the threshhold argument via SDF annotation.

Display tasks	**PLA modeling tasks**	**Value change dump (VCD) file tasks**
$display	$async$and$array	$dumpall
$displayb	$async$nand$array	$dumpfile
$displayh	$async$or$array	$dumpflush
$displayo	$async$nor$array	$dumplimit
$monitor	$async$and$plane	$dumpoff
$monitorb	$async$nand$plane	$dumpon
$monitorh	$async$or$plane	$dumpvars
$monitoro	$async$nor$plane	$dumpportson
$monitoroff	$sync$and$array	$dumpportsoff
$monitoron	$sync$nand$array	$dumpportsall
$strobe	$sync$or$array	$dumpportsflush
$strobeb	$sync$nor$array	$dumpports
$strobeh	$sync$and$plane	$dumpportslimit
$strobeo	$sync$nand$plane	
$write	$sync$or$plane	
$writeb	$sync$nor$plane	
$writeh		
$writeo		

File I/O tasks

$fclose	$fopen	$fwriteh
$fdisplay	$fread	$fwriteo
$fdisplayb	$fscanf	$readmemb
$fdisplayh	$fseek	$readmemh
$fdisplayo	$fstrobe	$rewind
$ferror	$fstrobeb	$sdf_annotate
$fflush	$fstrobeh	$sformat
$fgetc	$fstrobeo	$sscanf
$fgets	$ftell	$swrite
$fmonitor	$fwrite	$swriteb
$fmonitorb	$fwriteb	$swriteh
$fmonitorh		$swriteo
$fmonitoro		$ungetc

Verilog-XL compatible system tasks

The following system tasks are provided for compatibility with Verilog-XL. Although they are not part of the IEEE standard, they are described in an annex of the IEEE Std 1364.

```
$countdrivers
$getpattern
$sreadmemb
$sreadmemh
```

The following system tasks are also provided for compatibility with Verilog-XL; they are not described in the IEEE Std 1364.

`$deposit(variable, value);`

This system task sets a Verilog register or net to the specified value. **variable** is the register or net to be changed; **value** is the new value for the register or net. The value remains until there is a subsequent driver transaction or another $deposit task for the same register or net. This system task operates identically to the ModelSim **force -deposit** command.

`$disable_warnings("<keyword>"[,<module_instance>...]);`

This system task instructs ModelSim to disable warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you don't specify a module instance, ModelSim disables warnings for the entire simulation.

`$enable_warnings("<keyword>"[,<module_instance>...]);`

This system task enables warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you don't specify a module_instance, ModelSim enables warnings for the entire simulation.

`$system("<operating system shell command>");`

This system task executes the specified operating system shell command and displays the result. For example, to list the contents of the working directory on Unix:

```
$system("ls");
```

The following system tasks are extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as in Verilog-XL.

```
$recovery(reference event, data_event, removal_limit, recovery_limit,
[notifier], [tstamp_cond], [tcheck_cond], [delayed_reference],
[delayed_data])
```

The $recovery system task normally takes a recovery_limit as the third argument and an optional notifier as the fourth argument. By specifying a limit for both the third and fourth arguments, the $recovery timing check is transformed into a combination removal and recovery timing check similar to the $recrem timing check. The only difference is that the removal_limit and recovery_limit are swapped.

```
$setuphold(clk_event, data_event, setup_limit, hold_limit, [notifier],
[tstamp_cond], [tcheck_cond], [delayed_clk], [delayed_data])
```

The tstamp_cond argument conditions the data_event for the setup check and the clk_event for the hold check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The tcheck_cond argument conditions the data_event for the hold check and the clk_event for the setup check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The delayed_clk argument is a net that is continuously assigned the value of the net specified in the clk_event. The delay is non-zero if the setup_limit is negative, zero otherwise.

The delayed_data argument is a net that is continuously assigned the value of the net specified in the data_event. The delay is non-zero if the hold_limit is negative, zero otherwise.

The delayed_clk and delayed_data arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the delayed_clk and delayed_data nets in place of the normal clk and data nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for delayed_clk and delayed_data such that the correct data is latched as long as a timing constraint has not been violated. See "Negative timing check limits" (UM-123) for more details.

The following system tasks are Verilog-XL system tasks that are not implemented in ModelSim Verilog, but have equivalent simulator commands.

`$input("filename")`
This system task reads commands from the specified filename. The equivalent simulator command is **do <filename>**.

`$list[(hierarchical_name)]`
This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the graphic interface Structure window. The corresponding source code is displayed in the Source window.

`$reset`
This system task resets the simulation back to its time 0 state. The equivalent simulator command is **restart**.

`$restart("filename")`
This system task sets the simulation to the state specified by filename, saved in a previous call to $save. The equivalent simulator command is **restore <filename>**.

`$save("filename")`
This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is **checkpoint <filename>**.

`$scope(hierarchical_name)`
This system task sets the interactive scope to the scope specified by hierarchical_name. The equivalent simulator command is **environment <pathname>**.

`$showscopes`
This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is **show**.

`$showvars`
This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is **show**.

ModelSim Verilog system tasks

The following system tasks are specific to ModelSim. They are not included in the IEEE Std 1364 nor are they likely supported in other simulators. Their use may limit the portability of your code.

`$coverage_save(<filename>, [<instancepath>], [<xml_output>])`

The $coverage_save() system task saves Code Coverage information to a file during a batch run that typically would terminate via the $finish call. If you don't specify <instancepath>, ModelSim saves all coverage data in the current design to the specified file. If you do specify <instancepath>, ModelSim saves data on that instance, and all instances below it (recursively), to the specified file.

If set to 1, the [<xml_output>] argument specifies that the output be saved in XML format.

See *Chapter 12 - Code Coverage* for more information on Code Coverage.

`$init_signal_driver`

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_driver (UM-534) in *Chapter 16 - Signal Spy* for complete details.

`$init_signal_spy`

The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net onto an existing Verilog register or VHDL signal. This system task allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_spy (UM-537) in *Chapter 16 - Signal Spy* for complete details.

`$signal_force`

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). A $signal_force works the same as the **force** command (CR-176) with the exception that you cannot issue a repeating force. See $signal_force (UM-539) in *Chapter 16 - Signal Spy* for complete details.

`$signal_release`

The $signal_release() system task releases a value that had previously been forced onto an existing VHDL signal or Verilog register or net. A $signal_release works the same as the **noforce** command (CR-204). See $signal_release (UM-541) in *Chapter 16 - Signal Spy*.

`$sdf_done`

This task is a "cleanup" function that removes internal buffers, called MIPDs, that have a delay value of zero. These MIPDs are inserted in response to the **-v2k_int_delay** argument to the **vsim** command (CR-357). In general the simulator will automatically remove all zero delay MIPDs. However, if you have $sdf_annotate() calls in your design that are not getting executed, the zero-delay MIPDs are not removed. Adding the $sdf_done task after your last $sdf_annotate() will remove any zero-delay MIPDs that have been created.

Compiler directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary.

Many of the compiler directives (such as **`timescale**) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default by a **`resetall** directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line could be significant. For example, if you have a file that defines some common macros for the entire design, then you might need to place it first in the list of files to be compiled.

The **`resetall** directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364):

```
`celldefine
‘default_decay_time
`default_nettype
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`protected
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

IEEE Std 1364 compiler directives

The following compiler directives are described in detail in the IEEE Std 1364.

```
`celldefine
`default_nettype
`define
`else
`elsif
`endcelldefine
`endif
`ifdef
‘ifndef
`include
‘line
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef
```

Verilog-XL compatible compiler directives

The following compiler directives are provided for compatibility with Verilog-XL.

`‘default_decay_time <time>`
This directive specifies the default decay time to be used in trireg net declarations that do not explicitly declare a decay time. The decay time can be expressed as a real or integer number, or as "infinite" to specify that the charge never decays.

`` `delay_mode_distributed ``
This directive disables path delays in favor of distributed delays. See "Delay modes" (UM-142) for details.

`` `delay_mode_path ``
This directive sets distributed delays to zero in favor of path delays. See "Delay modes" (UM-142) for details.

`` `delay_mode_unit ``
This directive sets path delays to zero and non-zero distributed delays to one time unit. See "Delay modes" (UM-142) for details.

`` `delay_mode_zero ``
This directive sets path delays and distributed delays to zero. See "Delay modes" (UM-142) for details.

`` `uselib ``
This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler arguments. See "Verilog-XL `uselib compiler directive" (UM-114) for details.

The following Verilog-XL compiler directives are silently ignored by ModelSim Verilog. Many of these directives are irrelevant to ModelSim Verilog, but may appear in code being ported from Verilog-XL.

```
`accelerate
`autoexpand_vectornets
`disable_portfaults
`enable_portfaults
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`remove_gatenames
`remove_netnames
`suppress_faults
```

The following Verilog-XL compiler directives produce warning messages in ModelSim Verilog. These are not implemented in ModelSim Verilog, and any code containing these directives may behave differently in ModelSim Verilog than in Verilog-XL.

```
`default_trireg_strength
`signed
`unsigned
```

ModelSim compiler directives

The following directives are specific to ModelSim and are not compatible with other simulators (see note below).

`‘protect ... ‘endprotect`

This directive pair allows you to encrypt selected regions of your source code. The code in **`protect** regions has all debug information stripped out. This behaves exactly as if using the **-nodebug** argument except that it applies to selected regions of code rather than the whole file. This enables usage scenarios such as making module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

The **`protect** directive is ignored by default unless you use the **+protect** argument to **vlog** (CR-345). Once compiled, the original source file is copied to a new file with a ".vp" suffix in the current work directory. This new file can be delivered and used as a replacement for the original source file.

The **+protect** argument is not required when compiling *.vp* files because the **`protect** directives are converted to **`protected** directives which are processed even if **+protect** is omitted.

`protect** and **`protected directives cannot be nested.

If any **`include** directives occur within a protected region, the compiler generates a copy of the include file with a ".vp" suffix and protects the entire contents of the include file.

If errors are detected in a protected region, the error message always reports the first line of the protected block.

The $sdf_annotate() system task cannot be used to SDF-annotate code bracketed by `protect..`endprotect.

Though other simulators have a **`protect** directive, the algorithm ModelSim uses to encrypt source files is different. Hence, even though an uncompiled source file with **`protect** is compatible with another simulator, once the source is compiled in ModelSim, you could not simulate it elsewhere.

6 - Verilog PLI / VPI

Chapter contents

Introduction

This chapter describes the ModelSim implementation of the Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface). Both interfaces provide a mechanism for defining system tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI (see "Third party PLI applications" (UM-175)). In addition, you may write your own PLI/VPI applications.

ModelSim Verilog implements the PLI as defined in the IEEE Std 1364, with the exception of the **acc_handle_datapath()** routine. We did not implement the **acc_handle_datapath()** routine because the information it returns is more appropriate for a static timing analysis tool.

The VPI is partially implemented as defined in the IEEE Std 1364-2001. The list of currently supported functionality can be found in the following file:

```
<install_dir>/modeltech/docs/technotes/Verilog_VPI.note
```

The IEEE Std 1364 is the reference that defines the usage of the PLI/VPI routines. This manual only describes details of using the PLI/VPI with ModelSim Verilog.

Registering PLI applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, ModelSim Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of s_tfcell structures. This structure is declared in the veriuser.h include file as follows:

```
typedef int (*p_tffn)();

typedef struct t_tfcell {
    short type;/* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data;/* passed as data argument of callback function */
    p_tffn checktf;  /* argument checking callback function */
    p_tffn sizetf;   /* function return size callback function */
    p_tffn calltf;   /* task or function call callback function */
    p_tffn misctf;   /* miscellaneous reason callback function */
    char *tfname;/* name of system task or function */

        /* The following fields are ignored by ModelSim Verilog */
    int forwref;
    char *tfveritool;
    char *tferrmessage;
    int hash;
    struct t_tfcell *left_p;
    struct t_tfcell *right_p;
    char *namecell_p;
    int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the calltf function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the data field (many PLI applications don't use this field). The type field defines the entry as either a system task (USERTASK) or a system function that returns either a register (USERFUNCTION) or a real (USERREALFUNCTION). The tfname field is the system task or function name (it must begin with $). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF() for each system task or function defined. The mti_RegisterUserTF() function is declared in veriuser.h as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
    {0}  /* last entry must be 0 */
};
```

Alternatively, you can add an init_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if a PLI shared library does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see "Compiling and linking PLI/VPI C applications" (UM-159)). The PLI applications are specified as follows (note that on a Windows platform the file extension would be .dll):

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a -pli argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

The various methods of specifying PLI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

Registering VPI applications

Each VPI application must register its system tasks and functions and its callbacks with the simulator. To accomplish this, one or more user-created registration routines must be called at simulation startup. Each registration routine should make one or more calls to vpi_register_systf() to register user-defined system tasks and functions and vpi_register_cb() to register callbacks. The registration routines must be placed in a table named vlog_startup_routines so that the simulator can find them. The table must be terminated with a 0 entry.

Example

```
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }

PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }

PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }

PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }

PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }

void RegisterMySystfs( void )
  {

      vpiHandle tmpH;
      s_cb_data callback;
      s_vpi_systf_data systf_data;

      systf_data.type        = vpiSysFunc;
      systf_data.sysfunctype = vpiSizedFunc;
      systf_data.tfname      = "$myfunc";
      systf_data.calltf      = MyFuncCalltf;
      systf_data.compiletf   = MyFuncCompiletf;
      systf_data.sizetf      = MyFuncSizetf;
      systf_data.user_data   = 0;
      tmpH = vpi_register_systf( &systf_data );
      vpi_free_object(tmpH);

      callback.reason    = cbEndOfCompile;
      callback.cb_rtn    = MyEndOfCompCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);

      callback.reason    = cbStartOfSimulation;
      callback.cb_rtn    = MyStartOfSimCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);
  }

void (*vlog_startup_routines[ ] ) () = {
    RegisterMySystfs,
      0    /* last entry must be 0 */
};
```

Loading VPI applications into the simulator is the same as described in "Registering PLI applications" (UM-155).

PLI and VPI applications can co-exist in the same application object file. In such cases, the applications are loaded at startup as follows:

- If an init_usertfs() function exists, then it is executed and only those system tasks and functions registered by calls to mti_RegisterUserTF() will be defined.
- If an init_usertfs() function does not exist but a veriusertfs table does exist, then only those system tasks and functions listed in the veriusertfs table will be defined.
- If an init_usertfs() function does not exist and a veriusertfs table does not exist, but a vlog_startup_routines table does exist, then only those system tasks and functions and callbacks registered by functions in the vlog_startup_routines table will be defined.

As a result, when PLI and VPI applications exist in the same application object file, they must be registered in the same manner. VPI registration functions that would normally be listed in a vlog_startup_routines table can be called from an init_usertfs() function instead.

Compiling and linking PLI/VPI C applications

The following platform-specific instructions show you how to compile and link your PLI/VPI C applications so that they can be loaded by ModelSim. Microsoft Visual C/C++ is supported for creating Windows DLLs while Gcc and cc compilers are supported for creating UNIX shared libraries.

The PLI/VPI routines are declared in the include files located in the ModelSim *<install_dir>/modeltech/include* directory. The acc_user.h file declares the ACC routines, the veriuser.h file declares the TF routines, and the vpi_user.h file declares the VPI routines.

The following instructions assume that the PLI or VPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see "Specifying the PLI/VPI file to load" (UM-168).

Windows platforms

```
cl -c -I<install_dir>\modeltech\include app.c
link -dll -export:<init_function> app.obj \
    <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

For the Verilog PLI, the <init_function> should be "init_usertfs". Alternatively, if there is no init_usertfs function, the <init_function> specified on the command line should be "veriusertfs". For the Verilog VPI, the <init_function> should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

The PLI and VPI have been tested with DLLs built using Microsoft Visual C/C++ compiler version 4.1 or greater.

The gcc compiler *cannot* be used to compile PLI/VPI applications under Windows. This is because gcc does not support the Microsoft *.lib/.dll* format.

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

If you need to run the "Performance Analyzer" (UM-407) on a design that contains PLI/VPI code, add these two switches to the link command shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

32-bit Linux platform

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

gcc compiler:

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -shared -E -Bsymbolic -o app.so app.o -lc
```

When using -Bsymbolic with ld, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored. If you are using ModelSim on Redhat version 6.0 through 7.1, you also need to add the -noinhibit-exec switch when you specify -Bsymbolic.

The compiler switch -freg-struct-return must be used when compiling any FLI application code that contains foreign functions that return real or time values.

64-bit Linux for IA64 platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

gcc compiler (gcc 3.2 or later)

```
gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library libm, specify '-lm' to the 'ld' command:

```
gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

32-bit Solaris platform

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

gcc compiler

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -G -B symbolic -o app.so app.o -lc
```

cc compiler

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -G -B symbolic -o app.so app.o -lc
```

When using **-B symbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

If *app.so* is not in your current directory you must tell Solaris where to search for the shared object. You can do this one of two ways:

- Add a path before *app.so* in the foreign attribute specification. (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
  LD_LIBRARY_PATH= <*library path without filename*>

64-bit Solaris platform

gcc compiler

```
gcc -c -I<install_dir>/modeltech/include -m64 -fpic app.c
gcc -shared -o app.so -m64 app.o
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the LD_LIBRARY_PATH environment variable.

cc compiler

```
cc -v -xarch=v9 -O -I<install_dir>/modeltech/include -c app.c
ld -G -B symbolic app.o -o app.so
```

When using **-B symbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

32-bit HP700 platform

A shared library is created by creating object files that contain position-independent code (use the **+z** or **-fpic** compiler argument) and by linking as a shared library (use the **-b** linker argument).

If your PLI/VPI application uses anything from a system library, you’ll need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

gcc compiler

```
gcc -c -fpic -I/<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o -lc
```

cc compiler

```
cc -c +z +DD32 -I/<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o -lc
```

Note that **-fpic** may not work with all versions of gcc.

64-bit HP platform

cc compiler

```
cc -v +DD64 -O -I<install_dir>/modeltech/include -c app.c
ld -b -o app.so app.o -lc
```

64-bit HP for IA64 platform

cc compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

```
cc -c +DD64 -I/<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

```
cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
ld -b -o math_app.sl math_app.o -lm
```

32-bit IBM RS/6000 platform

ModelSim loads shared libraries on the IBM RS/6000 workstation. The shared library must import ModelSim's PLI/VPI symbols, and it must export the PLI or VPI application's initialization function or table. ModelSim's export file is located in the ModelSim installation directory in *rs6000/mti_exports*.

If your PLI/VPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command. The resulting object must be marked as shared reentrant using these **gcc** or **cc** compiler commands for AIX 4.x:

gcc compiler

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -o app.sl app.o -bE:app.exp \
    -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

cc compiler

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -o app.sl app.o -bE:app.exp \
    -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

The *app.exp* file must export the PLI/VPI initialization function or table. For the PLI, the exported symbol should be "init_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". For the VPI, the exported symbol should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the shared object.

When using AIX 4.3 in 32-bit mode, you must add the **-DUSE_INTTYPES** switch to the compile command lines. This switch prevents a name conflict that occurs between *inttypes.h* and *mti.h*.

64-bit IBM RS/6000 platform

Only version 4.3 of AIX supports the 64-bit platform. A gcc 64-bit compiler is not available at this time.

cc compiler

```
cc -c -q64 -I/<install_dir>/modeltech/include app.c
ld -o app.s1 app.o -b64 -bE:app.exports \
    -bI:/<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

Compiling and linking PLI/VPI C++ applications

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

Since ModelSim's PLI/VPI functions have a standard C prototype, you must prevent the C++ compiler from mangling the PLI/VPI function names. This can be accomplished by using the following type of extern:

```
extern "C"
{
  <PLI/VPI application function prototypes>
}
```

The header files *veriuser.h*, *acc_user.h*, and *vpi_user.h* already include this type of extern. You must also put the PLI/VPI shared library entry point (veriusertfs, init_usertfs, or vlog_startup_routines) inside of this type of extern.

The following platform-specific instructions show you how to compile and link your PLI/VPI C++ applications so that they can be loaded by ModelSim. Microsoft Visual C++ is supported for creating Windows DLLs while GNU C++ and native C++ compilers are supported for creating UNIX shared libraries.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see "Specifying the PLI/VPI file to load" (UM-168).

Windows platforms

Microsoft Visual C++:

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj \
     <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

The **-GX** argument enables exception handling.

For the Verilog PLI, the **<init_function>** should be "init_usertfs". Alternatively, if there is no init_usertfs function, the **<init_function>** specified on the command line should be "veriusertfs". For the Verilog VPI, the **<init_function>** should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

The GNU C++ compiler *cannot* be used to compile PLI/VPI applications under Windows. This is because GNU C++ does not support the Microsoft *.lib*/*.dll* format.

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

If you need to run the "Performance Analyzer" (UM-407) on a design that contains PLI/VPI code, add these two switches to the link command shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

32-bit Linux platform

GNU C++ version 2.95.3

```
c++ -c -fPIC -I<install_dir>/modeltech/include app.C
c++ -shared -fPIC -o app.so app.o
```

64-bit Linux for IA64 platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

gcc version 3.2 or later

```
c++ -c -fPIC -I/<install_dir>/include app.C
c++ -shared -fPIC -o app.sl app.o
```

If your PLI/VPI application requires a user or vendor-supplied C++ library, or an additional system library, you will need to specify that library when you link your PLI/VPI application.

32-bit Solaris platform

Sun WorkShop version 5.0

```
CC -c -Kpic -o app.o -I<install_dir>/modeltech/include app.C
CC -G -o app.so app.o -lCstd -lCrun
```

GNU C++ version 2.95.3

```
c++ -c -fPIC -I<install_dir>/modeltech/include app.C
c++ -shared -fPIC -o app.so app.o
```

LD_LIBRARY_PATH must be set to point to the directory containing *libstdc++.so* so that the simulator can find this shared object.

64-bit Solaris platform

Sun WorkShop version 5.0

```
CC -c -v -xcode=pic32 -xarch=v9 -o app.o \
   -I<install_dir>/modeltech/include app.C
CC -G -xarch=v9 -o app.so app.o -lCstd -lCrun
```

32-bit HP-UX platform

C++ shared libraries are supported only on HP-UX 11.0 and later operating system versions.

HP C++ version 3.25

```
aCC -c +DAportable +Z -o app.o -I<install_dir>/modeltech/include app.C
aCC -v -b -o app.so app.o -lstd -lstream -lCsup
```

HP C++ version 3.3 and above

For I/O streams such as *cout* to work correctly within shared objects, HP's new *iostream* library must be used. Access the library by compiling all C++ source files with the **-AA** option. When building the shared object, use **-lstd_v2** instead of **-lstd**, and use **-lCsup_v2** instead of **-lCsup**. See the release notes in */opt/aCC/newconfig* for more details.

```
aCC -c +DAportable +Z -AA -o app.o -I<install_dir>/modeltech/include app.C
aCC -v -b -o app.so app.o -lstd_v2 -lstream -lCsup_v2
```

GNU C++ version 2.95.3

```
c++ -c -fPIC -I<install_dir>/modeltech/include app.C
c++ -shared -fPIC -o app.so app.o
```

Exceptions are not supported.

When ModelSim loads GNU C++ shared libraries on HP-UX, it calls the constructors and destructors only for the shared libraries that it loads directly. Libraries loaded as a result of ModelSim loading a shared library do not have their constructors and destructors called.

64-bit HP-UX platform

HP C++ version 3.25

```
aCC -c +DA2.0W +z -o app.o -I<install_dir>/modeltech/include app.C
aCC -v +DA2.0W -b -o app.so app.o -lstd -lstream -lCsup
```

64-bit HP for IA64 platform

HP C++ (/opt/aCC/bin/aCC)

```
aCC -c +DD64 -z -o app.o -I/<install_dir>/include app.C
aCC -b +DD64 -z -o app.sl app.o -lstd_v2 -lCsup
```

If your PLI/VPI application requires a user or vendor-supplied C++ library, or an additional system library, you will need to specify that library when you link your PLI/VPI application.

32-bit IBM RS/6000 platform

IBM C++ version 3.6

```
xlC -c -o app.o -I<install_dir>/modeltech/include app.C
makeC++SharedLib -o app.sl \
     -bI:<install_dir>/modeltech/rs6000/mti_exports -p 10 app.o
```

64-bit IBM RS/6000 platform

IBM C++ version 3.6

```
xlC -q64 -c -o app.o -I<install_dir>/modeltech/include app.C
makeC++SharedLib -o app.sl -X64 \
     -bI:<install_dir>/modeltech/rs64/mti_exports -p 10 app.o
```

Specifying the PLI/VPI file to load

The PLI/VPI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a **-pli** argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

▶ **Note:** On Windows platforms, the file names shown above should end with *.dll* rather than *.so*.

The various methods of specifying PLI/VPI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

See also *Appendix A - ModelSim variables* for more information on the *modelsim.ini* file.

PLI example

The following example is a trivial, but complete PLI application.

```
hello.c:

    #include "veriuser.h"
    static PLI_INT32 hello()
    {
        io_printf("Hi there\n");
        return 0;
    }
    s_tfcell veriusertfs[] = {
        {usertask, 0, 0, 0, hello, 0, "$hello"},
        {0}  /* last entry must be 0 */
    };

hello.v:

    module hello;
        initial $hello;
    endmodule

Compile the PLI code for the Solaris operating system:

    % cc -c -I<install_dir>/modeltech/include hello.c
    % ld -G -o hello.sl hello.o

Compile the Verilog code:

    % vlib work
    % vlog hello.v

Simulate the design:

    % vsim -c -pli hello.sl hello
    # Loading work.hello
    # Loading ./hello.sl
    VSIM 1> run -all
    # Hi there
    VSIM 2> quit
```

VPI example

The following example is a trivial, but complete VPI application. A general VPI example can be found in *<install_dir>/modeltech/examples/vpi*.

```
hello.c:

    #include "vpi_user.h"
    static PLI_INT32 hello(PLI_BYTE8 * param)
    {
        vpi_printf( "Hello world!\n" );
        return 0;
    }

    void RegisterMyTfs( void )
    {
        s_vpi_systf_data systf_data;
        vpiHandle systf_handle;
        systf_data.type        = vpiSysTask;
        systf_data.sysfunctype = vpiSysTask;
        systf_data.tfname      = "$hello";
        systf_data.calltf      = hello;
        systf_data.compiletf   = 0;
        systf_data.sizetf      = 0;
        systf_data.user_data   = 0;
        systf_handle = vpi_register_systf( &systf_data );
        vpi_free_object( systf_handle );
    }

    void (*vlog_startup_routines[])() = {
        RegisterMyTfs,
        0
    };

hello.v:

    module hello;
        initial $hello;
    endmodule

Compile the VPI code for the Solaris operating system:

    % gcc -c -I<install_dir>/include hello.c
    % ld -G -o hello.sl hello.o

Compile the Verilog code:

    % vlib work
    % vlog hello.v

Simulate the design:

    % vsim -c -pli hello.sl hello
    # Loading work.hello
    # Loading ./hello.sl
    VSIM 1> run -all
    # Hello world!
    VSIM 2> quit
```

The PLI callback reason argument

The second argument to a PLI callback function is the reason argument. The values of the various reason constants are defined in the veriuser.h include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the misctf callback functions under the following circumstances:

`reason_endofcompile`
For the completion of loading the design.

`reason_finish`
For the execution of the $finish system task or the **quit** command.

`reason_startofsave`
For the start of execution of the **checkpoint** command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to tf_write_save() until it is called with reason_save.

`reason_save`
For the execution of the **checkpoint** command. This is when the PLI application must save its state with calls to tf_write_save().

`reason_startofrestart`
For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to tf_read_restart() until it is called with reason_restart. The reason_startofrestart value is passed only for a restore command, and not in the case that the simulator is invoked with -restore.

`reason_restart`
For the execution of the **restore** command. This is when the PLI application must restore its state with calls to tf_read_restart().

`reason_reset`
For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be reloaded. (See the **`-keeploaded`** (CR-360) and **`-keeploadedrestart`** (CR-360) arguments to **vsim** for related information.)

`reason_endofreset`
For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`
For the execution of the $stop system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`
For the execution of the **environment** command or selecting a scope in the Structure window. Also for the call to acc_set_interactive_scope() if the callback_flag argument is non-zero.

`reason_paramvc`
For the change of value on the system task or function argument.

`reason_synch`
For the end of time step event scheduled by tf_synchronize().

`reason_rosynch`
For the end of time step event scheduled by tf_rosynchronize().

`reason_reactivate`
For the simulation event scheduled by tf_setdelay().

`reason_paramdrc`
Not supported in ModelSim Verilog.

`reason_force`
Not supported in ModelSim Verilog.

`reason_release`
Not supported in ModelSim Verilog.

`reason_disable`
Not supported in ModelSim Verilog.

The sizetf callback function

A user-defined system function specifies the width of its return value with the sizetf callback function, and the simulator calls this function while loading the design. The following details on the sizetf callback function are not found in the IEEE Std 1364:

- If you omit the sizetf function, then a return width of 32 is assumed.
- The sizetf function should return 0 if the system function return value is of Verilog type "real".
- The sizetf function should return -32 if the system function return value is of Verilog type "integer".

PLI object handles

Many of the object handles returned by the PLI ACC routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the acc_close() routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after acc_close() is called. The following object types are created on demand in ModelSim Verilog:

```
accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and
    acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)
```

If your PLI application uses these types of objects, then it is important to call acc_close() to free the memory allocated for these objects when the application is done using them.

If your PLI application places value change callbacks on accRegBit or accTerminal objects, *do not* call acc_close() while these callbacks are in effect.

Third party PLI applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a veriuser.c file. The veriuser.c file contains the registration information as described above in "Registering PLI applications" (UM-155). To prepare the application for ModelSim Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see "Compiling and linking PLI/VPI C applications" (UM-159)). For example, if you have a *veriuser.c* file and a library archive *libapp.a* file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriuser.c
% ld -G -o app.sl veriuser.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the **Veriuser** entry in the *modesim.ini* file, the **-pli** simulator argument, or the PLIOBJS environment variable (see "Registering PLI applications" (UM-155)).

▶ **Note:** On the HP700 platform, the object files must be compiled as position-independent code by using the **+z** compiler argument. Since, the object files supplied for Verilog-XL may be compiled for static linking, you may not be able to use the object files to create a dynamically loadable object for ModelSim Verilog. In this case, you must get the third party application vendor to supply the object files compiled as position-independent code.

Support for VHDL objects

The PLI ACC routines also provide limited support for VHDL objects in either an all VHDL design or a mixed VHDL/Verilog design. The following table lists the VHDL objects for which handles may be obtained and their type and fulltype constants:

Type	Fulltype	Description
accArchitecture	accArchitecture	instantiation of an architecture
accArchitecture	accEntityVitalLevel0	instantiation of an architecture whose entity is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel0	instantiation of an architecture which is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel1	instantiation of an architecture which is marked with the attribute VITAL_Level1
accArchitecture	accForeignArch	instantiation of an architecture which is marked with the attribute FOREIGN and which does not contain any VHDL statements or objects other than ports and generics
accArchitecture	accForeignArchMixed	instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics
accBlock	accBlock	block statement
accForLoop	accForLoop	for loop statement
accForeign	accShadow	foreign scope created by mti_CreateRegion()
accGenerate	accGenerate	generate statement
accPackage	accPackage	package declaration
accSignal	accSignal	signal declaration

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the Structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes. However, some of these objects can be manipulated through the Model*Sim* VHDL foreign interface (mti_* routines). See the *FLI Reference Manual* for more information.

IEEE Std 1364 ACC routines

ModelSim Verilog supports the following ACC routines, described in detail in the IEEE Std 1364.

acc_append_delays	acc_append_pulsere	acc_close
acc_collect	acc_compare_handles	acc_configure
acc_count	acc_fetch_argc	acc_fetch_argv
acc_fetch_attribute	acc_fetch_attribute_int	acc_fetch_attribute_str
acc_fetch_defname	acc_fetch_delay_mode	acc_fetch_delays
acc_fetch_direction	acc_fetch_edge	acc_fetch_fullname
acc_fetch_fulltype	acc_fetch_index	acc_fetch_location
acc_fetch_name	acc_fetch_paramtype	acc_fetch_paramval
acc_fetch_polarity	acc_fetch_precision	acc_fetch_pulsere
acc_fetch_range	acc_fetch_size	acc_fetch_tfarg
acc_fetch_itfarg	acc_fetch_tfarg_int	acc_fetch_itfarg_int
acc_fetch_tfarg_str	acc_fetch_itfarg_str	acc_fetch_timescale_info
acc_fetch_type	acc_fetch_type_str	acc_fetch_value
acc_free	acc_handle_by_name	acc_handle_calling_mod_m
acc_handle_condition	acc_handle_conn	acc_handle_hiconn
acc_handle_interactive_scope	acc_handle_loconn	acc_handle_modpath
acc_handle_notifier	acc_handle_object	acc_handle_parent
acc_handle_path	acc_handle_pathin	acc_handle_pathout
acc_handle_port	acc_handle_scope	acc_handle_simulated_net
acc_handle_tchk	acc_handle_tchkarg1	acc_handle_tchkarg2
acc_handle_terminal	acc_handle_tfarg	acc_handle_itfarg
acc_handle_tfinst	acc_initialize	acc_next
acc_next_bit	acc_next_cell	acc_next_cell_load
acc_next_child	acc_next_driver	acc_next_hiconn
acc_next_input	acc_next_load	acc_next_loconn
acc_next_modpath	acc_next_net	acc_next_output
acc_next_parameter	acc_next_port	acc_next_portout

acc_next_primitive	acc_next_scope	acc_next_specparam
acc_next_tchk	acc_next_terminal	acc_next_topmod
acc_object_in_typelist	acc_object_of_type	acc_product_type
acc_product_version	acc_release_object	acc_replace_delays
acc_replace_pulsere	acc_reset_buffer	acc_set_interactive_scope
acc_set_pulsere	acc_set_scope	acc_set_value
acc_vcl_add	acc_vcl_delete	acc_version

acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

IEEE Std 1364 TF routines

ModelSim Verilog supports the following TF routines, described in detail in the IEEE Std 1364.

io_mcdprintf	io_printf	mc_scan_plusargs
tf_add_long	tf_asynchoff	tf_iasynchoff
tf_asynchon	tf_iasynchon	tf_clearalldelays
tf_iclearalldelays	tf_compare_long	tf_copypvc_flag
tf_icopypvc_flag	tf_divide_long	tf_dofinish
tf_dostop	tf_error	tf_evaluatep
tf_ievaluatep	tf_exprinfo	tf_iexprinfo
tf_getcstringp	tf_igetcstringp	tf_getinstance
tf_getlongp	tf_igetlongp	tf_getlongtime
tf_igetlongtime	tf_getnextlongtime	tf_getp
tf_igetp	tf_getpchange	tf_igetpchange
tf_getrealp	tf_igetrealp	tf_getrealtime
tf_igetrealtime	tf_gettime	tf_igettime
tf_gettimeprecision	tf_igettimeprecision	tf_gettimeunit
tf_igettimeunit	tf_getworkarea	tf_igetworkarea
tf_long_to_real	tf_longtime_tostr	tf_message
tf_mipname	tf_imipname	tf_movepvc_flag
tf_imovepvc_flag	tf_multiply_long	tf_nodeinfo
tf_inodeinfo	tf_nump	tf_inump
tf_propagatep	tf_ipropagatep	tf_putlongp
tf_iputlongp	tf_putp	tf_iputp
tf_putrealp	tf_iputrealp	tf_read_restart
tf_real_to_long	tf_rosynchronize	tf_irosynchronize
tf_scale_longdelay	tf_scale_realdelay	tf_setdelay
tf_isetdelay	tf_setlongdelay	tf_isetlongdelay
tf_setrealdelay	tf_isetrealdelay	tf_setworkarea
tf_isetworkarea	tf_sizep	tf_isizep

tf_spname	tf_ispname	tf_strdelputp
tf_istrdelputp	tf_strgetp	tf_istrgetp
tf_strgettime	tf_strlongdelputp	tf_istrlongdelputp
tf_strrealdelputp	tf_istrrealdelputp	tf_subtract_long
tf_synchronize	tf_isynchronize	tf_testpvc_flag
tf_itestpvc_flag	tf_text	tf_typep
tf_itypep	tf_unscale_longdelay	tf_unscale_realdelay
tf_warning	tf_write_save	

Verilog-XL compatible routines

The following PLI routines are not defined in IEEE Std 1364, but ModelSim Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc_decompile_expr** routine. The condition argument must be a handle obtained from the acc_handle_condition routine. The value returned by **acc_decompile_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the aof_hightime argument.

Using 64-bit ModelSim with 32-bit PLI/VPI Applications

If you have 32-bit PLI/VPI applications and wish to use 64-bit ModelSim, you will need to port your code to 64 bits by moving from the ILP32 data model to the LP64 data model. We strongly recommend that you consult the 64-bit porting guides for Sun and HP.

64-bit support in the PLI

The PLI function acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

PLI/VPI tracing

The foreign interface tracing feature is available for tracing PLI and VPI function calls. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files that can be used to replay what the foreign interface code did.

The purpose of tracing files

The purpose of the logfile is to aid you in debugging PLI or VPI code. The primary purpose of the replay facility is to send the replay files to MTI support for debugging co-simulation problems, or debugging PLI/VPI problems for which it is impractical to send the PLI/VPI code. We still need you to send the VHDL/Verilog part of the design to actually execute a replay, but many problems can be resolved with the trace only.

Invoking a trace

To invoke the trace, call **vsim** (CR-357) with the **-trace_foreign** argument:

Syntax

```
vsim
  -trace_foreign <action> [-tag <name>]
```

Arguments

`<action>`

Specifies one of the following actions:

Value	Action	Result
1	create log only	writes a local file called "mti_trace_<tag>"
2	create replay only	writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c"
3	create both log and replay	

`-tag <name>`

Used to give distinct file names for multiple traces. Optional.

Examples

```
vsim -trace_foreign 1 mydesign
```

Creates a logfile.

```
vsim -trace_foreign 3 mydesign
```

Creates both a logfile and a set of replay files.

```
vsim -trace_foreign 1 -tag 2 mydesign
```

Creates a logfile with a tag of "2".

The tracing operations will provide tracing during all user foreign code-calls, including PLI/VPI user tasks and functions (calltf, checktf, sizetf and misctf routines), and Verilog VCL callbacks.

Debugging PLI/VPI application code

ModelSim Versions 5.7 and later offer the optional C Debug feature. This tool allows you to interactively debug SystemC/C/C++ source code with the open-source **gdb** debugger. See *Chapter 14 - C Debug* for details. If you don’t have access to C Debug, continue reading for instructions on how to attach to an external C debugger.

In order to debug your PLI/VPI application code in a debugger, your application code must be compiled with debugging information (for example, by using the **-g** option) and without optimizations (for example, don’t use the **-O** option). You must then load **vsim** into a debugger. Even though **vsim** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **vsimk**, the simulation kernal where your application code is loaded (for example, "`ddd `which vsimk``"), or you can attach the debugger to an already running **vsim** process. In the second case, you must attach to the PID for **vsimk**, and you must specify the full path to the **vsimk** executable (for example, "`gdb $MTI_HOME/sunos5/vsimk 1234`").

On Solaris, AIX, and Linux systems you can use either **gdb** or **ddd**. On HP-UX systems you can use the **wdb** debugger from HP. You will need version 1.2 or later.

Since initially the debugger recognizes only **vsim's** PLI/VPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI function that is called by your application code. An easy way to set an entry point is to put a call to acc_product_version() as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run **vsim** with the usual arguments (e.g., "run -c top").

On HP-UX you might see some warning messages that **vsim** does not have debugging information available. This is normal. If you are using Exceed to access an HP machine from Windows NT, it is recommended that you run **vsim** in command line or batch mode because your NT machine may hang if you run **vsim** in GUI mode. Click on the "go" button, or use F5 or the **go** command to execute **vsim** in **wdb**.

When the breakpoint is reached, the shared library containing your application code has been loaded. In some debuggers you must use the **share** command to load the PLI/VPI application's symbols.

On HP-UX you might see a warning about not finding "__dld_flags" in the object file. This warning can be ignored. You should see a list of libraries loaded into the debugger. It should include the library for your PLI/VPI application. Alternatively, you can use **share** to load only a single library.

At this point all of the PLI/VPI application's symbols should be visible. You can now set breakpoints in and single step through your PLI/VPI application code.

7 - SystemC simulation

Chapter contents

This chapter describes how to compile and simulate SystemC designs with ModelSim. Proper name-binding is critical for your success. Read "Name association (binding)" (UM-204) for information on correctly naming signals, ports and modules in your SystemC design. ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.0.1 reference simulator. It is recommended that you obtain the OSCI functional specification as a reference manual. Visit *http://www.systemc.org* for details.

In addition to the functionality described in the OSCI specification, ModelSim for SystemC includes the following features:

- Single common Graphic Interface for SystemC and HDL languages.
- Extensive support for mixing SystemC, VHDL, and Verilog in the same design (SDF annotation for HDL only). For detailed information on mixing SystemC with HDL see *Chapter 8 - Mixed-language simulations*.

Supported platforms and compiler versions

SystemC runs on a subset of ModelSim supported platforms. The table below shows the currently supported platforms and compiler versions:

Platform	Supported compiler versions
HP-UX 11.0 or later	aCC 3.45 with associated patches
RedHat Linux 7.2 RedHat Linux Enterprise version 2.1	gcc 3.2.3
RedHat Linux 7.3 or later	gcc 3.2 *or* gcc 3.2.3
SunOS 5.6 or later	gcc 3.2

Important: ModelSim SystemC has been tested with the gcc versions available from *ftp.model.com/pub/gcc*. Customized versions of gcc may cause problems. We strongly encourage you to download and use the gcc versions available on our FTP site (login as anonymous).

Building gcc with custom configuration options

We only test with our default options. **If you use advanced gcc configuration options, we cannot guarantee that ModelSim will work with those options.**

To use a custom gcc build, set the CppPath variable in the *modelsim.ini* file. This variable specifies the pathname to the compiler binary you intend to use.

When using a custom gcc, ModelSim requires that the custom gcc be built with several specific configuration options. These vary on a per-platform basis as shown in the following table:

Platform	Mandatory configuration options
Linux	none
Solaris	--with-gnu-ld --with-ld=/path/to/binutils-2.14/bin/ld --with-gnu-as --with-as=/path/to/binutils-2.14/bin/as
HP-UX	N/A

If you don't have a GNU binutils2.14 assembler and linker handy, you can use the as and ld programs distributed with ModelSim. They are located inside the built-in gcc in directory *<install_dir>/modeltech/gcc-3.2-<mtiplatform>/lib/gcc-lib/<gnuplatform>/3.2*.

By default ModelSim also uses the following options when configuring built-in gcc.

- --disable-nls
- --enable-languages=c,c++

These are not mandatory, but they do reduce the size of the gcc installation.

Usage flow for SystemC-only designs

ModelSim allows users to simulate SystemC, either alone or in combination with other VHDL/Verilog modules. The following is an overview of the usage flow for strictly SystemC designs. More detailed instructions are presented in the sections that follow.

1 Create and map the working design library with the **vlib** and **vmap** statements, as appropriate to your needs.

2 Modify the SystemC source code as follows:

- Replace **sc_main()** with an SC_MODULE, and potentially add a process to contain any testbench code
- Replace **sc_start()** by using the **run** (CR-246) command in the GUI
- Remove calls to **sc_initialize()**
- Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macro
- Verify that SystemC signal, ports and modules are explicitly named to avoid port binding and debugging errors. See "Name association (binding)" (UM-204).

3 Analyze the SystemC source using **sccom** (CR-248). **sccom** invokes the native C++ compiler to create the C++ object files in the design library.

See "Using sccom vs. raw C++ compiler" (UM-195) for information on when you are required to use sccom vs. another C++ compiler.

4 Perform a final link of the C++ source using **sccom -link** (UM-197). This process creates a shared object file in the current work library which will be loaded by **vsim** at runtime. **sccom -link** must be re-run before simulation if any new **sccom** compiles were performed.

5 Simulate the design using the standard **vsim** command.

6 Simulate the design using the **run** command, entered at the **vsim** command prompt.

7 Debug the design using ModelSim GUI features, including the Source and Wave windows.

Compiling SystemC designs

To compile SystemC designs, you must

- create a design library
- make a few modifications to the SystemC source code
- run the **sccom** (CR-248) SystemC compiler.
- run the **sccom** (CR-248) SystemC linker (sccom -link)

Creating a design library

Before you can compile your design, you must create a library in which to store the compilation results. Use **vlib** (CR-344) to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX commands – always use the **vlib** command (CR-344).

See "Design libraries" (UM-53) for additional information on working with libraries.

Modifying SystemC source code

Several modifications are required to your original SystemC source code.

- Convert **sc_main()** to a module.
  In order for ModelSim to run the SystemC/C++ source code, the control function of **sc_main()** must be replaced by a constructor placed within a module at the top level of the design. The example shown below uses a module called **mytop**. Any testbench code inside **sc_main()** should be moved to a process, normally an SC_THREAD process. Also, any **sc_clock()** functions must be moved into the constructor.
- Replace the **sc_start()** function with the **run** command and options.
  ModelSim uses the **run** command and its options in place of the **sc_start()** function. If **sc_main()** has multiple **sc_start()** mixed in with the testbench code, then use an **SC_THREAD()** with wait statements to emulate the same behavior. An example of this is shown below
- Remove calls to **sc_initialize()**.
  vsim calls **sc_initialize()** by default at the end of elaboration.
- Export all top SystemC modules.
  For SystemC designs, you must export all top level modules in your design to ModelSim. You do this with the **SC_MODULE_EXPORT(<sc_module_name>)** macro. SystemC templates are not supported as top level or boundary modules. See "Templatized SystemC modules" (UM-196). The **sc_module_name** is the name of the top level module to be simulated in ModelSim. You must specify this macro in any C++ source (*.cpp*) file. If the macro is contained in a header file instead of a C++ source file, an error may result. See "sccom -link errors" (UM-197) for more information.
- Replace any VCD dump file generation functions with appropriate GUI commands.

Examples

The following is a simple example of how to convert **sc_main** to a module and elaborate it with **vsim**.

Original code (partial)	Modified code (partial)

`int sc_main(int argc, char* argv[])`
`{`
`     sc_signal<bool> mysig;`
`     mymod mod("mod");`
`     mod.outp(mysig);`
 	
`     sc_start(100, SC_NS);`
`}`
	`SC_MODULE(mytop)`
	`{`
	`     sc_signal<bool> mysig;`
	`     mymod mod;`
	
	`     SC_CTOR(mytop)`
	`         : mysig("mysig"),`
	`           mod("mod")`
	`     {`
	`         mod.outp(mysig);`
	`     }`
	`};`
	
	`SC_MODULE_EXPORT(mytop);`

The run command equivalent to the `sc_start(100, SC_NS)` statement is:

```
run 100 ns
```

This next example is slightly more complex, illustrating the use of **sc_main()** and signal assignments, and how you would get the same behavior using ModelSim.

Original OSCI code (partial)

```
int sc_main(int, char**)
{
    sc_signal<bool> reset;
    counter_top top("top");
    sc_clock CLK("CLK", 10, SC_NS, 0.5,
0.0, SC_NS, false);

    top.reset(reset);

    reset.write(1);
    sc_start(5, SC_NS);
    reset.write(0);
    sc_start(100, SC_NS);
    reset.write(1);
    sc_start(5, SC_NS);
    reset.write(0);
    sc_start(100, SC_NS);
}
```

Modified ModelSim code (partial)

```
SC_MODULE(new_top)
{
    sc_signal<bool> reset;
    counter_top top;
    sc_clock CLK;

    void sc_main_body();

    SC_CTOR(new_top)
         : reset("reset"),
           top("top")
         { CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS,
false)
         top.reset(reset);
         SC_THREAD(sc_main_body);
    }
};

void
new_top::sc_main_body()
{
    reset.write(1);
    wait(5, SC_NS);
    reset.write(0);
    wait(100, SC_NS);
    reset.write(1);
    wait(5, SC_NS);
    reset.write(0);
    wait(100, SC_NS);
}

SC_MODULE_EXPORT(new_top);
```

Invoking the SystemC compiler

ModelSim compiles one or more SystemC design units with a single invocation of **sccom** (CR-248), the SystemC compiler. The design units are compiled in the order that they appear on the command line. For SystemC designs, all design units must be compiled just as they would be for any C++ compilation. An example of an **sccom** command might be:

```
sccom -I ../myincludes mytop.cpp mydut.cpp
```

Compiling optimized and/or debug code

By default, **sccom** invokes the C++ compiler (g++ or aCC) without any optimizations. If desired, you can enter any g++/aCC optimization arguments at the **sccom** command line.

Also, source level debug of SystemC code is not available by default in ModelSim. To compile your SystemC code for debug, use the g++/aCC **-g** argument on the **sccom** command line.

Specifying an alternate g++ installation

We recommend using the version of g++ that is shipped with ModelSim on its various supported platforms. However, if you want to use your own installation, you can do so by setting the CppPath variable in the *modelsim.ini* file to the g++ executable location.

For example, if your g++ executable is installed in */u/abc/gcc-3.2/bin*, then you would set the variable as follows:

```
CppPath /u/abc/gcc-3.2/bin
```

Maintaining portability between OSCI and ModelSim

If you intend to simulate on both ModelSim and the OSCI reference simulator, you can use the MTI_SYSTEMC macro to execute the ModelSim specific code in your design only when running ModelSim. The MTI_SYSTEMC macro is defined in the *systemc.h* header file, read automatically upon compile. By including #ifdef/else statements in the code, you can avoid having two copies of the design.

Using the original and modified code shown in the example shown on page 191, you might write the code as follows:

```
#ifdef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
     sc_signal<bool> mysig;
     mymod mod;

     SC_CTOR(mytop)
         : mysig("mysig"),
           mod("mod")
     {
         mod.outp(mysig);
     }
};

SC_MODULE_EXPORT(top);

#else   //Otherwise, it compiles this
int sc_main(int argc, char* argv[])
{
     sc_signal<bool> mysig;
     mymod mod("mod");
     mod.outp(mysig);

     sc_start(100, SC_NS);
}
#endif
```

Restrictions on compiling with HP aCC

ModelSim uses the **aCC -AA** option by default when compiling C++ files on HP-UX. It does this so *cout* will function correctly in the *systemc.so* file. The **-AA** option tells **aCC** to use ANSI-compliant <iostream> rather than cfront-style <iostream.h>. Thus, all C++-based objects in a program must be compiled with **-AA**. This means you must use <iostream> and "using" clauses in your code. Also, you cannot use the **-AP** option, which is incompatible with **-AA**.

Switching platforms and compilation

Compiled SystemC libraries are platform dependent. If you move between platforms, you must remove all SystemC files from the working library and then recompile your SystemC source files. To remove SystemC files from the working directory, use the **vdel** (CR-315) command with the **-allsystemc** argument.

If you attempt to load a design that was compiled on a different platform, an error such as the following occurs:

```
# vsim work.test_ringbuf
# Loading work/systemc.so

# ** Error: (vsim-3197) Load of "work/systemc.so" failed:
work/systemc.so: ELF file data encoding not little-endian.

# ** Error: (vsim-3676) Could not load shared library
work/systemc.so for SystemC module 'test_ringbuf'.

# Error loading design
```

You can type **verror 3197** at the **vsim** command prompt and get details about what caused the error and how to fix it.

Using sccom vs. raw C++ compiler

When compiling complex C/C++ testbench environments, it is common to compile code with many separate runs of the compiler. Often users compile code into archives (.a files), and then link the archives at the last minute using the -L and -l link options.

When using ModelSim's SystemC, you may wish to compile a portion of your C design using raw g++ or aCC instead of **sccom**. Perhaps you have some legacy code or some non-SystemC utility code that you want to avoid compiling with **sccom**. You can do this, however, some caveats and rules apply.

Rules for sccom use

The rules governing when and how you must use **sccom** are as follows:

1. You must compile all code that references SystemC types or objects using **sccom** (CR-248).
2. When using **sccom**, you should not use the -I compiler option to point the compiler at any search directories containing OSCI SystemC header files. **sccom** does this for you accurately and automatically.
3. If you do use the raw C++ compiler to compile C/C++ functionality into archives or shared objects, you must then link your design using the -L and -l options with the **sccom -link** command. These options effectively pull the non-SystemC C/C++ code into a simulation image that is used at runtime.

Failure to follow the above rules can result in link-time or elaboration-time errors due to mismatches between the OSCI SystemC header files and the ModelSim SystemC header files.

Rules for using raw g++ to compile non-SystemC C/C++ code

If you use raw g++ to compile your non-systemC C/C++ code, the following rules apply:

1. The -fPIC option to g++ should be used during compilation at the sccom command line.
2. For C++ code, you must use the built-in g++ delivered with ModelSim, or (if using a custom g++) use the one you built and specified with the CppPath .ini variable.

Otherwise binary incompatibilities may arise between code compiled by sccom and code compiled by raw g++.

Rules for using raw HP aCC to compile non-SystemC C/C++ code

If you use HP's aCC compiler to compile your non-systemC C/C++ code, the following rules apply:

1. For C++ code, you should use the +Z and -AA options during compilation
2. You must use HP aCC version 3.45 or higher.

Issues with C++ templates

Templatized SystemC modules

Templatized SystemC modules are not supported for use at:

- the top level of the design
- the boundary between SystemC and higher level HDL modules (i.e. the top level of the SystemC branch)

To convert a top level templatized SystemC module, you can either specialize the module to remove the template, or you can create a wrapper module that you can use as the top module.

For example, let's say you have a templatized SystemC module as shown below:

```
template <class T>
class top : public sc_module
{
    sc_signal<T> sig1;
    .
    .
    .
};
```

You can specialize the module by setting T = int, thereby removing the template, as follows:

```
class top : public sc_module
{
    sc_signal<int> sig 1;
    .
    .
    .
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```
class modelsim_top : public sc_module
{
    top<int> actual_top;
    .
    .
    .
};

SC_MODULE_EXPORT(modelsim_top);
```

Organizing templatized code

Suppose you have a class template, and it contains a certain number of member functions. All those member functions must be visible to the compiler when it compiles any instance of the class. For class templates, the C++ compiler generates code for each unique instance of the class template. Unless it can see the full implementation of the class template, it cannot generate code for it thus leaving the invisible parts as undefined. Since it is legal to have undefined symbols in a .so, sccom -link will not produce any errors or warnings.

To make functions visible to the compiler, you should move them to the .h file. ModelSim requires all functions defined in a .h file to be inlined. For relevant information on inlining functions, see "Multiple symbol definition errors" (UM-208).

Linking the compiled source

Once the design has been compiled, it must be linked using the **sccom** (CR-248) command with the **-link** argument.

sccom -link

The **sccom -link** command **argument** collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library. If you have changed your SystemC source code and recompiled it using **sccom**, then you must relink the design by running **sccom -link** before invoking **vsim**. Otherwise, your changes to the code are not recognized by the simulator. Remember that any dependent .a or .o files should be listed on the **sccom -link** command line before the .a or .o on which it depends. For more details on dependencies and other syntax issues, see **sccom** (CR-248).

sccom -link errors

Most errors occurring during **sccom -link** are due to multiple symbol definitions caused by incorrect symbol definitions in header files or by ModelSim's name association. For information on fixing errors encountered during linking, see "Errors during loading" (UM-206)

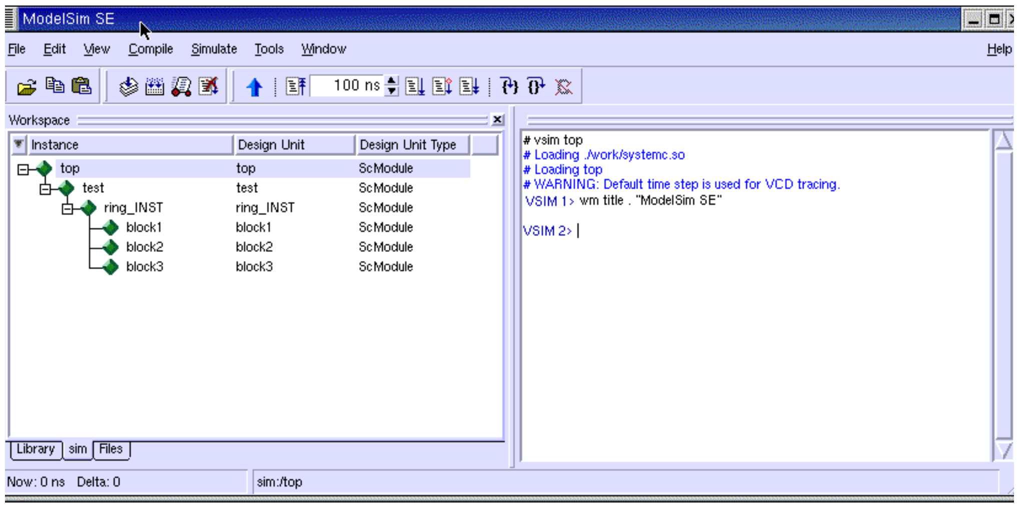
Simulating SystemC designs

After compiling the SystemC source code, you can simulate your design with **vsim** (CR-357). This section discusses simulation from the operating system command prompt.

For SystemC, invoke **vsim** (CR-357) with the top-level module of the design. This example invokes **vsim** (CR-357) on a design:

```
vsim top_level_module
```

When the GUI comes up, you can expand the hierarchy of the design to view the SystemC modules. SystemC objects are denoted by a green diamond.



Instance	Design Unit	Design Unit Type
top	top	ScModule
test	test	ScModule
ring_INST	ring_INST	ScModule
block1	block1	ScModule
block2	block2	ScModule
block3	block3	ScModule

```
# vsim top
# Loading ./work/systemc.so
# Loading top
# WARNING: Default time step is used for VCD tracing.
VSIM 1> wm title . "ModelSim SE"

VSIM 2> |
```

Simulator resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The default resolution limit is set to the value specified by the **Resolution** (UM-624) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-238) with the **simulator state** option.

Overriding the resolution

You can override ModelSim's default resolution by specifying the **-t** option on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps topmod
```

You need to be careful when doing this type of operation. If the resolution set by **-t** is larger than a delay value in your design (i.e. sc_wait (4,SC_PS);), the delay values in that design unit are rounded to the closest multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

In addition, you must keep in mind the relationship between the simulator's resolution and the SystemC time units specified in the source code. For example, with a time unit usage of:

```
sc_wait(10, SC_PS);
```

a simulator resolution of 10ps would be fine. No rounding off of the ones digits in the time units would occur. However, a specification of:

```
sc_wait(9, SC_PS);
```

requires setting the resolution limit to 1ps in order to avoid inaccuracies caused by rounding.

SystemC defaults to 1ps resolution.That means it is possible for the source code to contain calls which don't explicitly specify units. In such cases the SystemC resolution is 1ps. For example:

```
sc_clock("clk", 9);
```

In SystemC, the *sc_set_time_resolution()* function is used to change the default units during or after elaboration. This function is not supported in ModelSim, since by the time it could be called from an executing SystemC model, it would be too late to change the simulator resolution.

Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be set unnecessarily small because in some cases performance will be degraded.

Initialization and cleanup of SystemC state-based code

Constructors and Destructors should be reserved for creating and destroying SystemC design objects, such as `sc_modules` or `sc_signals`. The following 2.1 virtual functions should be used to initialize and clean up state-based code, such as logfiles or the VCD trace functionality of SystemC:

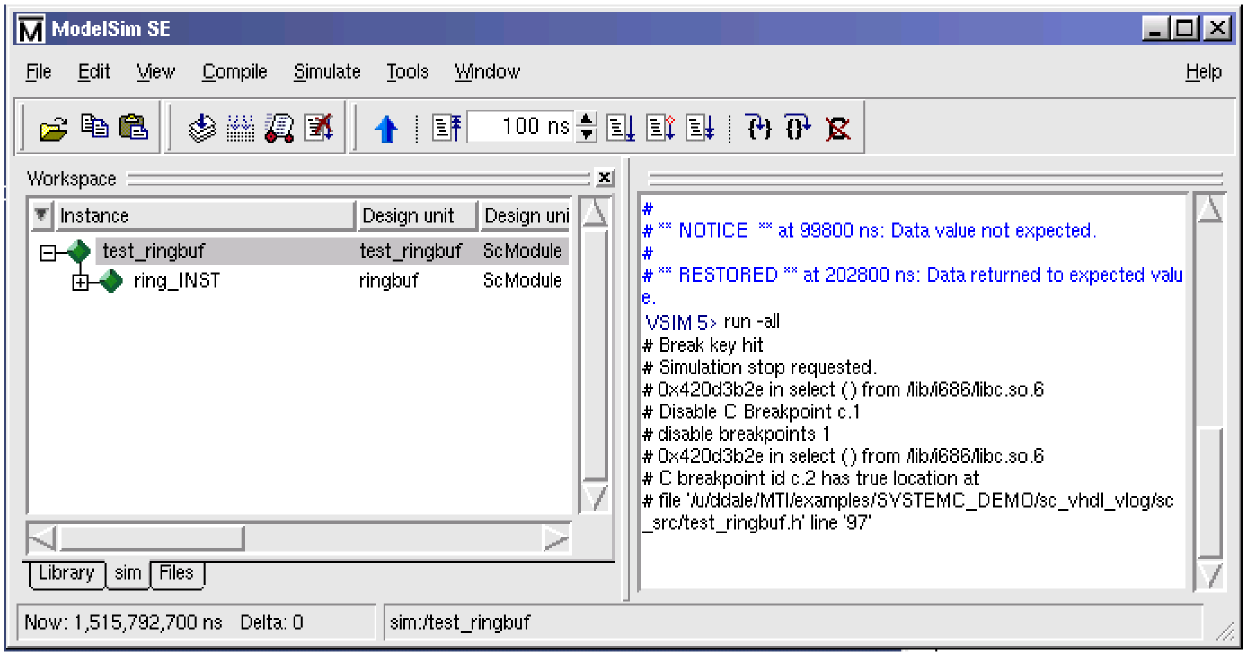
- end_of_construction ()
  Called after all constructors are called, but before port binding.
- end_of_elaboration ()
  Called at the end of elaboration after port binding. This function is available in the SystemC 2.0.1 reference simulator.
- start_of_simulation ()
  Called before simulation starts. Simulation specific initialization code can be placed in this function.
- end_of_simulation ()
  Called before ending the current simulation session.

The call sequence for these functions with respect to the SystemC object construction and destruction is as follows:

1. Constructors
2. end_of_construction ()
3. end_of_elaboration ()
4. start_of_simulation ()
5. end_of_simulation ()
6. Destructors

Debugging the design

All ModelSim's GUI debugging features within all windows, with the exception of Dataflow, are fully available for use with SystemC.



Instance	Design unit	Design uni
test_ringbuf	test_ringbuf	ScModule
ring_INST	ringbuf	ScModule

```
#
# ** NOTICE  ** at 99800 ns: Data value not expected.
#
# ** RESTORED ** at 202800 ns: Data returned to expected valu
e.
VSIM 5> run -all
# Break key hit
# Simulation stop requested.
# 0x420d3b2e in select () from /lib/i686/libc.so.6
# Disable C Breakpoint c.1
# disable breakpoints 1
# 0x420d3b2e in select () from /lib/i686/libc.so.6
# C breakpoint id c.2 has true location at
# file '/u/ddale/MTI/examples/SYSTEMC_DEMO/sc_vhdl_vlog/sc
_src/test_ringbuf.h' line '97'
```

Now: 1,515,792,700 ns Delta: 0 sim:/test_ringbuf

Source-level debug

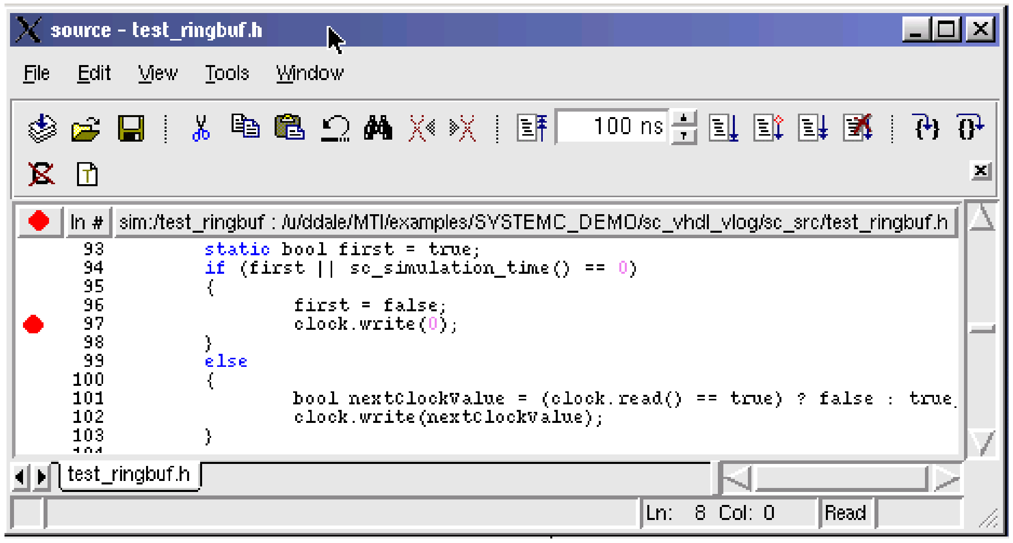
In order to debug your SystemC source code, you must compile the design for debug using the **-g** C++ compiler option. You can add this option directly to the **sccom** (CR-248) command line on a per run basis, with a command such as:

```
sccom mytop -g
```

Or, if you plan to use it every time you run the compiler, you can specify it in the *modelsim.ini* file with the **SccomCppOptions** variable. See "[sccom] SystemC compiler control variables" (UM-620) for more information.

The source code debugger, C Debug, is automatically invoked when the design is compiled for debug in this way.

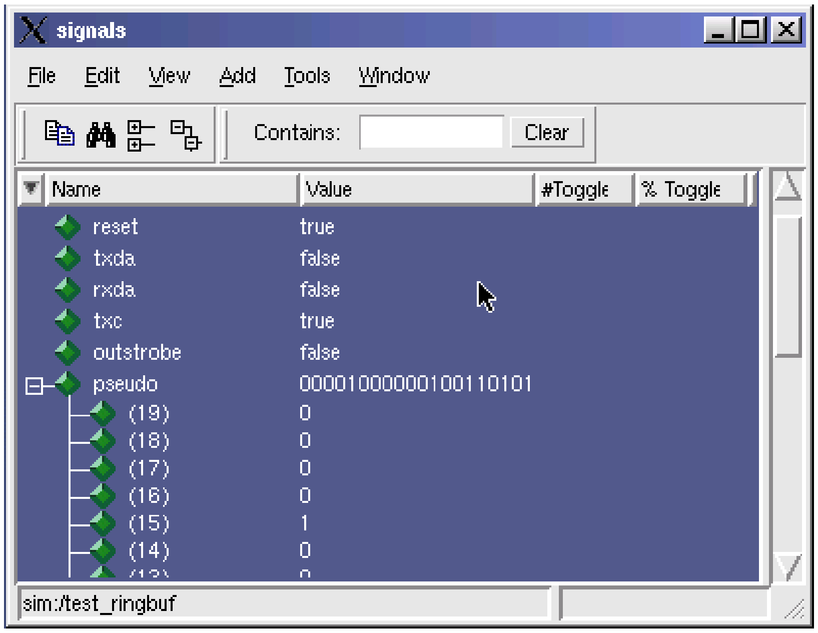
You can set breakpoints in the Source window, and single-step through your SystemC/C++ source code. .


source - test_ringbuf.h
File Edit View Tools Window
100 ns
ln #
sim:/test_ringbuf : /u/ddale/MTI/examples/SYSTEMC_DEMO/sc_vhdl_vlog/sc_src/test_ringbuf.h
test_ringbuf.h
Ln: 8 Col: 0
Read

```
        static bool first = true;
        if (first || sc_simulation_time() == 0)
        {
                first = false;
                clock.write(0);
        }
        else
        {
                bool nextClockValue = (clock.read() == true) ? false : true;
                clock.write(nextClockValue);
        }
```

The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Try to avoid setting breakpoints in constructors of SystemC objects; it may crash the debugger.

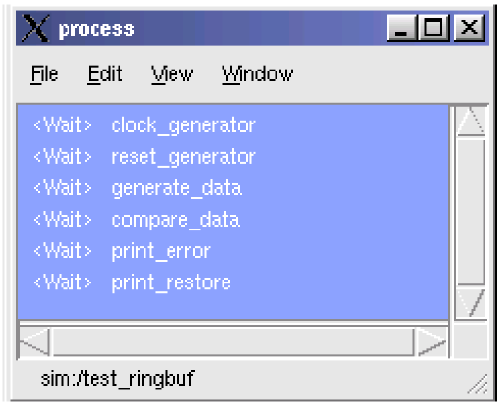
You can view and expand SystemC items in the Signals window.



Name	Value	#Toggle	% Toggle
reset	true		
txda	false		
rxda	false		
txc	true		
outstrobe	false		
pseudo	00001000000100110101		
(19)	0		
(18)	0		
(17)	0		
(16)	0		
(15)	1		
(14)	0		

sim:/test_ringbuf

You can also view the processes in the Process window.



```
<Wait>  clock_generator
<Wait>  reset_generator
<Wait>  generate_data
<Wait>  compare_data
<Wait>  print_error
<Wait>  print_restore
```

sim:/test_ringbuf

Differences between ModelSim and the OSCI simulator

ModelSim is based upon the 2.0.1 reference simulator provided by OSCI. However, there are some minor but key differences to understand:

- **vsim** calls **sc_initialize()** by default at the end of elaboration. The user has to explicitly call **sc_initialize()** in the reference simulator. You should remove calls to **sc_initialize()** from your code.
- The default time resolution of the reference simulator is 1ps. For **vsim** it is 1ns. The user can set the time resolution by using **vsim** command with the **-t** option or by modifying the value of the **resolution** variable in the *modelsim.ini* file.
- All SystemC processes without a **dont_initialize()** modifier are executed once at the end of elaboration. This can cause print messages to appear from user models before the first VSIM> prompt occurs. This behavior is normal and necessary in order to achieve compliance with both the SystemC and HDL LRM's.
- The **run** command in ModelSim is equivalent to **sc_start()**. In the reference simulator, **sc_start()** runs the simulation for the duration of time specified by its argument. In ModelSim the **run** command (CR-246) runs the simulation for the amount of time specified by its argument.
- The **delta_count()** function in the reference simulator returns the cumulative delta count. In **vsim**, it returns the delta count since the last time advance. The delta count in **vsim** is reset after every time advance.
- The **sc_cycle()**, **sc_start()**, **sc_main()** & **sc_set_time_resolution()** functions are not supported in ModelSim.

Name association (binding)

SystemC simulation objects such as modules, primitive channels and ports can be explicitly named by passing a name to the constructors of said objects. If an object is not constructed with an explicit name, then the OSCI reference simulator generates an internal name for it, using names such as "s0", "s1", etc..

ModelSim has implemented its own name association technology for SystemC, attempting to give reasonable names to the child objects of SC_MODULES, i.e. names that match the C++ source code names. ModelSim's name association automatically binds the C++ object name to any unnamed object.

For example, if the design has a primitive channel `sc_signal<bool> foo`; that is constructed without an explicit name, it is named `foo` in the simulator's database. Automatic name binding is enabled for each **sc_module** which makes use of the SC_CTOR constructor macro. If a module in the design doesn't use the SC_CTOR constructor macro, name binding can be enabled by adding the SC_MTI_BIND_NAME macro anywhere inside a public: access area of the module's declaration. See the following sample code:

```
SC_MODULE( mod_b )
{
public:
    sc_in<int> in;
    sc_out<int> out;

    SC_MTI_BIND_NAME;
private:
    int a;

public:
    mod_b(sc_module_name name);

};
```

Automatic name binding is supported only for modules declared in header (*.h*, *.hxx*) files. It is not supported when modules are declared in C++ source files (*.cpp*, *.cxx*, *.cc*, etc.).

Disabling automatic name binding

If a C++ source file contains a module that uses an SC_CTOR (or the SC_MTI_BIND_NAME) macro, you must disable automatic name binding. Otherwise, an **sccom** error results. You can disable automatic name binding when you compile your C++ source code using the **-nonamebind** argument to the **sccom** (CR-248) command. Disabling the name binding can also be useful as a workaround to symbol collisions at the time of linking the compiled source (see "sccom -link errors" (UM-197).

Fixed point types

Contrary to OSCI, ModelSim compiles the SystemC kernel with support for fixed point types. If you want to compile your own SystemC code to enable that support, you'll need to define the compile time macro SC_INCLUDE_FX. You can do this in one of two ways:

- enter the g++/aCC argument -DSC_INCLUDE_FX on the **sccom** (CR-248) command line, such as:

  ```
  sccom -DSC_INCLUDE_FX top.cpp
  ```

- add a define statement to the C++ source code before the inclusion of the *systemc.h*, as shown below:

  ```
  #define SC_INCLUDE_FX
  #include "systemc.h"
  ```

OSCI 2.1 features supported

ModelSim is fully compliant with the OSCI version 2.0.1. In addition, the following 2.1 features are supported:

- end_of_construction()
- start_of_simulation()
- end_of_simulation()

For more information regarding these functions, see "Initialization and cleanup of SystemC state-based code" (UM-200).

Troubleshooting SystemC

In the process of modifying your SystemC design to run on ModelSim, you may encounter several common errors. This section highlights some actions you can take to correct such errors.

Errors during compilation

ModelSim's name association feature (**gensrc**) runs primarily at **sccom -link** time. A key element of the feature is that C++ source files are generated in the work library. The C++ source files include user header files that define user **sc_module** classes. These files are compiled by the C++ compiler during sccom -link operation. Occasionally, this compilation phase generates errors, such as the following:

```
/_sc/gensrc_obj/gensrc_0.cpp:4:\
gates.h:8: redefinition of `struct my_gate'
gates.h:8: previous definition of `struct my_gate'
** Error: (sccom-6142) Compilation failed.
```

All known gensrc compilation errors occur due to lack of include guards in user header files. An include guard is a construct used to guard against redundant text inclusion in a *.cpp* file during compilation.

A typical include guard for a header file named *filename.h* would look like this:

```
#ifndef INCLUDED_FILENAME_H
#define INCLUDED_FILENAME_H
<main body of filename.h goes here>
typedef unsigned long u_long; // Example contents
#endif
```

When this construct is used, it makes it OK for users to "#include" in their header file more than once during the same compilation, e.g.:

```
#include "filename.h"
#include "filename.h"
```

However, this typically doesn't happen. More often something like this happens:

```
#include "foo.h"
#include "bar.h"
```

in which both *foo.h* and *bar.h* have a #include *filename.h* directive.

Errors during loading

When simulating your SystemC design, you might get a "failed to load sc lib" message because of an undefined symbol, looking something like this:

```
# Loading /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so

# ** Error: (vsim-3197) Load of "/home/cmg/newport2_systemc/chip/vhdl/work/
systemc.so" failed: ld.so.1:

/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file

/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:
```

```
referenced symbol not found.

# ** Error: (vsim-3676) Could not load shared library /home/cmg/
newport2_systemc/chip/vhdl/work/systemc.so for SystemC module 'host_xtor'.
```

Source of undefined symbol message

The causes for such an error could be:

- missing definition
- bad link order specified in sccom -link
- multiply defined symbols

Missing definition

If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

This should appear in any header files include in your C++ sources compiled by **sccom**. It tells the compiler to expect a regular C function; otherwise the compiler decorates the name for C++ and then the symbol can't be found.

Also, be sure that you actually linked with an object file that fully defines the symbol. You can use the "nm" utility on Unix platforms to test your SystemC object files and any libraries you link with your SystemC sources. For example, assume you ran the following commands:

```
sccom test.cpp
sccom -link libSupport.a
```

If there is an unresolved symbol and it is not defined in your sources, it should be correctly defined in any linked libraries:

```
nm libSupport.a | grep "mySymbol"
```

Misplaced "-link" option

The order in which you place the **-link** option within the **sccom -link** command is critical. There is a big difference between the following two commands:

```
sccom -link liblocal.a
```

and

```
sccom libmystuff.a -link
```

The first command ensures that your SystemC object files are seen by the linker before the library "liblocal.a" and the second command ensures that "liblocal.a" is seen first. Some linkers can look for undefined symbols in libraries that follow the undefined reference while others can look both ways. For more information on command syntax and dependencies, see **sccom** (CR-248).

Multiple symbol definition errors

The most common type of error found during **sccom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one *.o* file. The error message looks something like this:

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':

work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'

work/sc/test_ringbuf.o(.text+0x4): first defined here
```

A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e. not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

Text in *.h* files is included into *.cpp* files by the C++ preprocessor. By the time the compiler sees the text, it's just as if you had typed the entire text from the *.h* file into the *.cpp* file. So a *.h* file included into two *.cpp* files results in lots of duplicate text being processed by the C++ compiler when it starts up. Include guards are a common technique to avoid duplicate text problems. See "Errors during compilation" (UM-206) for more information on include guards.

If an *.h* file has an out-of-line function defined, and that *.h* file is included into two *.c* files, then the out-of-line function symbol will be defined in the two corresponding. *o* files. This leads to a multiple symbol definition error during sccom -link.

To solve this problem, add the "inline" keyword to give the function "internal linkage". This makes the function internal to the *.o* file, and prevents the function's symbol from colliding with a symbol in another *.o* file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a *.cpp* file.

Multiple symbol definitions caused by ModelSim's name association

Another cause of errors is due to ModelSim's name association feature. It is important to realize that the name association feature automatically generates *.cpp* files in the work library. These files "include" your header files. Thus, while it might appear as though you have included your header file in only one *.cpp* file, from the linker's point of view, it is included in multiple *.cpp* files.

If name association is causing multiple symbol definition errors, you should eliminate the errors by using the techniques mentioned above (i.e. adding the "inline" or "static" keywords, as appropriate). Another solution is to use the **sccom -nonamebind** argument to turn off name association. However, this is not recommended, since design debug will be heavily compromised without name association.

For related information, see "Name association (binding)" (UM-204).

8 - Mixed-language simulations

Chapter contents

ModelSim single-kernel simulation allows you to simulate designs that are written in VHDL, Verilog, and/or SystemC. The boundaries between languages are enforced at the level of a design unit. This means that although a design unit itself must be entirely of one language type, it may instantiate design units from another language. Any instance in the design hierarchy may be a design unit from another language without restriction.

Usage flow for mixed-language simulations

The usage flow for mixed-language designs is as follows:

1 Analyze HDL source code using **vcom** or **vlog** and C++ source code using **sccom**. Analyze all modules in the design following order-of-analysis rules.

- For SystemC designs with HDL instances:
  You must create a SystemC foreign module declaration for all Verilog and VHDL instances (UM-234).
- For Verilog/VHDL designs with SystemC instances:
  You must export any SystemC instances that will be directly instantiated by Verilog/VHDL using the SC_EXPORT_MODULE macro. Exported SystemC modules can be instantianted just as you would instantiate any Verilog/VHDL module or design unit.

2 Simulate the design by invoking **vsim**.

- For designs containing SystemC code:
  You must first prepare the design by running **sccom -link** (UM-234) prior to running simulation.

3 Issue **run** commands from the ModelSim GUI.

4 Debug your design using ModelSim GUI features. Note that objects inside SystemC modules are unavailable for viewing in the Dataflow window.

Separate compilers, common design libraries

VHDL source code is compiled by **vcom** (CR-303) and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in the working library. Likewise, Verilog source code is compiled by **vlog** (CR-345) and the resulting design units (modules and UDPs) are stored in the working library.

SystemC/C++ source code is compiled with the **sccom** command (CR-248). The resulting object code is compiled into the working library.

Design libraries can store any combination of design units from any of the supported languages, provided the design unit names do not overlap (VHDL design unit names are changed to lower case). See "Design libraries" (UM-53) for more information about library management.

Access limitations in mixed-language designs

The Verilog language allows hierarchical access to objects throughout the design. This is not the case with VHDL or SystemC. You *cannot* directly read or change a VHDL or SystemC object (signal, variable, generic, etc.) with a hierarchical reference within a mixed-language design. Furthermore, you cannot directly access a Verilog object up or down the hierarchy if there is an interceding VHDL or SystemC block.

You have two options for accessing VHDL objects or Verilog objects "obstructed" by an interceding block: 1) propagate the value through the ports of all design units in the hierarchy; 2) use the Signal Spy procedures or system tasks (see *Chapter 16 - Signal Spy* for details).

To access obstructed SystemC objects, propagate the value through the ports of all design units in the hierarchy.

Simulator resolution limit

If the root of the mixed design is VHDL, then VHDL simulator resolution rules are used (see "Simulator resolution limit" (UM-211) for VHDL details). If the root of the mixed design is Verilog, Verilog rules are used (see "Simulator resolution limit" (UM-117) for Verilog details), but no Verilog modules that are instantiated under VHDL models are considered when looking for the minimum simulation precision.

Note that the OSCI SystemC simulator allows users to dynamically change the simulator's resolution by using the *sc_set_time_resolution()* API. This is not permitted in ModelSim. You must be aware of your required SystemC resolution in advance, and then make sure ModelSim is running with a resolution at least as fine as the SystemC resolution.

If a design contains SystemC modules and Verilog modules, the default resolution (specified in *modelsim.ini* or by using the **vsim -t** command line option) must be at least as fine as the finest Verilog ‘timescale setting in the design. Otherwise an elaboration error occurs.

Runtime modeling semantics

The ModelSim simulator is compliant with all pertinent Language Reference Manuals. To achieve this compliance, the sequence of operations in one simulation iteration (i.e. delta cycle) is as follows:

- SystemC processes are run
- Signal updates are made
- HDL processes are run

Mapping data types

Cross-language (HDL) instantiation does not require any extra effort on your part. As ModelSim loads a design it detects cross-language instantiations – made possible because a design unit's language type can be determined as it is loaded from a library – and the necessary adaptations and data type conversions are performed automatically. SystemC and HDL cross-language instantiation requires minor modification of SystemC source code (addition of SC_EXPORT_MODULE, sc_foreign_module, etc.).

A VHDL instantiation of Verilog may associate VHDL signals and values with Verilog ports and parameters. Likewise, a Verilog instantiation of VHDL may associate Verilog nets and values with VHDL ports and generics. The same holds true for SystemC and VHDL/Verilog ports. However, SystemC does not support cross-language generic/ parameter propagation at this time.

ModelSim automatically maps between the language data types as shown in the sections below.

Verilog to VHDL mappings

VHDL generics

VHDL type	Verilog type
integer	integer or real
real	integer or real
time	integer or real
physical	integer or real
enumeration	integer or real
string	string literal

When a scalar type receives a real value, the real is converted to an integer by truncating the decimal portion.

Type time is treated specially: the Verilog number is converted to a time value according to the **‘timescale** directive of the module.

Physical and enumeration types receive a value that corresponds to the position number indicated by the Verilog number. In VHDL this is equivalent to T'VAL(P), where T is the type, VAL is the predefined function attribute that returns a value given a position number, and P is the position number.

Verilog parameters

VHDL type	Verilog type
integer	integer
real	real

VHDL type	Verilog type
string	string

The type of a Verilog parameter is determined by its initial value.

Verilog ports

The allowed VHDL types for ports connected to Verilog nets and for signals connected to Verilog ports are:

Allowed VHDL types
bit
bit_vector
std_logic
std_logic_vector
vl_logic
vl_logic_vector

The vl_logic type is an enumeration that defines the full state set for Verilog nets, including ambiguous strengths. The bit and std_logic types are convenient for most applications, but the vl_logic type is provided in case you need access to the full Verilog state set. For example, you may wish to convert between vl_logic and your own user-defined type. The vl_logic type is defined in the vl_types package in the pre-compiled **verilog** library. This library is provided in the installation directory along with the other pre-compiled libraries (**std** and **ieee**). The source code for the vl_types package can be found in the files installed with ModelSim. (See *<install_dir>\modeltech\vhdl_src\verilog\vltypes.vhd*.)

Verilog states

Verilog states are mapped to std_logic and bit as follows:

Verilog	std_logic	bit
HiZ	'Z'	'0'
Sm0	'L'	'0'
Sm1	'H'	'1'
SmX	'W'	'0'
Me0	'L'	'0'
Me1	'H'	'1'
MeX	'W'	'0'
We0	'L'	'0'

Verilog	std_logic	bit
We1	'H'	'1'
WeX	'W'	'0'
La0	'L'	'0'
La1	'H'	'1'
LaX	'W'	'0'
Pu0	'L'	'0'
Pu1	'H'	'1'
PuX	'W'	'0'
St0	'0'	'0'
St1	'1'	'1'
StX	'X'	'0'
Su0	'0'	'0'
Su1	'1'	'1'
SuX	'X'	'0'

For Verilog states with ambiguous strength:

- bit receives '0'
- std_logic receives 'X' if either the 0 or 1 strength component is greater than or equal to strong strength
- std_logic receives 'W' if both the 0 and 1 strength components are less than strong strength

VHDL to Verilog mappings

VHDL type bit is mapped to Verilog states as follows:

bit	Verilog
'0'	St0
'1'	St1

VHDL type std_logic is mapped to Verilog states as follows:

std_logic	Verilog
'U'	StX
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'–'	StX

Verilog and SystemC signal interaction and mappings

SystemC has a more complex signal-level interconnect scheme than Verilog. Design units are interconnected via hierarchical and primitive channels. An sc_signal<> is one type of primitive channel. The following section discusses how various SystemC channel types map to Verilog wires when connected to each other across the language boundary.

Channel and Port type mapping

The following port type mapping table lists all channels. Three types of primitive channels and 1 hierarchical channel are supported on the language boundary (SystemC modules connected to Verilog modules).

Channels	**Ports**	**Verilog mapping**
sc_signal<type>	sc_in<type> sc_out<type> sc_inout<type>	Depends on type. See table entitled "Data type mapping" (UM-218).
sc_signal_rv<width>	sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width>	wire [width-1:0]
sc_signal_resolved	sc_in_resolved sc_out_resolved sc_inout_resolved	wire [width-1:0]
sc_clock	sc_in_clk sc_out_clk sc_inout_clk	wire
sc_mutex	N/A	Not supported on language boundary
sc_fifo	sc_fifo_in sc_fifo_out sc_fifo_inout	Not supported on language boundary
sc_semaphore	N/A	Not supported on language boundary
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary

Data type mapping

SystemC's sc_signal<> types are mapped to Verilog types as follows:

SystemC	Verilog
bool, sc_bit	wire
sc_logic	wire
sc_bv<width>	wire [width-1:0]
sc_lv<width>	wire [width-1:0]
sc_int<width>, sc_uint<width>	wire [width-1:0]
char, unsigned char	wire [7:0]
int, unsigned int	wire [31:0]
long, unsigned long	wire [31:0]
sc_bigint<width>, sc_biguint<width>	Not supported on language boundary
sc_fixed<W,I,Q,O,N>, sc_ufixed<W,I,Q,O,N>	Not supported on language boundary
short, unsigned short	Not supported on language boundary
long long, unsigned long long	Not supported on language boundary
float	Not supported on language boundary
double	Not supported on language boundary
enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
struct	Not supported onlanguage boundary
union	Not supported on language boundary
bit_fields	Not supported on language boundary

Port direction

Verilog port directions are mapped to SystemC as follows:

Verilog	SystemC
input	sc_in<type>, sc_in_resolved, sc_in_rv<width>
output	sc_out<type>, sc_out_resolved, sc_out_rv<width>
inout	sc_inout<type>, sc_inout_resolved, sc_inout_rv<width>

Verilog to SystemC state mappings

Verilog states are mapped to sc_logic, sc_bit, and bool as follows:

Verilog	sc_logic	sc_bit	bool
HiZ	'Z'	'0'	false
Sm0	'0'	'0'	false
Sm1	'1'	'1'	true
SmX	'X'	'0'	false
Me0	'0'	'0'	false
Me1	'1'	'1'	true
MeX	'X'	'0'	false
We0	'0'	'0'	false
We1	'1'	'1'	true
WeX	'X'	'0'	false
La0	'0'	'0'	false
La1	'1'	'1'	true
LaX	'X'	'0'	false
Pu0	'0'	'0'	false
Pu1	'1'	'1'	true
PuX	'X'	'0'	false
St0	'0'	'0'	false
St1	'1'	'1'	true
StX	'X'	'0'	false
Su0	'0'	'0'	false
Su1	'1'	'1'	true

Verilog	sc_logic	sc_bit	bool
SuX	'X'	'0'	false

For Verilog states with ambiguous strength:

- sc_bit receives '1' if the value component is 1, else it receives ’0’
- bool receives true if the value component is 1, else it receives false
- sc_logic receives 'X' if the value component is X, H, or L
- sc_logic receives '0' if the value component is 0
- sc_logic receives ’1’ if the value component is 1

SystemC to Verilog state mappings

SystemC type bool is mapped to Verilog states as follows:

bool	Verilog
false	St0
true	St1

SystemC type sc_bit is mapped to Verilog states as follows:

sc_bit	Verilog
'0'	St0
'1'	St1

SystemC type sc_logic is mapped to Verilog states as follows:

sc_logic	Verilog
'0'	St0
'1'	St1
'Z'	HiZ
'X'	StX

VHDL and SystemC signal interaction and mappings

SystemC has a more complex signal-level interconnect scheme than VHDL. Design units are interconnected via hierarchical and primitive channels. An sc_signal<> is one type of primitive channel. The following section discusses how various SystemC channel types map to VHDL types when connected to each other across the language boundary.

Port type mapping

The following port type mapping table lists all channels. Three types of primitive channels and 1 hierarchical channel are supported on the language boundary (SystemC modules connected to VHDL modules)..

Channels	Ports	VHDL mapping
sc_signal<type>	sc_in<type> sc_out<type> sc_inout<type>	Depends on type. See table entitled "Data type mapping" (UM-221) below.
sc_signal_rv<width>	sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width>	std_logic_vector(width-1 downto 0)
sc_signal_resolved	sc_in_resolved sc_out_resolved sc_inout_resolved	std_logic
sc_clock	sc_in_clk sc_out_clk sc_inout_clk	bit/std_logic/boolean
sc_mutex	N/A	Not supported on language boundary
sc_fifo	sc_fifo_in sc_fifo_out sc_fifo_inout	Not supported on language boundary
sc_semaphore	N/A	Not supported on language boundary
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary

Data type mapping

SystemC's sc_signal types are mapped to VHDL types as follows

SystemC	VHDL
bool, sc_bit	bit/std_logic/boolean
sc_logic	std_logic
sc_bv<width>	bit_vector(width-1 downto 0)

SystemC	VHDL
sc_lv<width>	std_logic_vector(width-1 downto 0)
sc_int<W>, sc_uint<width>	bit_vector(width-1 downto 0)
char, unsigned char	bit_vector(7 downto 0)
int, unsigned int	bit_vector(31 downto 0)
long, unsigned long	bit_vector(31 downto 0)
sc_bigint<width>, sc_biguint<width>	Not supported on language boundary
sc_fixed<W,I,Q,O,N>, sc_ufixed<W,I,Q,O,N>	Not supported on language boundary
short, unsigned short	Not supported on language boundary
long long, unsigned long	Not supported on language boundary
float	Not supported on language boundary
double	Not supported on language boundary
enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
structure	Not supported onlanguage boundary
union	Not supported on language boundary
bit_fields	Not supported on language boundary

Port direction mapping

VHDL port directions are mapped to SystemC as follows:

VHDL	SystemC
in	sc_in<type>, sc_in_resolved, sc_in_rv<w>
out	sc_out<type>, sc_out_resolved, sc_out_rv<w>
inout	sc_inout<type>, sc_inout_resolved, sc_inout_rv<w>
buffer	sc_out<type>, sc_out_resolved, sc_out_rv<w>

VHDL to SystemC state mapping

VHDL states are mapped to sc_logic, sc_bit, and bool as follows:

std_logic	sc_logic	sc_bit	bool
'U'	'X'	'0'	false
'X'	'X'	'0'	false
'0'	'0'	'0'	false
'1'	'1'	'1'	true
'Z'	'Z'	'0'	false
'W'	'X'	'0'	false
'L'	'0'	'0'	false
'H'	'1'	'1'	true
'-'	'X'	'0'	false

SystemC to VHDL state mapping

SystemC type bool is mapped to VHDL boolean as follows:

bool	VHDL
false	false
true	true

SystemC type sc_bit is mapped to VHDL bit as follows:

sc_bit	VHDL
'0'	'0'
'1'	'1'

SystemC type sc_logic is mapped to VHDL std_logic states as follows:

sc_logic	std_logic
'0'	'0'
'1'	'1'
'Z'	'Z'
'X'	'X'

VHDL: instantiating Verilog

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. You can reference a Verilog module in the entity aspect of a component configuration – all you need to do is specify a module name instead of an entity name. You can also specify an optional secondary name for an optimized sub-module. Further, you can reference a Verilog configuration in the configuration aspect of a VHDL component configuration - just specify a Verilog configuration name instead of a VHDL configuration name.

Verilog instantiation criteria

A Verilog design unit may be instantiated within VHDL if it meets the following criteria:

- The design unit is a module or configuration. UDPs are not allowed.
- The ports are named ports (see "Modules with unnamed ports" (UM-228) below).
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in VHDL).

Component declaration

A Verilog module that is compiled into a library can be referenced from a VHDL design as though the module is a VHDL entity. Likewise, a Verilog configuration can be referenced as though it were a VHDL configuration.

The interface to the module can be extracted from the library in the form of a component declaration by running **vgencomp** (CR-318). Given a library and module name, **vgencomp** (CR-318) writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose:

- bit and bit_vector
- vl_logic and vl_logic_vector

VHDL and Verilog identifiers

The VHDL identifiers for the component name, port names, and generic names are the same as the Verilog identifiers for the module name, port names, and parameter names. If a Verilog identifier is not a valid VHDL 1076-1987 identifier, it is converted to a VHDL 1076-1993 extended identifier (in which case you must compile the VHDL with the -93 or higher switch). Any uppercase letters in Verilog identifiers are converted to lowercase in the VHDL identifier, except in the following cases:

- The Verilog module was compiled with the -93 switch. This means **vgencomp** (CR-318) should use VHDL 1076-1993 extended identifiers in the component declaration to preserve case in the Verilog identifiers that contain uppercase letters.

- The Verilog module, port, or parameter names are not unique unless case is preserved. In this event, **vgencomp** (CR-318) behaves as if the module was compiled with the -93 switch for those names only.

If you use Verilog identifiers where the names are unique by case only, use the **-93** argument when compiling mixed-language designs.

Examples

Verilog identifier	VHDL identifier
topmod	topmod
TOPMOD	topmod
TopMod	topmod
top_mod	top_mod
_topmod	_topmod\
\topmod	topmod
\\topmod\	\topmod\

If the Verilog module is compiled with -93:

Verilog identifier	VHDL identifier
topmod	topmod
TOPMOD	\TOPMOD\
TopMod	\TopMod\
top_mod	top_mod
_topmod	_topmod\
\topmod	topmod
\\topmod\	\topmod\

vgencomp component declaration

vgencomp (CR-318) generates a component declaration according to these rules:

Generic clause

A generic clause is generated if the module has parameters. A corresponding generic is defined for each parameter that has an initial value that does not depend on any other parameters.

The generic type is determined by the parameter's initial value as follows:

Parameter value	**Generic type**
integer	integer
real	real
string literal	string

The default value of the generic is the same as the parameter's initial value.

Examples

Verilog parameter	**VHDL generic**
parameter p1 = 1 - 3;	p1 : integer := -2;
parameter p2 = 3.0;	p2 : real := 3.000000;
parameter p3 = "Hello";	p3 : string := "Hello";

Port clause

A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named Verilog port.

You can set the VHDL port type to bit, std_logic, or vl_logic. If the Verilog port has a range, then the VHDL port type is bit_vector, std_logic_vector, or vl_logic_vector. If the range does not depend on parameters, then the vector type will be constrained accordingly, otherwise it will be unconstrained.

Examples

Verilog port	**VHDL port**
input p1;	p1 : in std_logic;
output [7:0] p2;	p2 : out std_logic_vector(7 downto 0);
output [4:7] p3;	p3 : out std_logic_vector(4 to 7);
inout [width-1:0] p4;	p4 : inout std_logic_vector;

Configuration declarations are allowed to reference Verilog modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a Verilog instance to configure the instantiations within the Verilog module.

Modules with unnamed ports

Verilog allows modules to have unnamed ports, whereas VHDL requires that all ports have names. If any of the Verilog ports are unnamed, then all are considered to be unnamed, and it is not possible to create a matching VHDL component. In such cases, the module may not be instantiated from VHDL.

Unnamed ports occur when the module port list contains bit-selects, part-selects, or concatenations, as in the following example:

```
module m(a[3:0], b[1], b[0], {c,d});
  input [3:0] a;
  input [1:0] b;
  input c, d;
endmodule
```

Note that *a[3:0]* is considered to be unnamed even though it is a full part-select. A common mistake is to include the vector bounds in the port list, which has the undesired side effect of making the ports unnamed (which prevents the user from connecting by name even in an all Verilog design).

Most modules having unnamed ports can be easily rewritten to explicitly name the ports, thus allowing the module to be instantiated from VHDL. Consider the following example:

```
module m(y[1], y[0], a[1], a[0]);
  output [1:0] y;
  input [1:0] a;
endmodule
```

Here is the same module rewritten with explicit port names added:

```
module m(.y1(y[1]), .y0(y[0]), .a1(a[1]), .a0(a[0]));
  output [1:0] y;
  input [1:0] a;
endmodule
```

"Empty" ports

Verilog modules may have "empty" ports, which are also unnamed, but they are treated differently from other unnamed ports. If the only unnamed ports are "empty", then the other ports may still be connected to by name, as in the following example:

```
module m(a, , b);
  input a, b;
endmodule
```

Although this module has an empty port between ports "a" and "b", the named ports in the module can still be connected to from VHDL.

Verilog: instantiating VHDL

You can reference a VHDL entity or configuration from Verilog as though the design unit is a module or a configuration of the same name.

VHDL instantiation criteria

A VHDL design unit may be instantiated within Verilog if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type bit, bit_vector, std_ulogic, std_ulogic_vector, vl_ulogic, vl_ulogic_vector, or their subtypes. The port clause may have any mix of these types.
- The generics are of type integer, real, time, physical, enumeration, or string. String is the only composite type allowed.

Entity/architecture names and escaped identifiers

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise. Since instantiation bindings are not determined at compile time in Verilog, you must instruct the simulator to search your libraries when loading the design. See "Library usage" (UM-111) for more information.

Alternatively, you can employ the escaped identifier to provide an extended form of instantiation:

```
\mylib.entity(arch) u1 (a, b, c);
\mylib.entity u1 (a, b, c);
\entity(arch) u1 (a, b, c);
```

If the escaped identifier takes the form of one of the above and is not the name of a design unit in the work library, then the instantiation is broken down as follows:

- library = mylib
- design unit = entity
- architecture = arch

Named port associations

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

Named port associations are *not* case sensitive unless a VHDL port name is an extended identifier (1076-1993). If the VHDL port name is an extended identifier, the association is case sensitive and the VHDL identifier's leading and trailing backslashes are removed before comparison.

Generic associations

Generic associations are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. Parameter assignment to generics is not case sensitive.

The **defparam** statement is not allowed for setting generic values.

SDF annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See "SDF for mixed VHDL and Verilog designs" (UM-554) for more information.

SystemC: instantiating Verilog

To instantiate Verilog modules into a SystemC design, you must first create a "SystemC foreign module declaration" (UM-231) for each Verilog module. Once you have created the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

Verilog instantiation criteria

A Verilog design unit may be instantiated within SystemC if it meets the following criteria:

- The design unit is a module (UDPs and Verilog primitives are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- The Verilog module name must be a valid C++ identifier.
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in SystemC).

The current release does not allow users to perform parameter overrides when instantiating Verilog from SystemC.

A Verilog module that is compiled into a library can be instantiated in a SystemC design as though the module were a SystemC module by passing the Verilog module name to the foreign module constructor. For an illustration of this, see "Example #1" (UM-232).

SystemC and Verilog identifiers

The SystemC identifiers for the module name and port names are the same as the Verilog identifiers for the module name and port names. Verilog identifiers must be valid C++ identifiers. SystemC and Verilog are both case sensitive.

SystemC foreign module declaration

In cases where you want to run a mixed simulation with SystemC and Verilog, you must generate and declare a foreign module that stands in for each Verilog module instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)
- modifying your SystemC source code manually

Using scgenmod

After you have analyzed the design, you can generate a foreign module declaration with an **scgenmod** command (CR-251) similar to the following:

```
scgenmod mod1
```

where *mod1* is a Verilog module. A foreign module declaration for the specified module is written to stdout.

Guidelines for manual creation

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to VHDL or Verilog ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For VHDL, the HDL name can be in the format **[<lib>.]<primary>[(<secondary>)]** or **[<lib>.]<conf>**. For Verilog, the HDL name is simply the Verilog module name corresponding to the foreign module, or **[<lib>].<module>**.

Example #1

A sample Verilog module to be instantiated in a SystemC design is:

```
module vcounter (clock, topcount, count);

    input clock;
    input topcount;
    output count;

    reg count;
    ...

endmodule
```

The SystemC foreign module declaration for the above Verilog module is:

```
class counter : public sc_foreign_module {
    public:

    sc_in<bool> clock;
    sc_in<sc_logic> topcount;
    sc_out<sc_logic> count;

counter(sc_module_name nm)
    : sc_foreign_module(nm, "lib.vcounter"),
    clock("clock"),
    topcount("topcount"),
    count("count")
    {}
};
```

The Verilog module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the SystemC instance name.

Example #2

Another variation of the SystemC foreign module declaration for the same Verilog module might be:

```
class counter : public sc_foreign_module {
    public:
        ...
        ...
        ...

counter(sc_module_name nm, char* hdl_name)
    : sc_foreign_module(nm, hdl_name),
    clock("clock"),
        ...
        ...
        ...

{}
};
```

The instantiation of this module would be:

```
counter dut("dut", "lib.counter");
```

Verilog: instantiating SystemC

You can reference a SystemC module from Verilog as though the design unit is a module of the same name.

SystemC instantiation criteria

A SystemC module can be instantiated in Verilog if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- SystemC modules are exported using the SC_MODULE_EXPORT macro. See "Exporting SystemC modules" (UM-234).
- The module ports are as listed in the table shown in "Channel and Port type mapping" (UM-217).
- Port data type mapping must match exactly. See the table in "Data type mapping" (UM-218).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module declaration. Named port associations are case sensitive.

Since there is no concept of "parameters" in SystemC, it is illegal to place parameter overrides on instantiations of sc_modules.

Exporting SystemC modules

To be able to instantiate a SystemC module from Verilog (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"

SC_MODULE_EXPORT(transceiver);
```

sccom -link

The **sccom -link** command collects the object files created in the work library, and uses them to build a shared library (.so) in the current work library. If you have changed your SystemC source code and recompiled it using **sccom**, then you must run **sccom -link** before invoking **vsim**. Otherwise your changes to the code are not recognized by the simulator.

SystemC: instantiating VHDL

To instantiate VHDL design units into a SystemC design, you must first generate a SystemC foreign module declaration (UM-231) for each VHDL design unit you want to instantiate. Once you have generated the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

VHDL instantiation criteria

A VHDL design unit may be instantiated from SystemC if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type bit, bit_vector, std_ulogic, std_ulogic_vector, or their subtypes. The port clause may have any mix of these types.

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

SystemC foreign module declaration

In cases where you want to run a mixed simulation with SystemC and VHDL, you must create and declare a foreign module that stands in for each VHDL design unit instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)
- modifying your SystemC source code manually

Using scgenmod

After you have analyzed the design, you can generate a foreign module declaration with an scgenmod command similar to the following:

```
scgenmod mod1
```

Where *mod1* is either a Verilog module or a VHDL entity. A foreign module declaration for the specified module is written to stdout.

Guidelines for manual creation

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to VHDL or Verilog ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For VHDL, the HDL name can be in the format **[<lib>.]<primary>[(<secondary>)]** or **[<lib>.]<conf>**. For Verilog, the HDL name is simply the Verilog module name corresponding to the foreign module, or [<lib>].<module>.

Example

A sample VHDL design unit to be instantiated in a SystemC design is:

```
entity counter is
    port (count : buffer bit_vector(8 downto 1);
        clk   : in bit;
        reset : in bit);
end;

architecture only of counter is
    ...
    ...

end only;
```

The SystemC foreign module declaration for the above VHDL module is:

```
class counter : public sc_foreign_module {
public:

sc_in<bool> clk;
sc_in<bool> reset;
sc_out<sc_logic> count;

counter(sc_module_name nm)
    : sc_foreign_module(nm, "work.counter(only)"),
    clk("clk"),
    reset("reset"),
    count("count")
    {}
};
```

The VHDL module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the SystemC instance name.

VHDL: instantiating SystemC

To instantiate SystemC in a VHDL design, you must create a component declaration for the SystemC module. Once you have generated the component declaration, you can instantiate the SystemC component just like any other VHDL component.

SystemC instantiation criteria

A SystemC design unit may be instantiated within VHDL if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- The SystemC design unit is exported using the SC_MODULE_EXPORT macro.
- The module ports are as listed in the table in "Data type mapping" (UM-221)
- Port data type mapping must match exactly. See the table in "Port type mapping" (UM-221).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module. Named port associations are case sensitive.

ModelSim does not support generic overrides across SystemC language boundaries.

Component declaration

A SystemC design unit can be referenced from a VHDL design as though it is a VHDL entity. The interface to the design unit can be extracted from the library in the form of a component declaration by running **vgencomp**. Given a library and a SystemC module name, **vgencomp** writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose:

- bit and bit_vector

VHDL and SystemC identifiers

The VHDL identifiers for the component name and port names are the same as the SystemC identifiers for the module name and port names. If a SystemC identifier is not a valid VHDL 1076-1987 identifier, it is converted to a VHDL 1076-1993 extended identifier (in which case you must compile the VHDL with the -93 or later switch).

Examples

SystemC identifier	VHDL identifier
topmod	topmod
TOPMOD	topmod
TopMod	topmod

SystemC identifier	VHDL identifier
top_mod	top_mod
_topmod	_topmod\

vgencomp component declaration

vgencomp (CR-318) generates a component declaration according to these rules:

Port clause

A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named SystemC port.

You can set the VHDL port type to bit or std_logic. If the SystemC port has a range, then the VHDL port type is bit_vector or std_logic_vector.

Examples

SystemC port	VHDL port
sc_in<sc_logic>p1;	p1 : in std_logic;
sc_out<sc_lv<8>>p2;	p2 : out std_logic_vector(7 downto 0);
sc_inout<sc_lv<8>>p3;	p3 : inout std_logic_vector(7 downto 0)

Configuration declarations are allowed to reference SystemC modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a SystemC instance to configure the instantiations within the SystemC module.

Exporting SystemC modules

To be able to instantiate a SystemC module within VHDL (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"

SC_MODULE_EXPORT(transceiver);
```

sccom -link

The **sccom -link** command collects the object files created in the work library, and uses them to build a shared library (.so) in the current work library. If you have changed your SystemC source code and recompiled it using **sccom**, then you must run **sccom -link** before invoking **vsim**. Otherwise your changes to the code are not recognized by the simulator.

9 - WLF files (datasets) and virtuals

Chapter contents

A ModelSim simulation can be saved to a wave log format (WLF) file for future viewing or comparison to a current simulation. We use the term "dataset" to refer to a WLF file that has been reopened for viewing.

You can open more than one WLF file for simultaneous viewing. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different datasets.

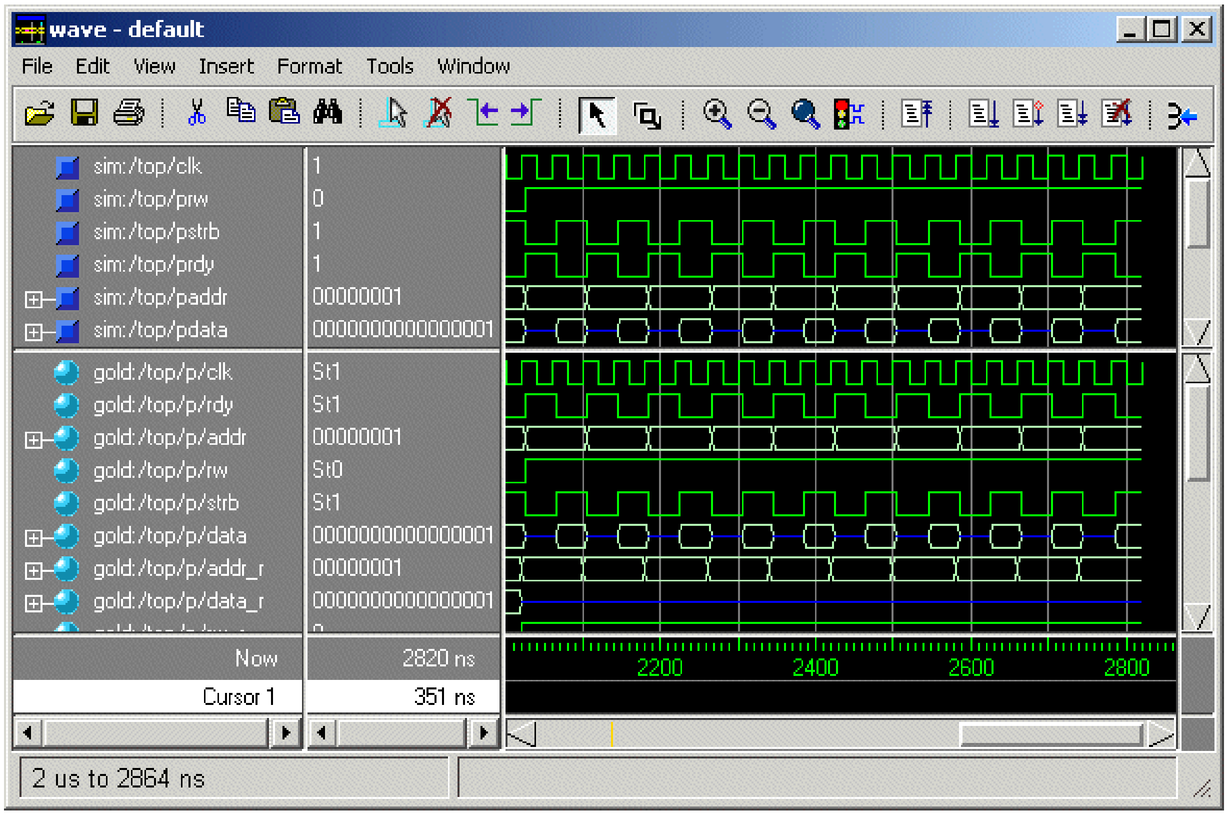
WLF files (datasets)

Wave Log Format (WLF) files are recordings of simulation runs. The files contain data from logged items (e.g., signals and variables) and the design hierarchy in which the logged items are found. You can record the entire design or choose specific items.

The WLF file provides you with precise in-simulation and post-simulation debugging capability. Any number of WLF files can be reloaded for viewing or comparing to the active simulation.

A dataset is a previously recorded simulation that has been loaded into ModelSim. Each dataset has a logical name to let you indicate the dataset to which any command applies. This logical name is displayed as a prefix. The current, active simulation is prefixed by "sim:", while any other datasets are prefixed by the name of the WLF file by default.

Two datasets are displayed in the Wave window below. The current simulation is shown in the top pane and is indicated by the "sim" prefix. A dataset from a previous simulation is shown in the bottom pane and is indicated by the "gold" prefix.



sim:/top/clk	1
sim:/top/prw	0
sim:/top/pstrb	1
sim:/top/prdy	1
sim:/top/paddr	00000001
sim:/top/pdata	0000000000000001
gold:/top/p/clk	St1
gold:/top/p/rdy	St1
gold:/top/p/addr	00000001
gold:/top/p/rw	St0
gold:/top/p/strb	St1
gold:/top/p/data	0000000000000001
gold:/top/p/addr_r	00000001
gold:/top/p/data_r	0000000000000001
Now	2820 ns
Cursor 1	351 ns

2 us to 2864 ns

The simulator resolution (see "Simulator resolution limit" (UM-67)) must be the same for all datasets you’re comparing, including the current simulation. If you have a WLF file that is in a different resolution, you can use the wlfman command (CR-370) to change it.

Saving a simulation to a WLF file

If you add items to the Dataflow, List, or Wave windows, or log items with the **log** command, the results of each simulation run are automatically saved to a WLF file called *vsim.wlf* in the current directory. If you run a new simulation in the same directory, the *vsim.wlf* file is overwritten with the new results.

If you want to save the WLF file and not have it overwritten, select **File > Save> sim dataset** (Main window) or **File > Save Dataset > sim** (Wave window). Or, you can use the **-wlf <filename>** argument to the **vsim** command (CR-357) or the **dataset save** command (CR-148).

Important: If you do not use **dataset save** or **dataset snapshot**, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you don't end the simulation in this manner, the WLF file will not close properly, and ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions. If you end up with a "damaged" WLF file, you can try to "repair" it using the **wlfrecover** command (CR-387).

Hiding library cell signals when saving a waveform file

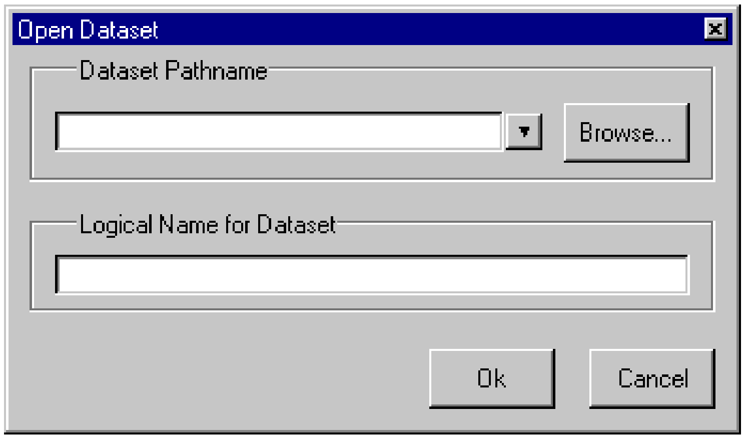
Gate-level simulations may result in large waveform files because the internal signals of your library cells are saved. The following method will prevent these signals from being saved in a Verilog design.

If your cells are enclosed in Verilog `celldefine and `endcelldefine preprocessor directives, you can specify -fast on the vlog command line when compiling the cell library. This will basically hide the internal signals so they will not be saved. A further benefit of this methodology is that the cells compiled with -fast will consume less memory.

See "Compiling with -fast" (UM-127) for further details on using -fast.

Opening datasets

To open a dataset, select either **File > Open > Dataset** (Main window) or use the **dataset open** command (CR-146).



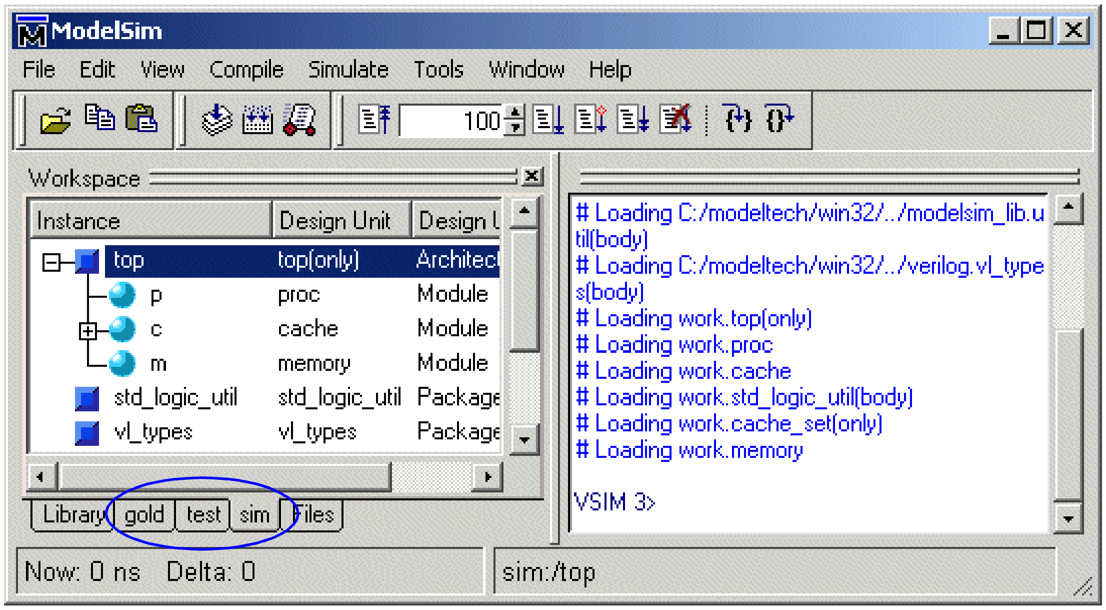
The Open Dataset dialog includes the following options.

- **Dataset Pathname**
  Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset**
  This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing dataset structure

Each dataset you open creates a Structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays the same data as the "Structure window" (UM-331).

The graphic below shows three Structure tabs: one for the active simulation (*sim*) and one each for two datasets (*gold* and *test*).


ModelSim
File Edit View Compile Simulate Tools Window Help
100
Workspace
Library gold test sim Files
Now: 0 ns Delta: 0
sim:/top

Instance	Design Unit	Design U
top	top(only)	Architect
p	proc	Module
c	cache	Module
m	memory	Module
std_logic_util	std_logic_util	Package
vl_types	vl_types	Package

```
# Loading C:/modeltech/win32/../modelsim_lib.util(body)
# Loading C:/modeltech/win32/../verilog.vl_types(body)
# Loading work.top(only)
# Loading work.proc
# Loading work.cache
# Loading work.std_logic_util(body)
# Loading work.cache_set(only)
# Loading work.memory

VSIM 3>
```

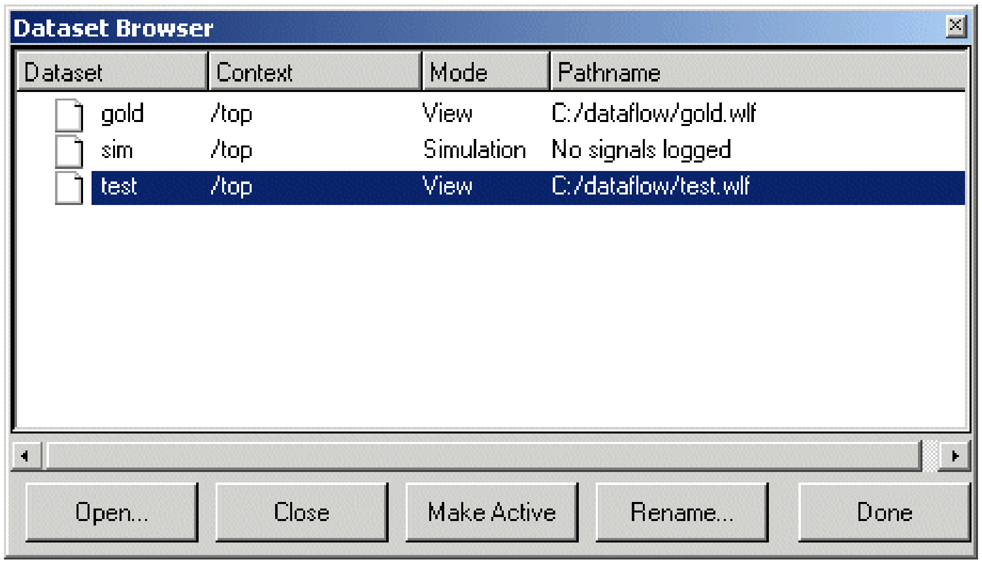
If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking and dragging them.

Each Structure tab has a context menu that you access by clicking the right mouse button anywhere within the Structure tab. See "Structure window context menu" (UM-333) for details.

Managing multiple datasets

GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **View** > **Datasets** (Main window).



Dataset	Context	Mode	Pathname
gold	/top	View	C:/dataflow/gold.wlf
sim	/top	Simulation	No signals logged
test	/top	View	C:/dataflow/test.wlf

The Dataset Browser dialog box includes the following options.

- **Open**
  Opens the Open Dataset dialog box (see "Opening datasets" (UM-242)) so you can open additional datasets.
- **Close**
  Closes the selected dataset. This will also remove the dataset's Structure tab in the Main window workspace.
- **Make Active**
  Makes the selected dataset "active." You can also effect this change by double-clicking the dataset name. Active dataset means that if you type a region path as part of a command and omit the dataset prefix, the active dataset will be assumed. It is equivalent to typing `env <dataset>:` at the VSIM prompt. The active dataset is displayed at the bottom of the Main window.
- **Rename**
  Allows you to assign a new logical name for the selected dataset.

Command line

You can open multiple datasets when the simulator is invoked by specifying more than one **vsim -view <filename>** option. By default the dataset prefix will be the filename of the WLF file. You can specify a different dataset name as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

```
-view <dataset>=<filename>
```

For example: **vsim -view foo=vsim.wlf**

ModelSim designates one of the datasets to be the "active" dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's Structure tab, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the **environment** command (CR-166) to change the active dataset.

Design regions and signal names can be fully specified over multiple WLF files by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out

view:/top/alu/out

golden:.top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command (CR-166), specifying the dataset without a path. For example:

```
env foo:
```

sets the active dataset to **foo** and the current context to the context last specified for **foo**. The context is then applied to any unlocked windows.

The current context of the current dataset (usually referred to as just "current context") is used for finding objects specified without a path.

The Signals window can be locked to a specific context of a dataset. Being locked to a dataset means that the window will update only when the content of that dataset changes. If locked to both a dataset and a context (e.g., test: /top/foo), the window will update only when that specific context changes. You specify the dataset to which the window is locked by selecting **File > Environment** (Signals window).

Restricting the dataset prefix display

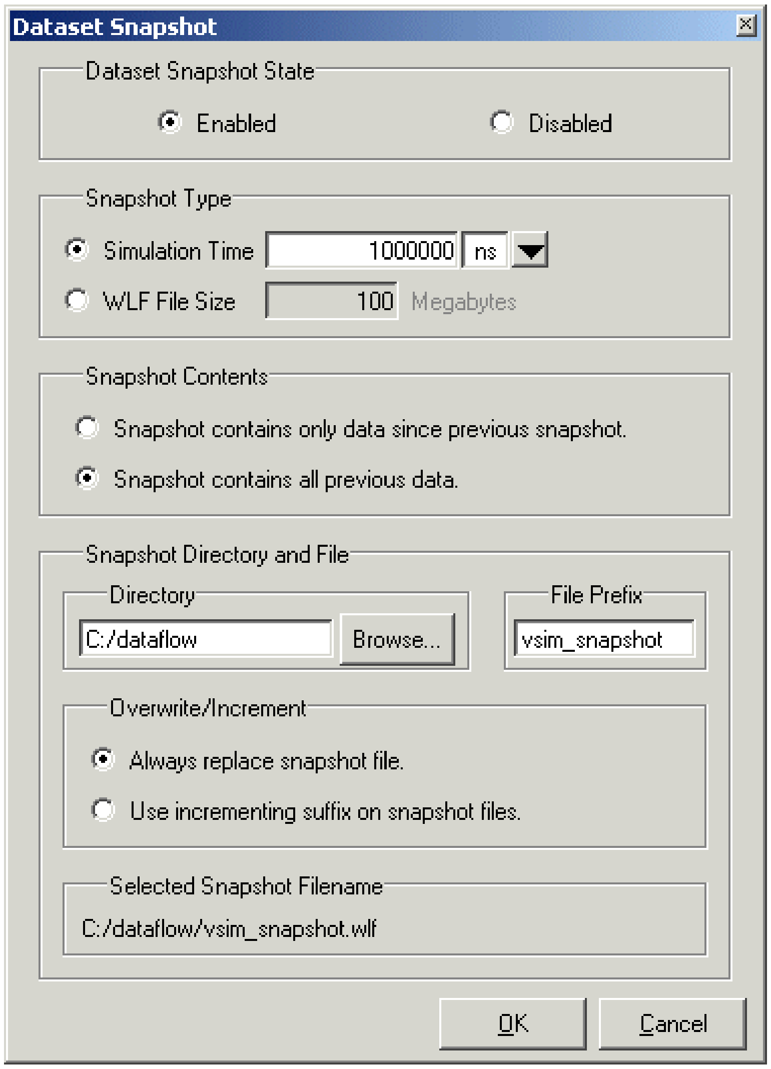
The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. Either edit the *pref.tcl* file directly or use the **Tools > Edit Preferences** (Main window) command to change the variable value.

Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-166) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

Saving at intervals with Dataset Snapshot

Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic "snapshots" of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Once you have logged the appropriate items, select **Tools** > **Dataset Snapshot** (Wave window).


Dataset Snapshot

Dataset Snapshot State
[✓ Enabled] Disabled

Snapshot Type
[✓ Simulation Time] 1000000 ns
WLF File Size 100 Megabytes

Snapshot Contents
Snapshot contains only data since previous snapshot.
[✓ Snapshot contains all previous data.]

Snapshot Directory and File
Directory: C:/dataflow
File Prefix: vsim_snapshot

Overwrite/Increment
[✓ Always replace snapshot file.]
Use incrementing suffix on snapshot files.

Selected Snapshot Filename
C:/dataflow/vsim_snapshot.wlf

The Dataset Snapshot dialog includes these options:

Dataset Snapshot State

- **Enabled/Disabled**
  Enable or disable Dataset Snapshot. All other dialog options are unavailable if Disabled is selected.

Snapshot Type

- **Simulation Time**
  Specifies that data is copied to the specified snapshot file every <x> time units. Default is 1000000 time units.
- **WLF File Size**
  Specifies that data is copied to the specified snapshot file whenever the current simulation WLF file reaches <*x*> megabytes. Default is 100 MB.

Snapshot Contents

- **Snapshot contains only data since previous snapshot**
  Specifies that each snapshot contains only data since the last snapshot. This option causes ModelSim to clear the current simulation WLF file each time a snapshot is taken.
- **Snapshot contains all previous data**
  Specifies that each snapshot contains all data from the time signals were first logged. The entire contents of the current simulation WLF file are saved each time a snapshot is taken.

Snapshot Directory and File

- **Directory**
  The directory in which ModelSim saves the snapshot files.
- **File Prefix**
  The name of the snapshot files. ModelSim adds *.wlf* to the snapshot files.

Overwrite / Increment

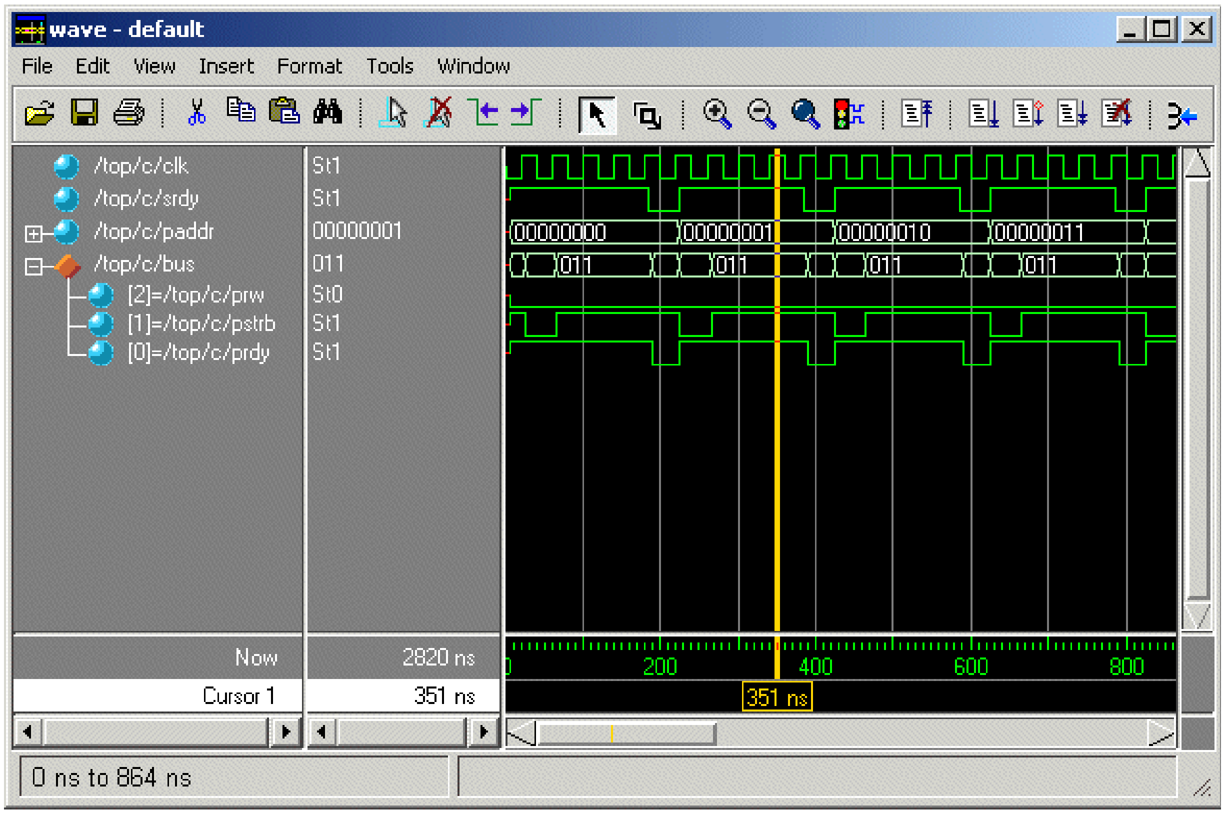
- **Always replace snapshot file**
  Specifies that a single file is created for all snapshots. Each new snapshot overwrites the previous.
- **Use incrementing suffix on snapshot files**
  Specifies that a new file is created for each snapshot. Each new snapshot creates a separate file (e.g., *vsim_snapshot_1.wlf*, *vsim_snapshot_2.wlf*, etc.).

Virtual Objects (User-defined buses, and more)

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the ModelSim simulation kernel. ModelSim supports the following kinds of virtual objects:

- Virtual signals (UM-248)
- Virtual functions (UM-249)
- Virtual regions (UM-250)
- Virtual types (UM-250)

Virtual objects are indicated by an orange diamond as illustrated by *bus* below:



Virtual signals

Virtual signals are aliases for combinations or subelements of signals written to the WLF file by the simulation kernel. They can be displayed in the Signals, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Tools > Combine Signals** (Wave and List windows) command or use the **virtual signal** command (CR-339). Once created, virtual signals can be dragged and dropped from the Signals window to the Wave and List windows.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The **virtual hide** command (CR-330) can be used to hide the display of the broken-down bits if you don't want them cluttering up the Signals window.

If the virtual signal has elements from more than one WLF file, it will be automatically installed in the virtual region *virtuals:/Signals*.

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the scalar elements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (CR-337). By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

Implicit and explicit virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Signals window or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

Virtual functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and can be dependent on simulation time. They can be displayed in the Signals, Wave, and List windows and accessed by the **examine** command (CR-167), but cannot be set by the **force** command (CR-176).

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual signal can be any of the types supported in the GUI expression syntax: integer, real, boolean, std_logic, std_logic_vector, and arrays and records of these types. Verilog types are converted to VHDL 9-state std_logic equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the **virtual function** command (CR-327).

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Signals, Wave, or List window. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

Virtual regions are created and attached using the **virtual region** command (CR-336).

Virtual types

User-defined enumerated types can be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

Virtual types are created using the **virtual type** command (CR-342).

Dataset, WLF file, and virtual commands

The table below provides a brief description of the actions associated with datasets, WLF files, and virtual commands. For complete details about syntax, arguments, and usage, refer to the *ModelSim Command Reference*.

Command name	**Action**
dataset alias (CR-141)	assigns an additional name (alias) to a dataset
dataset clear (CR-142)	removes all event data from the current simulation WLF file while keeping all currently logged signals logged
dataset close (CR-143)	closes the specified dataset
dataset info (CR-144)	reports a variety of information about a dataset
dataset list (CR-145)	lists all open datasets
dataset open (CR-146)	opens a WLF file
dataset rename (CR-147)	assigns a new logical name to the specified dataset
dataset save (CR-148)	saves the current simulation to a WLF file
dataset snapshot (CR-149)	saves the current simulation to a WLF file at regular intervals
log (CR-187)	creates a WLF file for the current simulation
nolog (CR-205)	suspends writing of data to the WLF file for the specified signals
searchlog (CR-255)	searches one or more of the currently open WLF files for a specified condition
virtual function (CR-327)	creates a new signal that consists of logical operations on existing signals and simulation time
virtual region (CR-336)	creates a new user-defined design hierarchy region
virtual signal (CR-339)	creates a new signal that consists of concatenations of signals and subelements
virtual type (CR-342)	creates a new enumerated type
vsim (CR-357) -wlf <filename>	creates a WLF file for the simulation which can be reopened as a dataset
wlf2log (CR-381)	translates a ModelSim WLF file (*vsim.wlf*) to a QuickSim II logfile
wlfman (CR-384)	allows you to get information about and manipulate WLF files
wlfrecover (CR-387)	attempts to "repair" WLF files that are incomplete due to a crash or the file being copied prior to completion of the simulation

10 - Graphic interface

Chapter contents

The example graphics in this chapter illustrate ModelSim's graphic interface within a Windows environment; however, ModelSim's interface remains consistent across all supported platforms. Your operating system provides the basic window-management frames, while ModelSim controls all internal window features such as menus, buttons, and scroll bars.

Because ModelSim's graphic interface is based on Tcl/Tk, you are able to customize your simulation environment. Easily-accessible preference variables and configuration commands give you control over the use and placement of windows, menus, menu options, and buttons.

Window overview

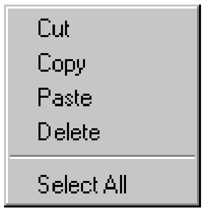
The ModelSim simulation and debugging environment consists of many windows. Multiple windows of each type can be used during simulation (with the exception of the Main window). To make an additional window select **File > New > Window**. A brief description of each window follows:

- Main window (UM-262)
  The initial window that appears upon startup. All subsequent ModelSim windows are opened from the Main window. This window contains the session transcript and the Workspace, which can contain Project, Library, Structure, and Files tabs.
- Dataflow window (UM-270)
  Displays the "physical" connectivity of your VHDL/Verilog design and lets you trace events (causality).
- List window (UM-286)
  Shows the simulation values of selected VHDL signals and variables; Verilog nets, registers, and variables; and SystemC primitive channels (signals) in tabular format.
- Memory window (UM-302)
  Displays memories in the current design context.
- Process window (UM-314)
  Displays a list of processes and SystemC method and thread processes in the region currently selected in the Structure window.
- Signals window (UM-316)
  Shows the names and current values of VHDL signals, and Verilog nets, registers, and variables in the region currently selected in the Structure window. For a selected SystemC module, SystemC primitive channels are shown.
- Source window (UM-325)
  Displays the HDL or C++ source code for the design. (Your source code can remain hidden if you wish.
- Structure window (UM-331)
  Displays the hierarchy of structural elements such as VHDL component instances, packages, blocks, generate statements; Verilog module instances, named blocks, tasks and functions; and SystemC modules. In versions 5.5 and later, this same information is displayed in the Main window workspace.
- Variables window (UM-334)
  Displays VHDL constants, generics, variables, and Verilog registers and variables in the current process and their current values.
- Wave window (UM-337)
  Displays waveforms, and current values for the VHDL signals and variables; Verilog nets, registers, and variables; and SystemC primitive channels (signals) you have selected. Current and past simulations can be compared side-by-side in one Wave window.

Common window features

ModelSim's graphic interface provides many features that add to its usability; features common to many of the windows are described below.

Feature	Feature applies to these windows
Quick access toolbars (UM-256)	Dataflow, Main, Source, and Wave windows
Drag and drop (UM-258)	Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows
Automatic window updating (UM-258)	Dataflow, Memory Process, Signals, and Structure windows
Finding names and searching for values (UM-259)	various windows
Sorting items (UM-259)	Process, Signals, Source, Structure, Variables and Wave windows
Multiple window copies (UM-259)	all windows except the Main window
Menu tear off (UM-259)	all windows
Customizing menus and buttons (UM-260)	all windows
Combining items in the List window (UM-292), Combining items in the Wave window (UM-345)	List and Wave windows
Tree window hierarchical view (UM-261)	Structure, Signals, Variables, and Wave windows



- Cut/Copy/Paste/Delete into any entry box by clicking the right mouse button in the entry box.
- Standard cut/copy/paste shortcut keystrokes – ^X/^C/^V – will work in all entry boxes.
- When the focus changes to an entry box, the contents of that box are selected (highlighted). This allows you to replace the current contents of the entry box with new contents with a simple paste command, without having to delete the old value.
- Dialog boxes will appear on top of their parent window (instead of the upper left corner of the screen).
- You can change the title of any window with the -title switch of the **view** command. See **view** command (CR-320) for details.

- The middle mouse button will allow you to paste the following into the transcript window:

  –text currently selected in the transcript window,

  –a current primary X-Windows selection (can be from another application), or

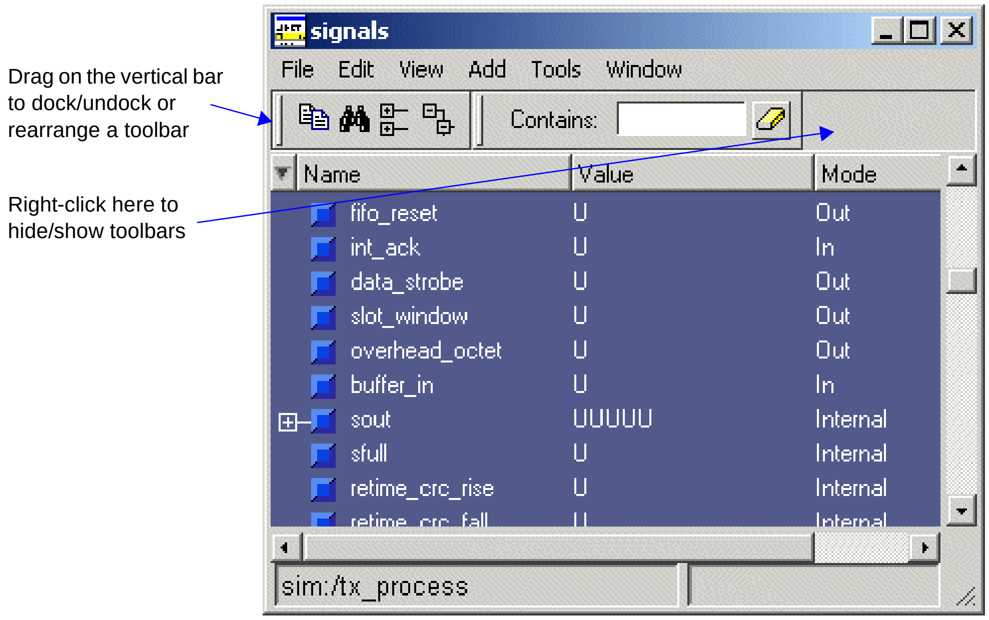
  –contents of the clipboard.

  Selecting text in the transcript window makes it the current primary X-Windows selection. This way you can copy transcript window selections to other X-Windows windows (xterm, emacs, etc.).
- The **Edit** > **Paste** operation in the Transcript pane will ONLY paste from the clipboard.
- All menus highlight their accelerator keys.

Quick access toolbars

Toolbar buttons in several windows provide access to commonly used commands and functions. These toolbars can be docked and undocked (moved to or from the main toolbar area) by clicking and dragging on the vertical bar at the left-edge of a toolbar.

You can also hide/show the various toolbars. To hide or show a toolbar, right-click on a blank spot of the main toolbar area and select a toolbar from the list.


Drag on the vertical bar to dock/undock or rearrange a toolbar
Right-click here to hide/show toolbars
signals
File Edit View Add Tools Window
Contains:
sim:/tx_process

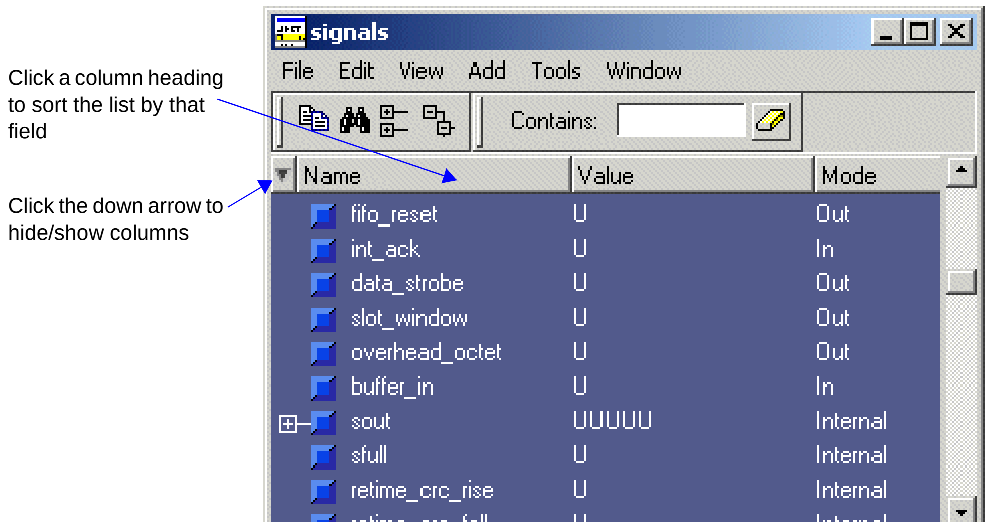
Name	Value	Mode
fifo_reset	U	Out
int_ack	U	In
data_strobe	U	Out
slot_window	U	Out
overhead_octet	U	Out
buffer_in	U	In
sout	UUUUU	Internal
sfull	U	Internal
retime_crc_rise	U	Internal
retime_crc_fall	U	Internal

To reset the toolbars to their original state, right-click on a blank spot of the main toolbar area and select **Reset**.

Columnar information display

Many windows (e.g., Main, Signals, Structure) display information in a columnar format. You can sort by any of the columns by clicking the column heading. Click once to sort in ascending order; click again to sort in descending order.

Also, you can hide or show columns by either right-clicking a column heading and selecting an item from the context menu or by clicking the column-list drop down arrow and selecting an item.


Click a column heading to sort the list by that field
Click the down arrow to hide/show columns
signals
File Edit View Add Tools Window
Contains:

Name	Value	Mode
fifo_reset	U	Out
int_ack	U	In
data_strobe	U	Out
slot_window	U	Out
overhead_octet	U	Out
buffer_in	U	In
sout	UUUUU	Internal
sfull	U	Internal
retime_crc_rise	U	Internal

Docking and undocking panes

Several windows are made up of multiple "panes." When you see a double bar at the top edge of a window area, it means you can click and drag the pane to "undock" it from the parent window. Once the pane is undocked, it becomes a free-floating window.

To redock a floating pane, click on the double bar at the top of the window and drag it back into the parent window.

Drag and drop

Drag and drop of items is possible between the following windows. Using the left mouse button, click and release to select an item, then click and hold to drag it.

- **Drag items from these windows:**
  Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows
- **Drop items into these windows:**
  Dataflow, List, Structure, and Wave windows

Drag and drop works to rearrange items *within* the List and Wave windows as well.

Automatic window updating

Selecting an item in the following windows automatically updates other related ModelSim windows as indicated below:

Select an item in this window	To update these windows
Dataflow window (UM-270)	Memory window (UM-302)
	Process window (UM-314)
	Signals window (UM-316)
	Source window (UM-325)
	Structure window (UM-331)
	Variables window (UM-334)
Process window (UM-314)	Dataflow window (UM-270)
	Signals window (UM-316)
	Source window (UM-325)
	Variables window (UM-334)
Signals window (UM-316)	Dataflow window (UM-270)
Structure window (UM-331) or structure pane in Main window Workspace	Memory window (UM-302)
	Process window (UM-314)
	Signals window (UM-316)
	Source window (UM-325)
	Variables window (UM-334)

Finding names and searching for values

- **Find** item names with the **Edit > Find** menu selection in these windows: Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows.

  A **Find** request that starts with a backslash (\ ) forces case sensitivity. Elsewhere in the pattern backslashes are used to escape special interpretation of basic regular expression characters. To search explicitly for a backslash character, it is necessary to escape the character. For example, to match \Arch Signal 1\, the pattern \\Arch... is required.
- **Search** for item values with the **Edit > Search** menu selection in these windows: List and Wave windows.

Sorting items

Use the **View > Sort** menu selection in the Process, Signals, Structure, Variables, and Wave windows to sort items in ascending, descending or declaration order.

Names such as *net_1*, *net_10*, and *net_2* will sort numerically in the Signals and Wave windows.

Multiple window copies

Select **File > New Window** to create multiple copies of the same window type. The new window will become the default window for that type.

Saving window layout

You can save the current positions and sizes of ModelSim windows as a default. Follow these steps to save the layout as a default:

1. Position and size the windows the way you want them to display.
2. Select **Tools > Save Preferences** (Main window) and save the *modelsim.tcl* file into the desired directory.
3. Modify the "Working Directory" of your ModelSim shortcut to point at the directory, or set the MODELSIM_TCL environment variable to point at the *modelsim.tcl* file (see "Creating environment variables in Windows" (UM-615) for more details).

Context menus

Context menus refer to menus that "pop-up" in the middle of the interface by clicking the right mouse button. The commands on the menu change depending on where in the interface you click. In other words, the menus change based on the context of their use. These menus are available in the following windows: Dataflow, List, Main, Memory, Signals, Source, Structure, and Wave.

Menu tear off

All window menus can be "torn off " to create a separate menu window. To tear off, click on the menu, then select the dotted-line button at the top of the menu.

Customizing menus and buttons

Menus can be added, deleted, and modified in all windows. Custom buttons can also be added to window toolbars. See

- "The Button Adder" (UM-400) for more information.

Controlling fonts in an X-session

When executed via an X-session (e.g., Exceed, VNC), ModelSim uses font definitions from the .Xdefaults file. To ensure that the fonts look correct, create a .Xdefaults file with the following lines:

```
vsim*Font: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*SystemFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*StandardFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*MenuFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
```

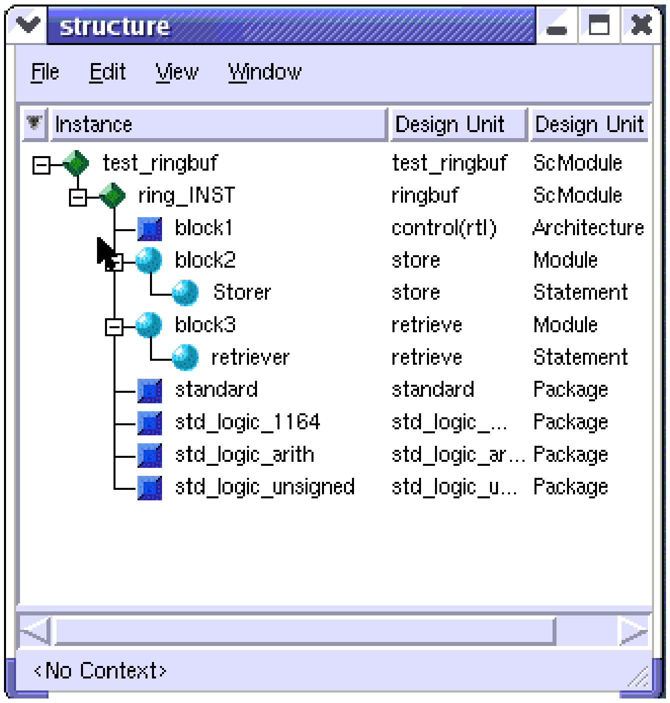
Alternatively, you can choose a different font. Use the program "xlsfonts" to identify which fonts are available on your system.

Also, the following command can be used to update the X resources if you make changes to the .Xdefaults and wish to use those changes on a Linux/UNIX machine:

```
xrdb -merge .Xdefaults
```

Tree window hierarchical view

ModelSim provides a hierarchical, or "tree view" of your design in various windows (e.g., Main, Signals, Structure).



Instance	Design Unit	Design Unit
test_ringbuf	test_ringbuf	ScModule
ring_INST	ringbuf	ScModule
block1	control(rtl)	Architecture
block2	store	Module
Storer	store	Statement
block3	retrieve	Module
retriever	retrieve	Statement
standard	standard	Package
std_logic_1164	std_logic_...	Package
std_logic_arith	std_logic_ar...	Package
std_logic_unsigned	std_logic_u...	Package

Depending on which window you are viewing, you will see various design items. Icons denote the item type as follows:

- Blue circle – Verilog item
- Blue square – VHDL item
- Green diamond – SystemC item
- Orange diamond – Virtual item
- Yellow triangle – Comparison item

See the individual window descriptions later in the chapter for which items are viewable in which windows.

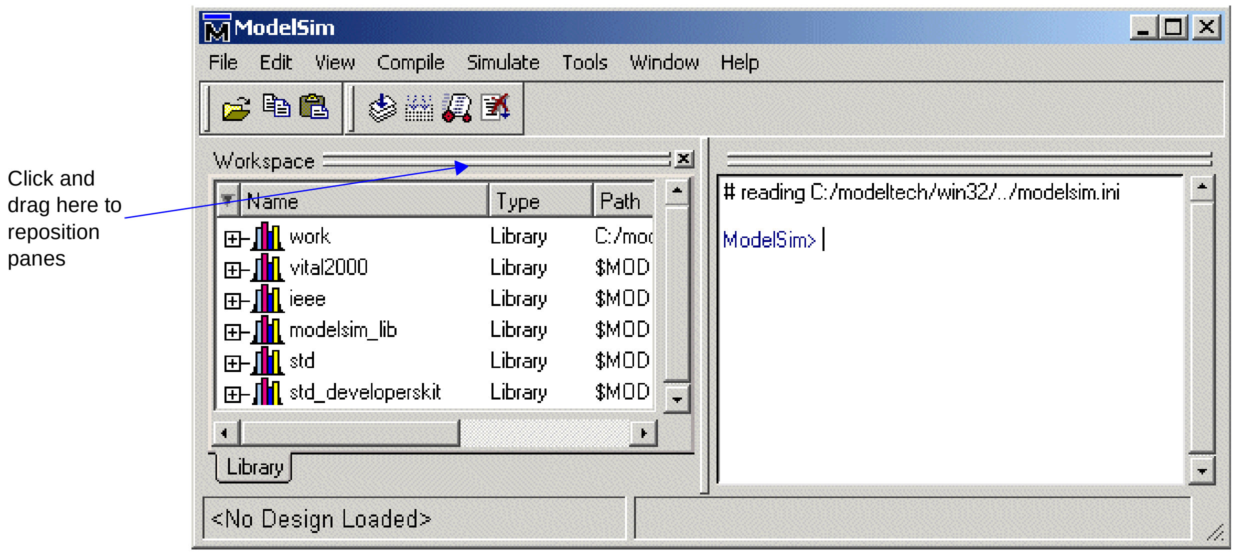
Viewing the hierarchy

Whenever you see a tree view, you can use the mouse to collapse or expand the hierarchy. Select the symbols as shown below to change the view of the structure.

Symbol	Description
[+]	click a plus box to expand the item and view the structure
[-]	click a minus box to hide a hierarchy that has been expanded

Main window

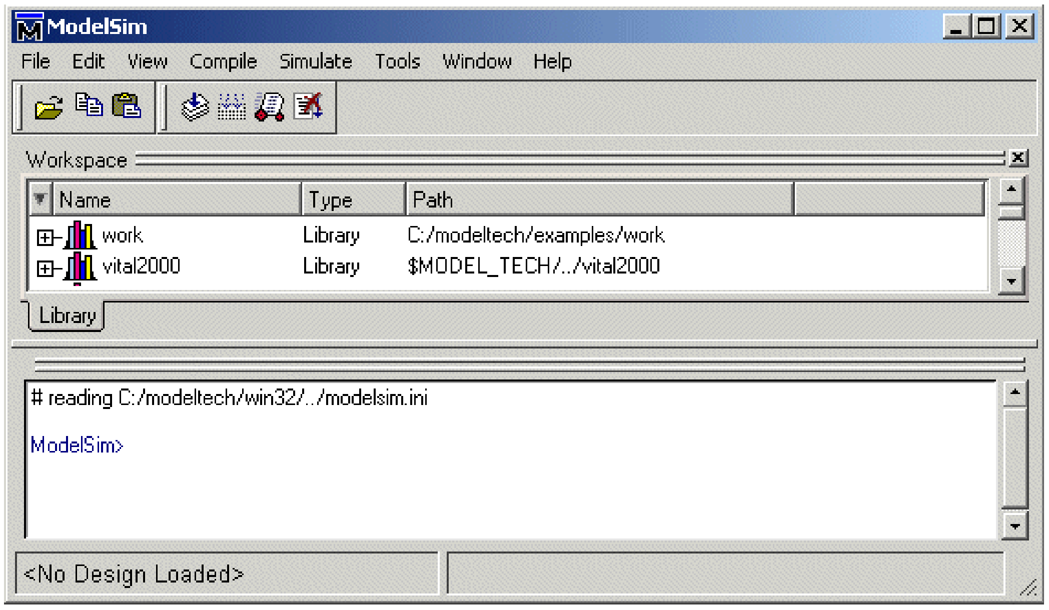
The Main window is pictured below as it appears when ModelSim is first invoked. Note that your operating system graphic interface provides the window-management frame only; ModelSim handles all internal-window features including menus, buttons, and scroll bars.


Click and drag here to reposition panes
ModelSim
File Edit View Compile Simulate Tools Window Help
Workspace
Library
<No Design Loaded>

Name	Type	Path
work	Library	C:/moc
vital2000	Library	$MOD
ieee	Library	$MOD
modelsim_lib	Library	$MOD
std	Library	$MOD
std_developerskit	Library	$MOD

```
# reading C:/modeltech/win32/../modelsim.ini

ModelSim> |
```

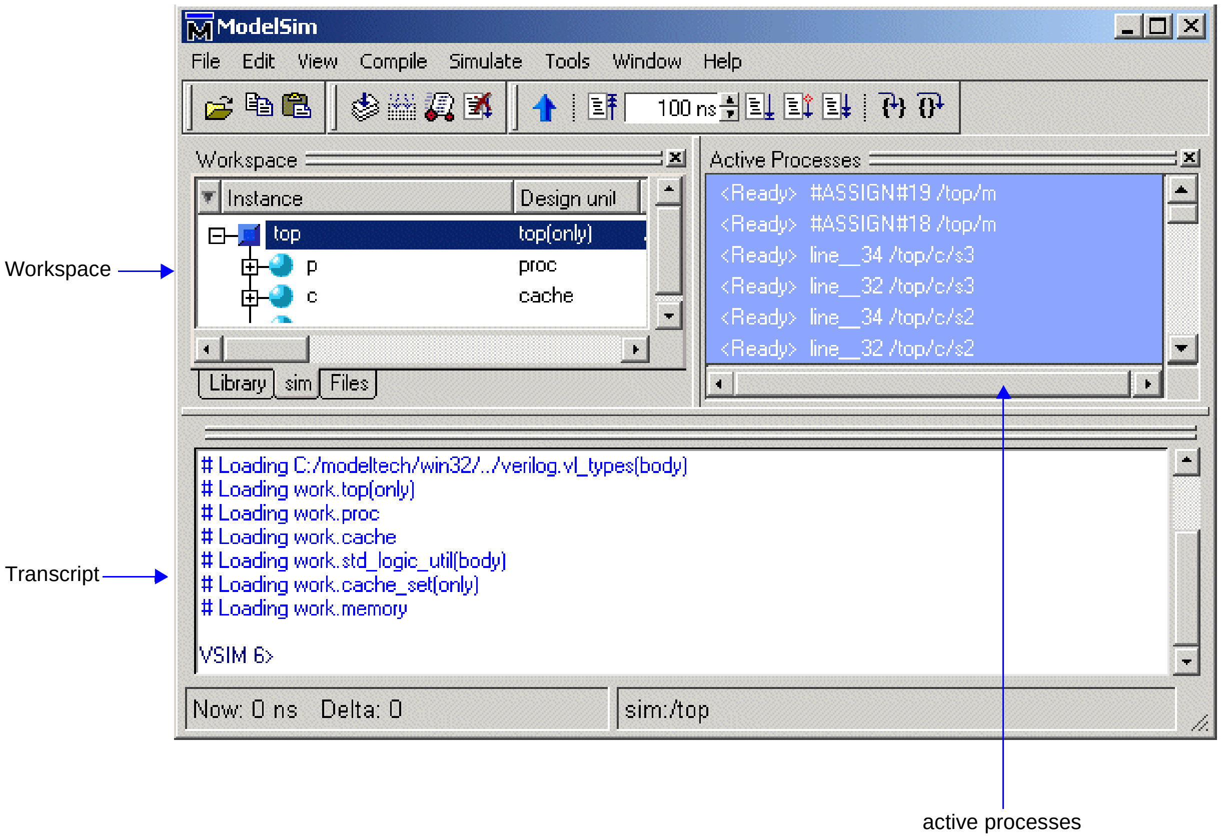
You can customize the Main window layout–click and drag on the bars noted in the graphic above to change the position of the panes and toolbars. You can also change the relative size of each pane by dragging on its border. The graphic below shows a customized layout.



Name	Type	Path
work	Library	C:/modeltech/examples/work
vital2000	Library	$MODEL_TECH/../vital2000

```
# reading C:/modeltech/win32/../modelsim.ini

ModelSim>
```

The graphic below shows the Main window as it might appear when you have a project and a design loaded.



```
<Ready> #ASSIGN#19 /top/m
<Ready> #ASSIGN#18 /top/m
<Ready> line__34 /top/c/s3
<Ready> line__32 /top/c/s3
<Ready> line__34 /top/c/s2
<Ready> line__32 /top/c/s2
```

```
# Loading C:/modeltech/win32/../verilog.vl_types(body)
# Loading work.top(only)
# Loading work.proc
# Loading work.cache
# Loading work.std_logic_util(body)
# Loading work.cache_set(only)
# Loading work.memory

VSIM 6>
```

```
Now: 0 ns  Delta: 0    sim:/top
```

The menu bar at the top of the window provides access to a wide variety of simulation commands and ModelSim preferences. The toolbar provides buttons for quick access to the many common commands. The status bar at the bottom of the window gives you information about the data in the active ModelSim window. The panes display different parts of your design or different features of ModelSim. The panes, menu bar, toolbar, and status bar are described in detail below.

Workspace

The Workspace is available in ModelSim versions 5.5 and later. It provides convenient access to projects, libraries, design files, compiled design units, simulation/dataset structures, and Waveform Comparison objects. It can be hidden or displayed by selecting **View** > **Workspace** (Main window).

The Workspace can display five types of tabs as shown in the graphic above.

- **Project tab**
  Shows all files that are included in the open project. See *Chapter 2 - Projects* for details.
- **Library tab**
  Shows design libraries and compiled design units. See "Managing library contents" (UM-57) for details.

- **Structure tabs**
  Shows a hierarchical view of the active simulation and any open datasets. This is the same data that is displayed in the "Structure window" (UM-331). There is one tab for the current simulation (named "sim") and one tab for each open dataset. See "Viewing dataset structure" (UM-243) for details.
- **Files tab**
  Shows the source files for the loaded design.
- **Compare tab**
  Shows comparison objects that were created by doing a waveform comparison. See *Chapter 13 - Waveform Compare* for details.

Transcript

The Transcript portion of the Main window maintains a running history of commands that are invoked and messages that occur as you work with ModelSim. When a simulation is running, the Transcript displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface.

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see "Mouse and keyboard shortcuts" (UM-269) for details).

Saving the Main window transcript file

Variable settings determine the filename used for saving the Main window transcript. If either **PrefMain(file)** in the *modelsim.tcl* file or **TranscriptFile** in the *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the **TranscriptFile** variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, you can use the **File** > **Transcript** > **Save Transcript As**, or **File** > **Transcript** > **Save Transcript** menu items. The initial save must be made with the **Save Transcript As** selection, which stores the filename in the Tcl variable **PrefMain(saveFile)**. Subsequent saves can be made with the **Save Transcript** selection. Since no automatic saves are performed for this file, it is written only when you invoke a **Save** command. The file is written to the specified directory and records the contents of the transcript at the time of the save.

Using the saved transcript as a macro (DO file)

Saved transcript files can be used as macros (DO files). See the **do** command (CR-156) for more information.

Active processes

This pane displays all processes that are scheduled to run during the current simulation cycle. You can hide or display this pane by selecting **View** > **Active Processes** (Main window). This same data can be displayed in the "Process window" (UM-314).

The Main window menu bar

This section provides information on select menu commands available in the Main window. Many of the commands are also available from a context menu (click right or 3rd mouse button within the window panes).

File menu

New	Folder – create a new folder in the current directory Library – create a new design library and mapping; see "Creating a library" (UM-56)
Open	Exclusion File – open an exclusion filter file; see "Excluding items from coverage" (UM-443)
Import	Library – import FPGA libraries; see "Importing FPGA libraries" (UM-68)
Save	Exclusion File – saves an exclusion filter file; see "Excluding items from coverage" (UM-443)
Change Directory	this command is disabled if you have a project or dataset open or a simulation running
Transcript	Save Transcript – save the Main window transcript; see "Saving the Main window transcript file" (UM-264)
Add to Project	these commands are only available if you have a project open; see *Chapter 2 - Projects*

View menu

Coverage	provides these options: Current Exclusions – hide or show the Exclusions pane Missed Coverage – hide or show the Missed Coverage pane Instance Coverage – hide or show the Instance Coverage pane Details – hide or show the Details pane See *Chapter 12 - Code Coverage* for details on these panes.
Encoding	select from alphabetical list of encoding names that enable proper display of character representations used by various operating systems or file systems, such as Unicode, ASCII, or Shift-JIS.
Properties	show information about the item selected in the Workspace
Project Settings	show information about the open project; disabled if you don’t have a project open

Compile menu

Compile	compile source files; disabled if you have a project open
Compile Options	set various compile options; see "Setting default compile options" (UM-370); disabled if you have a project open
SystemC Link	collects the object files created in the different design libraries, and uses them to build a shared library (*.so*) in the current work library
Compile All	compile all files in the open project; disabled if you don't have a project open
Compile Selected	compile the files selected in the project tab; disabled if you don't have a project open
Compile Order	set the compile order of the files in the open project; see "Changing compile order" (UM-42) for details; disabled if you don't have a project open
Compile Report	report on the compilation history of the selected file(s) in the project; disabled if you don't have a project open
Compile Summary	report on the compilation history of all files in the project; disabled if you don't have a project open

Simulate menu

Simulate	load the selected design unit with the specified options; see "Simulating with the graphic interface" (UM-377)
Simulation Options	set various simulation options; see "Setting default simulation options" (UM-386)
Run	Restart – reload the design elements and reset the simulation time to zero; only design elements that have changed are reloaded; you specify whether to maintain the following after restart–List and Wave window environment, breakpoints, logged signals, virtual definitions, and assertion settings; see also the **restart** command (CR-240)

Tools menu

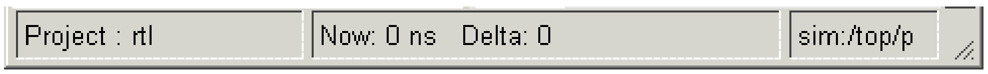
Waveform Compare	see "Waveform Compare menu" (UM-468)
Coverage	load, merge, report on, or clear coverage data
Profile	see "Profile menu" (UM-417)
C Debug (available only on Unix)	see "C Debug menu reference" (UM-488)
Breakpoints	open the Breakpoints dialog box; see "Setting file-line breakpoints" (UM-329) for details
Execute Macro	call and execute a *.do* or *.tcl* macro file
Macro Helper	**UNIX only** - invoke the Macro Helper tool; see also "The Macro Helper" (UM-401)
Tcl Debugger	invoke the Tcl debugger, TDebug; see also "The Tcl Debugger" (UM-402)
TclPro Debugger	invoke TclPro Debugger by Scriptics®, if installed. TclPro Debugger can be acquired from Scriptics.
Options (all options are set for the current session only)	provides these options: Transcript File – set a transcript file to save for this session only Command History – set a file for saving command history only, no comments Save File – set filename for Save Transcript, and Save Transcript As Saved Lines – limit the number of lines saved in the transcript (default is 5000) Line Prefix – specify the comment prefix for the transcript Update Rate – specify the update frequency for the Main window status bar ModelSim Prompt – change the title of the ModelSim prompt VSIM Prompt – change the title of the VSIM prompt Paused Prompt – change the title of the Paused prompt HTML Viewer – specify the path to your browser; used for displaying online help PDF Viewer – specify the path to your PDF viewer; used to display documentation
Edit Preferences	set various preference variables; see "Preference variables located in Tcl files" (UM-631) for more information
Save Preferences	save current ModelSim settings to a Tcl preference file; see "Preference variables located in Tcl files" (UM-631) for more information

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Layout Style[a]	provides these options: Default - restore the windows to version 5.5 layout Millennium - restore the windows to version 5.6 layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically
Customize	use The Button Adder (UM-400) to define and add a button to either the tool or status bar of the specified window

a.You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

The Main window status bar


Project : rtl
Now: 0 ns Delta: 0
sim:/top/p

Fields at the bottom of the Main window provide the following information about the current simulation:

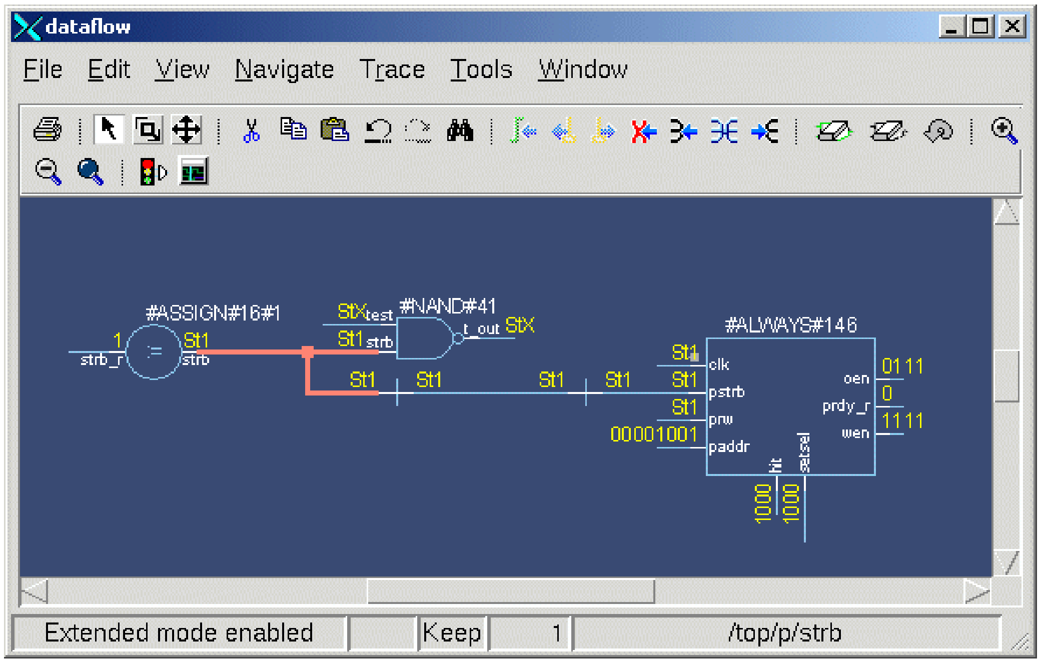
Field	Description
Project	name of the current project
Now	the current simulation time, using the default resolution units (see "Simulating with the graphic interface" (UM-377)), or a larger time unit if one can be used without a fractional remainder
Delta	the current simulation iteration number
environment	name of the current context (item selected in the Structure window (UM-331))

Mouse and keyboard shortcuts

See "Main and Source window mouse and keyboard shortcuts" (UM-639).

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.



Items you can view

The Dataflow window displays processes; signals, nets, and registers; and interconnect. The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See "Symbol mapping" (UM-283) for details.

Note: You cannot view SystemC items in the Dataflow window.

Adding items to the window

You can use any of the following methods to add items to the Dataflow window:

- drag and drop items from other windows
- use the Navigate menu options in the Dataflow window
- use the **add dataflow** command (CR-54)
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add items to the window. The commands include:

View region — clear the window and display all signals from the current region

Add region — display all signals from the current region without first clearing window

View all nets — clear the window and display all signals from the entire design

Add ports — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added items in order to reduce clutter. You can easily view readers by selecting an item and invoking **Navigate > Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

Links to other windows

The Dataflow window has links to other windows as described below:

Window	Link
Main window (UM-262)	select a signal or process in the Dataflow window, and the Structure pane updates if that item is in a different design unit
Process window (UM-314)	select a process in either window, and that process is highlighted in the other
Signals window (UM-316)	select a signal in either window, and that signal is highlighted in the other
Wave window (UM-337)	• trace through the design in the Dataflow window, and the associated signals are added to the Wave window • move a cursor in the Wave window, and the values update in the Dataflow window
Source window (UM-325)	select an item in the Dataflow window, and the Source window updates if that item is in a different source file

Dataflow window menu bar

This section provides information on select menu commands available in the Dataflow window. Many of the commands are also available from the context menu (click right or 3rd mouse button).

Edit menu

Erase selected	clear the selected object from the window
Erase highlight	remove green highlighting from interconnect lines
Regenerate	clear and redraw the display using an optimal layout

View menu

Show Wave	open the embedded wave viewer pane
Select	set left mouse button to select mode and middle mouse button to zoom mode
Zoom	set left mouse button to zoom mode and middle mouse button to pan mode
Pan	set left mouse button to pan mode and middle mouse button to zoom mode
Default	set mouse to default mode

Navigate menu

Expand net to drivers	display driver(s) of the selected signal, net, or register
Expand net to readers	display reader(s) of the selected signal, net, or register
Expand net	display driver(s) and reader(s) of the selected signal, net, or register
Expand to design inputs	display the top-level driver of the net, which will most likely be in a testbench or in the top entity or module
Expand to hierarchy inputs	display the primary driver (port) of the net within its level of hierarchy
Hide selected	remove the selected component and all other components from the same region and replace them with a single component representing that region
Show selected	expand the selected component to show all underlying components

View region	clear the window and display all signals from the current region
Add region	display all signals from the current region without first clearing the window
View all nets	clear the window and display all signals from the entire design
Add ports	add port symbols to the port signals in the current region

Trace menu

TraceX^{TM}	step back to the last driver of an unknown (X) value
ChaseX^{TM}	jump to the source of an unknown (X) value
TraceX Delay	step back in time to the last driver of an unknown (X) value
ChaseX Delay	jump back in time to the point where the output value transitions to X
Trace next event	move the next event cursor to the next input event driving the selected output
Trace event set	jump to the source of the selected input event
Trace event reset	return the next event cursor to the selected output

Tools menu

Load built-in symbol map	load a .bsm file for mapping symbol instances; see "Symbol mapping" (UM-283)
Load symlib library	load a user-defined symbol library
Create symlib index	create an index for a user-defined symbol library
Options	configure Dataflow window preferences

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Exploring the connectivity of your design

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

You can expand the view of your design using menu commands or your mouse. To expand with the mouse, simply double click a signal, register, or process. Depending on the specific item you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

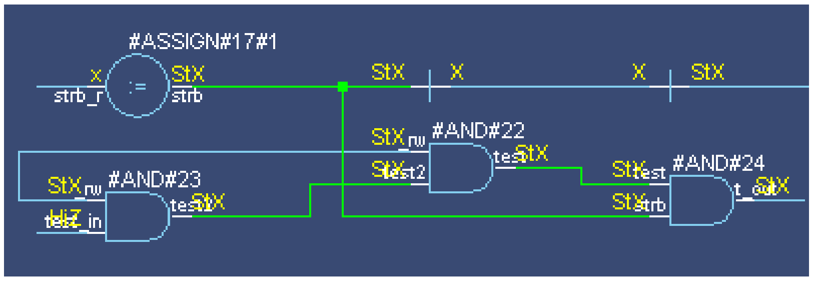
Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described below:

	Expand net to all drivers display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
	Expand net to all readers display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers

As you expand the view, note that the "layout" of the design may adjust to best show the connectivity. For example, the location of an input signal may shift from the bottom to the top of a process.

Tracking your path through the design

You can quickly traverse through many components in your design. To help mark your path, the items that you have expanded are highlighted in green.



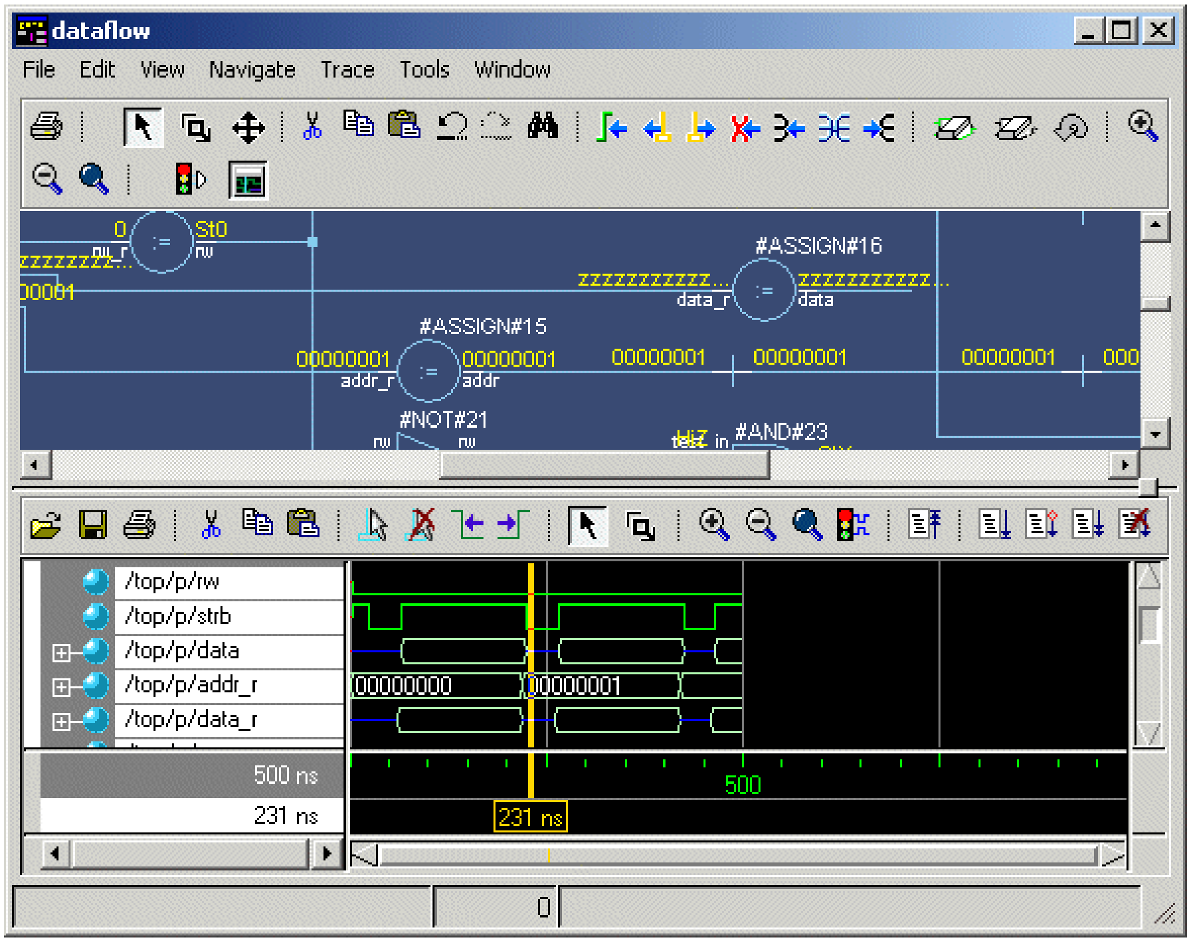
You can clear this highlighting using the **Edit > Erase highlight** command.

The embedded wave viewer

Another way of exploring your design is to use the Dataflow window's embedded wave viewer. This viewer closely resembles, in appearance and operation, the stand-alone Wave window (see "Wave window" (UM-337) for more information).

The wave viewer is opened using the **View > Show Wave** command.

One common scenario is to place signals in the wave viewer and the Dataflow panes, run the design for some amount of time, and then use time cursors to investigate value changes. In other words, as you place and move cursors in the wave viewer pane (see "Using time cursors in the Wave window" (UM-358) for details), the signal values update in the Dataflow pane.



Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See "Tracing events (causality)" (UM-277) for another example of using the embedded wave viewer.

Zooming and panning

The Dataflow window offers several tools for zooming and panning the display.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

🔍	**Zoom In** zoom in by a factor of two from the current view	🔍	**Zoom Out** zoom out by a factor of two from current view
🔍	**Zoom Full** zoom out to view the entire schematic		

Zooming with the mouse

To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **View > Zoom** and then use the left mouse button.

Four zoom options are possible by clicking and dragging in different directions:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected
- Up-Left: Zoom Full

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

Panning with the mouse

You can pan with the mouse in two ways: 1) enter Pan Mode by selecting **View > Pan** and then drag with the left mouse button to move the design; 2) hold down the <Ctrl> key and drag with the middle mouse button to move the design.

Tracing events (causality)

One of the most useful features of the Dataflow window is tracing an event to see the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see "The embedded wave viewer" (UM-275) for more details).

In short you identify an output of interest in the Dataflow pane and then use time cursors in the wave viewer pane to identify events that contribute to the output.

The process for tracing events is as follows:

1 Log all signals before starting the simulation (add log -r /*).

2 After running a simulation for some period of time, open the Dataflow window and the wave viewer pane.

3 Add a process or signal of interest into the Dataflow window (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.

4 Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.

5 Select **Trace** > **Trace next event**.

A second cursor is added at the most recent input event.

6 Keep selecting **Trace > Trace next event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave pane.

7 Now select **Trace** > **Trace event set**.

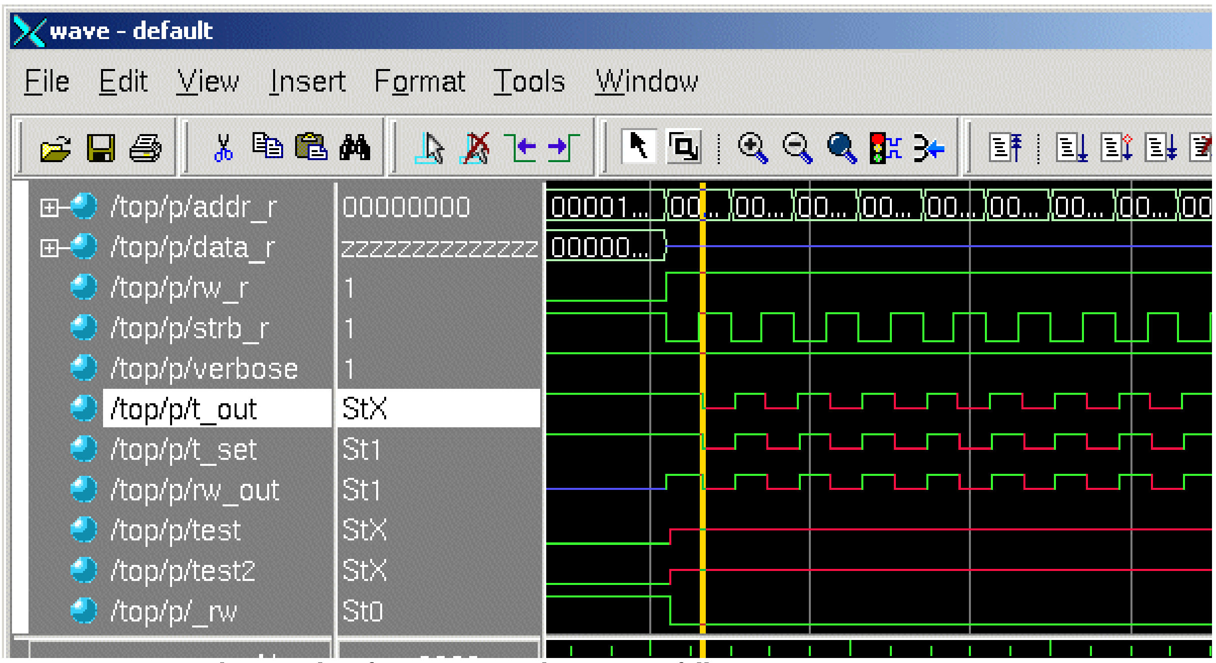
The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.

8 To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace** > **Trace event reset**.

Tracing the source of an unknown (X)

Another useful debugging option is locating the source of an unknown (X). Unknown values are most clearly seen in the Wave window—the waveform displays in red when a value is unknown.



/top/p/addr_r	00000000
/top/p/data_r	zzzzzzzzzzzzzz
/top/p/rw_r	1
/top/p/strb_r	1
/top/p/verbose	1
/top/p/t_out	StX
/top/p/t_set	St1
/top/p/rw_out	St1
/top/p/test	StX
/top/p/test2	StX
/top/p/_rw	St0

The procedure for tracing an unknown is as follows:

1 Load your design.

2 Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /*** will log all signals in the design).

3 Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.

4 Put a cursor on the time at which the signal value is unknown.

5 Add the signal of interest to the Dataflow window, making sure the signal is selected.

6 Select **Trace > TraceX**, **Trace > TraceX Delay, Trace > ChaseX**, or **Trace > ChaseX Delay**.

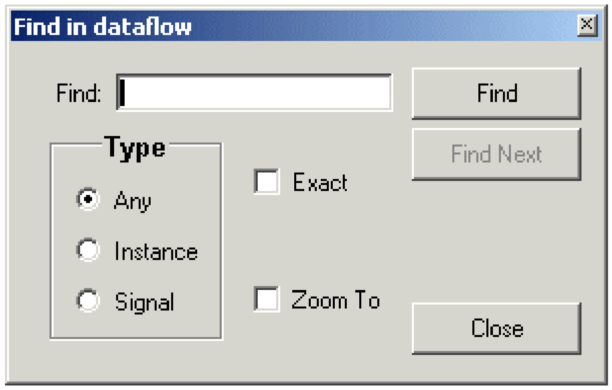
These commands behave as follows:

TraceX / TraceX Delay— Step back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with backannotated delays.

ChaseX / ChaseX Delay — "Jumps" through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in time to the point where the output value transitions to X. **ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with backannotated delays.

Finding items by name in the Dataflow window

Select **Edit > Find** to search for signal, net, or register names or an instance of a component.



Enter an item name and specify whether it is an instance of a process (Instance); a signal, net, or register (Signal); or either (Any).

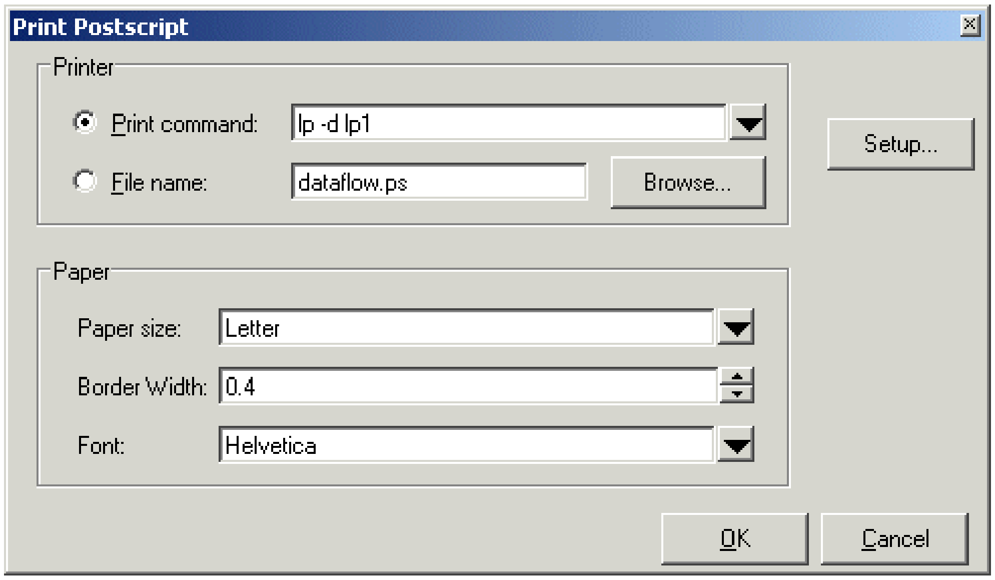
Specify **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

If you want to zoom in on the located item, select Zoom To. You can continue searching using the Find Next button.

Printing and saving the display

Saving a .eps file and printing under UNIX

Select **File > Print Postscript** to print the Dataflow display in UNIX, or save the waveform as a .eps file on any platform.



Print command: [✓] lp -d lp1
File name: dataflow.ps
Paper size: Letter
Border Width: 0.4
Font: Helvetica

The **Print Postscript** dialog box includes these options:

Printer

- **Print command**
  Enter a UNIX print command to print the display in a UNIX environment.
- **File name**
  Enter a filename for the encapsulated Postscript (.eps) file to create; or browse to a previously created .eps file and use that filename.

Paper

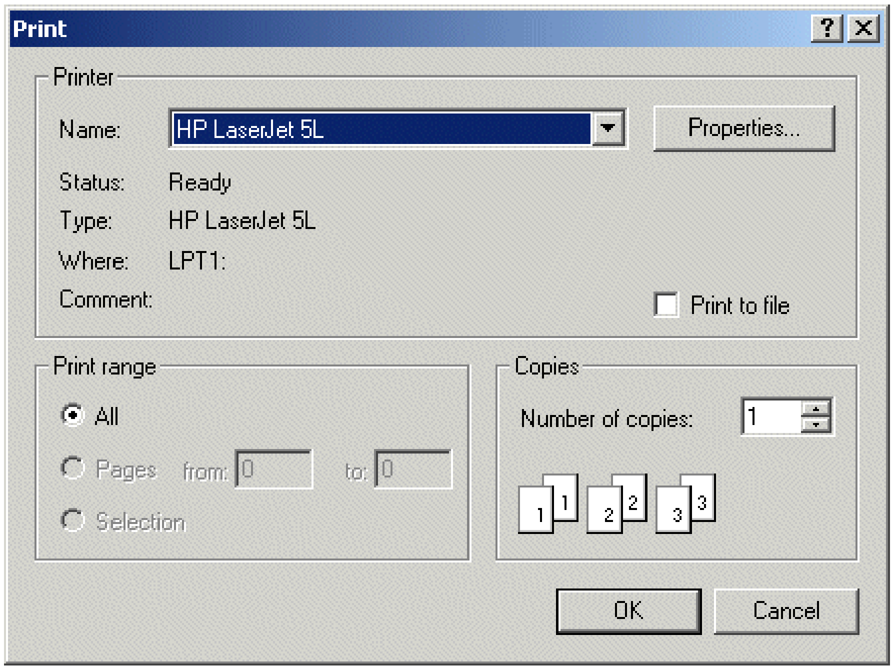
- **Paper size**
  Select the paper size used by the printer.
- **Border width**
  Specify the border in inches.
- **Font**
  Specify the font to use for printing.

Setup button

See "Printer Page Setup" (UM-366).

Printing on Windows platforms

Select **File** > **Print** to print the Dataflow display or to save the display to a file.


Print

Printer

Name: HP LaserJet 5L

Status: Ready

Type: HP LaserJet 5L

Where: LPT1:

Comment:

[ ] Print to file

Print range

[✓ All]

Pages from: 0 to: 0

Selection

Copies

Number of copies: 1

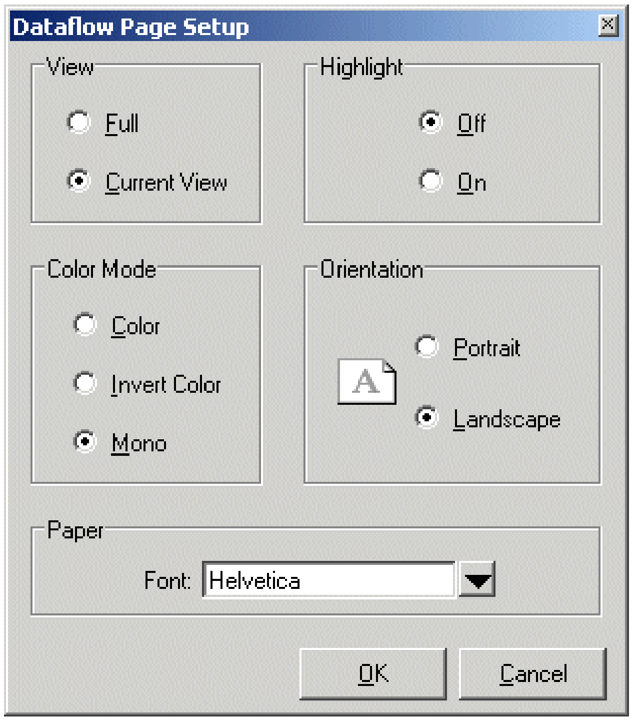
The **Print** dialog box includes these options:

Printer

- **Name**
  Choose the printer from the drop-down menu. Set printer properties with the *Properties* button.
- **Status**
  Indicates the availability of the selected printer.
- **Type**
  Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.
- **Where**
  The printer port for the selected printer.
- **Comment**
  The printer comment from the printer properties dialog box.
- **Print to file**
  Make this selection to print the display to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers create a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the **File** > **Print Postscript** menu selection.

Configuring page setup

Clicking the Setup button in the Print Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens via **File > Page setup**).



The **Dataflow Page Setup** dialog box includes these options:

- **View**
  Specifies **Full** (everything in the window) or **Current View** (only that which is visible).
- **Highlight**
  Specifies that highlighting (see "Tracking your path through the design" (UM-274)) is **On** or **Off**.
- **Color Mode**
  Specifies **Color** (256 colors), **Invert Color** (gray-scale) or **Mono** (monochrome) color mode.
- **Orientation**
  Specifies **Landscape** (horizontal) or **Portrait** (vertical) orientation.
- **Paper**
  Specifies the font to use for printing.

Symbol mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. This is done through a file containing name pairs, one per line, where the first name is the concatenation of the design unit and process names, (DUname.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).p1 XOR
org(only).p1 OR
andg(only).p1 AND
```

Entities and modules are mapped the same way:

```
AND1 AND
AND2 AND  # A 2-input and gate
AND3 AND
AND4 AND
AND5 AND
AND6 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file *dataflow.bsm* (.bsm stands for "Built-in Symbol Map). It will read all files found.

User-defined symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview^{TM} widget Symlib format. For more specific details on this widget, see www.model.com/support/documentation/BOOK/nlviewSymlib.pdf.

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file *dataflow.sym*. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \
    port a in -loc -12 -15 0 -15 \
    pinattrdsp @name -cl 2 -15 8 \
    port b in -loc -12 15 0 15 \
    pinattrdsp @name -cl 2 15 8 \
    port cin in -loc 20 -40 20 -28 \
    pinattrdsp @name -uc 19 -26 8 \
    port cout out -loc 20 40 20 28 \
pinattrdsp @name -lc 19 26 8 \
    port sum out -loc 63 0 51 0 \
    pinattrdsp @name -cr 49 0 8 \
    path 10 0 0 7 \
    path 0 7 0 35 \
    path 0 35 51 17 \
    path 51 17 51 -17 \
    path 51 -17 0 -35 \
    path 0 -35 0 -7 \
    path 0 -7 10 0
```

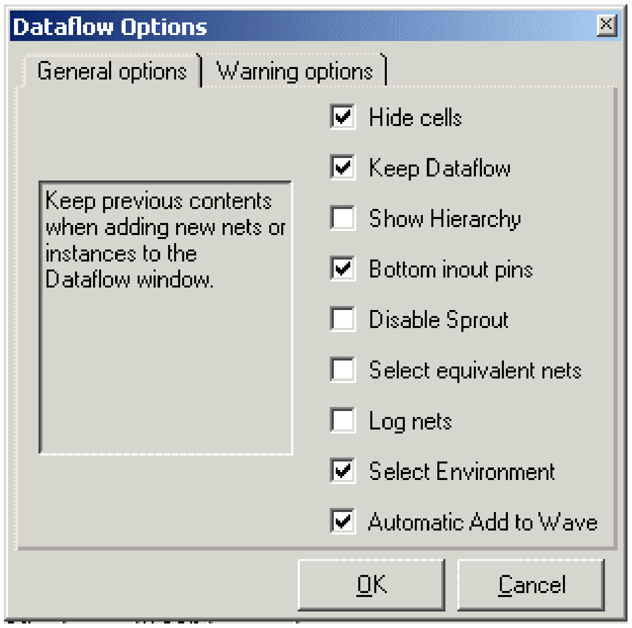
Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

Important: When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Tools > Create symlib index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index.

Configuring window options

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **Tools > Options** to open the Dataflow Options dialog box.



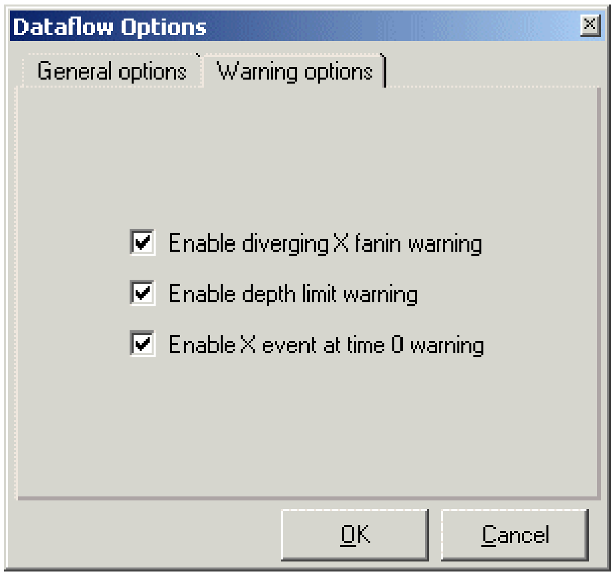
Keep previous contents when adding new nets or instances to the Dataflow window.

[✓] Hide cells
[✓] Keep Dataflow
[ ] Show Hierarchy
[✓] Bottom inout pins
[ ] Disable Sprout
[ ] Select equivalent nets
[ ] Log nets
[✓] Select Environment
[✓] Automatic Add to Wave

The **General options** tab includes these options:

- **Hide Cells**
  By default the Dataflow window automatically hides instances that have either 'celldefine, VITAL_LEVEL0, or VITAL_LEVEL1 attributes. Unchecking this disables automatic cell hiding.
- **Keep Dataflow**
  Keeps previous contents when adding new signals or processes to the window.
- **Show Hierarchy**
  Displays connectivity using hierarchical references. Note that selecting this will erase the current contents of the window.
- **Bottom inout pins**
  Places inout pins on the bottom of components rather than on the right with output pins.

- **Disable Sprout**
  Displays only the selected signal or process with its immediate fanin/fanout. Configures window to behave like the Dataflow window of versions prior to 5.6.
- **Select equivalent nets**
  If the item you select traverses hierarchy, then ModelSim selects all connected items across the hierarchy.
- **Log nets**
  Logs signals when they are added to the window.
- **Select Environment**
  Updates the Structure, Signals, and Source windows to reflect the net selected in the Dataflow window.
- **Automatic Add to Wave**
  Adds signals automatically to the Wave pane or window when executing ChaseX or TraceX.



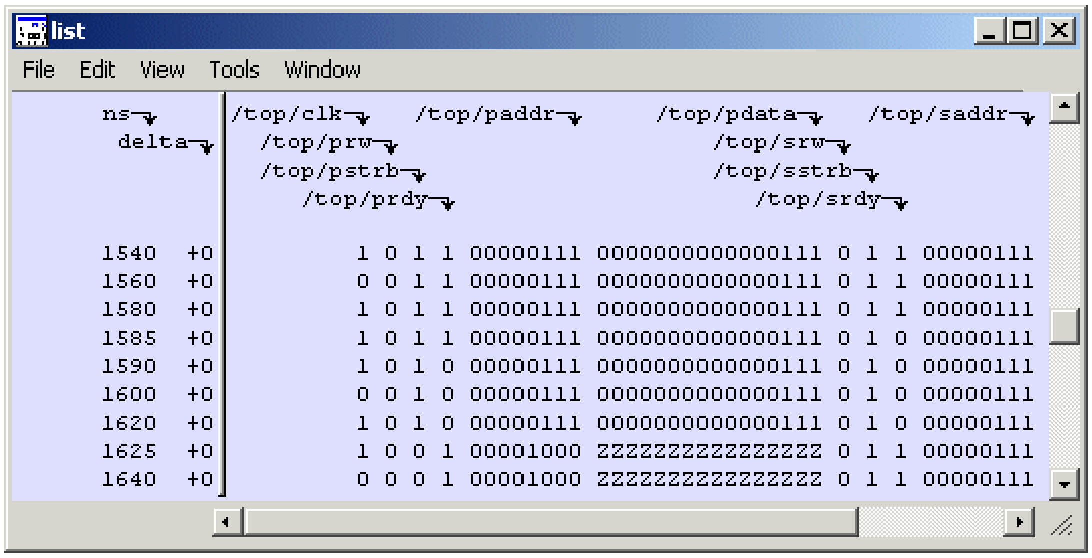
[✓] Enable diverging X fanin warning
[✓] Enable depth limit warning
[✓] Enable X event at time 0 warning

The **Warning options** tab includes these options:

- **Enable diverging X fanin warning**
  Enables the warning message, "ChaseX: diverging X fanin. Reduce the selection list and try again."
- **Enable depth limit warning**
  Enables the warning message, "ChaseX: Stop because depth limit reached! Possible loop?"
- **Enable X event at time 0 warning**
  Enables the warning message, ""Driving X event at time 0."

List window

The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable panes, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.



```
     ns       /top/clk      /top/paddr        /top/pdata      /top/saddr
      delta     /top/prw                           /top/srw
                /top/pstrb                         /top/sstrb
                   /top/prdy                          /top/srdy

    1540  +0           1 0 1 1 00000111 0000000000000111 0 1 1 00000111
    1560  +0           0 0 1 1 00000111 0000000000000111 0 1 1 00000111
    1580  +0           1 0 1 1 00000111 0000000000000111 0 1 1 00000111
    1585  +0           1 0 1 1 00000111 0000000000000111 0 1 0 00000111
    1590  +0           1 0 1 0 00000111 0000000000000111 0 1 0 00000111
    1600  +0           0 0 1 0 00000111 0000000000000111 0 1 0 00000111
    1620  +0           1 0 1 0 00000111 0000000000000111 0 1 0 00000111
    1625  +0           1 0 0 1 00001000 ZZZZZZZZZZZZZZZZ 0 1 1 00000111
    1640  +0           0 0 0 1 00001000 ZZZZZZZZZZZZZZZZ 0 1 1 00000111
```

Items you can view

The following type of items can be viewed in the List window:

VHDL

signals, aliases, process variables, and shared variables

Verilog

nets, registers, and variables

SystemC

primitive channels and ports

Comparisons

comparison regions and comparison signals; see *Chapter 13 - Waveform Compare* for more information

Virtuals

Virtual signals and functions

Constants, generics, and parameters are not viewable in the List or Wave windows.

Adding items to the List window

Before adding items to the List window you may want to set the window display properties (see "Setting List window display properties" (UM-293)). You can add items to the List window in several ways.

Adding items with drag and drop

You can drag and drop items into the List window from the Signals, Source, Process, Variables, Wave, or Structure window. Select the items in the first window, then drop them into the List window. Depending on what you select, all items or any portion of the design may be added.

Adding items from the Main window command line

Invoke the **add list** (CR-55) command to add one or more individual items; separate the names with a space:

```
add list <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list *
```

Or add all the items in the design with:

```
add list -r /*
```

Adding items with a List window format file

To use a List window format file you must first save a format file for the design you are simulating. The saved format file can then be used as a DO file to recreate the List window formatting. Follow these steps:

- Add items to your List window.
- Edit and format the items to create the view you want (see "Editing and formatting items in the List window" (UM-290)).
- Save the format to a file by selecting **File** > **Save Format** (List window).

To use the format file, start with a blank List window, and run the DO file in one of two ways:

- Invoke the **do** (CR-156) command from the command line:
  ```
  do <my_list_format>
  ```
- Select **File** > **Load Format** from the List window menu bar.

List window format files are design-specific; use them only with the design you were simulating when they were created. If you try to use the wrong format file, ModelSim will advise you of the items it expects to find.

The List window menu bar

This section provides information on select menu commands available in the List window.

File menu

Write List	save the List window data to a text file in one of three formats; see "Saving List window data to a file" (UM-301) for details
Save Format	save the current List window display and signal preferences to a DO (macro) file; running the DO file will reformat the List window to match the display as it appeared when the DO file was created
Load Format	run a List window format DO file previously saved with Save Format

Edit menu

Add Marker	add a time marker at the currently selected line; see "Setting time markers in the List window" (UM-300)
Delete Marker	delete the selected marker from the listing
Find	find the specified item label within the List window; see "Finding items by name in the List window" (UM-297)
Search	search the List window for a specified value, or the next transition for the selected signal; see "Searching for item values in the List window" (UM-298)

View menu

Signal Properties	set item properties; see "Editing and formatting items in the List window" (UM-290)
Goto	choose the time marker to go to from a list of current markers

Tools menu

Combine Signals	combine the selected fields into a user-defined bus; keep copies of the original items rather than moving them; see "Combining items in the List window" (UM-292)
Window Preferences	set display properties for items in the window; see "Setting List window display properties" (UM-293)

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

The List window context menu

Some commands like the following are available by clicking the right mouse button on an entry in the right-hand pane:

Examine	display the value of the item at the time selected
Annotate Diff	Add a note to explain a comparison difference. See *Chapter 13 - Waveform Compare* for further information.
Ignore Diff	Disregard the selected comparison difference. See *Chapter 13 - Waveform Compare* for further information.

Editing and formatting items in the List window

Once you have the items you want in the List window, you can edit and format the list to create the view you find most useful. (See also, "Adding items to the List window" (UM-287))

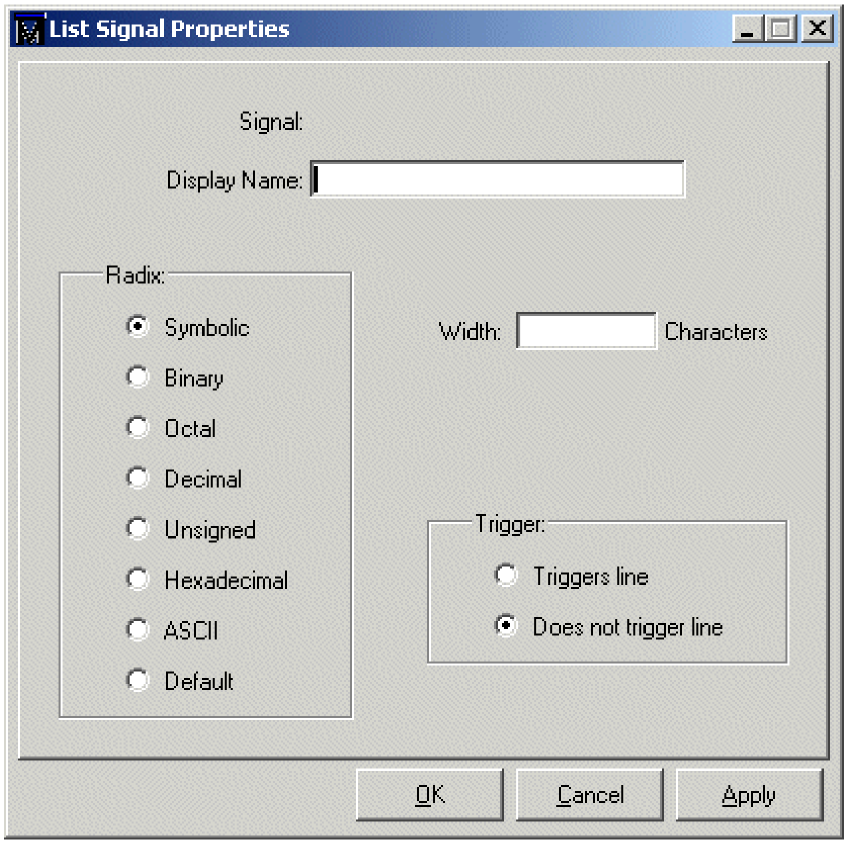
To edit an item:

Select the item's label at the top of the List window or one of its values from the listing. Move, copy or remove the item by selecting commands from the List window Edit menu (UM-288) menu.

You can also click+drag to move items within the window.

To format an item:

Select the item's label at the top of the List window or one of its values from the listing, then select **View > Signal Properties** (List window). The resulting List Signal Properties dialog box allows you to set the item's label, label width, triggering, and radix.


List Signal Properties

Signal:

Display Name:

Radix: [✓ Symbolic] Binary Octal Decimal Unsigned Hexadecimal ASCII Default

Width: Characters

Trigger: Triggers line [✓ Does not trigger line]

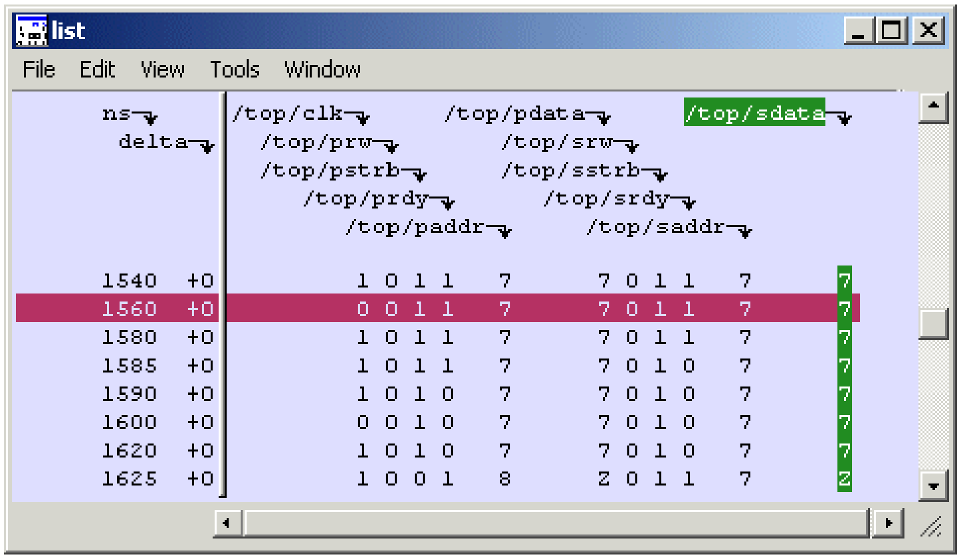
The **List Signal Properties** dialog box includes these options:

- **Signal**
  Shows the full pathname of the selected signal.
- **Display Name**
  Specifies the label that appears at the top of the List window column.

- **Radix**
  Specifies the radix (base) in which the item value is expressed. The default radix is symbolic, which means that for an enumerated type, the List window lists the actual values of the enumerated type of that item. You can change the default radix for the current simulation using either **Simulate > Simulation Options** (Main window) or the **radix** command (CR-235). You can change the default radix permanently by editing the DefaultRadix (UM-623) variable in the *modelsim.ini* file.

  For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the item value is converted to an appropriate representation in that radix. In the system initialization file, *modelsim.tcl*, you can specify the list translation rules for arrays of enumerated types for binary, octal, decimal, unsigned decimal, or hexadecimal item values in the design unit.

  Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image on page UM-286 (with symbolic values).



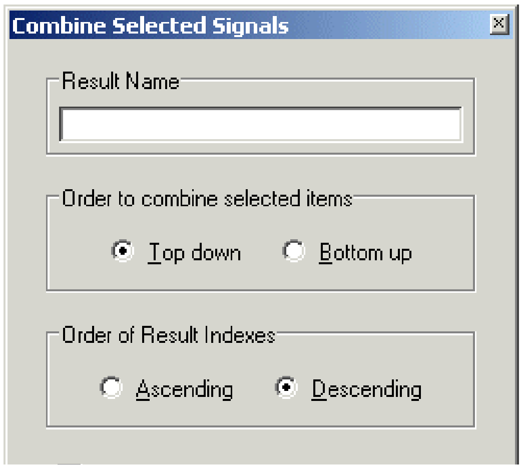
```
     ns      /top/clk      /top/pdata      /top/sdata
      delta    /top/prw      /top/srw
               /top/pstrb    /top/sstrb
                  /top/prdy     /top/srdy
                     /top/paddr    /top/saddr

  1540  +0     1 0 1 1  7    7 0 1 1  7   7
  1560  +0     0 0 1 1  7    7 0 1 1  7   7
  1580  +0     1 0 1 1  7    7 0 1 1  7   7
  1585  +0     1 0 1 1  7    7 0 1 0  7   7
  1590  +0     1 0 1 0  7    7 0 1 0  7   7
  1600  +0     0 0 1 0  7    7 0 1 0  7   7
  1620  +0     1 0 1 0  7    7 0 1 0  7   7
  1625  +0     1 0 0 1  8    2 0 1 1  7   2
```

- **Width**
  Allows you to specify the desired width of the column used to list the item value. The default is an approximation of the width of the current value.
- **Trigger: Triggers line**
  Specifies that a change in the value of the selected item causes a new line to be displayed in the List window.
- **Trigger: Does not trigger line**
  Specifies that a change in the value of the selected item does not affect the List window.

  The trigger specification affects the trigger property of the selected item. See also, "Setting List window display properties" (UM-293).

Combining items in the List window

You can combine signals in the List window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. To create a bus, select one or more signals in the List window and then choose **Tools > Combine Signals**.


Combine Selected Signals

Result Name

Order to combine selected items: [✓ Top down] Bottom up

Order of Result Indexes: Ascending [✓ Descending]

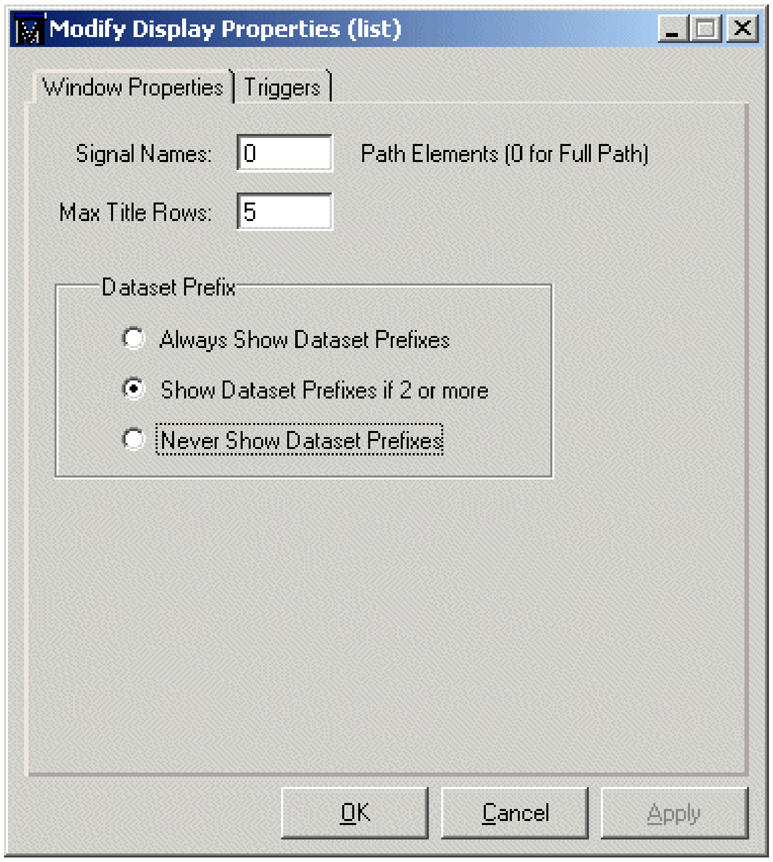
The **Combine Selected Signals** dialog box includes these options:

- **Name**
  Specifies the name of the newly created bus.
- **Order of Indexes**
  Specifies in which order the selected signals are indexed in the bus. If set to **Ascending**, the first signal selected in the List window will be assigned an index of 0. If set to **Descending**, the first signal selected will be assigned the highest index number. Note that the signals are added to the bus in the order that they appear in the window. Ascending and descending affect only the order and direction of the indexes of the bus.
- **Remove selected signals after combining**
  Specifies whether you want to remove the selected signals from the List window once the bus is created.

Setting List window display properties

Before you add items to the List window you can set the window’s display properties. To change when and how a signal is displayed in the List window, select **Tools > Window Preferences** (List window). The resulting Modify Display Properties dialog box contains tabs for Window Properties and Triggers.

Window Properties tab


Modify Display Properties (list)

Window Properties | Triggers

Signal Names: 0 Path Elements (0 for Full Path)

Max Title Rows: 5

Dataset Prefix

- Always Show Dataset Prefixes
- [✓ Show Dataset Prefixes if 2 or more]
- Never Show Dataset Prefixes

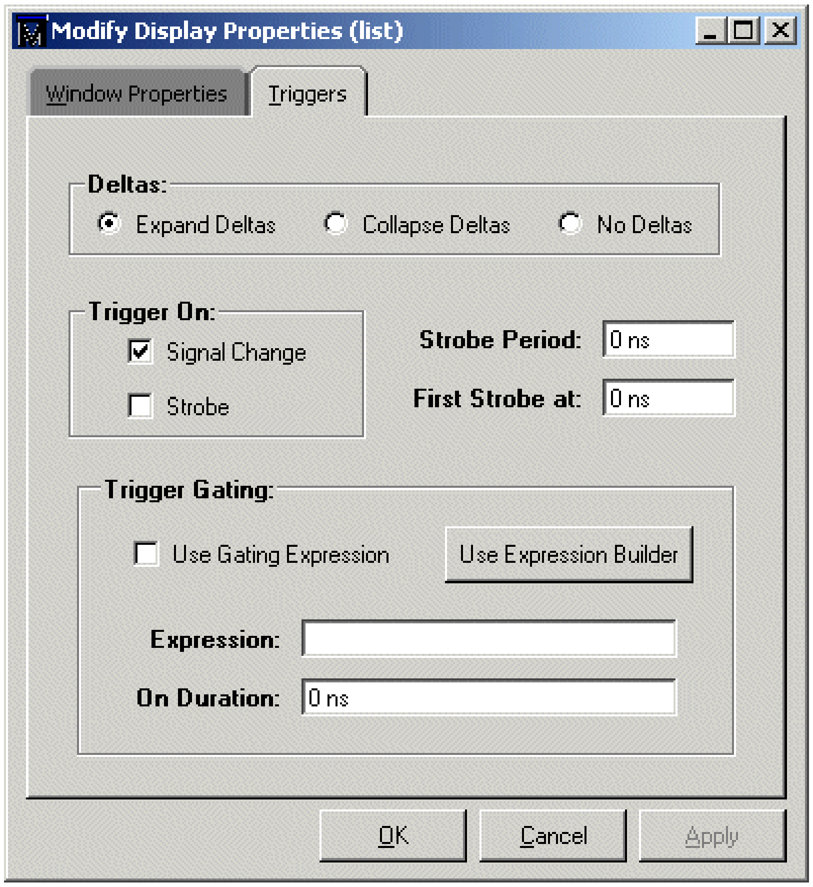
The **Window Properties** tab includes these options:

- **Signal Names**
  Sets the number of path elements to be shown in the List window. For example, "0" shows the full path. "1" shows only the leaf element.
- **Max Title Rows**
  Sets the maximum number of rows in the name pane.
- **Dataset Prefix: Always Show Dataset Prefixes**
  Displays the dataset prefix associated with each signal pathname. Useful for displaying signals from multiple datasets.
- **Dataset Prefix: Show Dataset Prefix if 2 or more**
  Displays dataset prefixes if there are signals in the window from 2 or more datasets.

- **Dataset Prefix: Never Show Dataset Prefixes**
  Turns off display of dataset prefixes.

Triggers tab

The **Triggers** tab controls the triggering for the display of new lines in the List window. You can specify whether an item trigger or a strobe trigger is used to determine when the List window displays a new line. If you choose **Trigger on: Signal Change**, then you can choose between collapsed or expanded delta displays. You can also choose a combination of signal and strobe triggers. To use gating, **Signal Change** or **Strobe** or both must be selected.


Modify Display Properties (list)

Window Properties | Triggers

Deltas: [✓ Expand Deltas] Collapse Deltas No Deltas

Trigger On: [✓ Signal Change] Strobe

Strobe Period: 0 ns

First Strobe at: 0 ns

Trigger Gating: Use Gating Expression

Expression:

On Duration: 0 ns

The **Triggers** tab includes the following options:

- **Expand Deltas**
  When selected with the **Trigger on: Signal Change** check box, displays a new line for each time step on which items change, including deltas within a single unit of time resolution.
- **Collapse Deltas**
  Displays only the final value for each time unit.
- **No Deltas**
  Hides the simulation cycle (delta) column.

- **Trigger On Signal Change**
  Triggers on signal changes. Defaults to all signals. Individual signals can be excluded from triggering by using the **View > Signal Properties** dialog box or by originally adding them with the **-notrigger** option to the **add list** command (CR-55).
- **Trigger On Strobe**
  Triggers on the **Strobe Period** you specify; specify the first strobe with **First Strobe at:**.
- **Use Gating Expression**
  Enables triggers to be gated on (a value of 1) or off (a value of 0) by the specified **Expression**, much like a hardware signal analyzer might be set up to start recording data on a specified setup of address bits and clock edges. Affects the display of data, not the acquisition of the data.
- **Use Expression Builder** (button)
  Opens the Expression Builder to help you write a gating expression. See "The GUI Expression Builder" (UM-395)
- **Expression**
  Enter the expression for trigger gating into this field, or use the Expression Builder (select the **Use Expression Builder** button). The expression is evaluated when the List window would normally have displayed a row of data (given the trigger on signals and strobe settings above).
- **On Duration**
  The duration for gating to remain open after the last list row in which the expression evaluates to true; expressed in x number of default timescale units. Gating is level-sensitive rather than edge-triggered.

List window gating information is saved as configuration statements when the list format is saved. The gating portion of a configuration statement might look like this:

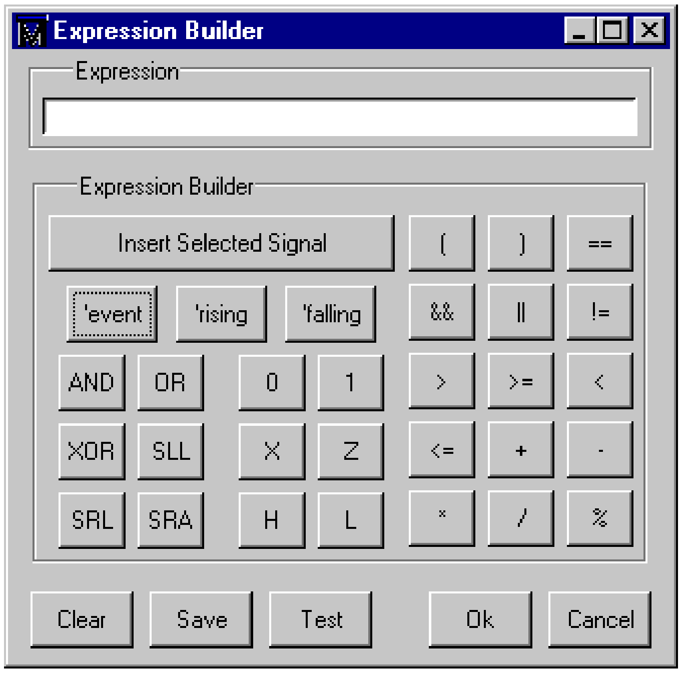
```
configure list -usegating 1
configure list -gateduration 100
configure list -gateexpr {<expression>}
```

Configuring a List trigger with the Expression Builder

This example shows you how to set a List window trigger based on a gating expression created with the ModelSim Expression Builder. Here is the procedure:

1 Select **Tools** > **Window Preferences** to access the Triggers tab.

2 Check the **Use Gating Expression** check box and click **Use Expression Builder**.



3 Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window.

4 Click **Insert Selected Signal** and then **'rising** in the Expression Builder.

5 Click OK to close the Expression Builder.

You should see the name of the signal plus "'rising" added to the Expression entry box of the Modify Display Properties dialog box.

6 Click **OK** to close the dialog.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**.

Sampling signals at a clock change

You can sample signals at a clock change easily using the **add list** command (CR-55) with the **-notrigger** argument. -notrigger disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

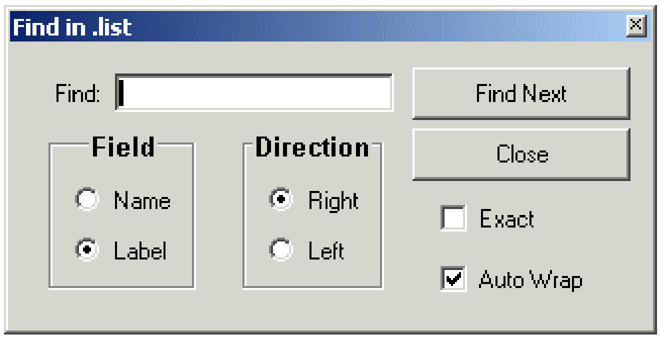
When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

1. Turn off the List window triggering on the clock signal, and then define a repeating strobe for the List window.
2. Define a "gating expression" for the List window that requires the clock to be in a specified state. See "Configuring a List trigger with the Expression Builder" (UM-296).

Finding items by name in the List window

The Find dialog box allows you to search for text strings in the List window. Select **Edit > Find** (List window) to bring up the Find dialog box.



Enter a text string and **Find** it by searching **Right** or **Left** through the List window display.

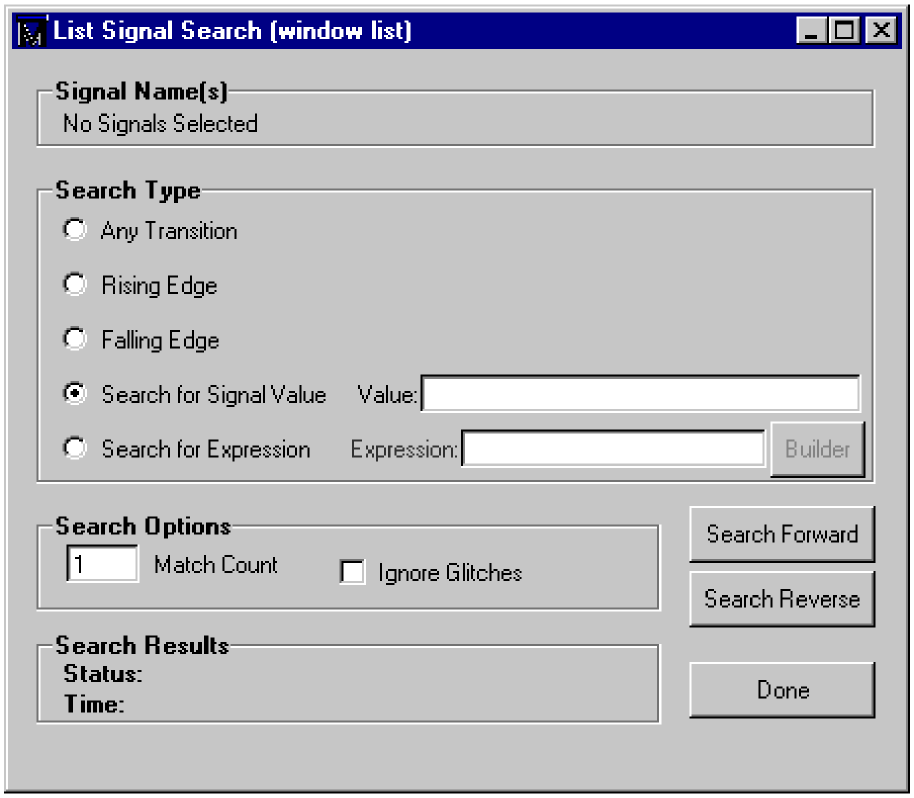
Specify **Name** to search the real pathnames of the items or **Label** to search their assigned names (see "Setting List window display properties" (UM-293)).

Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

Searching for item values in the List window

Select an item in the List window. Select **Edit > Search** (List window) to bring up the List Signal Search dialog box.



Signal Name(s) shows a list of the items currently selected in the List window. These items are the subject of the search. The search is based on these options:

- **Search Type: Any Transition**
  Searches for any transition in the selected signal(s).
- **Search Type: Rising Edge**
  Searches for rising edges in the selected signal(s).
- **Search Type: Falling Edge**
  Searches for falling edges in the selected signal(s).
- **Search Type: Search for Signal Value**
  Searches for the value specified in the **Value** field; the value should be formatted using VHDL or Verilog numbering conventions; see "Numbering conventions" (CR-21).

Note: If your signal values are displayed in binary radix, see "Searching for binary signal values in the GUI" (CR-30) for details on how signal values are mapped between a binary radix and std_logic.

- **Search Type: Search for Expression**
  Searches for the expression specified in the **Expression** field evaluating to a boolean true. Activates the **Builder** button so you can use "The GUI Expression Builder" (UM-395) if desired.

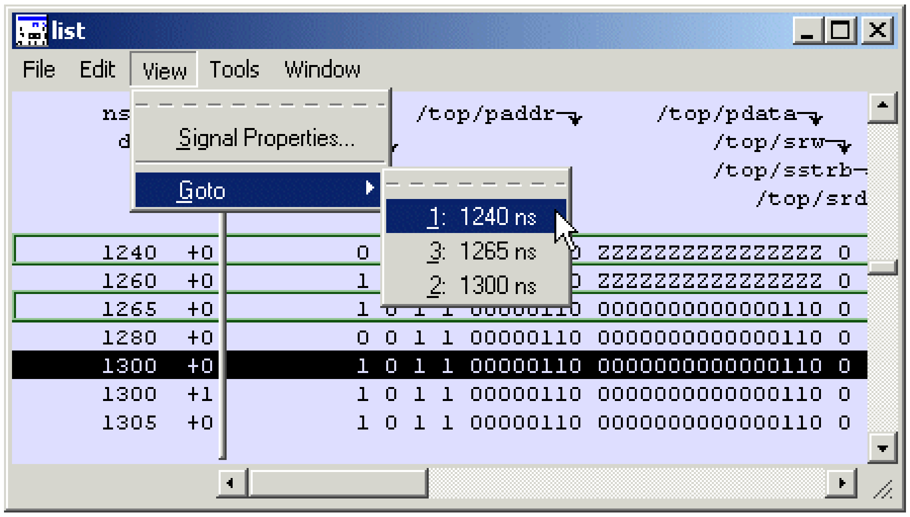
  The expression can involve more than one signal but is limited to signals logged in the List window. Expressions can include constants, variables, and DO files. If no expression is specified, the search will give an error. See "Expression syntax" (CR-24) for more information.
- **Search Options: Match Count**
  Indicates the number of transitions or matches to search.You can search for the nth transition or the nth match on value.
- **Search Options: Ignore Glitches**
  Ignores zero width glitches in VHDL signals and Verilog nets.

The **Search Results** are indicated at the bottom of the dialog box.

Setting time markers in the List window

Select **Edit > Add Marker** (List window) to tag the selected list line with a marker. The marker is indicated by a thin box surrounding the marked line. The selected line uses the same indicator, but its values are highlighted. Delete markers by first selecting the marked line, then selecting **Edit > Delete Marker**.

Finding a marker



```
      ns                    /top/paddr            /top/pdata
       d                                            /top/srw
                                                    /top/sstrb
                                                       /top/srd
    1240   +0          0                0 ZZZZZZZZZZZZZZZZ 0
    1260   +0          1                0 ZZZZZZZZZZZZZZZZ 0
    1265   +0          1 0 1 1 00000110 0000000000000110 0
    1280   +0          0 0 1 1 00000110 0000000000000110 0
    1300   +0          1 0 1 1 00000110 0000000000000110 0
    1300   +1          1 0 1 1 00000110 0000000000000110 0
    1305   +0          1 0 1 1 00000110 0000000000000110 0
```

Choose a specific marked line to view by selecting **View > Goto**. The marker name (on the **Goto** list) corresponds to the simulation time of the selected line.

Saving List window data to a file

Select **File > Write List** (List window) to save the List window data in one of these formats:

- **Tabular**
  writes a text file that looks like the window listing

```
ns   delta      /a       /b      /cin     /sum    /cout
0      +0        X        X        U        X        U
0      +1        0        1        0        X        U
2      +0        0        1        0        X        U
```

- **Events**
  writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI**
  writes a file in standard TSSI format; see also, the **write tssi** command (CR-395)

```
0 00000000000000010?????????
2 00000000000000010???????1?
3 00000000000000010??????010
4 00000000000000010000000010
100 00000001000000010000000010
```

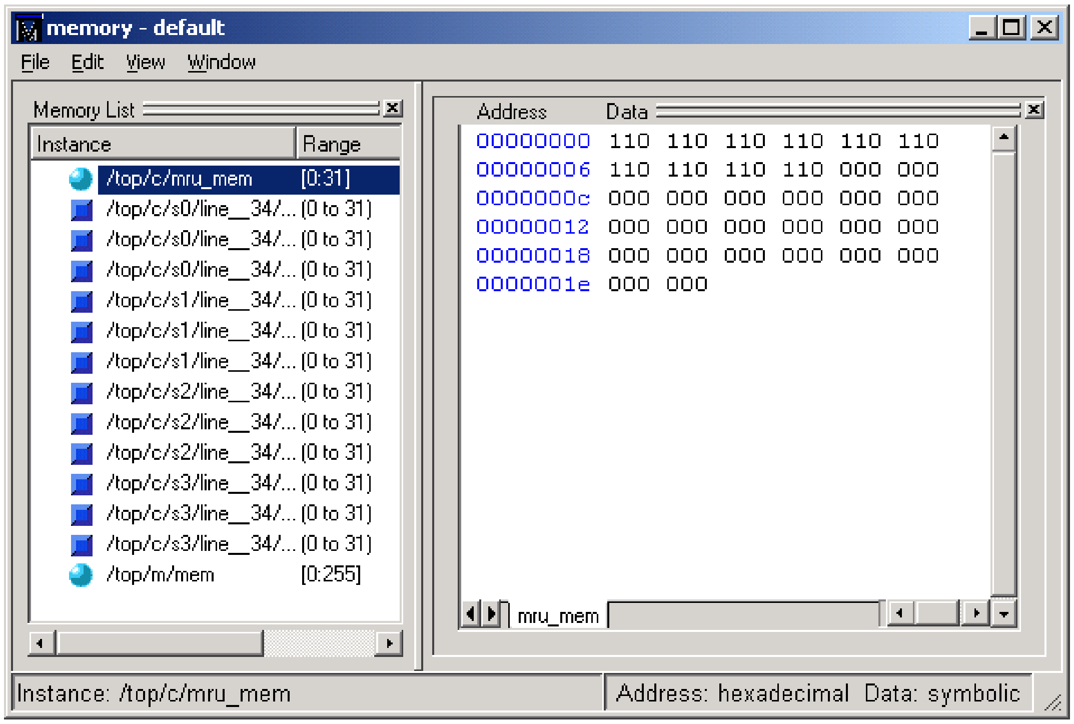
You can also save List window output using the **write list** command (CR-391).

List window keyboard shortcuts

See "List window keyboard shortcuts" (UM-642) .

Memory window

The memory window lists and displays the contents of the memories in your design. The window is divided into two adjustable panes, allowing you to scroll vertically through the memory contents displayed on the right, while keeping the memory list browser visible on the left.



Instance	Range
/top/c/mru_mem	[0:31]
/top/c/s0/line__34/...	[0 to 31]
/top/c/s0/line__34/...	[0 to 31]
/top/c/s0/line__34/...	[0 to 31]
/top/c/s1/line__34/...	[0 to 31]
/top/c/s1/line__34/...	[0 to 31]
/top/c/s1/line__34/...	[0 to 31]
/top/c/s2/line__34/...	[0 to 31]
/top/c/s2/line__34/...	[0 to 31]
/top/c/s2/line__34/...	[0 to 31]
/top/c/s3/line__34/...	[0 to 31]
/top/c/s3/line__34/...	[0 to 31]
/top/c/s3/line__34/...	[0 to 31]
/top/m/mem	[0:255]

```
00000000 110 110 110 110 110 110
00000006 110 110 110 110 000 000
0000000c 000 000 000 000 000 000
00000012 000 000 000 000 000 000
00000018 000 000 000 000 000 000
0000001e 000 000
```

Instance: /top/c/mru_mem   Address: hexadecimal Data: symbolic

Memories you can view

The memory browser identifies and lists the following types of arrays as memories:

- reg, wire, bit, and std_logic arrays

  Any signal or variable that is an array of two dimensions (including arrays of arrays) are identified as memories and listed if the base type is a Verilog reg or wire type, or a VHDL enumerated type with values in std_ulogic, bit, and all related sub-types.

- Integer arrays

  Single dimensional arrays of integers are interpreted as 2D memory arrays. In these cases, the word width listed in the Memory List pane is equal to the integer size, and the depth is the size of the array itself. The appearance of this type of array in the memory list can be disabled via the View menu or the ShowIntMem (UM-625) variable in the *modelsim.ini*.

- Single dimensional arrays of VHDL enumerated types other than std_logic or bit

  These enumerated type value sets must have values that are longer than one character. The listed width is the number of entries in the enumerated type definition and the depth is the size of the array itself. The appearance of this type of array in the memory list can be disabled via the View menu or the ShowEnumMem (UM-625) variable in the *modelsim.ini*.

- 3D or greater arrays

  Memories with three or more dimensions display with a plus sign '+' next to their names in the Memory List. Click the '+' to show the array indices under that level. When you finally expand down to the 2D level, you can click on the index, and the data for the selected 2D slice of the memory will appear in the memory contents pane.The appearance of this type of array in the memory list can be disabled via the View menu or the Show3DMem (UM-625) variable in the *modelsim.ini*.

The Memory window menu bar

This section provides information on select menu commands available in the Memory window. Several commands are also available on a context menu by right-clicking within the content or address pane.

File menu

Load	load memory data to the currently displayed memory instance from a file; see "Loading files and patterns" (UM-309)
Save	save currently displayed memory data (all or a range) to a file; see "Saving memory data to a file" (UM-312)
Environment	set the environment of the memory being viewed either to follow context selection or to the current context

Edit menu

Goto	go to specific memory address in currently displayed memory instance; see "Using the Goto dialog" (UM-307)
Change	change the memory contents for all addresses or a range of addresses in the currently displayed memory instance; see "Interactive memory initialization" (UM-311)
Find	find the specified text string within the Memory window; see "Finding a memory instance" (UM-308)
Data Search	searches for a specified memory data pattern in the currently displayed memory instance; see "Searching for a data pattern" (UM-307)

View menu

Memory Declaration	open up the Source window to the line of code where the currently displayed memory instance is defined
Split Screen	split the address pane horizontally into two identically-sized panes, one upper and one lower; see "Splitting the Data Screen" (UM-306).
Memory List	toggle on and off the display of the Memory List pane
ShowIntMem	toggle on and off the display in the Memory List pane of single dimensional arrays of integers
ShowEnumMem	toggle on and off the display in the Memory List pane of single dimensional arrays of VHDL enumerated types other than std_logic or bit
Show3DMem	toggle on and off the display in the Memory List pane of arrays of 3 or more dimensions
Display Options	set various window display options; see "Modifying the memory window display" (UM-305).

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Viewing memory contents

To bring up a Memory window, either select **View > Memory** from the menu bar, or enter **view memory** at the command prompt. Multiple memory windows can be viewed simultaneously.

Selecting memory instances

To select a memory instance for viewing its contents, you can:

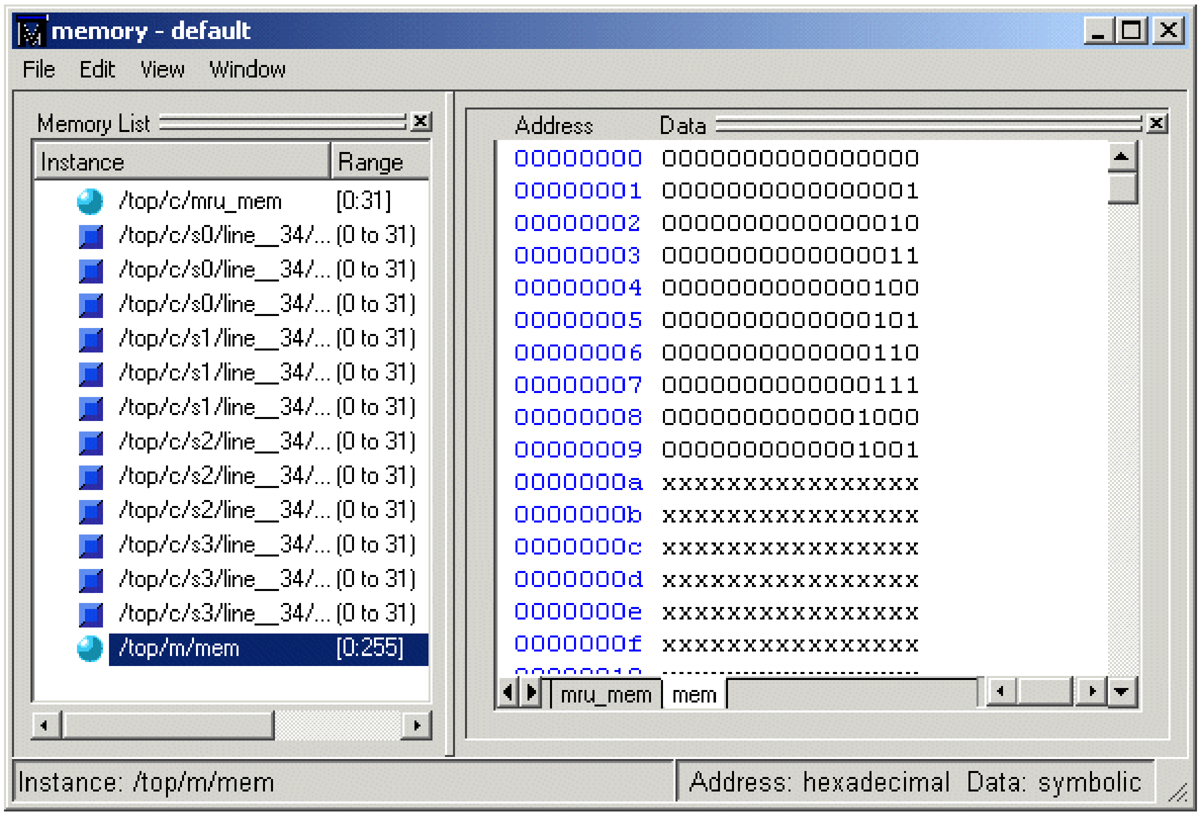
- Click on one of the memory instances appearing in the Memory List pane.
- Drag and drop any instance shown in the other ModelSim windows, such as Structure or Wave into the **Address / Data** pane. All memory instances in that level of hierarchy are displayed.
- Enter the command **add mem <instance>** at the vsim command prompt.

Viewing multiple memory instances

You can view multiple memory instances. A tab appears at the bottom of the Address / Data pane corresponding to each memory instance that is added to the view, as shown below. To close one instance or all instances, select **File > Close Instance** or **Close All**, respectively.

Modifying the memory window display

When you have added memory instances to the view, the Memory window appears as follows:



Instance	Range
/top/c/mru_mem	[0:31]
/top/c/s0/line__34/...	[0 to 31]
/top/c/s0/line__34/...	[0 to 31]
/top/c/s0/line__34/...	[0 to 31]
/top/c/s1/line__34/...	[0 to 31]
/top/c/s1/line__34/...	[0 to 31]
/top/c/s1/line__34/...	[0 to 31]
/top/c/s2/line__34/...	[0 to 31]
/top/c/s2/line__34/...	[0 to 31]
/top/c/s2/line__34/...	[0 to 31]
/top/c/s3/line__34/...	[0 to 31]
/top/c/s3/line__34/...	[0 to 31]
/top/c/s3/line__34/...	[0 to 31]
/top/m/mem	[0:255]

```
00000000 0000000000000000
00000001 0000000000000001
00000002 0000000000000010
00000003 0000000000000011
00000004 0000000000000100
00000005 0000000000000101
00000006 0000000000000110
00000007 0000000000000111
00000008 0000000000001000
00000009 0000000000001001
0000000a xxxxxxxxxxxxxxxx
0000000b xxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxx
0000000d xxxxxxxxxxxxxxxx
0000000e xxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxx
```

Instance: /top/m/mem   Address: hexadecimal Data: symbolic

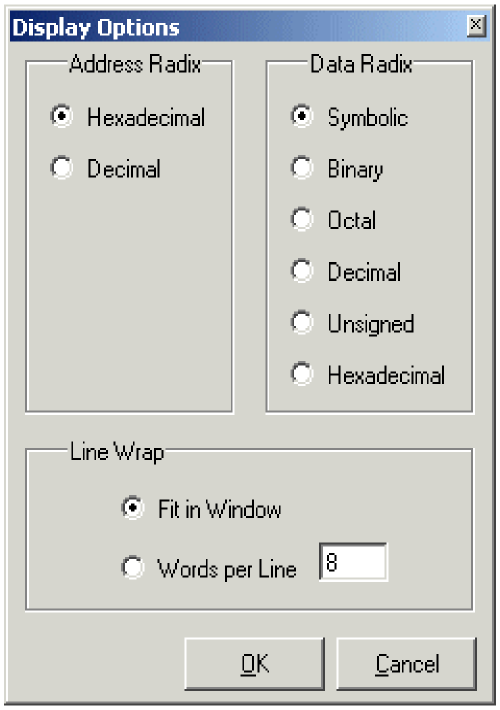
The display can be modified by setting different display options and splitting the Data / Address pane.

Setting display options

To change the display's address and data radix, or line wrapping of the selected memory, select **View > Display Options**. You can also right-click anywhere in the in the Address/ Data pane to bring up a pop-up menu containing **Display Options**.

The **Display Options** dialog box includes these options:

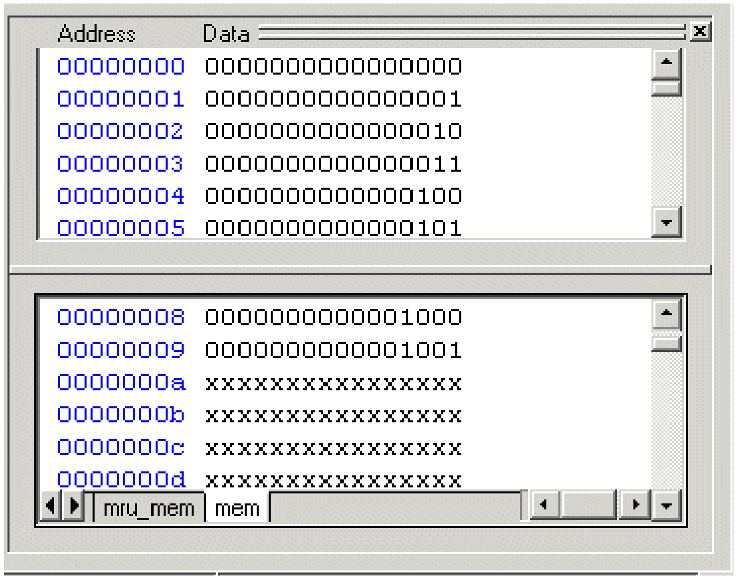
- **Address Radix**
  The radix for the address. Can be Hexadecimal or Decimal.
- **Data Radix**
  The radix for the data. Non-enumerated type memories can be Symbolic, Binary, Octal, Decimal, Unsigned, and Hexadecimal. Enumerated type memories are only symbolic data types, and all other options are grayed out.
- **Line Wrap**
  The number of words per line can be set, or arbitrarily determined based on the size of the window.



Address Radix: [✓ Hexadecimal] Decimal
Data Radix: [✓ Symbolic] Binary Octal Decimal Unsigned Hexadecimal
Line Wrap: [✓ Fit in Window] Words per Line 8

Splitting the Data Screen

To split the Address / Data pane into two screens displaying the contents of a single memory instance, select **View > Split Screen** (or right-click in the pane and select **Split Screen** from the pop-up menu). This allows you to view different address locations within the same memory instance simultaneously.

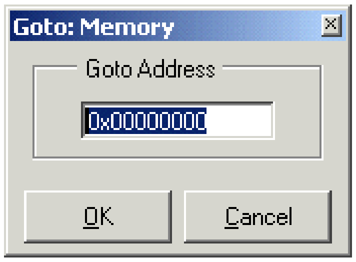


```
Address   Data
00000000  0000000000000000
00000001  0000000000000001
00000002  0000000000000010
00000003  0000000000000011
00000004  0000000000000100
00000005  0000000000000101

00000008  0000000000001000
00000009  0000000000001001
0000000a  xxxxxxxxxxxxxxxx
0000000b  xxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxx
0000000d  xxxxxxxxxxxxxxxx
```

Navigating to memory locations within a memory instance

Other than using the scroll bar to scroll up and down through the memory, you can navigate to specific memory locations within an instance in several ways.

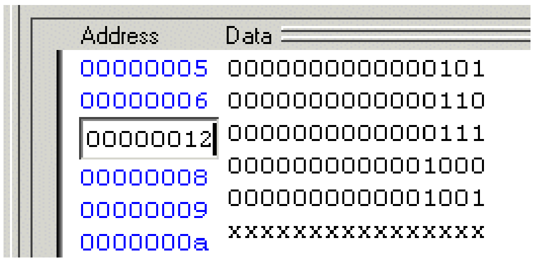
Using the Goto dialog



0x00000000

Select **Edit > Goto** to bring up the Goto Memory dialog. You can also right-click on the address column in the Address / Data display area, and select **Goto** from the pop-up menu. When selected, it brings up the Goto dialog box, shown here. Enter the desired address location into the field, select **OK**, and the data view shifts to display the data in that location.

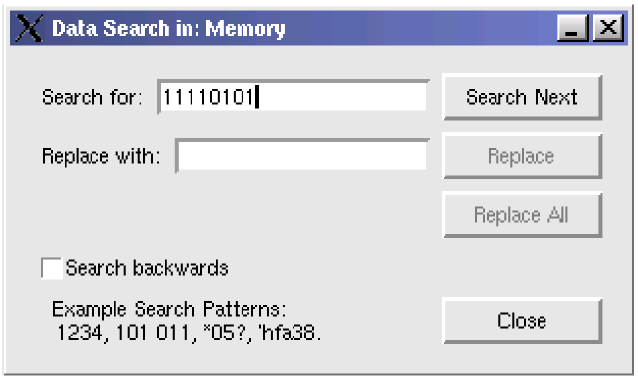
Direct address navigation



```
Address    Data
00000005   0000000000000101
00000006   0000000000000110
00000012   0000000000000111
00000008   0000000000001000
00000009   0000000000001001
0000000a   xxxxxxxxxxxxxxxx
```

You can navigate to any address location directly by editing the address in the address column. Double-click on any address, type in the desired address, and hit **Enter**. The address display scrolls to the specified location.

Searching for a data pattern



Search for: 11110101

Replace with:

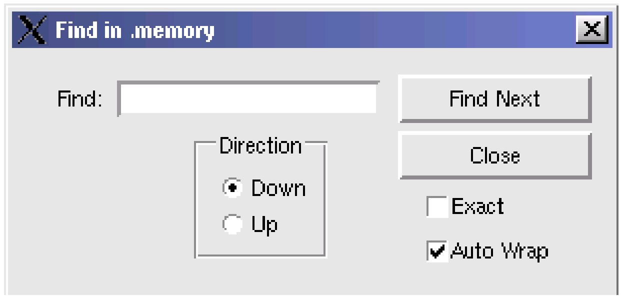
Search backwards

Example Search Patterns:
1234, 101 011, *05?, 'hfa38.

To find a particular data pattern, select **Edit > Data Search** or right-click in the data area of the pane, and select **Data Search**. The Data Search in Memory dialog box appears as shown here. Specify the pattern you want to find in the Search for: field and, optionally, a replacement pattern in the a the Replace with: field. The Search Next button performs the search and replace operation. Select Search backward to search and/or replace backward through the memory for the specified pattern. Select Close to close the Data Search dialog box.

Finding a memory instance

To find a particular data pattern, select **Edit > Find** or right-click in the data area of the pane, and select **Find**. The Find dialog box appears as shown here, containing a search pattern definition field and a Find Next button.



Select Exact match to search for patterns exactly matching the specified pattern. Select Search backward to search backward through the memory for the specified pattern. Select Close to close the Find dialog box.

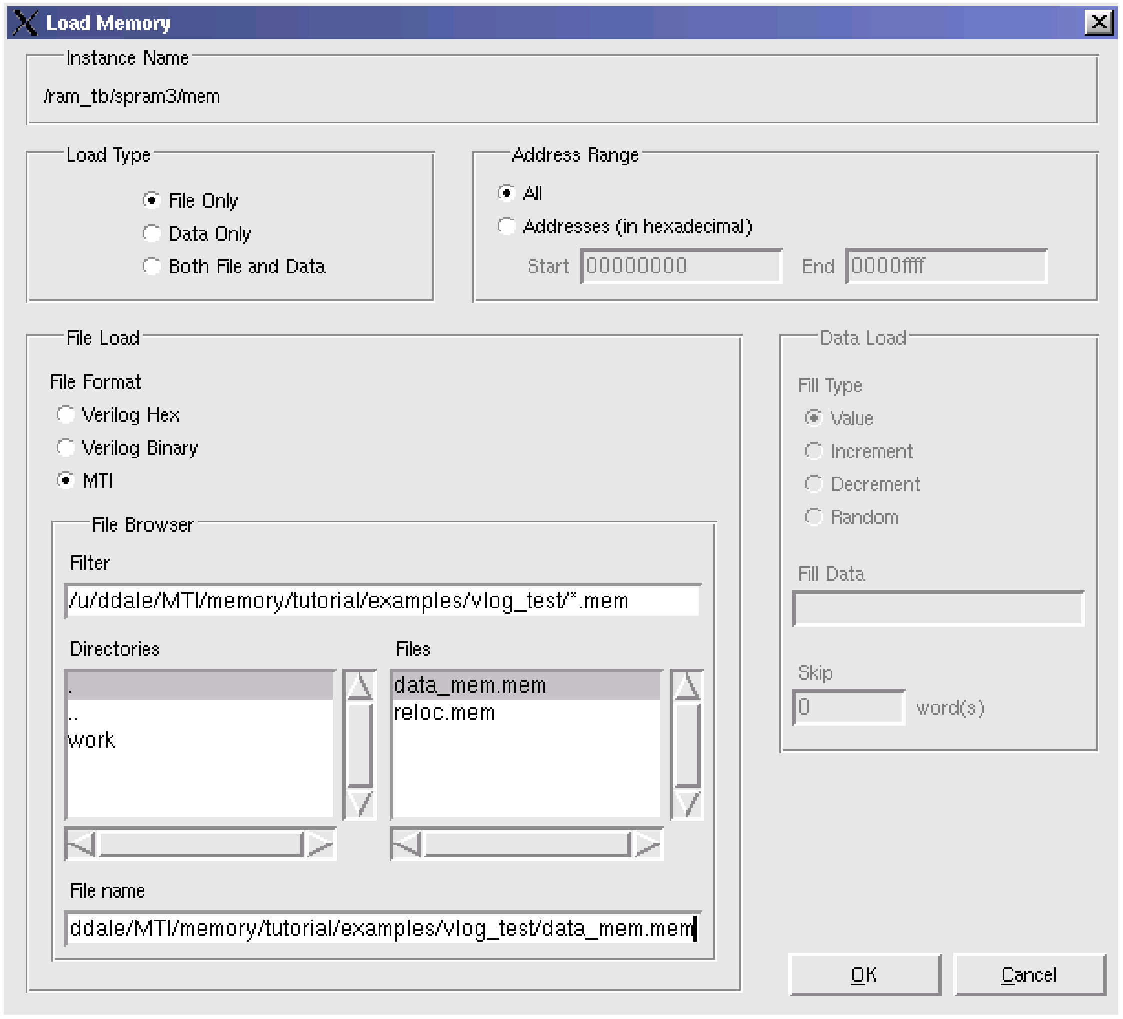
Initializing memories

You can initialize memories in your design by either loading the contents from a file, or by an interactive command. An entire memory, a specific range of addresses, or an individual word can be overwritten. Choose the type of Load operation to be performed in the Load Type area. The default load type is File Only. When either File Only or Data Only is selected, the unused section of the dialog box is grayed out.

Loading files and patterns

To initialize a memory from a file:

Select **File** > **Load**. The Load Memory dialog box appears, as follows:



Load Memory

Instance Name
/ram_tb/spram3/mem

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

Address Range
[✓ All] Addresses (in hexadecimal)
Start 00000000 End 0000ffff

File Load
File Format
Verilog Hex Verilog Binary [✓ MTI]

File Browser
Filter
/u/ddale/MTI/memory/tutorial/examples/vlog_test/*.mem

Directories
.
..
work

Files
data_mem.mem
reloc.mem

File name
ddale/MTI/memory/tutorial/examples/vlog_test/data_mem.mem

Data Load
Fill Type
[✓ Value] Increment Decrement Random
Fill Data

Skip
0 word(s)

OK Cancel

The Load Memory dialog box includes these options:

- **Instance Name**
  Displays the name of the memory instance being loaded.

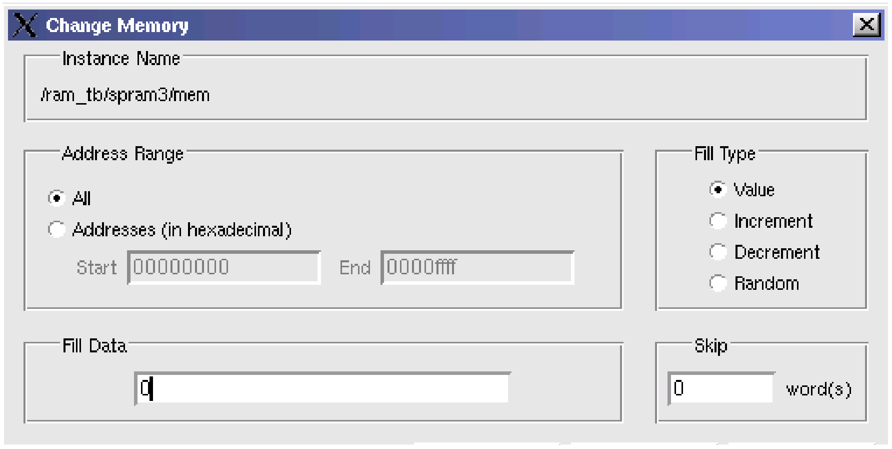
- **Load Type**
  Defines the type of load function you will perform. Your choices for loading data are: File Only, Data Only or Both File and Data.
- **Address Range**
  Specifies all addresses or a range of addresses in the memory that you want to load. The address radix of the displayed memory is shown in parentheses.
- **File Load**
  Contains all inputs related to loading from a file. This whole area of the dialog is grayed out if Load Type is specified as Data Only.
- **File Format**
  Specifies the format of the file to be loaded. Verilog Hex, Verilog Binary, or MTI format can be explicitly set, or the format can be determined automatically from the file (if the file was created with the **mem save** command).
- **Filter**
  Filters the file list.
- **File name**
  The name of the memory file to load. You can manually edit this field, or select a file from the Files list, and it will fill in automatically.
- **Data Load**
  Contains all inputs related to loading memory data. This area of the dialog is grayed out if Load Type is specified as File Only.
- **Fill Data**
  Specifies the fill data for addresses not contained in the load file.
- **Fill Type**
  Specifies how to apply the fill data, either directly as a value, or algorithmically. See the **mem load** command (CR-195) for more information on Fill Type and Fill Data.
- **Skip**
  Specifies the number of words to skip when applying a fill pattern sequence.

Interactive memory initialization

Memory contents can be modified interactively during simulation for greater ease in debugging your design. You can change the data values in multiple addresses in the memory by using the Change Memory dialog, or change individual data values by editing them directly in the data area of the Address / Data pane.

Changing data for multiple addresses

Select **Edit > Change** to open the Change Memory dialog box.



Change Memory

Instance Name: /ram_tb/spram3/mem

Address Range: [✓ All] Addresses (in hexadecimal)
Start: 00000000 End: 0000ffff

Fill Type: [✓ Value] Increment Decrement Random

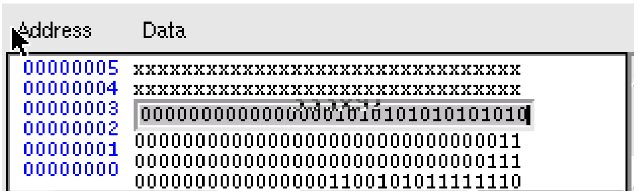
Fill Data: 0

Skip: 0 word(s)

The Change Memory dialog box includes the following:

- **Instance Name**
  Displays the name of the memory instance being loaded.
- **Address Range**
  Specifies all addresses or a starting and ending address to be changed. The address radix of the currently displayed memory is shown in parentheses.
- **Fill Data**
  Specifies the fill data for specified addresses.
- **Fill Type**
  Specifies how to apply the fill data, either directly as a value, or algorithmically. See the **mem load** command (CR-195) for more information on Fill Type and Fill Data.
- **Skip**
  Specifies the number of words to skip after applying a fill pattern sequence.

Changing data for individual addresses

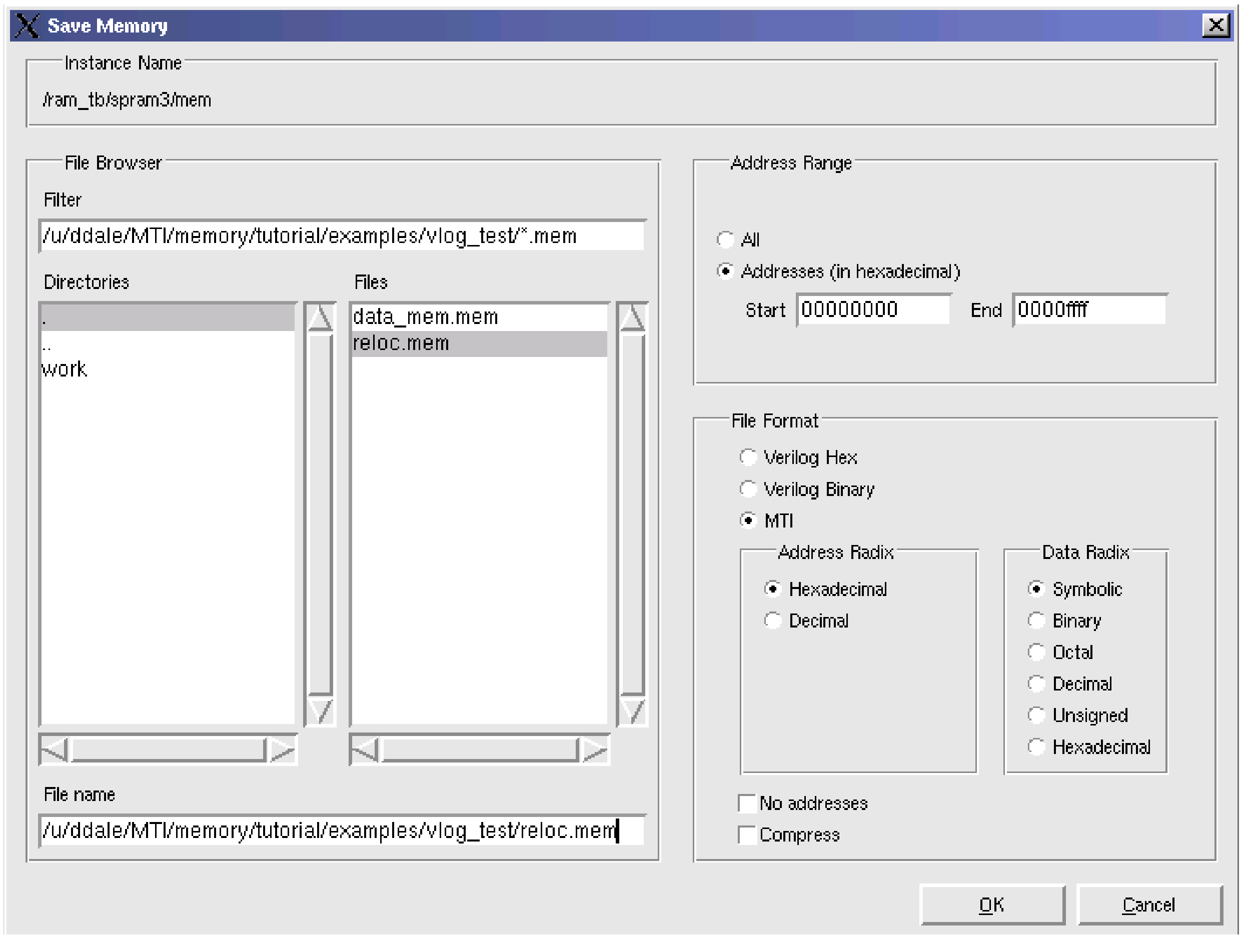


```
Address     Data
00000005    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000004    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003    00000000000000001010101010101010
00000002    00000000000000000000000000000011
00000001    00000000000000000000000000000111
00000000    00000000000000001100101011111110
```

To edit memory data in place, **Double click** (or right-click and select **Edit in Place)** on any word in the Address/Data pane of the Memory window. The data is highlighted. Type in the desired change. Pressing <Enter> commits the change; <Esc> aborts it. <Tab> scrolls down the list of data entries, while <Shift>-<Tab> scrolls up the list. As a short-cut, after editing one data value, you can double-click on another data value to commit the change and edit the second value.

Saving memory data to a file

To save the current memory data to a file, select **File > Save**.



Instance Name: /ram_tb/spram3/mem

Filter: /u/ddale/MTI/memory/tutorial/examples/vlog_test/*.mem

Directories: . .. work

Files: data_mem.mem reloc.mem

File name: /u/ddale/MTI/memory/tutorial/examples/vlog_test/reloc.mem

Address Range: All [✓ Addresses (in hexadecimal)] Start 00000000 End 0000ffff

File Format: Verilog Hex, Verilog Binary, [✓ MTI]

Address Radix: [✓ Hexadecimal] Decimal

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

No addresses

Compress

The Save Memory File dialog box includes the following:

- **File name**
  Name of file to be saved.
- **Address Range**
  Specifies all or a range of addresses to be saved into the file. The address radix
- **File Format**
  Specifies whether memory is to be saved in Verilog Hex, Verilog Binary, or MTI format. Also, specify the Address and Data radix for MTI format.
- **No addresses**
  Specifies that no addresses are to appear in the saved file. This enables the file to be reloaded anywhere in the memory.

- **Compress**
  Applies a simple ASCII compression to the saved file. The compression algorithm replaces repeating lines with a single asterisk, like is done with the Unix "od" command.

MTI memory data file format

The MTI memory data file format is as illustrated in the following example:

```
// memory data file
// (do not edit the following line - required for mem load use)
// format=mti addressradix=d dataradix=s direction=ascending
 0: 110 110 110 110 110 110
 6: 110 110 110 110 000 000
12: 000 000 000 000 000 000
18: 000 000 000 000 000 000
24: 000 000 000 000 000 000
30: 000 000
```

The possible format, address radix, data radix, and direction settings are as specified by the corresponding options in the **mem save** and **mem load** commands. See the **mem save** command (CR-198) and the **mem load** command (CR-195) for more information.

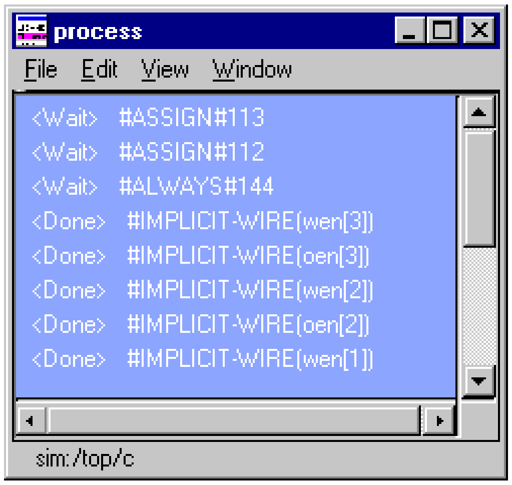
Process window

The Process window displays a list of processes, and SystemC method and thread processes. In ModelSim versions 5.7 and later, the information contained in the Process window can also be displayed in the Main window Workspace (UM-263).

If **View > Active** is selected then all processes, and SystemC methods and thread processes scheduled to run during the current simulation cycle are displayed along with the pathname of the instance in which each process is located. If **View > In Region** is selected then only the processes in the currently selected region are displayed.

Understanding process status

Each item in the scrollbox is preceded by one of the following indicators:



```
<Wait>  #ASSIGN#113
<Wait>  #ASSIGN#112
<Wait>  #ALWAYS#144
<Done>  #IMPLICIT-WIRE(wen[3])
<Done>  #IMPLICIT-WIRE(oen[3])
<Done>  #IMPLICIT-WIRE(wen[2])
<Done>  #IMPLICIT-WIRE(oen[2])
<Done>  #IMPLICIT-WIRE(wen[1])
sim:/top/c
```

- **<Ready>**
  Indicates that the process is scheduled to be executed within the current delta time.
- **<Wait>**
  Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period. SystemC items cannot be in a Wait state.
- **<Done>**
  Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run. SystemC items cannot be in a Done state.

If you select a "Ready" process, it will be executed next by the simulator.

Links to other windows

When you click on a process in the Process window, the following windows are updated:

Window updated	Result
Dataflow window (UM-270)	highlights the selected process
Memory window (UM-302)	shows the memory instances in that process
Signals window (UM-316)	shows the signals in the region in which the process is located
Source window (UM-325)	shows the associated source code
Structure window (UM-331)	shows the region in which the process is located

Window updated	**Result**
Variables window (UM-334)	shows the VHDL variables and Verilog registers and variables in the process

The Process window menu bar

This section provides information on select menu commands available in the Process window.

File menu

Save List	save the process tree to a text file
Environment	**Follow Context Selection**: update the window based on the selection in the Structure window (UM-331); **Fix to Current Context**: maintain the current view, do not update

View menu

Active	display all the processes that are scheduled to run during the current simulation cycle
In Region	display any processes that exist in the region that is selected in the Structure window
Sort	sort the process list in either ascending, descending, or declaration order

Window menu

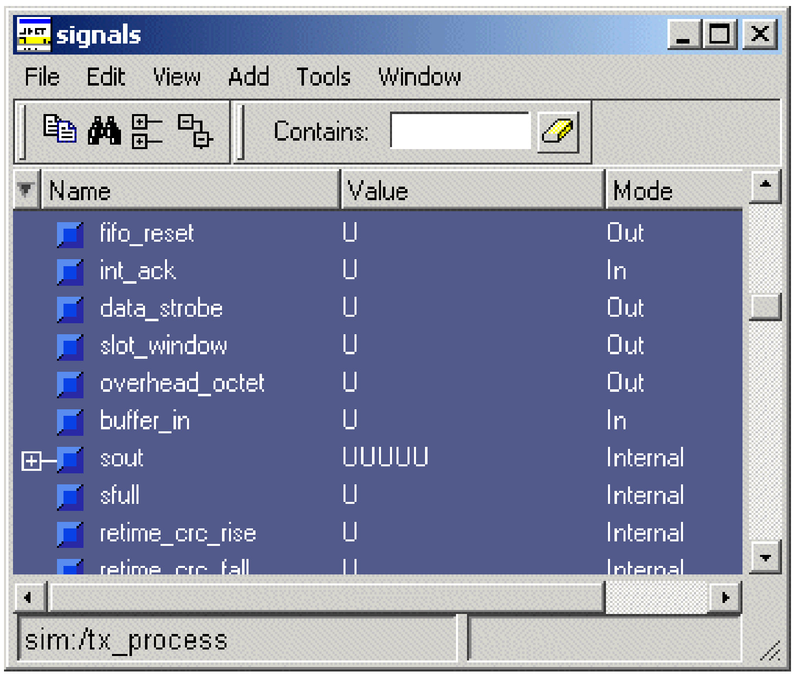
The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Signals window

The Signals window shows the names and current values of items in the current region (which is selected in the Structure window). The data in this pane is similar to that shown in the Wave window (UM-337), except that the values do not change dynamically with movement of the selected Wave window cursor.

Clicking on a signal name in the Signals window highlights that signal in the Dataflow and Wave windows. Double-clicking a signal highlights that signal in the Source window (opening a Source window if one is not open already). You can also right click a signal name and add it to the List or Wave window, or the current log file.

The items can be sorted in ascending, descending, or declaration order.



Name	Value	Mode
fifo_reset	U	Out
int_ack	U	In
data_strobe	U	Out
slot_window	U	Out
overhead_octet	U	Out
buffer_in	U	In
sout	UUUUU	Internal
sfull	U	Internal
retime_crc_rise	U	Internal
retime_crc_fall	U	Internal

sim:/tx_process

Items you can view

One entry is created for each of the following items in the design:

VHDL items

signals, aliases, generics, shared variables

Verilog items

nets, registers, variables, named events, and module parameters

SystemC items

primitive channels and ports

Virtual items

virtual signals and virtual functions; see "Virtual signals" (UM-248) for more information

VHDL composite types (arrays and record types) and Verilog vector nets, vector registers, and memories are shown in a hierarchical fashion. ModelSim indicates hierarchy with plus (expandable), minus (expanded), and blank (single level) boxes. See "Tree window hierarchical view" (UM-261) for more information.

The Signals window menu bar

This section provides information on select menu commands available in the Signals window. Several commands are also available on a context menu by right-clicking on a signal name.

File menu

Save List	save the signals tree to a text file
Environment	allow the window contents to change based on the current environment, or fix to a specific context or dataset
Close	close this copy of the Signals window; you can create a new window with **File** > **New** > **Window** from the "The Main window menu bar" (UM-265)

Edit menu

Force	apply stimulus to the specified signal; see "Forcing signal and net values" (UM-321)
Noforce	remove the effect of an active force
Clock	define clock signals; see"Defining clock signals in HDL designs" (UM-323)

View menu

Signal Declaration	open the source file in the Source window and highlight the signal declaration
Sort	sort the signals tree in either ascending, descending, or declaration order
Filter	choose the port and signal types to view; see "Filtering by signal type" (UM-319)

Add menu

Allows you to add the specified signals to the Wave or List windows or the current WLF file

Tools menu

Breakpoints	open the Breakpoints dialog; see "Creating and managing breakpoints" (UM-391)
Toggle Coverage	add or reset toggle coverage; see *Chapter 12 - Code Coverage* for details

Window menu

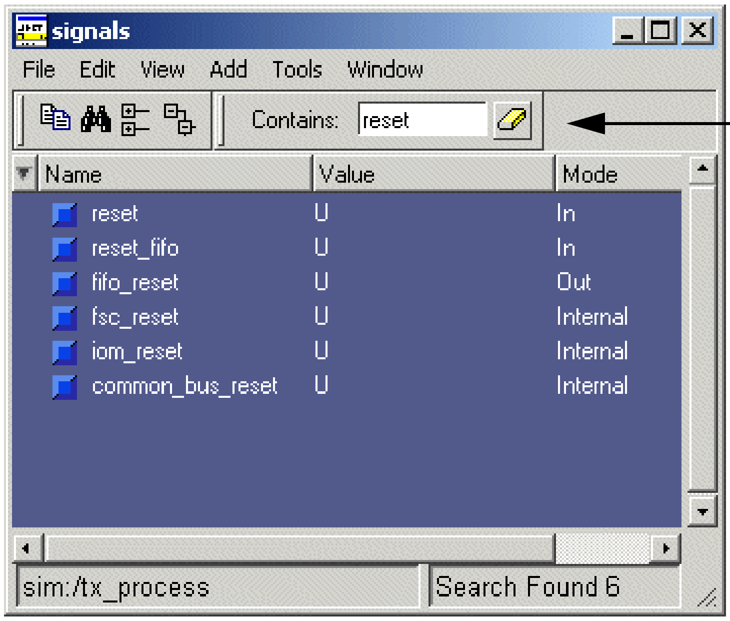
The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Filtering the signal list

You can filter the signal list by name or by signal type.

Filtering by name

To filter by name, start typing letters in the **Contains** field on the toolbar. As you type letters, the signals list filters to show only those signals that contain those letters.



Name	Value	Mode
reset	U	In
reset_fifo	U	In
fifo_reset	U	Out
fsc_reset	U	Internal
iom_reset	U	Internal
common_bus_reset	U	Internal

sim:/tx_process Search Found 6

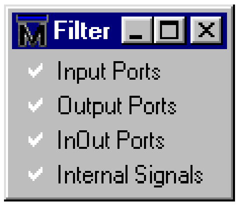
The signals list filters dynamically as you type letters in the Contains: field. Click the eraser icon to clear the field.

To display all signals again, click the Eraser icon to clear the entry.

Filters are stored relative to the region selected in the Structure window. If you re-select a region that had a filter applied, that filter is restored. This allows you to apply different filters to different regions.

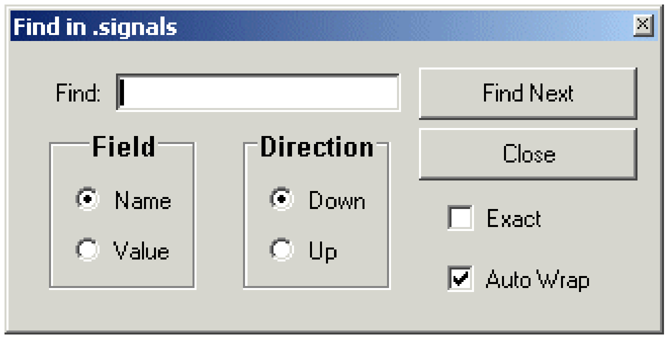
Filtering by signal type

The **View** > **Filter** menu selection allows you to specify which signal types to display in the Signals window. Multiple options can be selected.



Finding items in the Signals window

To find the specified text string within the Signals window, choose the **Name** or **Value** field to search and the search direction: **Down** or **Up**.



Field: [✓ Name] Value

Direction: [✓ Down] Up

Exact ☐

Auto Wrap ✓

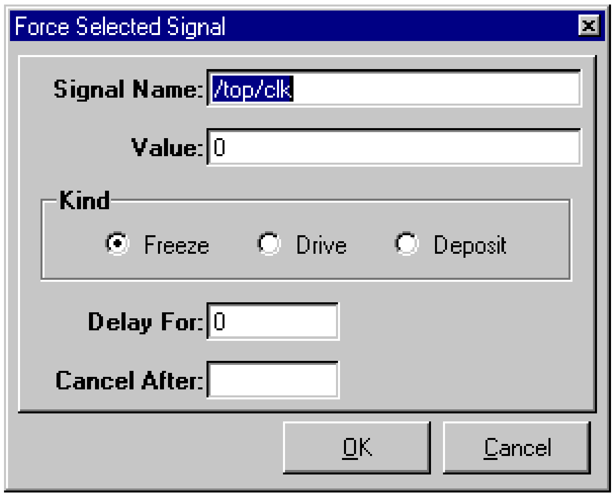
Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

You can also do a quick find from the keyboard. When the Signals window is active, each time you type a letter the signal selector (highlight) will move to the next signal whose name begins with that letter.

Forcing signal and net values

The **Edit > Force** command (unavailable for SystemC) displays a dialog box that allows you to apply stimulus to the selected signal or net. Multiple signals can be selected and forced; the force dialog box remains open until all of the signals are either forced, skipped, or you close the dialog box. To cancel a force command, use the **Edit > NoForce** command. See also the **force** command (CR-176).



Signal Name: /top/clk
Value: 0
Kind: [✓ Freeze] Drive Deposit
Delay For: 0
Cancel After:

The **Force** dialog box includes these options:

- **Signal Name**
  Specifies the signal or net for the applied stimulus.
- **Value**
  Initially displays the current value, which can be changed by entering a new value into the field. A value can be specified in radixes other than decimal by using the form (for VHDL and Verilog, respectively):

  ```
  base#value -or- b|o|d|h'value
  ```

  16#EE or h'EE, for example, specifies the hexadecimal value EE.
- **Kind: Freeze**
  Freezes the signal or net at the specified value until it is forced again or until it is unforced with a **noforce** command (CR-204).

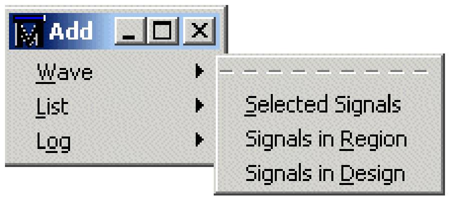
  Freeze is the default for Verilog nets and unresolved VHDL signals and **Drive** is the default for resolved signals.

  If you prefer **Freeze** as the default for resolved and unresolved signals, you can change the default force kind in the *modelsim.ini* file; see *Appendix A - ModelSim variables*.
- **Kind: Drive**
  Attaches a driver to the signal and drives the specified value until the signal or net is forced again or until it is unforced with a **noforce** command (CR-204). This type of force is illegal for unresolved VHDL signals.
- **Kind: Deposit**
  Sets the signal or net to the specified value. The value remains until there is a subsequent driver transaction, or until the signal or net is forced again, or until it is unforced with a **noforce** command (CR-204).

- **Delay For**
  Allows you to specify how many time units from the current time the stimulus is to be applied.
- **Cancel After**
  Cancels the **force** command (CR-176) after the specified period of simulation time.
- **OK**
  When you click the OK button, a **force** command (CR-176) is issued with the parameters you have set, and is echoed in the Main window. If more than one signal is selected to force, the next signal down appears in the dialog box each time the OK button is selected. Unique force parameters can be set for each signal.

Adding items to the Wave and List windows or a WLF file

Use the **Add** menu to add items from the Signals window to the Wave window (UM-337), List window (UM-286), or log file (WLF file). You can also access these same commands by right-clicking a signal in the window.



The WLF file is written as an archive file in binary format and is used to drive the List and Wave windows at a later time. Once signals are added to the WLF file they cannot be removed (though you can turn off logging with the **nolog** command (CR-205)). If you begin a simulation by invoking **vsim** (CR-357) with the **-view <WLF_fileame>** argument, ModelSim reads the WLF file to drive the Wave and List windows.

Choose one of the following options from the **Add** sub-menus:

- **Selected Signals**
  Adds only the item(s) selected in the Signals window.
- **Signals in Region**
  Adds all items in the region that is selected in the Structure window.
- **Signals in Design**
  Adds all items in the design.

Adding items from the Main window command line

Another way to add items to the Wave or List window or the WLF file is to enter one of the following commands at the VSIM prompt (choose either the **add list** (CR-55), **add wave** (CR-64), or **log** (CR-187) command):

```
add list | add wave | log <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list | add wave | log *
```

If the target window (Wave or List) is closed, ModelSim opens it when you when you invoke the command.

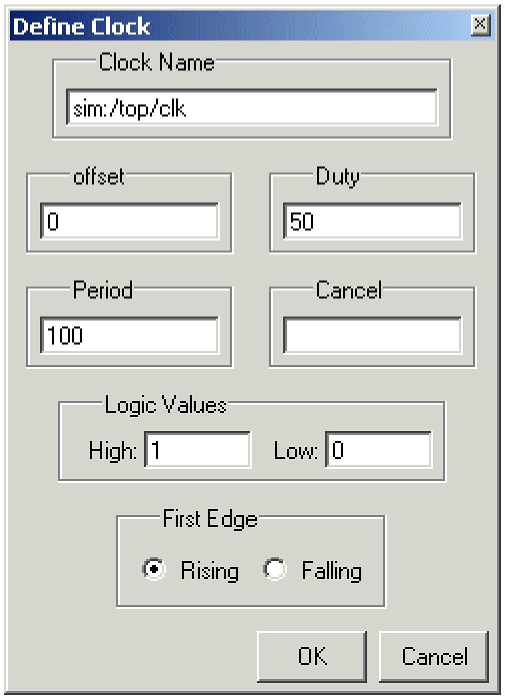
Setting signal breakpoints in HDL designs

You can set "Signal breakpoints" (UM-391) in the Signals window. When a signal breakpoint is hit, a message appears in the Main window Transcript stating which signal caused the breakpoint.

To insert a signal breakpoint, right-click a signal name and select **Insert Breakpoint**. See "Creating and managing breakpoints" (UM-391) for more information.

Defining clock signals in HDL designs

Select **Edit > Clock** to define clock signals by Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling. You can also specify a simulation period after which the clock definition should be cancelled.


Define Clock

Clock Name: sim:/top/clk

offset: 0

Duty: 50

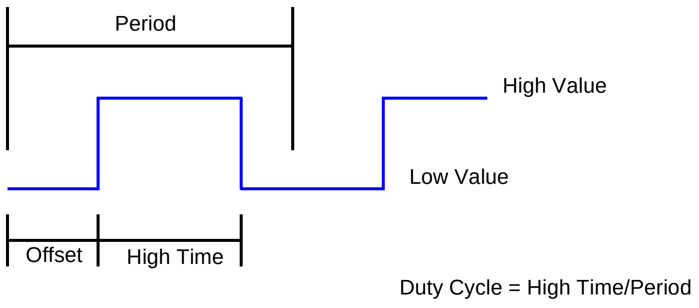
Period: 100

Cancel:

Logic Values: High: 1 Low: 0

First Edge: [✓ Rising] Falling

For clock signals starting on the rising edge, the definition for Period, Offset, and Duty Cycle is as follows:



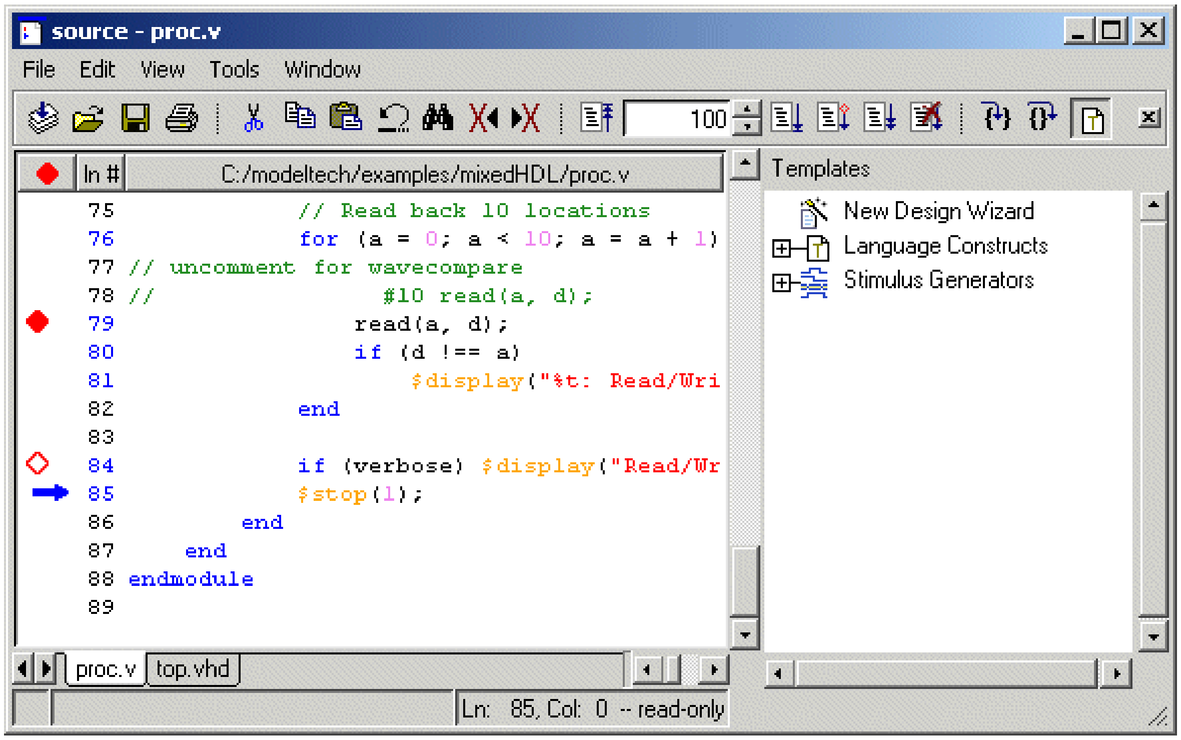
If the signal type is std_logic, std_ulogic, bit, verilog wire, verilog net, or any other logic type where 1 and 0 are valid, then 1 is the default High Value and 0 is the default Low Value. For other signal types, you will need to specify a High Value and a Low Value for the clock.

Source window

The Source window allows you to view and edit your source code. When you first load a design, the source file will display automatically if the Source window is open. Alternatively, you can select an item in a Structure tab of the Main window or use the **File > Open** command (Source window) to add a file to the window. (Your source code can remain hidden if you wish.

The window displays your source code with line numbers. As shown in the picture below, you may also see the following:

- Blue line numbers – denote lines on which you can set a breakpoint
- Blue arrow – denotes a process that you have selected in the Process window (UM-314) or the line corresponding to a breakpoint at which the simulator is currently stopped
- Red diamonds – denote file-line breakpoints; hollow diamonds denote breakpoints that are currently disabled
- File tabs – represent each open file
- Templates pane – displays HDL language templates (UM-397)



C:/modeltech/examples/mixedHDL/proc.v

```
75                // Read back 10 locations
76                for (a = 0; a < 10; a = a + 1)
77 // uncomment for wavecompare
78 //                   #10 read(a, d);
79                    read(a, d);
80                    if (d !== a)
81                        $display("%t: Read/Wri
82                end
83
84                if (verbose) $display("Read/Wr
85                $stop(1);
86            end
87        end
88 endmodule
89
```

Ln: 85, Col: 0 -- read-only

Note that files open by default in read-only mode. You can toggle this mode by selecting **Edit > read only**.

The Source window menu bar

This section provides information on select menu commands available in the Source window. Several commands are also available on a context menu by right-clicking in the body of the window.

File menu

Open Design Source	open a dialog that lists all source files for the current design
Use Source	specify an alternative file to use for the current source file; this alternative source mapping exists for the current simulation only
Source Directory	add to a list of directories to search for source files; you can set this permanently using the **SourceDir** variable in the *modelsim.tcl* file

Edit menu

To edit a source file, make sure **read only** is *not* selected on the Edit menu.

Clear highlights	clear highlights that result from double-clicking an error message or a line in a Performance Analyzer report
Comment Selected	turn the selected lines into comments by inserting the correct language comment character at the beginning of each line
Uncomment Selected	removes comment characters from the selected lines
Previous Coverage Miss	when simulating with Code Coverage (UM-419), find the previous line of code that was not used in the simulation
Next Coverage Miss	when simulating with Code Coverage (UM-419), find the next line of code that was not used in the simulation
read only	toggle the read-only status of the current source file

View menu

Show line numbers	toggle line numbers
Show language templates	toggle display of the HDL language templates (UM-397) pane
Show coverage data	toggle display of line hits when simulating with Code Coverage (UM-419)
Show branch coverage	toggle display of branch hits when simulating with Code Coverage (UM-419)

Show coverage numbers	toggle display of coverage numbers versus checkmarks when simulating with Code Coverage (UM-419)
Show coverage By Instance	toggle display of coverage numbers as sum of all instances or for each individual instance when simulating with Code Coverage (UM-419)
Properties	list a variety of information about the source file; for example, file type, file size, file modification date

Tools menu

Examine	display the current value of the selected item; same as the **examine** (CR-167) command; the item name is shown in the title bar
Describe	display information about the selected item; same as the **describe** command (CR-152); the item name is shown in the title bar
Compile	compile the currently active source file
Readers	list the names of all readers of the selected signal or net
Drivers	list the names of all drivers of the selected signal or net
C Debug	commands for using "C Debug" (UM-473); available on UNIX platforms only
Breakpoints	add, edit, or delete file-line and signal breakpoints; see "Creating and managing breakpoints" (UM-391)
Options	set various Source window options; see Options sub-menu below

Options sub-menu

Colorize Source	colorize key words, variables, and comments
Highlight Executable Lines	highlight the line numbers of executable lines
Middle Mouse Button Paste	enable/disable pasting by pressing the middle-mouse button
Verilog Highlighting	specify Verilog-style colorizing
VHDL Highlighting	specify VHDL-style colorizing
C Highlighting	specify C-style colorizing
Freeze File	maintain the same source file in the Source window (useful when you have two Source windows open; one can be updated from the Structure window (UM-331), the other frozen)
Freeze View	disable updating the source view from other windows
Auto-Indent Mode	indent code automatically when editing the file
Tab Stops	set tab stop distance in Source window (see "Setting tab stops in the Source window" (UM-330))
Examine Now	examine selected item at the current simulation time; this option affects the behavior of the Examine and Describe commands as well as the examine popup; see "Checking item values and descriptions" (UM-329)
Examine Current Cursor	examine selected item at the time of the current cursor in the Wave window; this option affects the behavior of the Examine and Describe commands as well as the examine popup; see "Checking item values and descriptions" (UM-329)

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Setting file-line breakpoints

You can easily set "File-line breakpoints" (UM-391) in the Source window using your mouse. Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint.

To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable/Enable Breakpoint**
  Deactivate or activate the selected breakpoint.
- **Edit Breakpoint**
  Open the **File Breakpoint** dialog to change breakpoint arguments; see "Adding a breakpoint" (UM-393) for a description of the dialog.
- **Edit All Breakpoints**
  Open the **Modify Breakpoints** dialog; see "Breakpoints dialog" (UM-392).

Checking item values and descriptions

There are two quick methods to determine the value and description of an item displayed in the Source window:

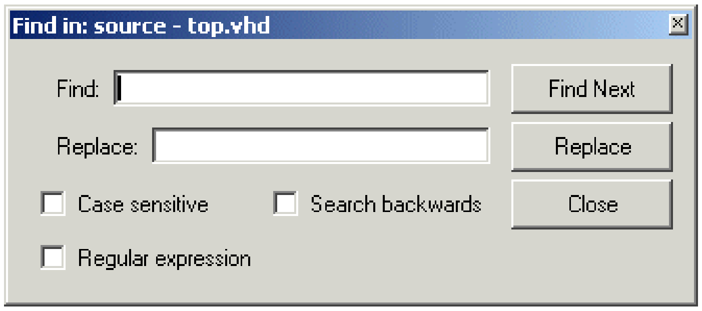
- select an item, then choose **Tools > Examine** or **Tools > Describe** from the Source window menu
- pause over an item with your mouse pointer to see an examine pop-up

Select **Tools > Options > Examine Now** or **Tools > Options > Examine Current Cursor** to determine at what simulation time the item is examined or described.

You can also invoke the **examine** (CR-167) and/or **describe** (CR-152) command on the command line or in a macro.

Finding and replacing in the Source window

The Find dialog box allows you to find and replace text strings or regular expressions in the Source window. Select **Edit > Find** or **Edit > Replace** to bring up the Find dialog box. If you select **Edit > Find**, the **Replace** field is absent from the dialog.



Enter the value to search for in the **Find** field. If you are doing a replace, enter the appropriate value in the **Replace** field. Optionally specify whether the entries are **case sensitive** and whether to **search backwards** from the current cursor location. Check the **Regular expression** checkbox if you are using regular expressions.

Setting tab stops in the Source window

You can set temporary tab stops in the Source window by selecting **Tools** > **Options** > **Tab Stops**. Follow these steps:

1 Select **Tools** > **Options** > **Tab Stops** (Source window).

2 In the dialog that appears, enter either a single number "n" and units, which sets a tab stop every n units, or enter a list of numbers which sets a tab at each location. Available units and their abbreviations are as follows:

Units	Abbreviations
centimeters	c, cm
millimeters	m, mm
inches	i, in
points	p
pixels (screen units)	u
characters	char, chars

If you don't specify units, they default to characters.

Here are three examples:

- Enter 5 to set a tab stop every 5 characters.
- Enter 10c to set a tab stop every 10 centimeters.
- Enter a list of numbers like the following to set tab stops at specific character locations: 21 49 77 105 133 161 189 217 245 273 301 329 357 385 413 441 469

Important: Do not use quotes or braces in the list (i.e., "21 49" or {21 49}); this will cause the GUI to hang.

If you want to set permanent tab stops, you have to edit the PrefSource(tabs) preference variable and then save a *modelsim.tcl* file. See "Preference variables located in Tcl files" (UM-631) for further details.

Structure window

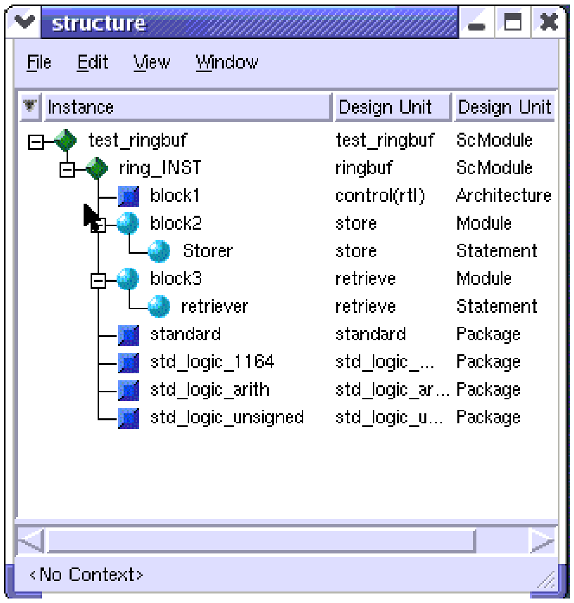
The Structure window provides a hierarchical view of the structure of your design. In ModelSim versions 5.5 and later, the information contained in the Structure window is also shown in the structure tabs of the Main window Workspace (UM-263). The Structure window does not display by default. You can display the Structure window at any time by selecting **View** > **Structure** (Main window).

An entry is created by each item within the design. (Your design structure can remain hidden if you wish. When you select a region in the Structure window, it becomes the *current region* and is highlighted. The Source window (UM-325) and Signals window (UM-316) change dynamically to reflect the information for that region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

Also, when you select a region in the Structure window, the Process window (UM-314) is updated if **In Region** is selected in that window. The Process window will in turn update the Variables window (UM-334).

Items you can view

The following items are represented by hierarchy within the Structure window.



Instance	Design Unit	Design Unit
test_ringbuf	test_ringbuf	ScModule
ring_INST	ringbuf	ScModule
block1	control(rtl)	Architecture
block2	store	Module
Storer	store	Statement
block3	retrieve	Module
retriever	retrieve	Statement
standard	standard	Package
std_logic_1164	std_logic_...	Package
std_logic_arith	std_logic_ar...	Package
std_logic_unsigned	std_logic_u...	Package

VHDL items

(indicated by a dark blue square icon)
component instantiations, generate statements, block statements, and packages

Verilog items

(indicated by a lighter blue circle icon)
module instantiations, named forks, named begins, tasks, and functions

SystemC items

(indicated by a green diamond icon)
SystemC module instantiations, primitive channels, method and thread processes

Virtual items

(indicated by an orange diamond icon)
virtual regions; see "Virtual Objects (User-defined buses, and more)" (UM-248) for more information.

You can expand and contract the display to view the hierarchical structure by clicking on the boxes that contain "+" or "-". Clicking "+" expands the hierarchy so the sub-elements of that item can be seen. Clicking "-" contracts the hierarchy.

Structure window menu bar

This section provides information on select menu commands available in the Signals window. Several commands are also available on a context menu by right-clicking in the right-hand pane of the window (see "Structure window context menu" (UM-333) for some details).

File menu

Save List	save the structure tree to a text file viewable with the ModelSim **notepad** (CR-207)
Environment	1) specify that the window contents change when the active dataset is changed; 2) fix the window contents to a specific dataset; or 3) change to a new root context

View menu

Sort	sort the structure tree in either ascending, descending, or declaration order

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

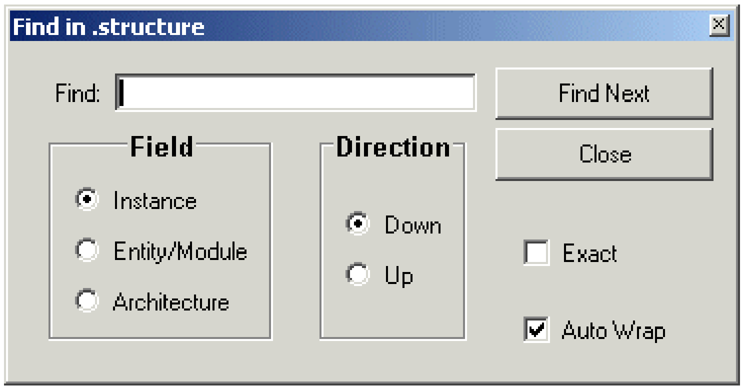
Structure window context menu

Access the following commands by clicking the right mouse button on an entry in the right-hand pane:

View Source	opens the source file in the Source window (UM-325); double-clicking will also open the source file
Add	adds the selected item to the Dataflow, List, or Wave window or to the current log file
Save List	writes the item names in the Structure tab to a text file
Coverage	provides access to the Coverage Reports and Clear Coverage Data commands; see *Chapter 12 - Code Coverage* for more details

Finding items in the Structure window

The Find dialog box allows you to search for text strings in the Structure window. Select **Edit > Find** (Structure window) to bring up the Find dialog box.



Enter the value to search for in the **Find** field. Specify whether you are looking for an **Instance**, **Entity/Module**, or **Architecture**. Also specify which direction to search.

Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

Variables window

The Variables window is divided into two window panes. The left pane lists the names of items within the current process. The right pane lists the current value(s) associated with each name. The pathname of the current process is displayed at the bottom of the window. (The internal variables of your design can remain hidden if you wish.

Items you can view

The following types of items can be viewed in the Variables window:

VHDL items

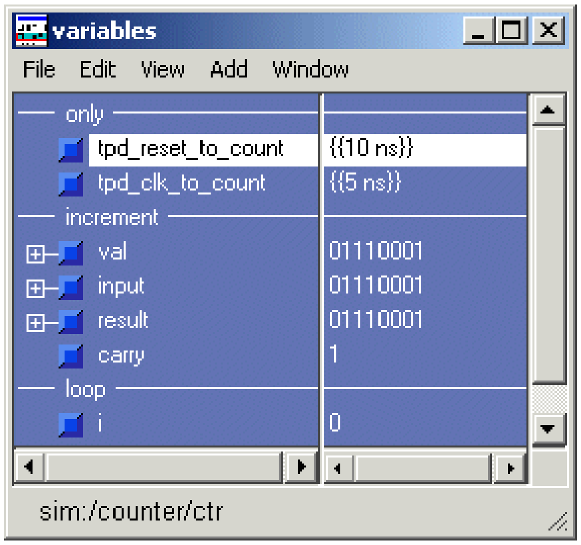
constants, generics, and variables

Verilog items

registers and variables

SystemC items

SystemC variables are not supported for viewing.



VHDL composite types (arrays and record types) and Verilog vector registers and memories are shown in a hierarchical fashion. ModelSim indicates hierarchy with plus (expandable), minus (expanded), and blank (single level) boxes. See "Tree window hierarchical view" (UM-261) for more information.

To change the value of a VHDL variable, constant, or generic or a Verilog register or variable, move the pointer to the desired name and click to highlight the selection. Select **Edit > Change** (Variables window) to bring up a dialog box that lets you specify a new value. You can enter any value that is valid for the variable. An array value must be specified as a string (without surrounding quotation marks). To modify the values in a record, you need to change each field separately.

Click on a process in the Process window to change the Variables window.

The Variables window menu bar

This section provides information on select menu commands available in the Variables window.

File menu

Save List	save the variable tree to a text file viewable with the ModelSim **notepad** (CR-207)
Environment	**Follow Process Selection**: update the window based on the selection in the Process window (UM-314) **Fix to Current Process**: maintain the current view, do not update
Close	close this copy of the Variables window

Edit menu

Change	change the value of the selected item(s)

View menu

Sort	sort the variables tree in either ascending, descending, or declaration order
Justify Values	justify values to the left or right margins of the window pane

Add menu

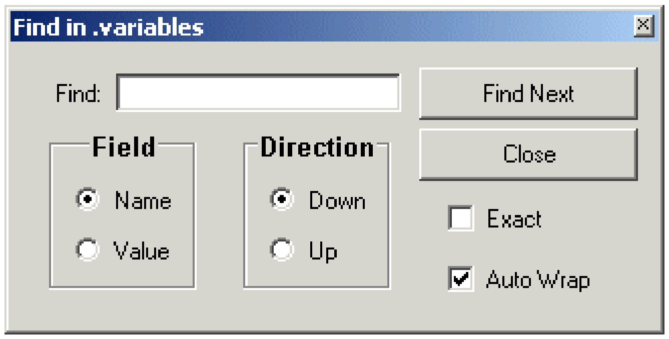
Add variables to the Wave or List windows or the current WLF file.

Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Finding items in the Variables window

To find the specified text string within the Variables window, choose the **Name** or **Value** field to search and the search direction: **Down** or **Up**.



Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

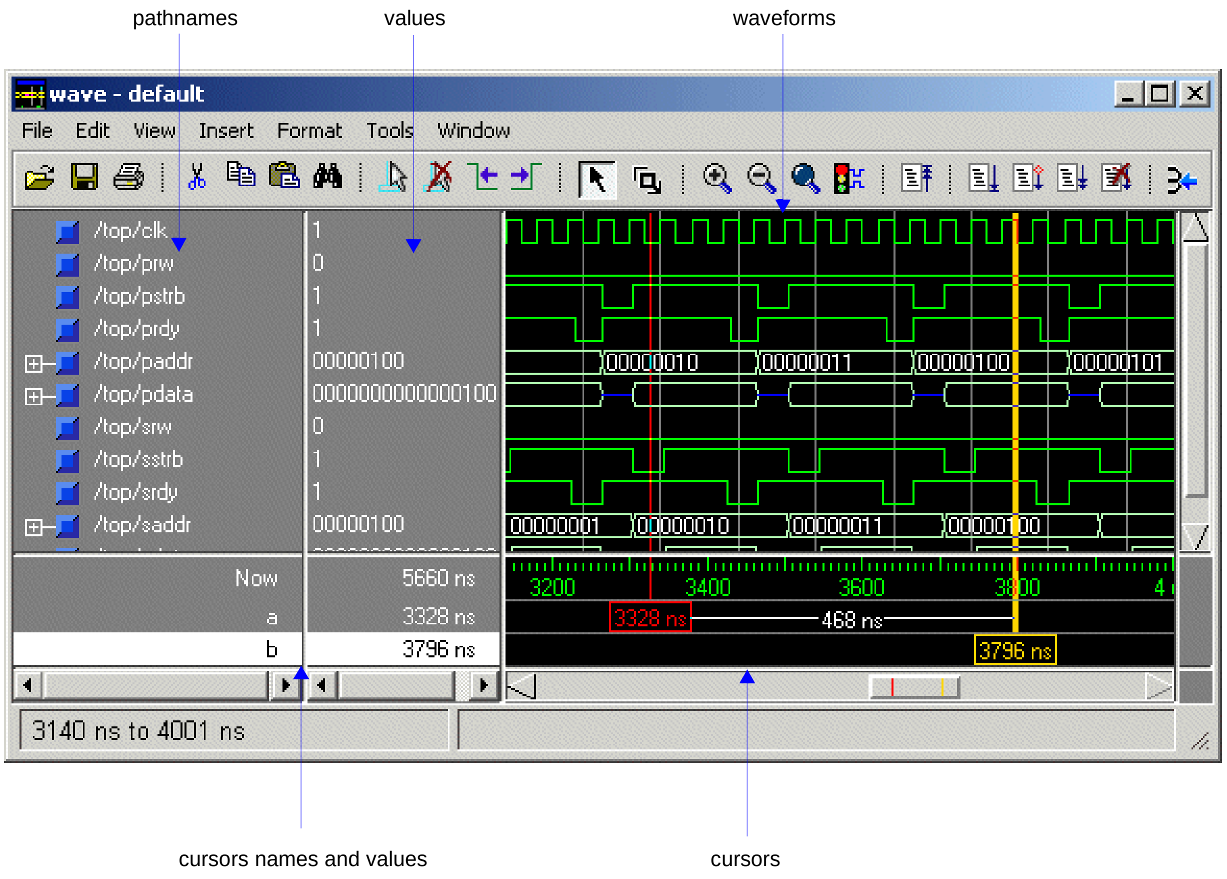
Check **Auto Wrap** to continue the search at the beginning of the window.

You can also do a quick find from the keyboard. When the Variables window is active, each time you type a letter the highlight will move to the next item whose name begins with that letter.

Wave window

The Wave window, like the List window, allows you to view the results of your simulation. In the Wave window, however, you can see the results as waveforms and their values.

The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.



```
/top/clk     1
/top/prw     0
/top/pstrb   1
/top/prdy    1
/top/paddr   00000100
/top/pdata   0000000000000100
/top/srw     0
/top/sstrb   1
/top/srdy    1
/top/saddr   00000100

Now   5660 ns
a     3328 ns
b     3796 ns

3140 ns to 4001 ns
```

Pathname pane

The pathname pane displays signal pathnames. Signals can be displayed with full pathnames, as shown here, or with only the leaf element displayed. You can increase the size of the pane by clicking and dragging on the right border. The selected signal is highlighted.

The white bar along the left margin indicates the selected dataset (see "Splitting Wave window panes" (UM-344)).

Value pane

The value pane displays the values of the displayed signals.

The radix for each signal can be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII, or default. The default radix can be set by selecting **Simulate** > **Simulation Options** (Main window) (see "Setting default simulation options" (UM-386)).

The data in this pane is similar to that shown in the Signals window (UM-316), except that the values change dynamically whenever a cursor in the waveform pane is moved.

Waveform pane

The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors. Signal values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. Each signal can be formatted individually. The default format is logic.

If you rest your mouse pointer on a signal in the waveform pane, a popup displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog (see "Setting Wave window display properties" (UM-352)).

Cursor panes

There are three cursor panes–the left pane shows the cursor names; the middle pane shows the current simulation time and the value for each cursor; and the right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors can be displayed. See "Using time cursors in the Wave window" (UM-358) for more information.

Items you can view

The following types of items can be viewed in the Wave window

VHDL items

(indicated by a dark blue square)
signals, aliases, process variables, and shared variables

Verilog items

(indicated by a light blue circle)
nets, registers, variables, and named events

SystemC items

(indicated by a green diamond)
primitive channels and ports

Virtual items

(indicated by an orange diamond)
virtual signals, buses, and functions, see; "Virtual Objects (User-defined buses, and more)" (UM-248) for more information

Comparison items

(indicated by a yellow triangle)
comparison region and comparison signals; see *Chapter 13 - Waveform Compare* for more information

Constants, generics, and parameters are not viewable in the Wave windows.

The data in the item values pane is very similar to the Signals window, except that the values change dynamically whenever a cursor in the waveform pane is moved.

At the bottom of the waveform pane you can see a time line, tick marks, and the time value of each cursor's position. As you click and drag to move a cursor, the time value at the cursor location is updated at the bottom of the cursor.

You can resize the window panes by clicking on the bar between them and dragging the bar to a new location.

Waveform and signal-name formatting are easily changed via the Format menu (UM-342). You can reuse any formatting changes you make by saving a Wave window format file, see "Adding items with a Wave window format file" (UM-339).

Adding items in the Wave window

Before adding items to the Wave window you may want to set the window display properties (see "Setting Wave window display properties" (UM-352)). You can add items to the Wave window in several ways.

Adding items from other window with drag and drop

You can drag and drop items into the Wave window from the List, Process, Signals, Source, Structure, or Variables window. Select the items in the first window, then drop them into the Wave window. Depending on what you select, all items or any portion of the design can be added.

Adding items from the command line

To add specific items to the window, enter (separate the item names with a space):

```
VSIM> add wave <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
VSIM> add wave *
```

Or add all the items in the design with:

```
VSIM> add wave -r /*
```

Adding items with a Wave window format file

Select **File** > **Open** > **Format** and specify a previously saved format file. See "Saving the Wave window format" (UM-340) for details on how to create a format file.

Saving the Wave window format

By default all Wave window information is forgotten once you close the Wave window. If you want to restore the Wave window to a previously configured layout, you must save a Wave window format file. Follow these steps:

1 Add the items you want to the Wave window.

2 Edit and format the items, see "Editing and formatting items in the Wave window" (UM-347) to create the view you want.

3 Save the format to a file by selecting **File** > **Save** > **Format** (Wave window).

To use the format file, start with a blank Wave window and run the DO file in one of two ways:

- Invoke the **do** command (CR-156) from the command line:

```
VSIM> do <my_wave_format>
```

- Select **File** > **Open** > **Format** (Wave window).

▶ **Note:** Wave window format files are design-specific; use them only with the design you were simulating when they were created.

The Wave window menu bar

This section provides information on select menu commands available in the Wave window. Many of these commands are also available via a context menu by clicking your right mouse button within the Wave window itself.

File menu

Open	Format – run a Wave window format (DO) file previously saved with Save Format; see "Saving the Wave window format" (UM-340)
Save	Format – save the current Wave window display and signal preferences to a DO (macro) file; see "Saving the Wave window format" (UM-340) Image – saves a bitmap file of the Wave window
Page Setup	configure page setup for printing; see "Printer Page Setup" (UM-366)
Print Postscript	save or print the waveform display as a Postscript file; see "Printing and saving waveforms" (UM-363) for details

Edit menu

Edit Cursor	open a dialog to specify the location of the selected cursor
Delete Cursor	delete the selected cursor from the window
Delete Window Pane	delete the selected window pane
Remove All (Panes and Signals)	removes all signals and additional window panes, leaving the window in its original state
Find	find the specified item label within the pathname pane or the specified value within the value pane
Search	search the waveform display for a specified value, or the next transition for the selected signal; see: "Searching for item values in the Wave window" (UM-356)

View menu

Mouse Mode	toggle mouse pointer between Select Mode (click left mouse button to select, drag with middle mouse button to zoom) and Zoom Mode (drag with left mouse button to zoom, click middle mouse button to select)
Signal Declaration	open the source file in the Source window and highlight the signal declaration for the currently selected signal
Cursors	choose a cursor to go to from a list of available cursors
Bookmarks	choose a bookmark to go to from a list of available bookmarks
Goto Time	scroll the Wave window so the specified time is in view; "g" hotkey produces the same result
Sort	sort the top-level items in the pathname pane; sort with full path or viewed name; use ascending or descending order
Justify Values	justify values to the left or right margins of the window pane
Refresh Display	clear the Wave window, empty the file cache, and rebuild the window from scratch
Signal Properties	set properties for the selected item; see "Editing and formatting items in the Wave window" (UM-347)

Insert menu

Divider	insert a divider at the current location
Breakpoint	add a breakpoint on the selected signal; see "Signal breakpoints" (UM-391)
Bookmark	add a bookmark with the current zoom range and scroll location; see "Saving zoom range and scroll position with bookmarks" (UM-361)
Cursor	add a cursor to the waveform pane
Window Pane	split the pathname, values and waveform window panes to provide room for a new waveset

Format menu

Radix	set the selected items' radix
Format	set the waveform format for the selected items – Literal, Logic, Event, Analog
Color	set the color for the selected items from a color palette
Height	set the waveform height in pixels for the selected items

Tools menu

Waveform Compare	see "Waveform Compare menu" (UM-468)
Breakpoints	add, edit, and delete signal breakpoints; see "Creating and managing breakpoints" (UM-391)
Bookmarks	add, edit, delete, and goto bookmarks; see "Saving zoom range and scroll position with bookmarks" (UM-361)
Dataset Snapshot	enable periodic saving of simulation data to a WLF file
Combine Signals	combine the selected items into a user-defined bus; see "Combining items in the Wave window" (UM-345)
Window Preferences	set various display properties; see "Setting Wave window display properties" (UM-352)

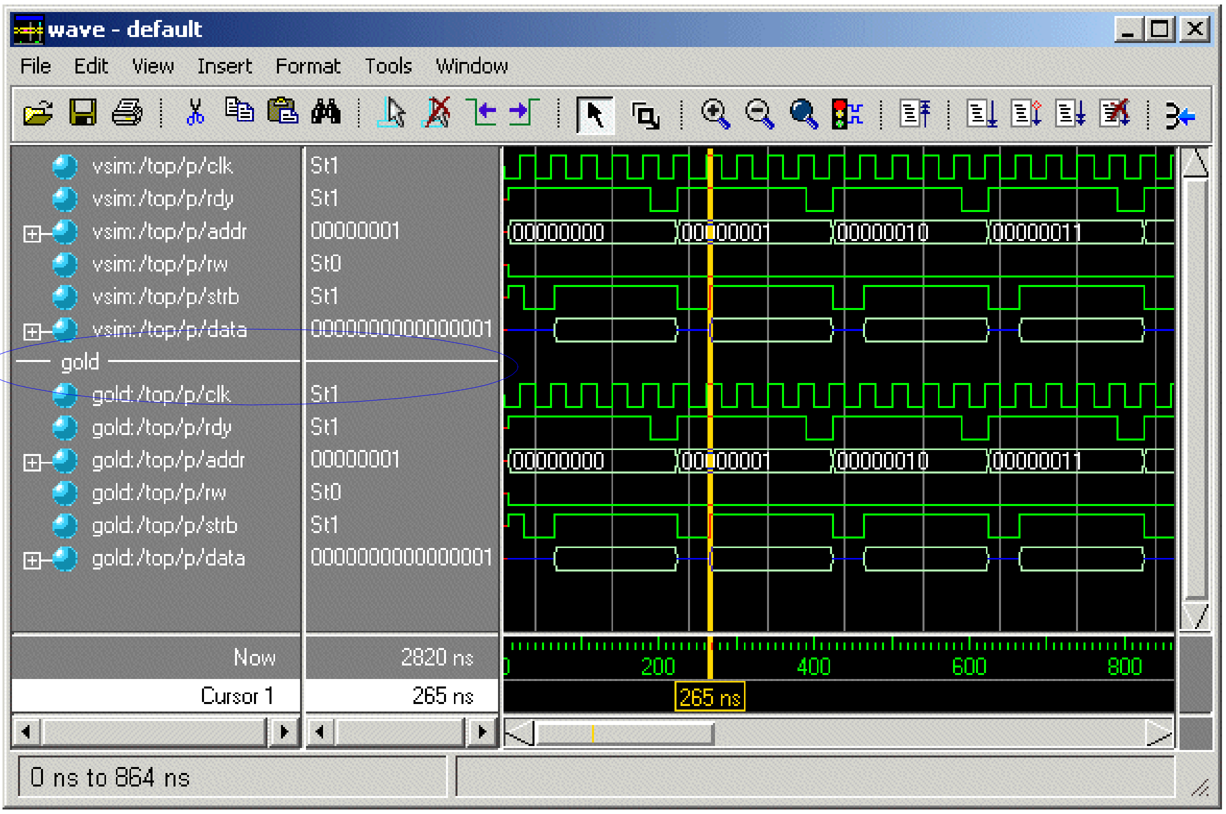
Window menu

The Window menu is identical in all windows. See "Window menu" (UM-268) for a description of the commands.

Using dividers

Dividers serve as a visual aid to signal debugging, allowing you to separate signals and waveforms for easier viewing. Dividing lines can be placed in the pathname and values window panes by selecting **Insert > Divider** (Wave window). Or, you can add a divider using the **-divider** argument to the **add wave** command (CR-64).

Dividing lines can be assigned any name or no name at all. The default name is "New Divider." In the illustration below, two datasets have been separated with a Divider called "gold." Notice that the waveforms in the waveform window pane have been separated by the divider as well.



```
vsim:/top/p/clk    St1
vsim:/top/p/rdy    St1
vsim:/top/p/addr   00000001
vsim:/top/p/rw     St0
vsim:/top/p/strb   St1
vsim:/top/p/data   0000000000000001
gold
gold:/top/p/clk    St1
gold:/top/p/rdy    St1
gold:/top/p/addr   00000001
gold:/top/p/rw     St0
gold:/top/p/strb   St1
gold:/top/p/data   0000000000000001
Now                2820 ns
Cursor 1           265 ns
0 ns to 864 ns
```

After you have added a divider, you can move it, change its properties (name and size), or delete it.

To move a divider — Click and drag the divider to the location you want.

To change a divider's name and size — Click the divider with the right (Windows) or third (UNIX) mouse button and select Divider Properties from the pop-up menu.

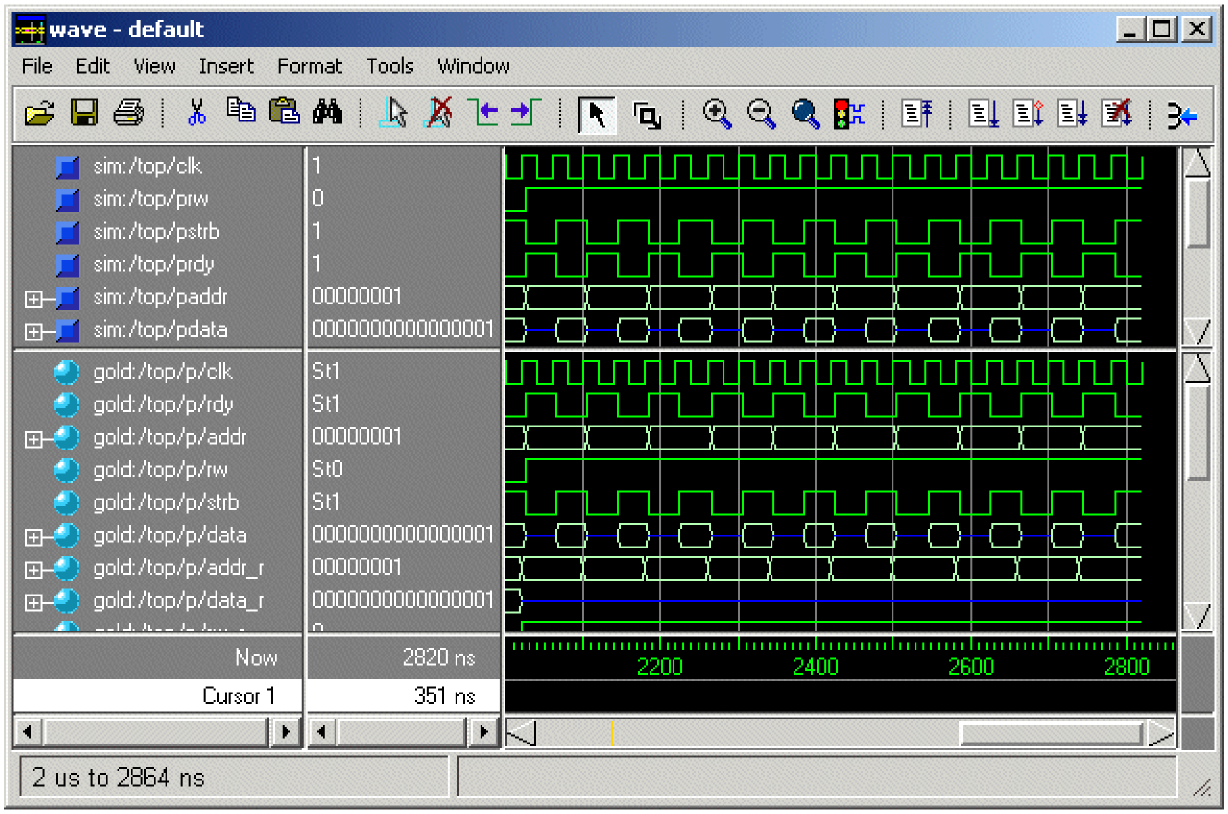
To delete a divider — Select the divider and either press the <Delete> key on your keyboard or select Delete from the pop-up menu.

Splitting Wave window panes

The pathnames, values, and waveforms panes of the Wave window display can be split to accommodate signals from one or more datasets. Selecting **Insert > Window Pane** (Wave window) creates a space below the selected dataset and makes the new window pane the selected pane. (The selected wave window pane is indicated by a white bar along the left margin of the pane.)

In the illustration below, the Wave window is split, showing the current active simulation with the prefix "sim," and a second view-mode dataset, with the prefix "gold."

For more information on viewing multiple simulations, see *Chapter 9 - WLF files (datasets) and virtuals*.



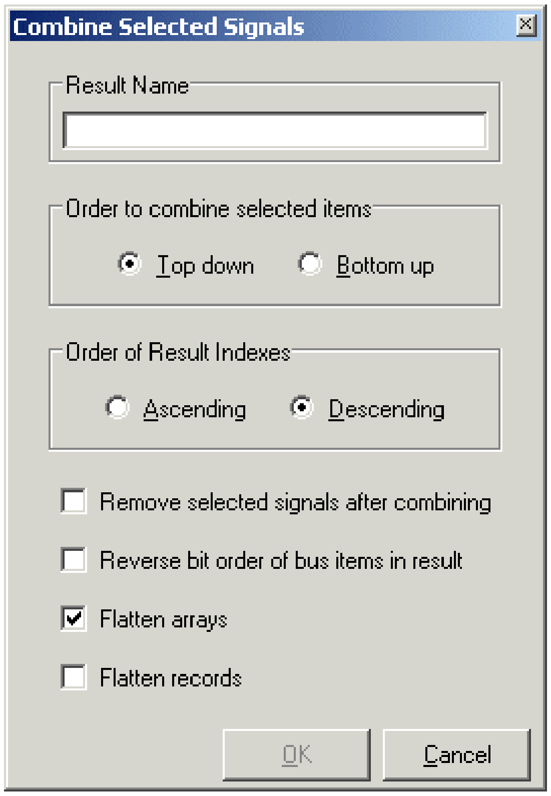
sim:/top/clk	1
sim:/top/prw	0
sim:/top/pstrb	1
sim:/top/prdy	1
sim:/top/paddr	00000001
sim:/top/pdata	0000000000000001
gold:/top/p/clk	St1
gold:/top/p/rdy	St1
gold:/top/p/addr	00000001
gold:/top/p/rw	St0
gold:/top/p/strb	St1
gold:/top/p/data	0000000000000001
gold:/top/p/addr_r	00000001
gold:/top/p/data_r	0000000000000001
Now	2820 ns
Cursor 1	351 ns

2 us to 2864 ns

Combining items in the Wave window

You can combine signals in the Wave window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. You can also do this from the ModelSim prompt using the **virtual signal** command (CR-339).

To create a bus, select one or more signals in the Wave window and then choose **Tools > Combine Signals**.

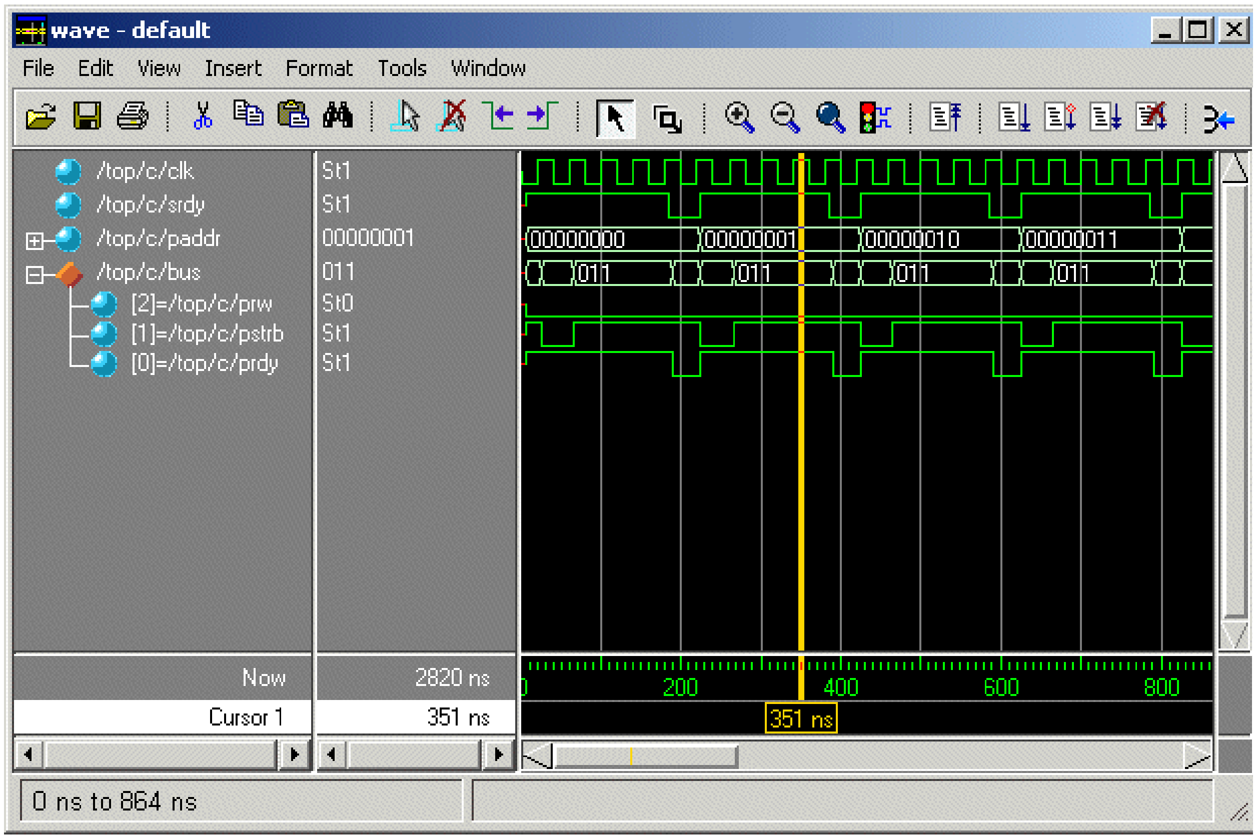


The **Combine Selected Signals** dialog box includes these options:

- **Result Name**
  Specifies the name of the newly created bus.
- **Order to combine selected items**
  Specifies the order in which to combine the selected signals. "Top down" specifies that the selected signals are ordered as they appear top-to-bottom in the Wave window. "Bottom up" reverses the order.
- **Order of Result Indexes**
  Specifies in which order the selected signals are indexed in the bus. If set to Ascending, the first signal selected in the Wave window will be assigned an index of 0. If set to Descending, the first signal selected will be assigned the highest index number.
- **Remove selected signals after combining**
  Specifies whether you want to remove the selected signals from the Wave window once the bus is created.

- **Reverse bit order of bus items in the result**
  If checked, the bits of each selected signal are reversed in the newly created bus. The order of the signals in the bus is not affected.
- **Flatten arrays**
  If checked, ModelSim combines the signals into one big array. If unchecked, ModelSim combines signals together without merging them into one array. The signals become elements of a record and retain their original names. When expanded, the new signal looks just like a group of signals.
- **Flatten records**
  If checked, causes elements of a record type signal to be pulled up to the top level. This option is the reverse of "Flatten arrays."

In the illustration below, three signals have been combined to form a new bus called "bus". Note that the component signals are listed in the order in which they were selected in the Wave window. Also note that the value of the bus is made up of the values of its component signals, arranged in a specific order. Virtual objects are indicated by an orange diamond.



Other virtual items in the Wave window

See "Virtual Objects (User-defined buses, and more)" (UM-248) for information about other virtual items viewable in the Wave window.

Displaying drivers of the selected waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar.
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see "Setting Wave window display properties" (UM-352))

This operation will open the Dataflow window and display the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window will also open showing the selected signal with a cursor at the selected time. The Dataflow window will show the signal(s) values at the current time cursor position.

Editing and formatting items in the Wave window

Once you have the items you want in the Wave window, you can edit and format the list in the pathname and values panes to create the view you find most useful. (See also, "Setting Wave window display properties" (UM-352).)

To edit an item:

Select the item's label in the pathname pane or its waveform in the waveform pane. Move, copy, or remove the item by selecting commands from the Wave window **Edit menu** (UM-341).

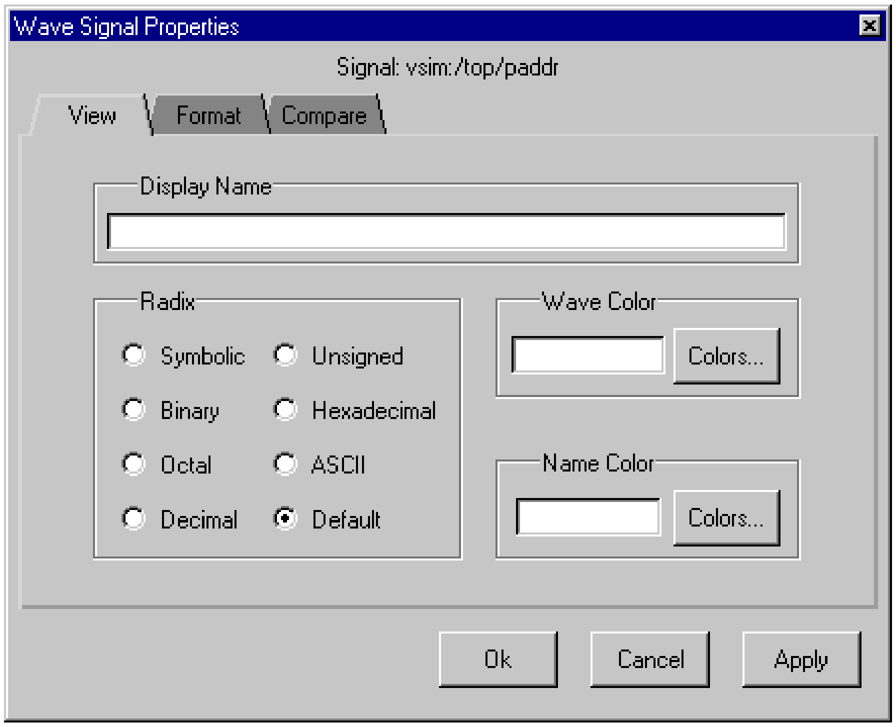
You can also **click+drag** to move items within the pathnames and values panes:

- to select several items:
  control+click to add or subtract from the selected group
- to move the selected items:
  re-click and hold on one of the selected items, then drag to the new location

To format an item:

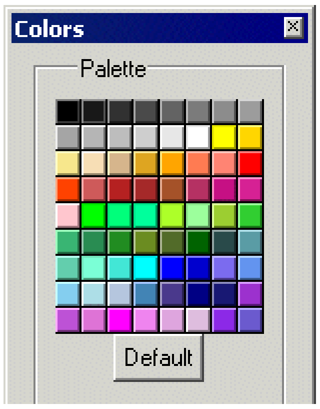
Select the item's label in the pathname pane or its waveform in the waveform pane, then select **View > Signal Properties** (Wave window) or use the selections in the **Format** menu.

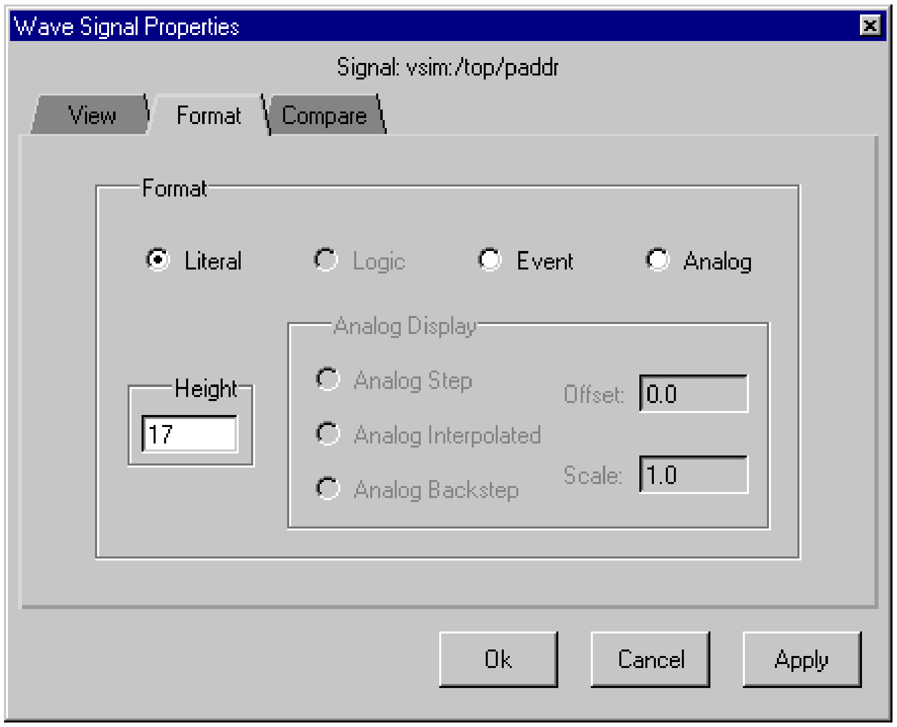
When you select **View > Signal Properties** the Wave Signal Properties dialog box opens. It has three tabs: View, Format, and Compare.



The **View** tab includes these options:

- **Display Name**
  Specifies a new name (in the pathname pane) for the selected signal.
- **Radix**
  Specifies the Radix of the selected signal(s). Setting this to default causes the signal's radix to change whenever the default is modified using the **radix** command (CR-235). Item values are not translated if you select Symbolic.
- **Wave Color**
  Specifies the waveform color. Select a new color from the color palette, or enter a color name. The Default button in the Colors palette allows you to return the selected item's color back to its default value.
- **Name Color**
  Specifies the signal name's color. Select a new color from the color palette, or enter a color name. The Default button in the Colors palette allows you to return the selected item's color back to its default value.





Signal: vsim:/top/paddr

Format: [✓ Literal] Logic Event Analog

Height: 17

Analog Step / Analog Interpolated / Analog Backstep

Offset: 0.0

Scale: 1.0

The **Format** tab includes these options:

- **Format: Literal**
  Displays the waveform as a box containing the item value (if the value fits the space available). This is the only format that can be used to list a record.

- **Format: Logic**
  Displays values as U, X, 0, 1, Z, W, L, H, or -.
- **Format: Event**
  Marks each transition during the simulation run.
- **Format: Analog [Step | Interpolated | Backstep]**
  Analog Step
  Displays the waveform in step style.

  Analog Interpolated
  Displays the waveform in interpolated style.

  Analog Backstep
  Displays the waveform in backstep style. Often used for power calculations.

  Offset and Scale
  Allows you to adjust the scale of the item as it is seen on the display. Offset is the number of pixels offset from zero. The scale factor reduces (if less than 1) or increases (if greater than 1) the number of pixels displayed.

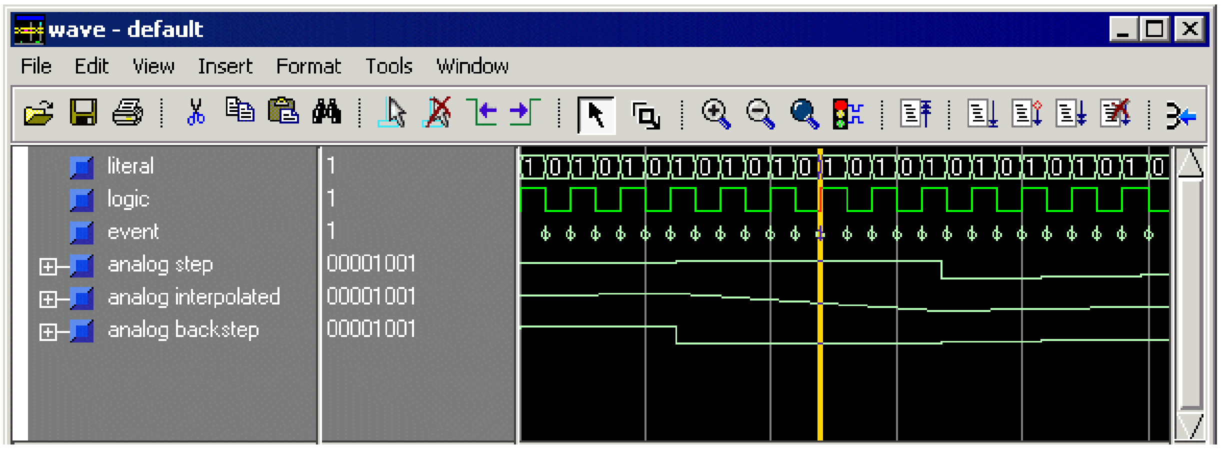
  Only the following types are supported in Analog format:

  VHDL types:
  All vectors - std logic vectors, bit vectors, and vectors derived from these types
  Scalar integers
  Scalar reals
  Scalar times

  Verilog types:
  All vectors
  Scalar reals
  Scalar integers

  SystemC types:
  Vector types (sc_int<>, sc_bigint<>, etc.)
  Scalar integers (char, short, int, long, etc.)
  float, double

The signals in the following illustration demonstrate the various signal formats.



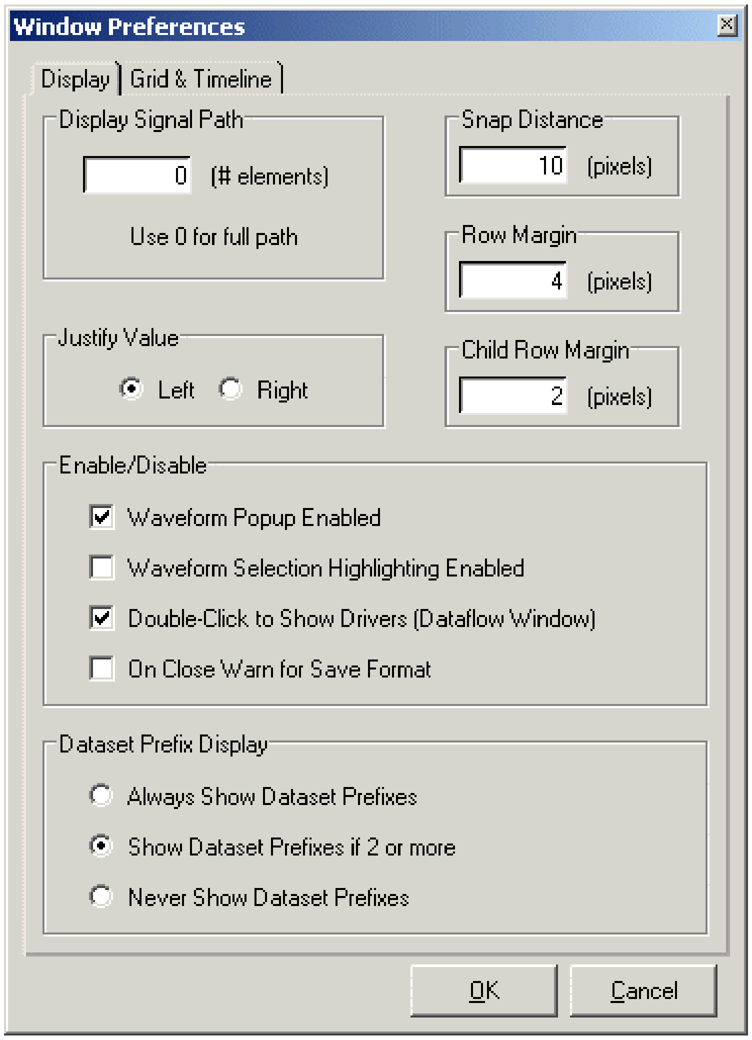
- **Height**
  Allows you to specify the height (in pixels) of the waveform.

The **Compare** tab includes the same options as those in the Add Signal Options dialog box (see "Comparison Method tab" (UM-463)).

Setting Wave window display properties

You can define display properties of the Wave window by selecting **Tools > Window Preferences** (Wave window). You can make these changes permanent by selecting **Tools > Save Preferences** (Main window). See "Preference variables located in Tcl files" (UM-631) for details on changing window properties permanently.

The dialog box has two tabs–**Display** and **Grid & Timeline**.



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

Snap Distance: 10 (pixels)

Row Margin: 4 (pixels)

Justify Value: [✓ Left] Right

Child Row Margin: 2 (pixels)

Enable/Disable
- [✓] Waveform Popup Enabled
- [ ] Waveform Selection Highlighting Enabled
- [✓] Double-Click to Show Drivers (Dataflow Window)
- [ ] On Close Warn for Save Format

Dataset Prefix Display
- Always Show Dataset Prefixes
- [✓ Show Dataset Prefixes if 2 or more]
- Never Show Dataset Prefixes

The **Display** tab includes the following options:

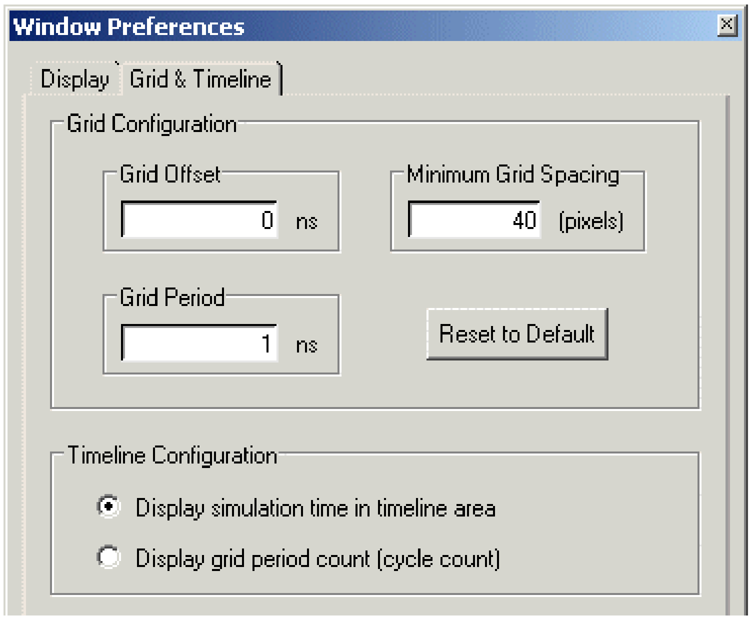
- **Display Signal Path**
  Sets the display to show anything from the full pathname of each signal (e.g., *sim:/top/clk*) to only its leaf element (e.g., *sim:clk*). A non-zero number indicates the number of path elements to be displayed. The default is Full Path.
- **Justify Value**
  Specifies whether the signal values will be justified to the left margin or the right margin in the values window pane.

- **Snap Distance**
  Specifies the distance the cursor needs to be placed from an item edge to jump to that edge (a 0 specification turns off the snap).
- **Row Margin**
  Specifies the distance in pixels between top-level signals.
- **Child Row Margin**
  Specifies the distance in pixels between child signals.
- **Waveform Popup Enable**
  Toggles on/off the popup that displays when you rest your mouse pointer on a signal or comparison object.
- **Waveform Selection Highlighting Enabled**
  Toggles on/off waveform highlighting. When enabled the waveform is highlighted if you select the waveform or its value.
- **Double-Click to Show Drivers (Dataflow Window)**
  Toggles on/off double-clicking to show the drivers of the selected waveform. See "Displaying drivers of the selected waveform" (UM-347) for more details.
- **On Close Warn for Save Format**
  Toggles on/off a message that prompts you to save the Wave window format when you close the window. See "Displaying drivers of the selected waveform" (UM-347) for more details.
- **Dataset Prefix**
  Specifies how signals from different datasets are displayed.

  Always Show Dataset Prefixes
  All dataset prefixes will be displayed along with the dataset prefix of the current simulation ("sim").

  Show Dataset Prefixes if 2 or more
  Displays all dataset prefixes if 2 or more datasets are displayed. "sim" is the default prefix for the current simulation.

  Never Show Dataset Prefixes
  No dataset prefixes will be displayed. This selection is useful if you are running only a single simulation.



Grid Configuration

Grid Offset: 0 ns

Minimum Grid Spacing: 40 (pixels)

Grid Period: 1 ns

Timeline Configuration

[✓ Display simulation time in timeline area]

Display grid period count (cycle count)

The **Grid & Timeline** tab is used to configure grid lines and the horizontal axis in the waveform pane. You can also access this tab by right-clicking in the cursor tracks at the bottom of the Wave window and selecting Grid & Timeline Properties. The tab has the following options:

- **Grid Offset**
  Specifies the time (in user time units) of the first grid line. Default is 0.
- **Grid Period**
  Specifies the time (in user time units) between subsequent grid lines. Default is 1.
- **Minimum Grid Spacing**
  Specifies the closest (in pixels) two grid lines can be drawn before intermediate lines will be removed. Default is 40.
- **Timeline Configuration**
  Specifies whether to display simulation time or grid period count on the horizontal axis. Default is to display simulation time.

Sorting a group of items

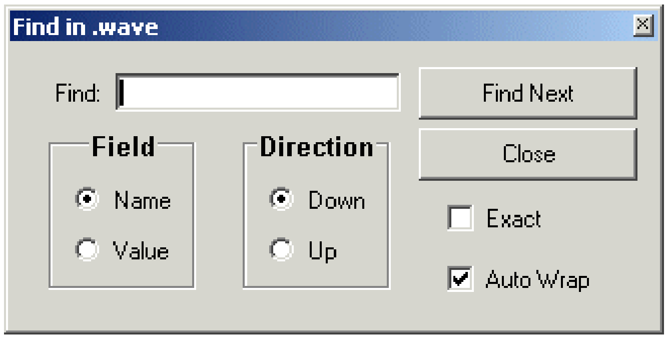
Select **View** > **Sort** to sort the items in the pathname and values panes.

Setting signal breakpoints

You can set "Signal breakpoints" (UM-391) in the Wave window. When a signal breakpoint is hit, a message appears in the Main window Transcript stating which signal caused the breakpoint.

To insert a signal breakpoint, right-click a signal and select **Insert Breakpoint**. A breakpoint will be set on the selected signal. See "Creating and managing breakpoints" (UM-391) for more information.

Finding items by name or value in the Wave window



Settings shown: Field [✓ Name] Value; Direction [✓ Down] Up; Exact ☐; [✓ Auto Wrap]

The Find dialog box allows you to search for text strings in the Wave window. Select **Edit > Find** (Wave window) to bring up the Find dialog box.

Choose either the Name or Value field to search and enter the value to search for in the Find field. **Find** the item by searching **Down** or **Up** through the Wave window display.

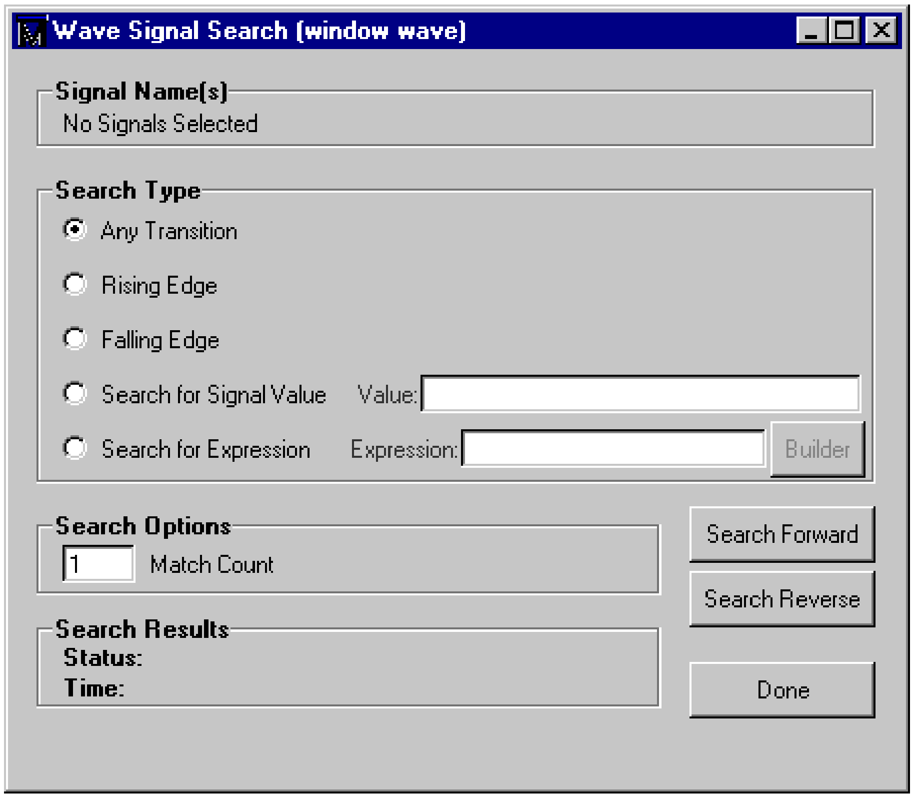
Check **Exact** if you only want to find items that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

Check **Auto Wrap** to continue the search at the beginning of the window.

The find operation works only within the active pane.

Searching for item values in the Wave window

Select an item in the Wave window and then select **Edit** > **Search** to bring up the Wave Signal Search dialog box.



The **Wave Signal Search** dialog box includes these options:

You can locate values for the **Signal Name(s)** shown at the top of the dialog box. The search is based on these options:

- **Search Type: Any Transition**
  Searches for any transition in the selected signal(s).
- **Search Type: Rising Edge**
  Searches for rising edges in the selected signal(s).
- **Search Type: Falling Edge**
  Searches for falling edges in the selected signal(s).
- **Search Type: Search for Signal Value**
  Searches for the value specified in the **Value** field; the value should be formatted using VHDL or Verilog numbering conventions; see "Numbering conventions" (CR-21).

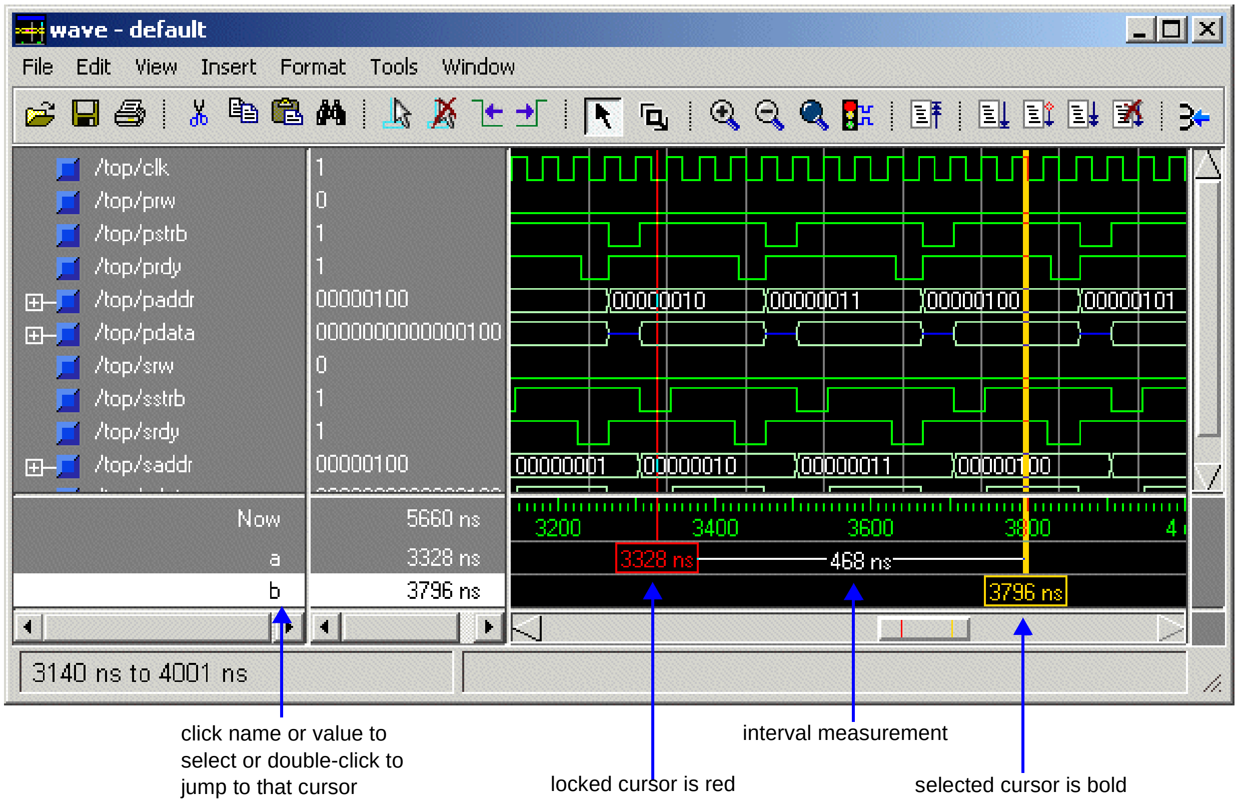
Note: If your signal values are displayed in binary radix, see "Searching for binary signal values in the GUI" (CR-30) for details on how signal values are mapped between a binary radix and std_logic.

- **Search Type: Search for Expression**
  Searches for the expression specified in the **Expression** field evaluating to a boolean true. Activates the **Builder** button so you can use "The GUI Expression Builder" (UM-395) if desired.

  The expression can involve more than one signal but is limited to signals logged in the Wave window. Expressions can include constants, variables, and DO files. If no expression is specified, the search will give an error. See "Expression syntax" (CR-24) for more information.
- **Search Options: Match Count**
  You can search for the nth transition or the nth match on value; **Match Count** indicates the number of transitions or matches to search for.

The **Search Results** are indicated at the bottom of the dialog box.

Using time cursors in the Wave window



When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. You can add cursors to the waveform pane by selecting **Insert > Cursor** (or the Add Cursor button shown below). The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines. Remove cursors by selecting them and selecting **Edit > Delete Cursor** (or the Delete Cursor button shown below).

	Insert Cursor add a cursor to the waveform window		**Delete Cursor** delete the selected cursor from the window

Naming cursors

By default cursors are named "Cursor <n>". To rename a cursor, click the name in the left-hand cursor pane with your right mouse button. Type a new name and press the <Enter> key on your keyboard.

Locking cursors

You can lock a cursor in position so it won't move. Select the cursor you wish to lock and select **Edit > Edit Cursor** (Wave window). In the dialog that appears, check **Lock cursor to specified time** and click OK. The cursor turns red and you can no longer drag it with the mouse.

As a convenience, you can hold down the <shift> key and click-and-drag a locked cursor. Once you let go of the cursor, it will be locked in the new position.

To unlock a cursor, select **Edit > Edit Cursor** and uncheck **Lock cursor to specified time**.

Finding cursors

The cursor value corresponds to the simulation time of that cursor. Choose a specific cursor view by selecting **View > Cursors**.

You can also access cursors by clicking a name or value in the left-hand cursor pane. Single-clicking selects a cursor; double-clicking jumps to a cursor. Alternatively, you can click a value with your second mouse button and type the value to which you want to scroll.

Making cursor measurements

Each cursor is displayed with a time box showing the precise simulation time at the bottom. When you have more than one cursor, each time box appears in a separate track at the bottom of the display. ModelSim also adds a delta measurement showing the time difference between two adjacent cursor positions.

If you click in the waveform display, the cursor closest to the mouse position is selected and then moved to the mouse position. Another way to position multiple cursors is to use the mouse in the time box tracks at the bottom of the display. Clicking anywhere in a track selects that cursor and brings it to the mouse position.

Cursors will "snap" to a waveform edge if you click or drag a cursor to within ten pixels of a waveform edge. You can set the snap distance in the Window Preferences dialog (select **Tools > Window Preferences**). You can position a cursor without snapping by dragging in the cursor track below the waveforms.

You can also move cursors to the next transition of a signal with these toolbar buttons:

←	**Find Previous Transition** locate the previous signal value change for the selected signal	→	**Find Next Transition** locate the next signal value change for the selected signal

Examining waveform values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See "Setting Wave window display properties" (UM-352).
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse.

Zooming - changing the waveform display range

Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

You can access Zoom commands from the **View** menu on the toolbar or by clicking the right mouse button in the waveform pane.

The **Zoom** menu options include:

- **Zoom Full**
  Redraws the display to show the entire simulation from time 0 to the current simulation time.
- **Zoom In**
  Zooms in by a factor of two, increasing the resolution and decreasing the visible range horizontally.
- **Zoom Out**
  Zooms out by a factor of two, decreasing the resolution and increasing the visible range horizontally.
- **Zoom Last**
  Restores the display to where it was before the last zoom operation.
- **Zoom Range**
  Brings up a dialog box that allows you to enter the beginning and ending times for a range of time units to be displayed.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

🔍	**Zoom In 2x** zoom in by a factor of two from the current view	🔍	**Zoom Out 2x** zoom out by a factor of two from current view
🔍	**Zoom Full** zoom out to view the full range of the simulation from time 0 to the current time	⧉	**Zoom Mode** change mouse pointer to zoom mode; see below

Zooming with the mouse

To zoom with the mouse, first enter zoom mode by selecting **View > Mouse Mode > Zoom Mode** (Wave window). The left mouse button (<Button-1>) then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

You can also enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.

With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

Zooming keyboard shortcuts

See "Wave window mouse and keyboard shortcuts" (UM-363) for a complete list of Wave window keyboard shortcuts.

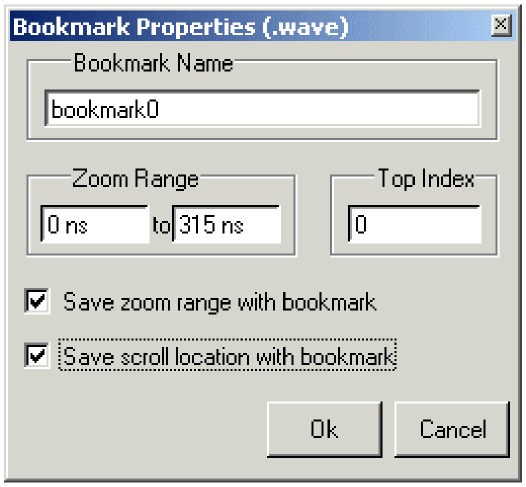
Saving zoom range and scroll position with bookmarks

Bookmarks allow you to save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name, and then access the named bookmark from the Bookmark menu.

Bookmarks are saved in the Wave format file (see "Adding items with a Wave window format file" (UM-339)) and are restored when the format file is read. There is no limit to the number of bookmarks you can save.

Bookmarks can also be created and managed from the command line. See the **bookmark add wave** command (CR-77) for details.

To add a bookmark, select **Insert > Bookmark** (Wave window).


Bookmark Properties (.wave)

Bookmark Name: bookmark0

Zoom Range: 0 ns to 315 ns

Top Index: 0

☑ Save zoom range with bookmark

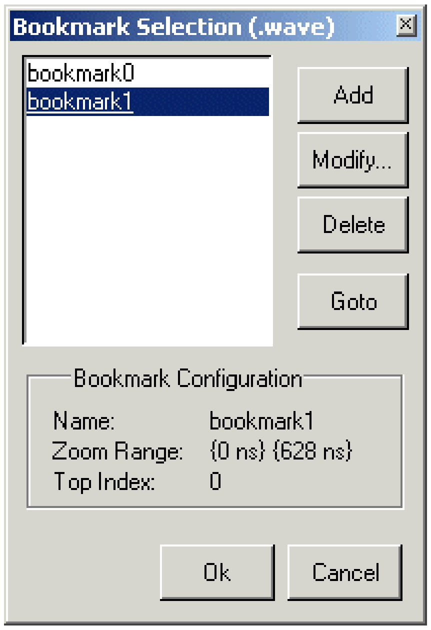
☑ Save scroll location with bookmark

The Bookmark Properties dialog includes the following options.

- **Bookmark Name**
  A text label to assign to the bookmark. The name will identify the bookmark on the **View** > **Bookmarks** menu.
- **Zoom Range**
  A starting value and ending value that define the zoom range.
- **Top Index**
  The item that will display at the top of the Wave window. For instance, if you specify 15, the Wave window will be scrolled down to show the 15th item in the window.
- **Save zoom range with bookmark**
  When checked the zoom range will be saved in the bookmark.
- **Save scroll location with bookmark**
  When checked the scroll location will be saved in the bookmark.

Once the bookmark is saved, select it by name from the **View** > **Bookmarks** menu, and the Wave window will be zoomed and scrolled accordingly.

To edit or delete a bookmark, select **Tools** > **Bookmarks** (Wave window).



Bookmark Configuration

Name: bookmark1
Zoom Range: {0 ns} {628 ns}
Top Index: 0

The Bookmark Selection dialog includes the following options.

- **Add** (bookmark add wave)
  Add a new bookmark.
- **Modify**
  Edit the selected bookmark.
- **Delete** (bookmark delete wave)
  Delete the selected bookmark.
- **Goto** (bookmark goto wave)
  Zoom and scroll the Wave window using the selected bookmark.

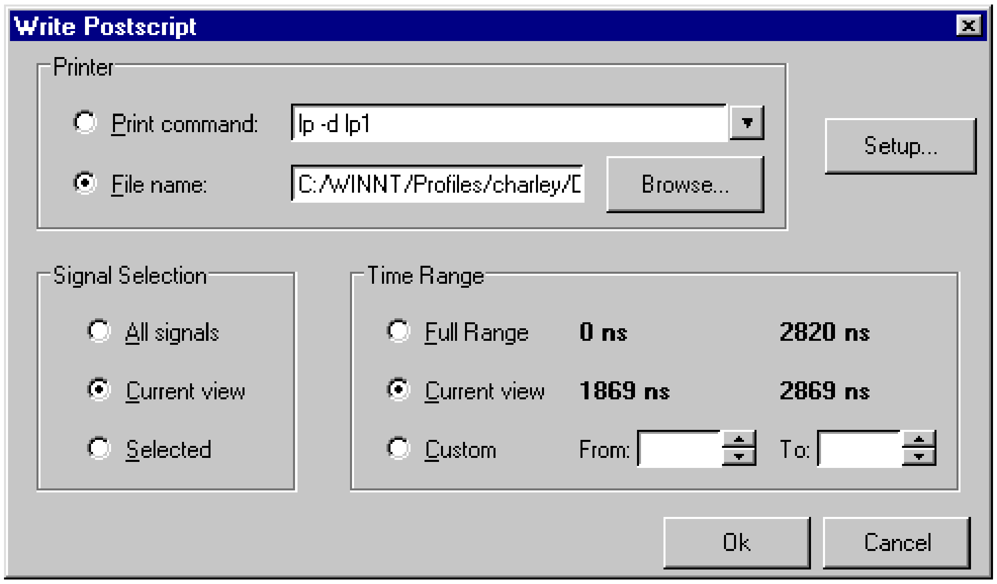
Wave window mouse and keyboard shortcuts

See "Wave window mouse and keyboard shortcuts" (UM-643).

Printing and saving waveforms

Saving a .eps file and printing under UNIX

Select **File** > **Print Postscript** (Wave window) to print all or part of the waveform in the current Wave window in UNIX, or save the waveform as a .eps file on any platform (see also the **write wave** command (CR-397)). Printing and writing preferences are controlled by the dialog box shown below.


Write Postscript

Printer
- Print command: lp -d lp1
- [✓ File name:] C:/WINNT/Profiles/charley/D

Signal Selection
- All signals
- [✓ Current view]
- Selected

Time Range
- Full Range 0 ns 2820 ns
- [✓ Current view] 1869 ns 2869 ns
- Custom From: To:

The **Write Postscript** dialog box includes these options:

Printer

- **Print command**
  Enter a UNIX print command to print the waveform in a UNIX environment.
- **File name**
  Enter a filename for the encapsulated Postscript (.eps) file to be created; or browse to a previously created .eps file and use that filename.

Signal Selection

- **All signals**
  Print all signals.
- **Current View**
  Print signals in the current view.
- **Selected**
  Print all selected signals.

Time Range

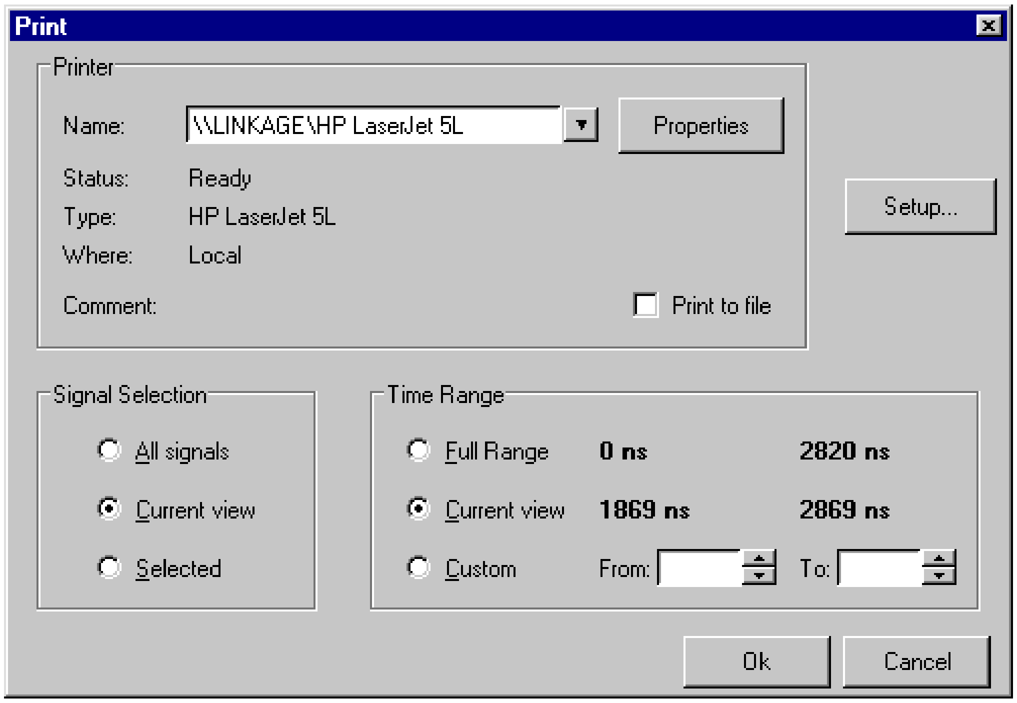
- **Full Range**
  Print all specified signals in the full simulation range.
- **Current view**
  Print the specified signals for the viewable time range.
- **Custom**
  Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "Printer Page Setup" (UM-366)

Printing on Windows platforms

Select **File > Print** (Wave window) to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers). Printing and writing preferences are controlled by the dialog box shown below.


Print

Printer

Name: \\LINKAGE\HP LaserJet 5L

Status: Ready

Type: HP LaserJet 5L

Where: Local

Comment:

[ ] Print to file

Signal Selection: All signals [✓ Current view] Selected

Time Range:

Full Range	0 ns	2820 ns
[✓ Current view]	1869 ns	2869 ns
Custom	From:	To:

Printer

- **Name**
  Choose the printer from the drop-down menu. Set printer properties with the **Properties** button.
- **Status**
  Indicates the availability of the selected printer.

- **Type**
  Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.
- **Where**
  The printer port for the selected printer.
- **Comment**
  The printer comment from the printer properties dialog box.
- **Print to file**
  Make this selection to print the waveform to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers create a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the **File > Print Postscript** menu selection.

Signal Selection

- **All signals**
  Print all signals.
- **Current View**
  Print signals in current view.
- **Selected**
  Print all selected signals.

Time Range

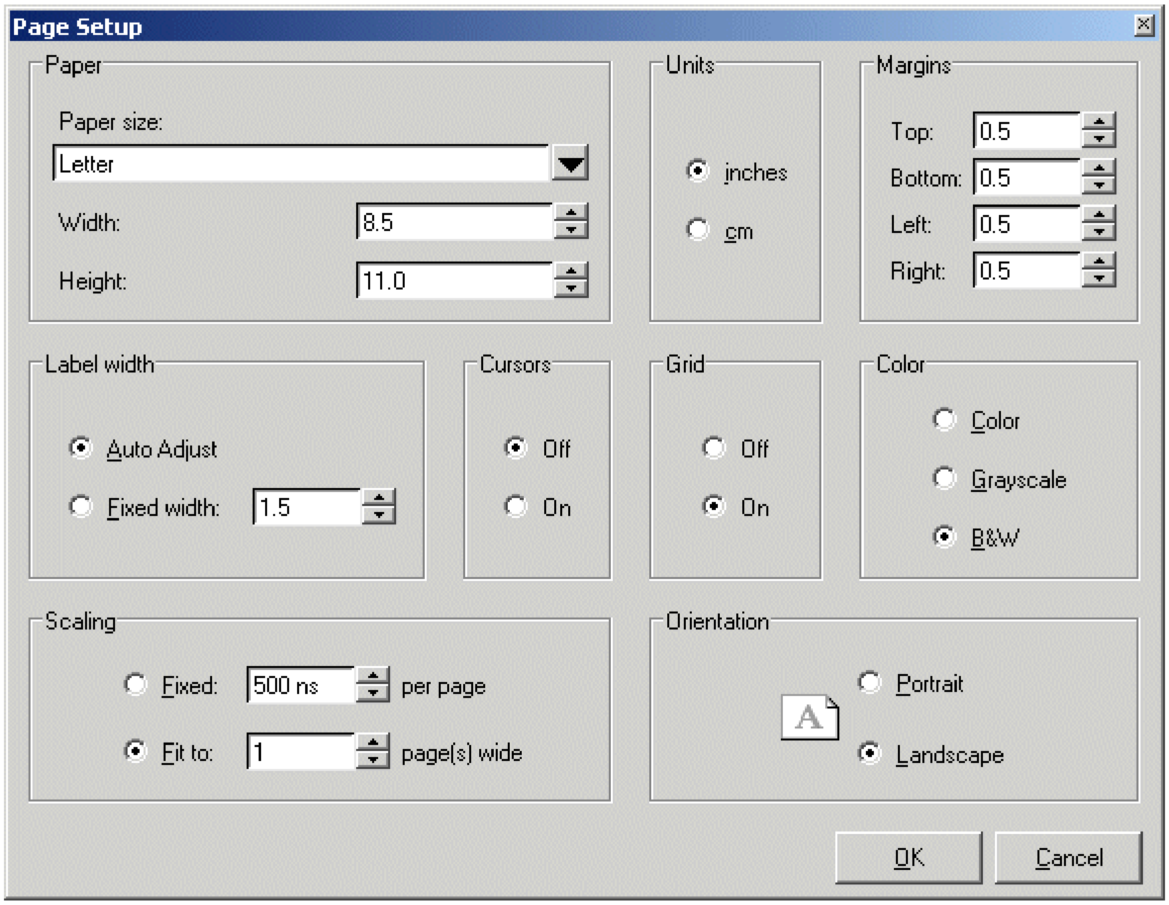
- **Full Range**
  Print all specified signals in the full simulation range.
- **Current view**
  Print the specified signals for the viewable time range.
- **Custom**
  Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "Printer Page Setup" (UM-366)

Printer Page Setup

Clicking the Setup button in the Write Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens via **File > Page setup**).



Page Setup

Paper
Paper size: Letter
Width: 8.5
Height: 11.0

Units
[✓ inches] cm

Margins
Top: 0.5
Bottom: 0.5
Left: 0.5
Right: 0.5

Label width
[✓ Auto Adjust] Fixed width: 1.5

Cursors
[✓ Off] On

Grid
Off [✓ On]

Color
Color Grayscale [✓ B&W]

Scaling
Fixed: 500 ns per page
[✓ Fit to:] 1 page(s) wide

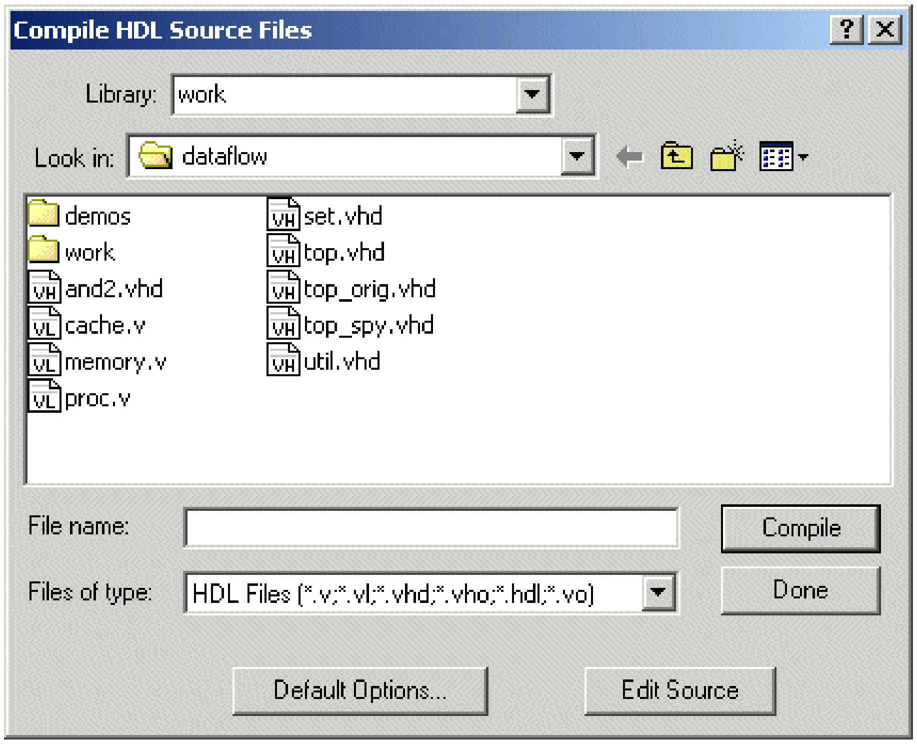
Orientation
Portrait [✓ Landscape]

- **Paper Size**
  Select your output page size from a number of options; also choose the paper width and height.
- **Units**
  Specify whether measurements are in inches or centimeters.
- **Margins**
  Specify the page margins; changing the **Margin** will change the **Scale** and **Page** specifications.
- **Label width**
  Specify Auto Adjust to accommodate any length label, or set a fixed label width.
- **Cursors**
  Turn printing of cursors on or off.

- **Grid**
  Turn printing of grid lines on or off.
- **Color**
  Select full color printing, grayscale, or black and white.
- **Scaling**
  Specify a **Fixed** output time width in nanoseconds per page – the number of pages output is automatically computed; or, select **Fit to** to define the number of pages to be output based on the paper size and time settings; if set, the time-width per page is automatically computed.
- **Orientation**
  Select the output page orientation, **Portrait** or **Landscape**.

Compiling with the graphic interface

You can use a project or the **Compile Source Files** dialog box to compile VHDL or Verilog designs. For information on compiling in a project, see "Getting started with projects" (UM-34). To open the Compile Source Files dialog, select **Compile > Compile** (Main window).



From the Compile Source Files dialog box you can:

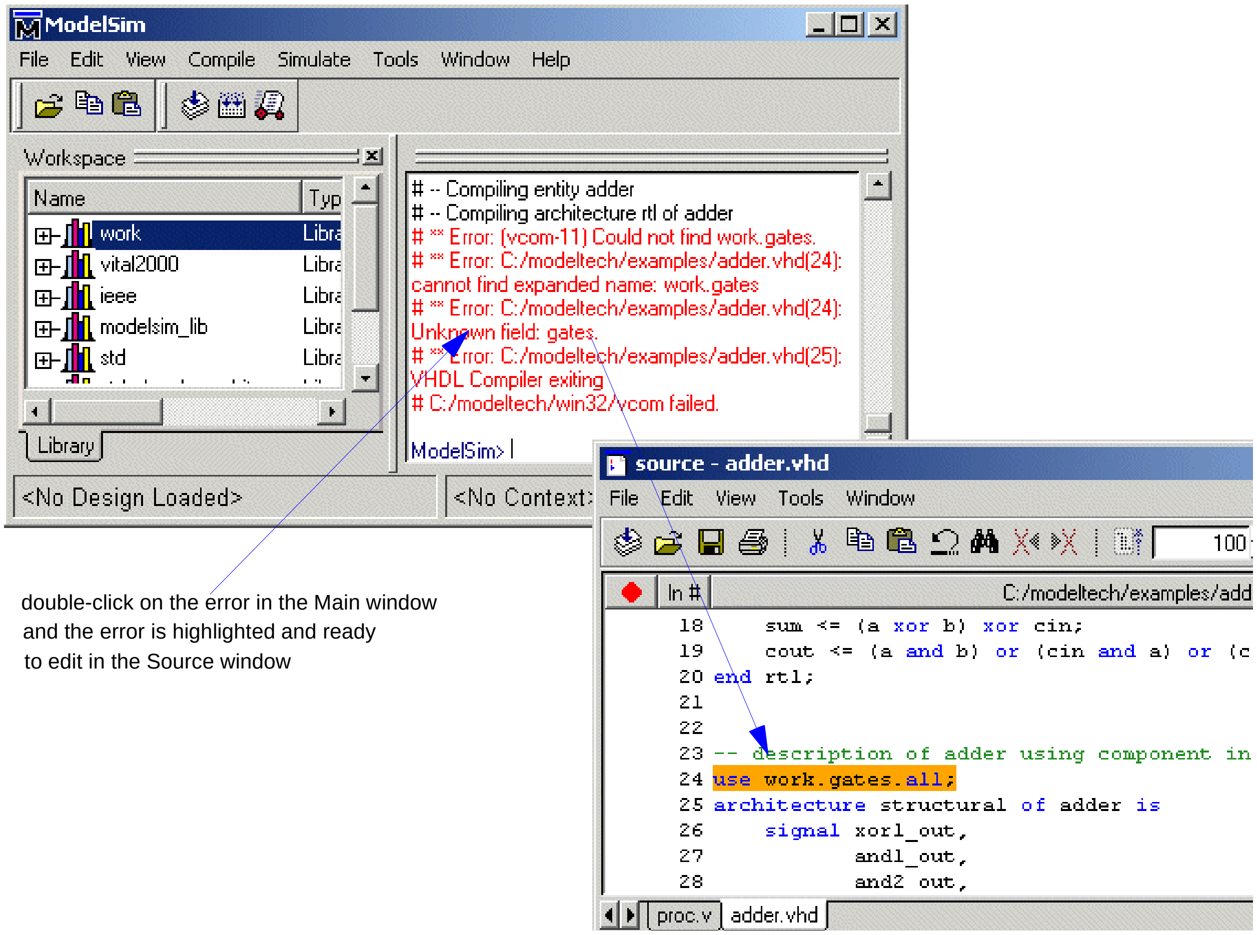
- select source files to compile in any language combination
- specify the target library for the compiled design units
- select among the compiler options for VHDL, Verilog, or SystemC

Select the **Default Options** button to change the compiler options, see "Setting default compile options" (UM-370) for details. The same Compiler Options dialog box can also be accessed by selecting **Compile > Compile Options** (Main window) or by selecting Compile Properties from the context menu in the Project tab.

Select the **Edit Source** button to view or edit a source file via the Compile dialog box. See "Source window" (UM-325) for additional source file editing information.

Locating source errors during compilation

If a compiler error occurs during compilation, a red error message is printed in the Main transcript. Double-click on the error message to open the source file in an editable Source window with the error highlighted.



```
# -- Compiling entity adder
# -- Compiling architecture rtl of adder
# ** Error: (vcom-11) Could not find work.gates.
# ** Error: C:/modeltech/examples/adder.vhd(24):
cannot find expanded name: work.gates
# ** Error: C:/modeltech/examples/adder.vhd(24):
Unknown field: gates.
# ** Error: C:/modeltech/examples/adder.vhd(25):
VHDL Compiler exiting
# C:/modeltech/win32/vcom failed.

ModelSim>
```

```
    sum <= (a xor b) xor cin;
    cout <= (a and b) or (cin and a) or (c
end rtl;


-- description of adder using component in
use work.gates.all;
architecture structural of adder is
    signal xor1_out,
           and1_out,
           and2_out,
```

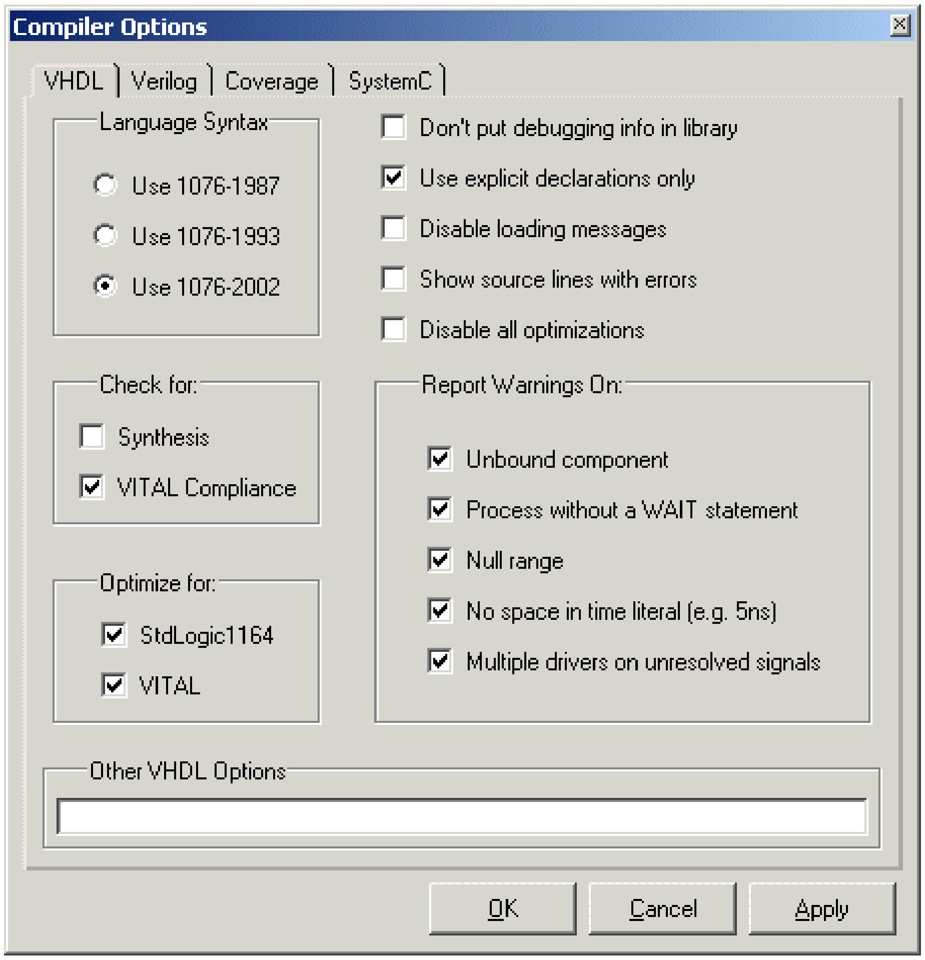
double-click on the error in the Main window
and the error is highlighted and ready
to edit in the Source window

Setting default compile options

Select **Compile** > **Compile Options** (Main window) to bring up the Compiler Options dialog.

Important: Note that changes made in the **Compiler Options** dialog box become the default for all future simulations.

VHDL compiler options tab



Compiler Options

VHDL | Verilog | Coverage | SystemC

Language Syntax
- [ ] Use 1076-1987
- [ ] Use 1076-1993
- [✓ Use 1076-2002]

- [ ] Don't put debugging info in library
- [✓] Use explicit declarations only
- [ ] Disable loading messages
- [ ] Show source lines with errors
- [ ] Disable all optimizations

Check for:
- [ ] Synthesis
- [✓] VITAL Compliance

Report Warnings On:
- [✓] Unbound component
- [✓] Process without a WAIT statement
- [✓] Null range
- [✓] No space in time literal (e.g. 5ns)
- [✓] Multiple drivers on unresolved signals

Optimize for:
- [✓] StdLogic1164
- [✓] VITAL

Other VHDL Options

OK Cancel Apply

The VHDL compiler options tab includes the following options:

- **Language Syntax**
  Specifies which version of the 1076 standard to use when compiling. The default for versions 5.8 and later is 2002. You can also set this with arguments to the **vcom** command (CR-303) or by editing the VHDL standard (UM-630) variable in the *modelsim.ini* file. Changing the setting in the *modelsim.ini* file will make the setting permanent.

- **Don’t put debugging info in library**
  Models compiled with this option do not use any of the ModelSim debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don’t compile with this option until you are done debugging. Same as the **-nodebug** argument to the **vcom** command (CR-303). Edit the NoDebug (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Use explicit declarations only**
  Used to ignore an error in packages supplied by some other EDA vendors; directs the compiler to resolve ambiguous function overloading in favor of the explicit function definition. Same as the **-explicit** argument to the **vcom** command (CR-303). Edit the Explicit (UM-619) variable in the *modelsim.ini* file to set a permanent default.

  Although it is not intuitively obvious, the = operator is overloaded in the **std_logic_1164** package. All enumeration data types in VHDL get an “implicit” definition for the = operator. So while there is no explicit = operator, there is an implicit one. This implicit declaration can be hidden by an explicit declaration of = in the same package (LRM Section 10.3). However, if another version of the = operator is declared in a different package than that containing the enumeration declaration, and both operators become visible through **use** clauses, neither can be used without explicit naming, for example:

  ```
  ARITHMETIC.”=”(left, right)
  ```

  This option allows the explicit = operator to hide the implicit one.
- **Disable loading messages**
  Disables loading messages in the Main window. Same as the **-quiet** argument for the **vcom** command (CR-303). Edit the Quiet (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Show source lines with errors**
  Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** argument to the **vcom** command (CR-303). Edit the Show_source (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Disable all optimizations**
  Instructs the compiler to remove all optimizations. Same as the **-O0** argument to the **vcom** command (CR-303). Useful when running "Code Coverage" (UM-419), where optimizations can skew results.

Check for:

- **Synthesis**
  Turns on limited synthesis-rule compliance checking. Checks only signals used (read) by a process; also, checks understand only combinational logic, not clocked logic. Edit the CheckSynthesis (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Vital Compliance**
  Toggle Vital compliance checking. Edit the NoVitalCheck (UM-619) variable in the *modelsim.ini* file to set a permanent default.

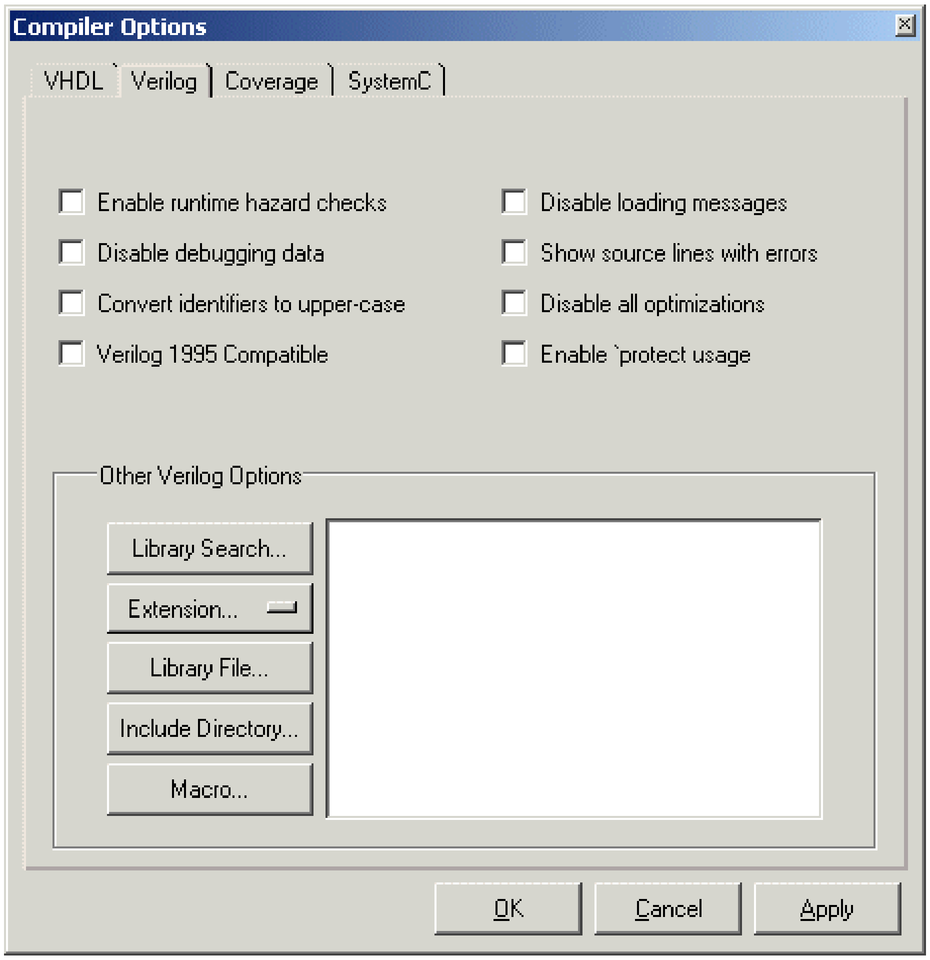
Report Warnings on:

- **Unbound component**
  Flags any component instantiation in the VHDL source code that has no matching entity in a library that is referenced in the source code, either directly or indirectly. Edit the Show_Warning1 (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Process without a WAIT statement**
  Flags any process that does not contain a wait statement or a sensitivity list. Edit the Show_Warning2 (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Null range**
  Flags any null range, such as 0 down to 4. Edit the Show_Warning3 (UM-620) variable in the *modelsim.ini* file to set a permanent default.
- **No space in time literal (e.g. 5ns)**
  Flags any time literal that is missing a space between the number and the time unit. Edit the Show_Warning4 (UM-620) variable in the *modelsim.ini* file to set a permanent default.
- **Multiple drivers on unresolved signals**
  Flags any unresolved signals that have multiple drivers. Edit the Show_Warning5 (UM-620) variable in the *modelsim.ini* file to set a permanent default.

Optimize for:

- **StdLogic1164**
  Causes the compiler to perform special optimizations for speeding up simulation when the multi-value logic package std_logic_1164 is used. Unless you have modified the std_logic_1164 package, this option should always be checked. Edit the Optimize_1164 (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Vital**
  Toggle acceleration of the Vital packages. Edit the NoVital (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Other VHDL options**
  Enter any other valid **vcom** arguments. See the **vcom** command (CR-303) in the *ModelSim Command Reference* for a complete list.

Verilog compiler options tab



- **Enable runtime hazard checks**
  Enables the run-time hazard checking code. Same as the **-hazards** argument to the **vlog** command (CR-345). Edit the Hazard (UM-618) variable in the *modelsim.ini* file to set a permanent default.
- **Disable debugging data**
  Models compiled with this option do not use any of the ModelSim debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you are done debugging. Same as the **-nodebug** argument for the **vlog** command (CR-345). Edit the NoDebug (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Convert identifiers to upper-case**
  Converts regular Verilog identifiers to uppercase. Allows case insensitivity for module names. Same as the **-u** argument to the **vlog** command (CR-345). Edit the UpCase (UM-618) variable in the *modelsim.ini* file to set a permanent default.

- **Verilog 1995 Compatible**
  Some requirements in Verilog 2001 conflict with requirements in the 1995 LRM. Use of this option ensures that code that was valid according to the 1995 LRM can still be compiled. Same as the **-vlog95compat** argument to the **vlog** command (CR-345). Edit the vlog95compat (UM-618) variable in the *modelsim.ini* file to set a permanent default.
- **Disable loading messages**
  Disables loading messages in the Main window. Same as the **-quiet** argument for the **vlog** command (CR-345). Edit the Quiet (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Show source lines with errors**
  Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** argument to the **vlog** command (CR-345). Edit the Show_source (UM-619) variable in the *modelsim.ini* file to set a permanent default.
- **Disable all optimizations**
  Instructs the compiler to remove all optimizations. Same as the **-O0** argument to the **vlog** command (CR-345). Useful when running "Code Coverage" (UM-419), where optimizations can skew results.
- **Enable `protect usage**
  Enables encryption of regions of your Verilog source code. See "ModelSim compiler directives" (UM-152) for more details. Same as the **+protect** argument for the **vlog** command (CR-345). Edit the Protect (UM-618) variable in the *modelsim.ini* file to set a permanent default.

Other Verilog Options:

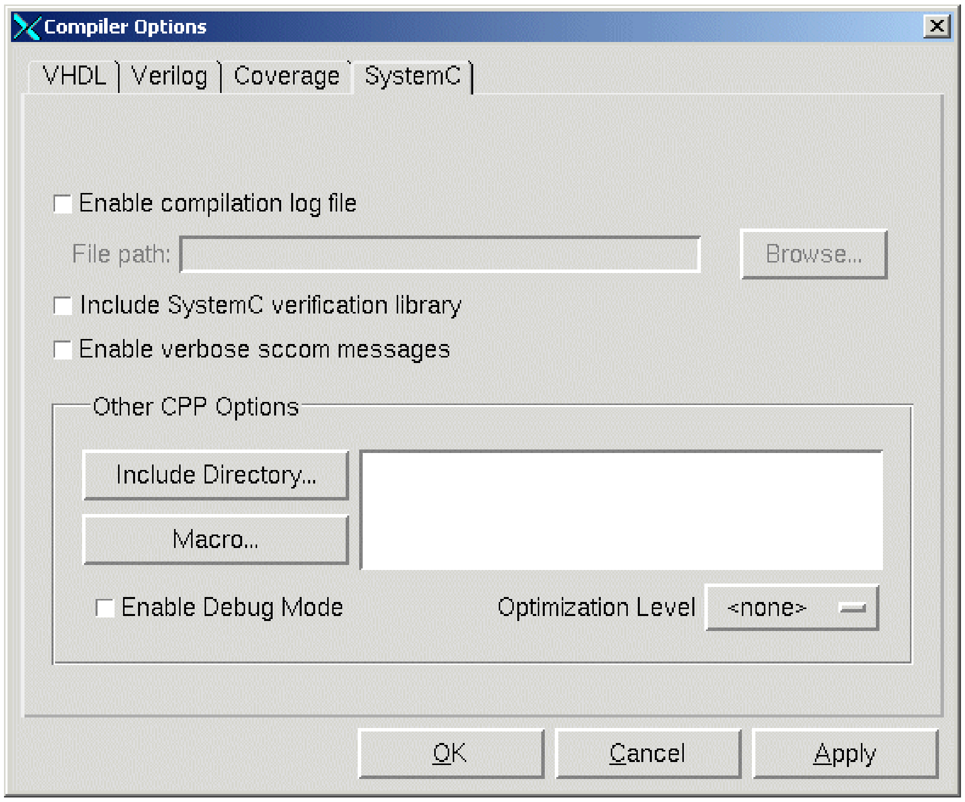
Specify any valid **vlog** command (CR-345) arguments. When you specify Other Verilog Options, they are saved into a file called *vlog.opt*. If you do this while a project is open, an OptionFile entry is written into your project file. If you do this when a project is not open, an OptionFile entry is written into the *modelsim.ini* file that you are currently using.

- **Library Search**
  Specifies the Verilog source library directory to search for undefined modules. Same as the **-y <library_directory>** argument for the **vlog** command (CR-345).
- **Extension**
  Specifies the suffix of files in the library directory. Multiple suffixes can be used. Same as the **+libext+<suffix>** argument for the **vlog** command (CR-345).
- **Library File**
  Specifies the Verilog source library file to search for undefined modules. Same as the **-v <library_file>** argument for the **vlog** command (CR-345).
- **Include Directory**
  Specifies a directory for files included with the **‘include filename** compiler directive. Same as the **+incdir+<directory>** argument for the **vlog** command (CR-345).
- **Macro**
  Defines a macro to execute during compilation. Same as the compiler directive: ‘define macro_name macro_text. Also the same as the **+define+<macro_name> [=<macro_text>]** argument for the **vlog** command (CR-345).

Coverage compiler options tab

The options on this tab are described in the section "Enabling Code Coverage" (UM-423).

SystemC compiler options tab



- **Enable compilation log file**
  Writes the compilation output to a file name, specified in the **File path** field. Same as the **-log** argument to the **sccom** command (CR-248).
- **Include SystemC verification library**
  Includes the SystemC verification library. Same as the **-scv** argument to the **sccom** command (CR-248).
- **Enable verbose sccom messages**
  Echoes subprocess invocations with command arguments. Same as the **-verbose** argument to the **sccom** command (CR-248).

Other CPP Options

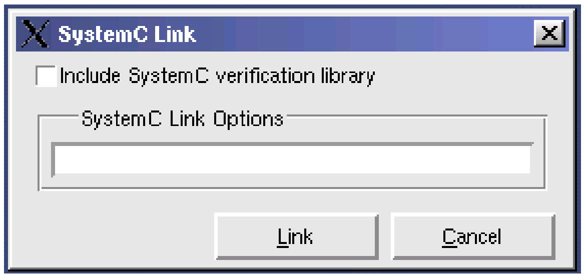
Specify any valid g++/aCC compiler options. All options are accepted, with the exception of the **-o** and **-c** options.

- **Include Directory**
  Includes a directory that contains source files. Same as the **-I** argument to g++/aCC.
- **Macro**
  Defines a macro. Same as the **-D** argument to g++/aCC.

- **Enable Debug Mode**
  Compiles SystemC code with debugging information. By default SystemC code is compiled without debugging information. Same as the **-g** argument to g++/aCC.
- **Optimization level**
  Specify optimization value you wish to use. By default, no optimization is performed. Same as the **-O#** argument to g++/aCC.

Setting SystemC link options

Before you can simulate a SystemC design, you must link the design. The SystemC linking collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library. To link the design using the GUI, select **Compile -> SystemC Link**. A dialog box opens, allowing you to enter any g++/aCC linking options your design requires.



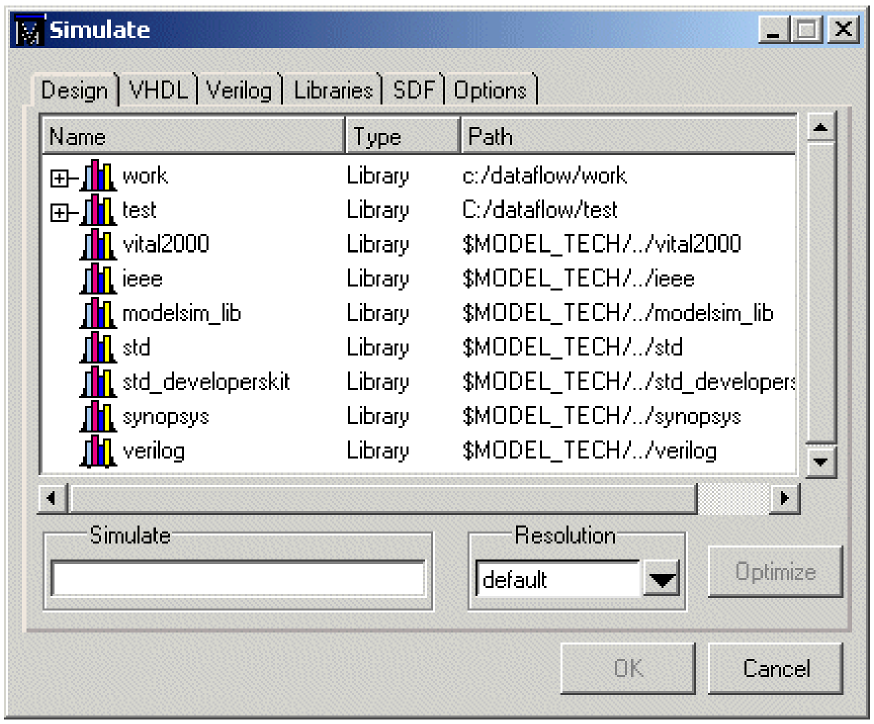
- **Include SystemC verification library**
  Includes the SystemC verification library. Same as the **-scv** argument to the **sccom** command (CR-248).
- **SystemC Link Options**
  Specify any valid g++/aCC linking options (e.g. -l, -L, etc.). All options are accepted.

Simulating with the graphic interface

You can use the Library tab in the workspace or the **Simulate** dialog box to simulate a compiled design. To simulate from the Library tab, simply double-click a design unit. To open the **Simulate** dialog, select **Simulate > Simulate** (Main window).

Six tabs - **Design**, **VHDL**, **Verilog**, **Libraries**, **SDF**, and **Options** - allow you to select various simulation options. You can switch between tabs to modify settings, then begin simulation by selecting the **OK** button.

Design tab



Name	Type	Path
work	Library	c:/dataflow/work
test	Library	C:/dataflow/test
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_developers
synopsys	Library	$MODEL_TECH/../synopsys
verilog	Library	$MODEL_TECH/../verilog

Resolution: default

The **Design** tab includes these options:

- **Simulate**
  Specifies the design unit(s) to simulate. You can simulate your Verilog top-level module(s), a VHDL top-level design unit, or your SystemC top-level module(s) in one of two ways:

  - Type a design unit name (configuration, module, or entity) into the field, separate additional names with a space. Specify library/design units with the following syntax:

  ```
  [<library_name>.]<design_unit>
  ```

  - Select a design unit from the list. You can select multiple design units from the list by using the control key when you click.

- **Resolution**
  (-t [<multiplier>]<time_unit>)
  The drop-down menu sets the simulator time units.

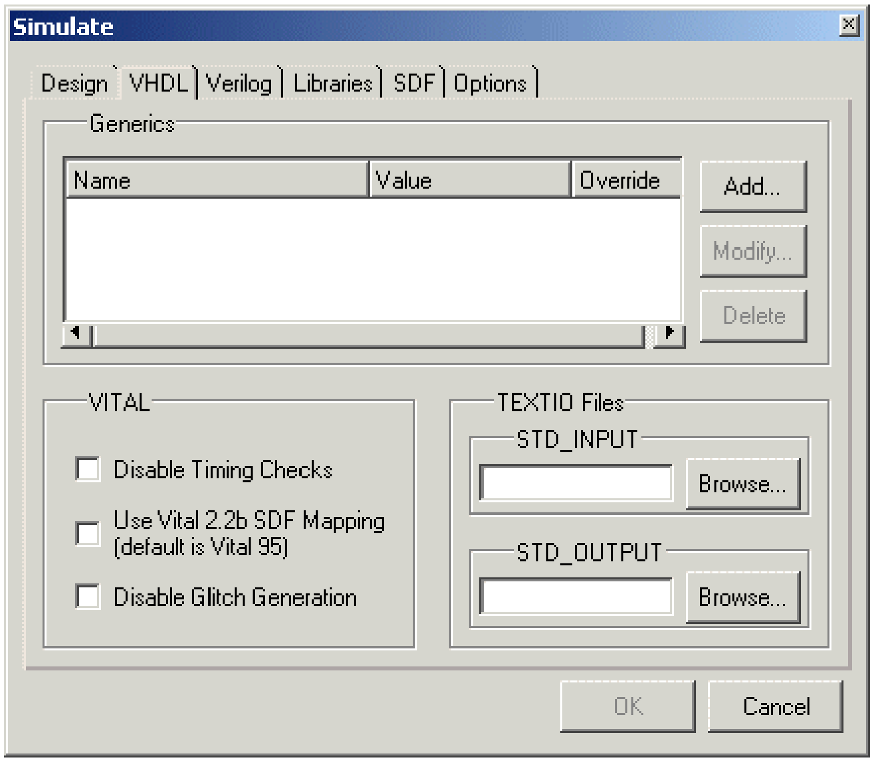
  Simulator time units can be expressed as any of the following:

Simulation time units	
1fs, 10fs, or 100fs	femtoseconds
1ps, 10ps, or 100ps	picoseconds
1ns, 10ns, or 100ns	nanoseconds
1us, 10us, or 100us	microseconds
1ms, 10ms, or 100ms	milliseconds
1sec, 10sec, or 100sec	seconds

  See also, "Simulator resolution limit" (UM-77).

- **Optimize**
  Recompile the selected Verilog design unit using +opt optimizations. Please read "Compiling for faster performance" (UM-127) before using this option.

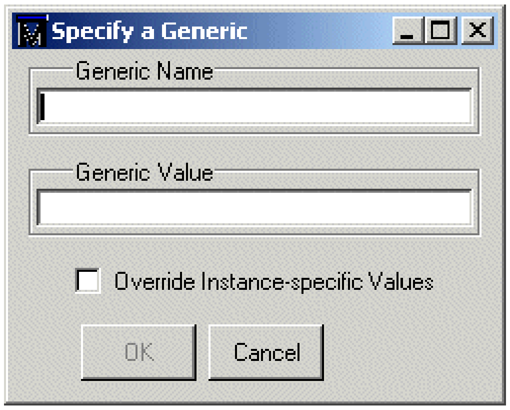
VHDL tab



The **VHDL** tab includes these options:

Generics

The **Add** button opens a dialog box (shown below) that allows you to specify the value of generics within the current simulation; generics are then added to the **Generics** list. You can also select a generic on the listing to **Delete** or **Edit**.



From the **Specify a Generic** dialog box you can set the following options.

- **Generic Name** (-g <Name>=<Value>)
  The name of the generic parameter. Type it in as it appears in the VHDL source (case is ignored).
- **Generic Value**
  Specifies a value for all generics in the design with the given name (above) that have not received explicit values in generic maps (such as top-level generics and generics that

would otherwise receive their default value). The value must be appropriate for the declared data type of the generic. No spaces are allowed in the specification (except within quotes) when specifying a string value.

- **Override Instance - specific Values** (-G <Name>=<Value>)
  Select to override generics that received explicit values in generic maps. The name and value are specified as above. The use of this switch is indicated in the **Override** column of the **Generics** list.

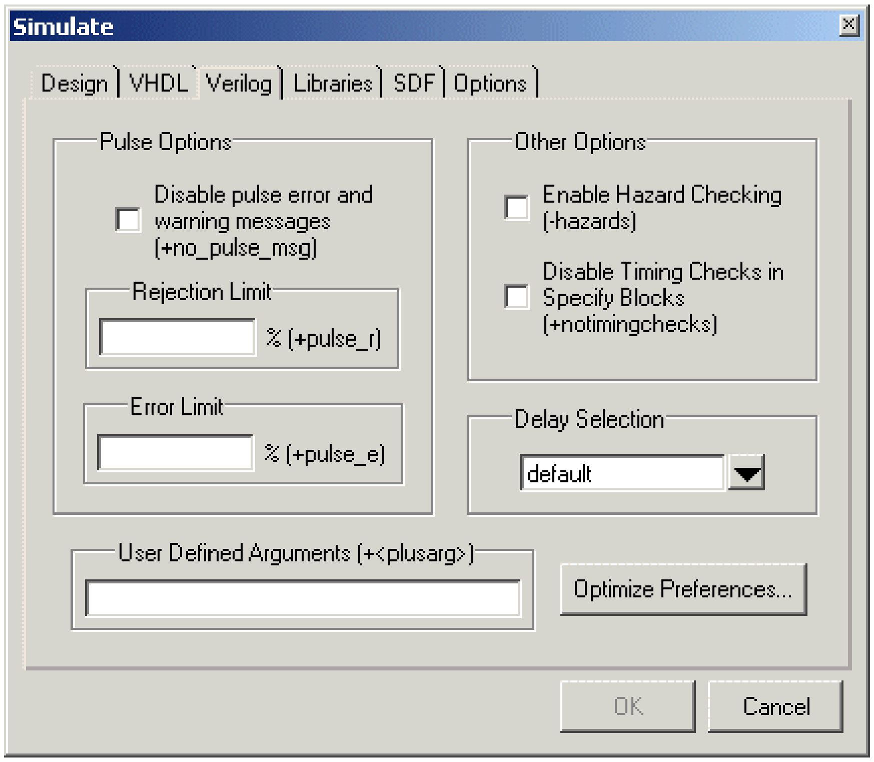
VITAL

- **Disable Timing Checks** (+notimingchecks)
  Disables timing checks generated by VITAL models.
- **Use Vital 2.2b SDF Mapping** (-vital2.2b)
  Selects SDF mapping for VITAL 2.2b (default is Vital95).
- **Disable Glitch Generation** (-noglitch)
  Disables VITAL glitch generation.

TEXTIO files

- **STD_INPUT** (-std_input <filename>)
  Specifies the file to use for the VHDL textio STD_INPUT file. Use the **Browse** button to locate a file within your directories.
- **STD_OUTPUT** (-std_output <filename>)
  Specifies the file to use for the VHDL textio STD_OUTPUT file. Use the **Browse** button to locate a file within your directories.

Verilog tab


Simulate
Design
VHDL
Verilog
Libraries
SDF
Options
Pulse Options
Disable pulse error and warning messages (+no_pulse_msg)
Rejection Limit
% (+pulse_r)
Error Limit
% (+pulse_e)
Other Options
Enable Hazard Checking (-hazards)
Disable Timing Checks in Specify Blocks (+notimingchecks)
Delay Selection
default
User Defined Arguments (+<plusarg>)
Optimize Preferences...
OK
Cancel

The **Verilog** tab includes these options:

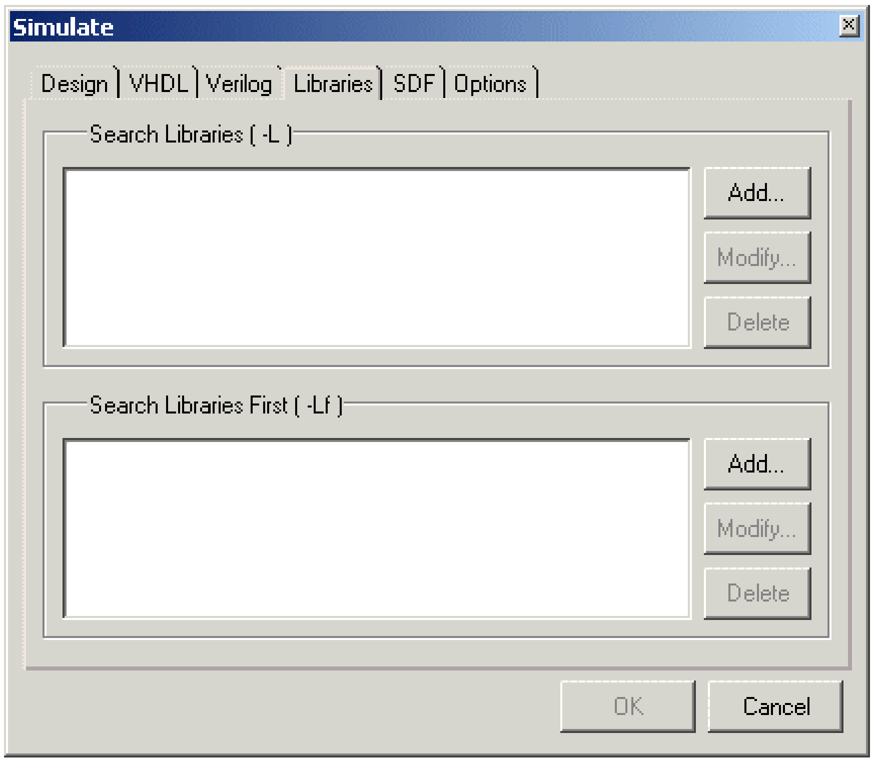
Pulse Options

- **Disable pulse error and warning messages** (+no_pulse_msg)
  Disables path pulse error warning messages.
- **Rejection Limit** (+pulse_r/<percent>)
  Sets the module path pulse rejection limit as a percentage of the path delay.
- **Error Limit** (+pulse_e/<percent>)
  Sets the module path pulse error limit as a percentage of the path delay.

Other Options

- **Enable Hazard Checking** (-hazards)
  Enables hazard checking in Verilog modules.
- **Disable Timing Checks in Specify Blocks** (+notimingchecks)
  Disables the timing check system tasks ($setup, $hold,...) in specify blocks.
- **Delay Selection** (+mindelays | +typdelays | +maxdelays)
  Use the drop-down menu to select timing for min:typ:max expressions.
- **User Defined Arguments** (+<plusarg>)
  Arguments are preceded with “+”, making them accessible through the Verilog PLI routine **mc_scan_plusargs**. The values specified in this field must have a "+" preceding them or ModelSim may parse them incorrectly.
- **Optimize Preferences** (-fast +acc)
  Enable design unit access for certain modules. See "Enabling design object visibility in optimized simulations" (UM-389) for details.

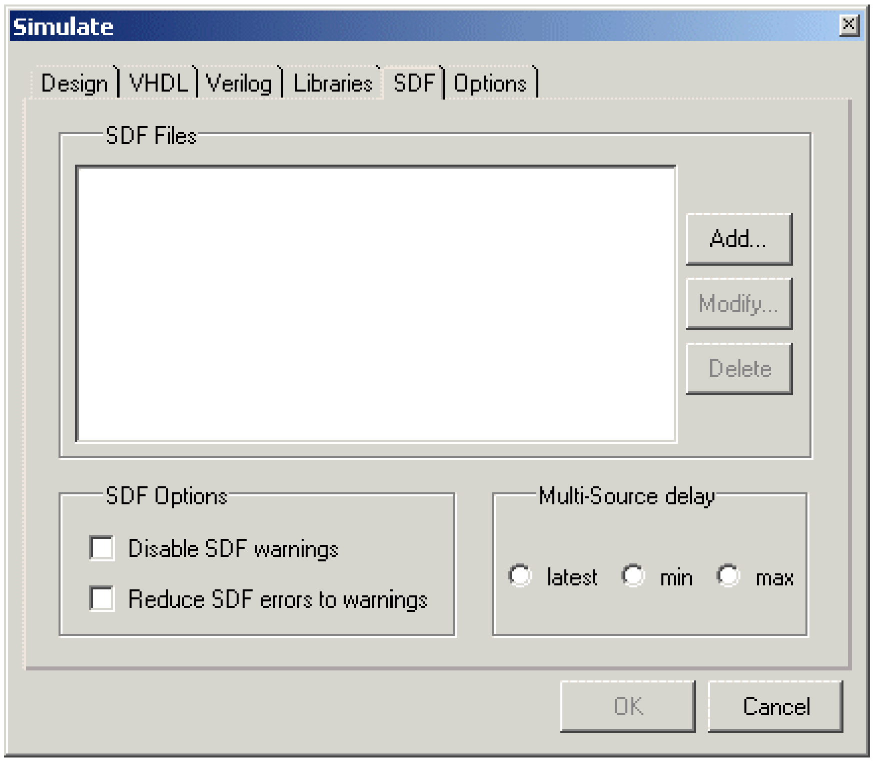
Libraries tab



The **Libraries** tab includes these options:

- **Search Libraries** (-L)
  Specifies the libraries to search for design units instantiated from Verilog.
- **Search Libraries First** (-Lf)
  Same as Search Libraries but these libraries are searched before ‘uselib.

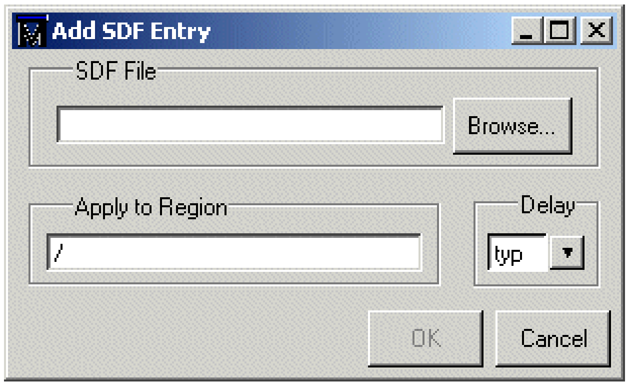
SDF tab



The **SDF** (Standard Delay Format) tab includes these options:

SDF Files

Click the **Add** button to specify the SDF files to load for the current simulation; files are then added to the **SDF Files** list. You may also select a file on the listing to **Delete** or **Modify** (opens the dialog box below).



Apply to Region: /
Delay: typ

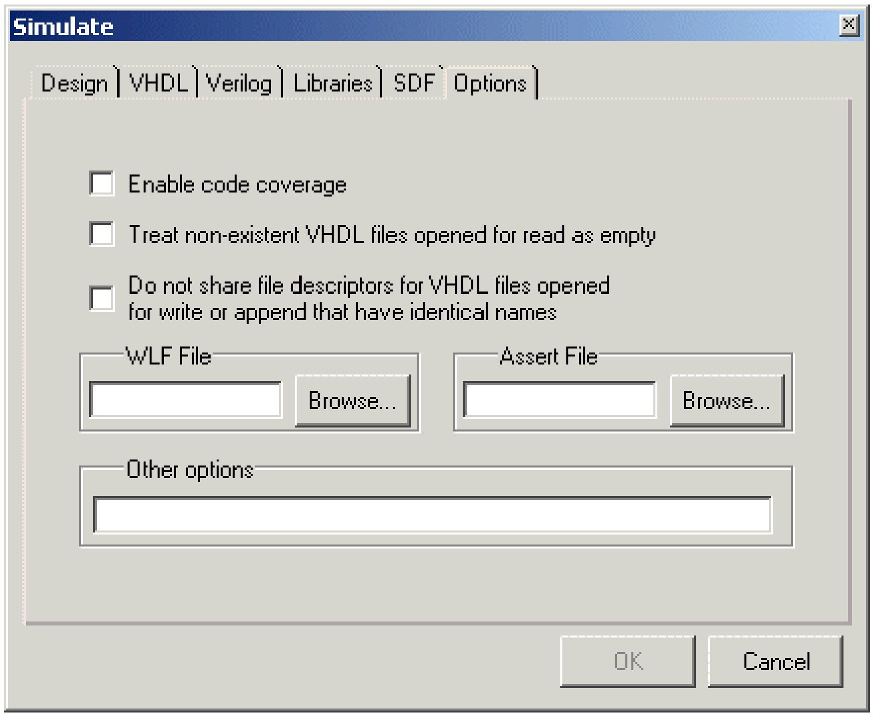
From the **Add SDF Entry** dialog box you can set the following options.

- **SDF file** ([<region>] = <sdf_filename>)
  Specifies the SDF file to use for annotation. Use the **Browse** button to locate a file within your directories.
- **Apply to region** ([<region>] = <sdf_filename>)
  Specifies the design region to use with the selected SDF options.
- **Delay** (-sdfmin | -sdftyp | -sdfmax)
  The drop-down menu selects delay timing (min, typ, or max) to be used from the specified SDF file. See also, "Specifying SDF files for simulation" (UM-544).

SDF options

- **Disable SDF warnings** (-sdfnowarn)
  Select to disable warnings from the SDF reader.
- **Reduce SDF errors to warnings** (-sdfnoerror)
  Change SDF errors to warnings so the simulation can continue.
- **Multi-Source Delay** (-multisource_delay <sdf_option>)
  Select **max**, **min**, or **latest** delay. Controls how multiple PORT or INTERCONNECT constructs that terminate at the same port are handled. By default, the Module Input Port Delay (MIPD) is set to the **max** value encountered in the SDF file. Alternatively, you can choose the **min** or **latest** of the values.

Options tab


Simulate
Design
VHDL
Verilog
Libraries
SDF
Options
Enable code coverage
Treat non-existent VHDL files opened for read as empty
Do not share file descriptors for VHDL files opened for write or append that have identical names
WLF File
Browse...
Assert File
Browse...
Other options
OK
Cancel

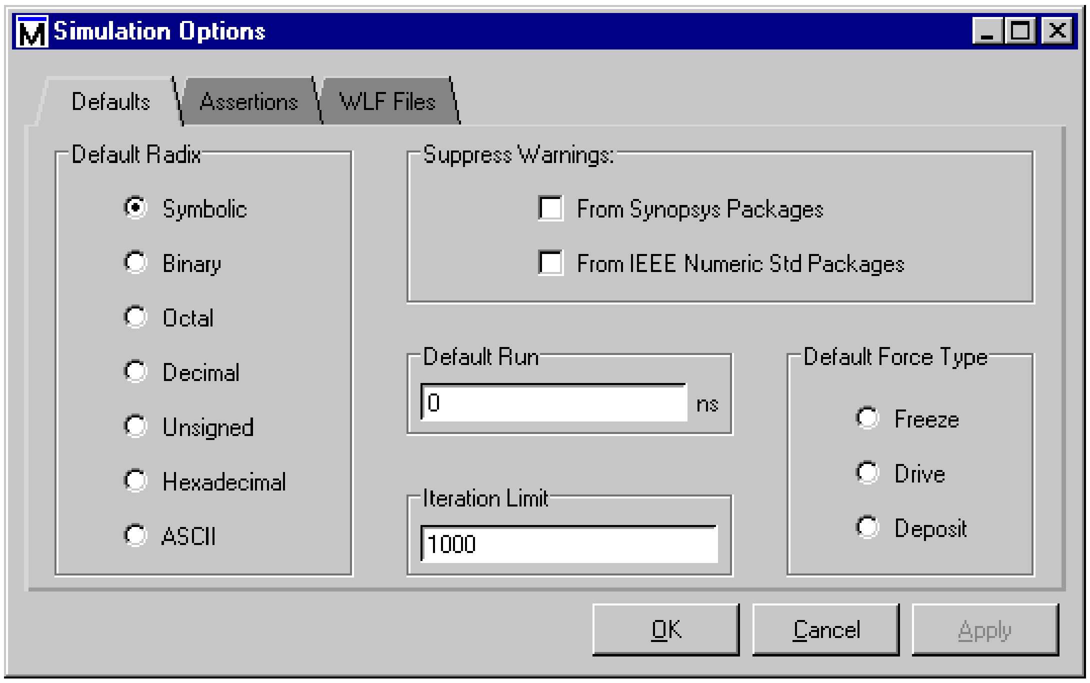
The **Options** tab includes these options:

- **Enable code coverage** (-coverage)
  Turn on collection of Code Coverage statistics. You must also specify which type of statistics you want when you compile the design. See *Chapter 12 - Code Coverage* for more information.
- **Treat non-existent VHDL files ...** (-absentisempty)
  Cause VHDL files opened for read that target non-existent files to be treated as empty, rather than ModelSim issuing fatal error messages.
- **Do not share file descriptors...** (-nofileshare)
  By default ModelSim shares a file descriptor for all VHDL files opened for write or append that have identical names. This option turns off file descriptor sharing.
- **WLF File** (-wlf <filename>)
  Specify the name of the wave log format (WLF) file to create. The default is vsim.wlf.
- **Assert File** (-assertfile <filename>)
  Designate an alternative file for recording assertion messages. By default assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file (see "Creating a transcript file" (UM-628)).
- **Other options**
  Specify any other **vsim** command (CR-357) arguments.

Setting default simulation options

Select **Simulate > Simulation Options** (Main window) to bring up the **Simulation Options** dialog box shown below. Changes made in the **Simulation Options** dialog box are the default for the current simulation only. Options can be saved as the default for future simulations by editing the simulator control variables in the *modelsim.ini* file; the variables to edit are noted in the text below.

Defaults tab



Default Radix: [✓ Symbolic] Binary Octal Decimal Unsigned Hexadecimal ASCII

Suppress Warnings: ☐ From Synopsys Packages ☐ From IEEE Numeric Std Packages

Default Run: 0 ns

Iteration Limit: 1000

Default Force Type: Freeze Drive Deposit

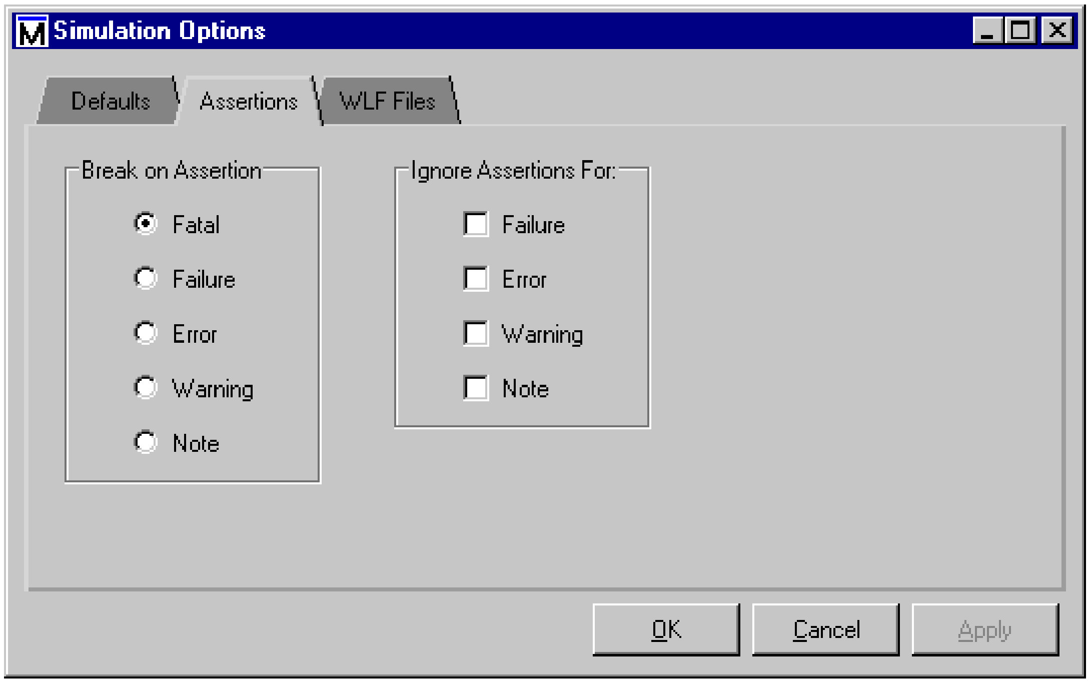
The **Defaults** tab includes these options:

- **Default Radix**
  Sets the default radix for the current simulation run. You can also use the **radix** (CR-235) command to set the same temporary default. A permanent default can be set by editing the DefaultRadix (UM-623) variable in the *modelsim.ini* file. The chosen radix is used for all commands (**force** (CR-176), **examine** (CR-167), **change** (CR-87) are examples) and for displayed values in the Signals, Variables, Dataflow, List, and Wave windows.
- **Suppress Warnings**
  Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. Edit the StdArithNoWarnings (UM-625) variable in the *modelsim.ini* file to set a permanent default.

  Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. Edit the NumericStdNoWarnings (UM-624) variable in the *modelsim.ini* file to set a permanent default.
- **Default Run**
  Sets the default run length for the current simulation. Edit the RunLength (UM-625) variable in the *modelsim.ini* file to set a permanent default.

- **Iteration Limit**
  Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. Edit the IterationLimit (UM-624) variable in the *modelsim.ini* file to set a permanent iteration limit default.
- **Default Force Type**
  Selects the default force type for the current simulation. Edit the DefaultForceKind (UM-623) variable in the *modelsim.ini* file to set a permanent default.

Assertions tab



Break on Assertion
[✓ Fatal] Failure Error Warning Note

Ignore Assertions For:
☐ Failure
☐ Error
☐ Warning
☐ Note

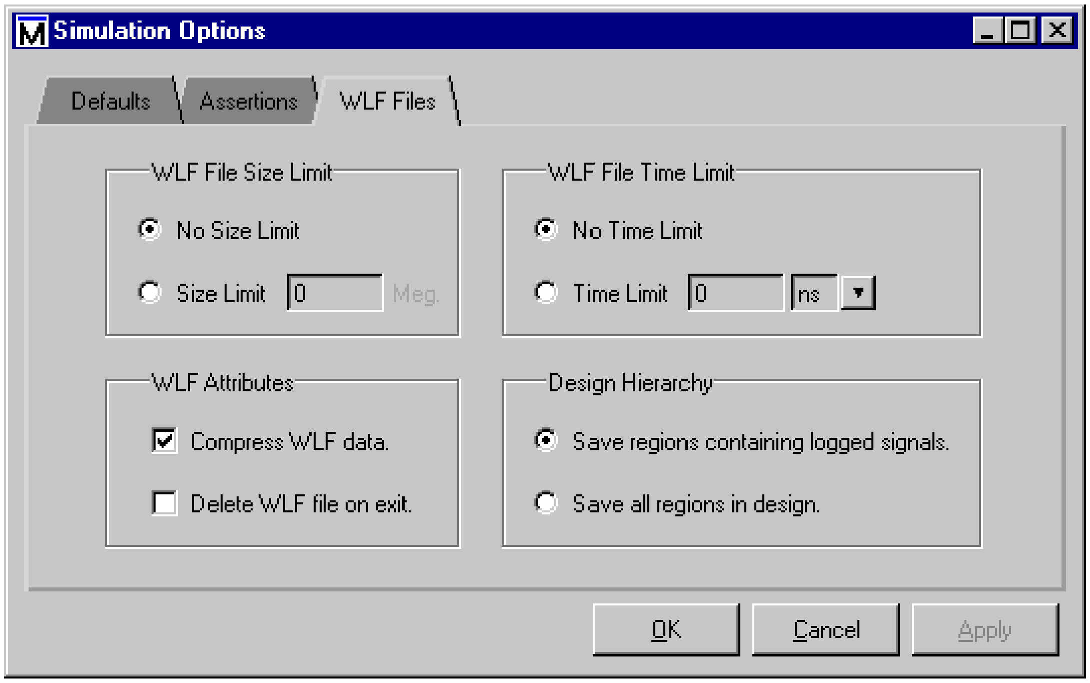
The **Assertions** tab includes these options:

- **Break on Assertion**
  Selects the assertion severity that will stop simulation. Edit the BreakOnAssertion (UM-622) variable in the *modelsim.ini* file to set a permanent default.
- **Ignore Assertions For**
  Selects the assertion type to ignore for the current simulation. Multiple selections are possible. Edit the IgnoreFailure, IgnoreError, IgnoreWarning, and IgnoreNote (UM-624) variables in the *modelsim.ini* file to set permanent defaults.

  When an assertion type is ignored, no message will be printed, nor will the simulation halt (even if break on assertion is set for that type).

Note: Assertions that appear within an instantiation or configuration port map clause conversion function will not stop the simulation regardless of the severity level of the assertion.

WLF Files tab



WLF File Size Limit
[● No Size Limit] ○ Size Limit 0 Meg.

WLF File Time Limit
[● No Time Limit] ○ Time Limit 0 ns

WLF Attributes
[✓ Compress WLF data.]
☐ Delete WLF file on exit.

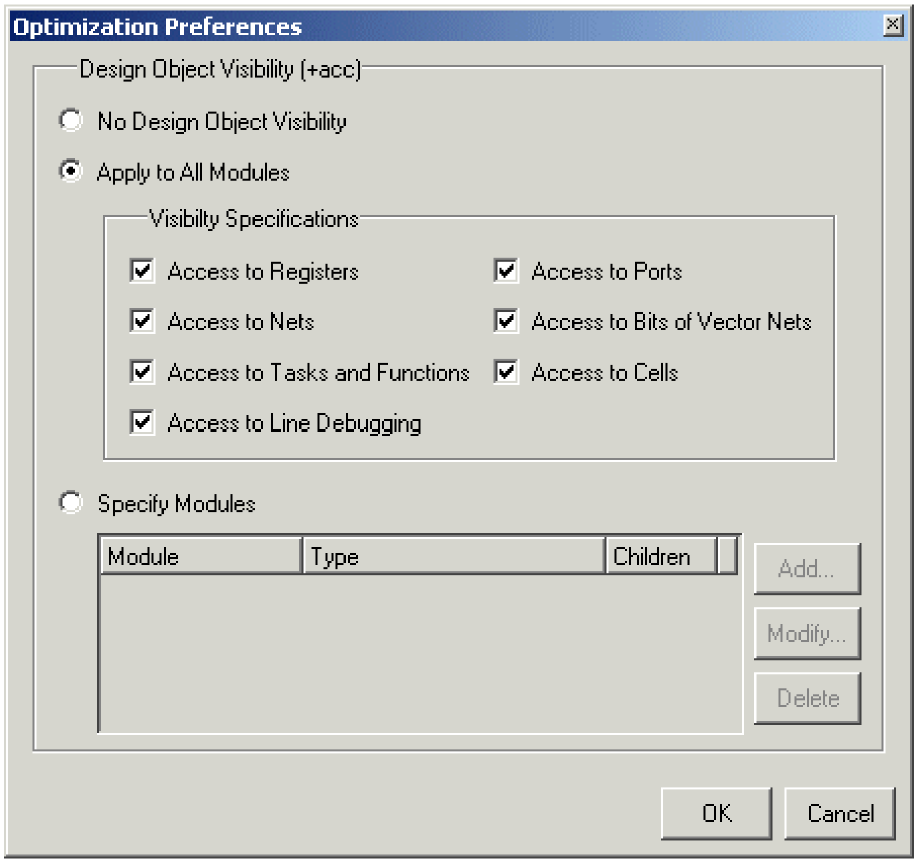
Design Hierarchy
[● Save regions containing logged signals.]
○ Save all regions in design.

The **WLF Files** tab includes these options:

- **WLF File Size Limit**
  Limits the WLF file by size (as closely as possible) to the specified number of megabytes. If both size and time limits are specified, the most restrictive is used. Setting it to 0 results in no limit. Edit the WLFSizeLimit (UM-626) variable in the *modelsim.ini* file to set a permanent default.
- **WLF File Time Limit**
  Limits the WLF file by size (as closely as possible) to the specified amount of time. If both time and size limits are specified, the most restrictive is used. Setting it to 0 results in no limit. Edit the WLFTimeLimit (UM-626) variable in the *modelsim.ini* file to set a permanent default.
- **WLF Attributes**
  Specifies whether to compress WLF files and whether to delete the WLF file when the simulation ends. You would typically only disable compression for troubleshooting purposes. Edit the WLFCompress (UM-626) variable in the *modelsim.ini* file to set a permanent default for compression. Edit the WLFDeleteOnQuit (UM-626) variable in the *modelsim.ini* file to set a permanent default for WLF file deletion.
- **Design Hierarchy**
  Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. Edit the WLFSaveAllRegions (UM-626) variable in the *modelsim.ini* file to set a permanent default.

Enabling design object visibility in optimized simulations

Designs simulated with **-fast** have limited access to design objects. See "Enabling design object visibility with the +acc option" (UM-133) for more details. On the "Verilog tab" (UM-381) of the Simulate dialog, you can select **Optimize Preferences** to selectively enable design object visibility.



Optimization Preferences

Design Object Visibility (+acc)

( ) No Design Object Visibility

(●) Apply to All Modules

Visibilty Specifications

- ☑ Access to Registers
- ☑ Access to Ports
- ☑ Access to Nets
- ☑ Access to Bits of Vector Nets
- ☑ Access to Tasks and Functions
- ☑ Access to Cells
- ☑ Access to Line Debugging

( ) Specify Modules

Module	Type	Children

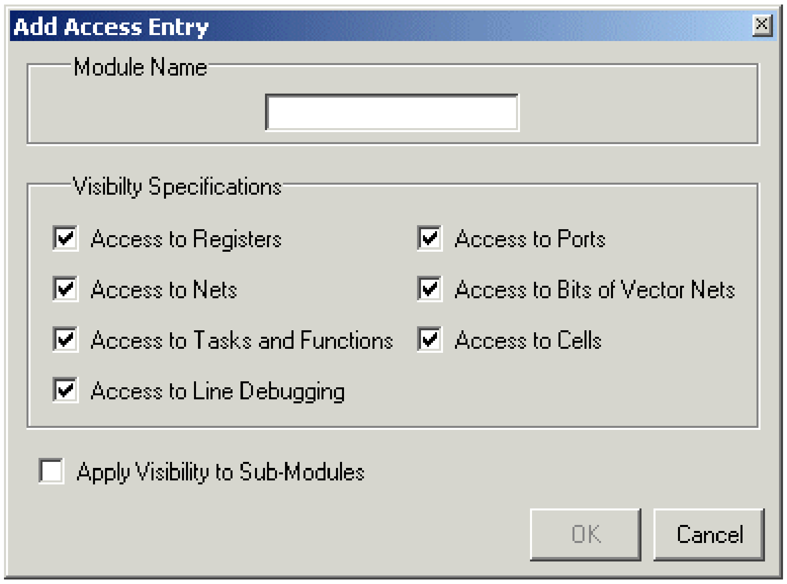
The **Optimization Preferences** dialog includes these options:

- **No Design Object Visibility**
  Default behavior where ModelSim optimizes at will without concern for underlying design object visibility.
- **Apply to All Modules** (+acc)
  Specifies visibility settings for all modules in your design. Please see "Enabling design object visibility with the +acc option" (UM-133) for more details. Options include:

Access to Registers (+acc=r)	Enable access to registers (including memories, integer, time, and real types).
Access to Nets (+acc=n)	Enable access to nets.
Access to Tasks and Functions (+acc=t)	Enable access to tasks and functions.

Access to Line Debugging (+acc=l)	Enable line number directives and process names for line debugging, profiling, and code coverage.
Access to Ports (+acc=p)	Enable access to ports.
Access to Bits of Vector Nets (+acc=b)	Enable access to individual bits of vector nets.
Access to Cells (+acc=c)	Enable access to library cells.

- **Specify Modules** (`+acc[=<spec>][+<module>[.]]`)
  Specifies visibility settings for individual modules in your design. Click Add to open the **Add Access Entry** dialog.



Add Access Entry

Module Name

Visibilty Specifications

- [✓] Access to Registers
- [✓] Access to Ports
- [✓] Access to Nets
- [✓] Access to Bits of Vector Nets
- [✓] Access to Tasks and Functions
- [✓] Access to Cells
- [✓] Access to Line Debugging

[ ] Apply Visibility to Sub-Modules

The **Add Access Entry** dialog includes these options:

- **Module Name**
  Specifies the module to which the visibility settings will apply.
- **Visibility Specifications**
  See above.
- **Apply Visibility to Sub-Modules**
  Specifies that the settings apply to all sub-modules of the specified Module Name.

Creating and managing breakpoints

ModelSim supports both signal (i.e., when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line. Breakpoints within SystemC portions of the design can only be set using File-line breakpoints (UM-391).

Signal breakpoints

Signal breakpoints (when conditions) instruct ModelSim to perform actions when the specified conditions are met. For example, you can break on a signal value or at a specific simulator time (see the **when** command (CR-375) for additional details). When a breakpoint is hit, a message in the Main window transcript identifies the signal that caused the breakpoint.

Setting signal breakpoints from the command line

You use the **when** command (CR-375) to set a signal breakpoint from the VSIM> prompt. See the *Command Reference* for further details.

Setting signal breakpoints from the GUI

Signal breakpoints are most easily set in the Signals window (UM-316) and the Wave window (UM-337). Right-click a signal and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Breakpoints** dialog.

Alternatively you can set signal breakpoints from the Breakpoints dialog (UM-392).

File-line breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops.

Since C Debug is invoked when you set a breakpoint within a SystemC module, your C Debug settings must be in place prior to setting a breakpoint. See Setting up C Debug (UM-475) for more information. Once invoked, C Debug can be exited using the C Debug menu.

Setting file-line breakpoints from the command line

You use the **bp** command (CR-81) to set a file-line breakpoint from the VSIM> prompt. See the *Command Reference* for further details.

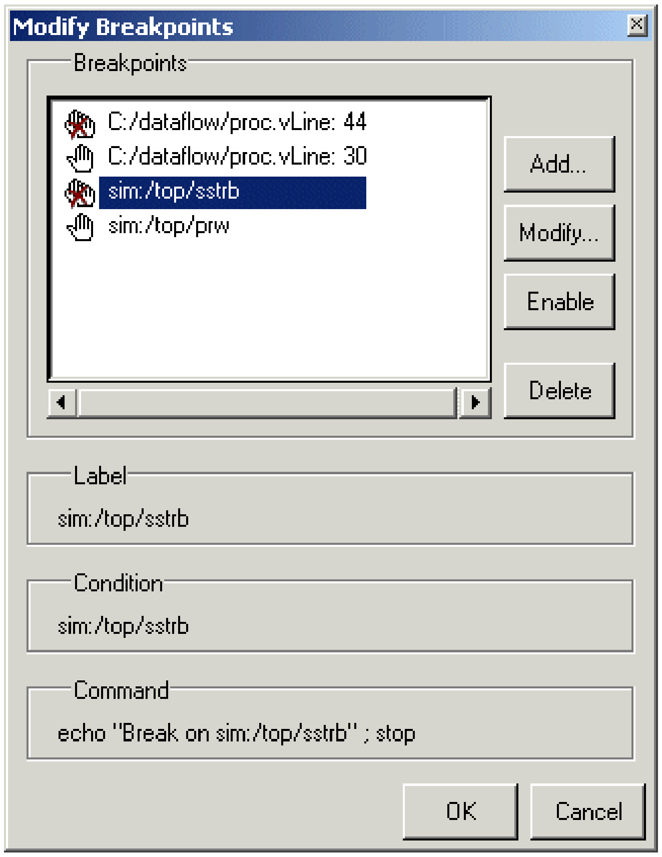
Setting file-line breakpoints from the GUI

File-line breakpoints are most easily set using your mouse in the Source window (UM-325). Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint. To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**.

Alternatively you can set file-line breakpoints from the Breakpoints dialog (UM-392).

Breakpoints dialog

The Breakpoints dialog box allows you to create and manage both Signal breakpoints (UM-391) and File-line breakpoints (UM-391). Select **Tools** > **Breakpoints** from the Main, Signals, Source, or Wave windows to open the dialog.



C:/dataflow/proc.vLine: 44
C:/dataflow/proc.vLine: 30
sim:/top/sstrb
sim:/top/prw

Label: sim:/top/sstrb

Condition: sim:/top/sstrb

Command: echo "Break on sim:/top/sstrb" ; stop

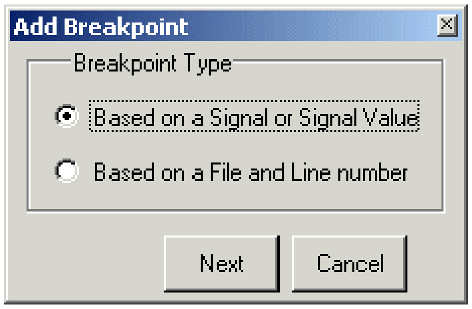
The **Breakpoints** dialog includes these options:

- **Breakpoints**
  List of all existing breakpoints. Breakpoints set from anywhere in the GUI, or from the command line, are listed. A red 'X' through the hand icon means the breakpoint is currently disabled.
- **Add**
  Create a new signal or file-line breakpoint. See below for more details.
- **Modify**
  Change properties of an existing breakpoint. See below for more details.
- **Disable/Enable**
  De-activate or activate the selected breakpoint.
- **Delete**
  Delete the selected breakpoint.
- **Label**
  Text label of the selected breakpoint.

- **Condition**
  The condition under which the breakpoint will be hit.
- **Command**
  The command that will be executed when the breakpoint is hit.

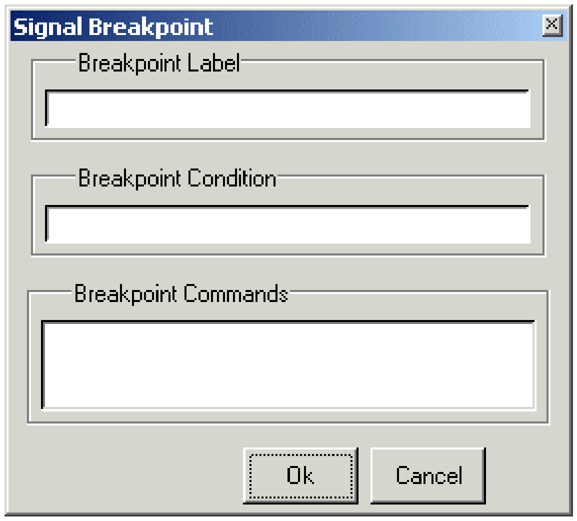
Adding a breakpoint

Click Add to add a new breakpoint, and you will see the Add Breakpoint dialog.



[✓ Based on a Signal or Signal Value]
Based on a File and Line number

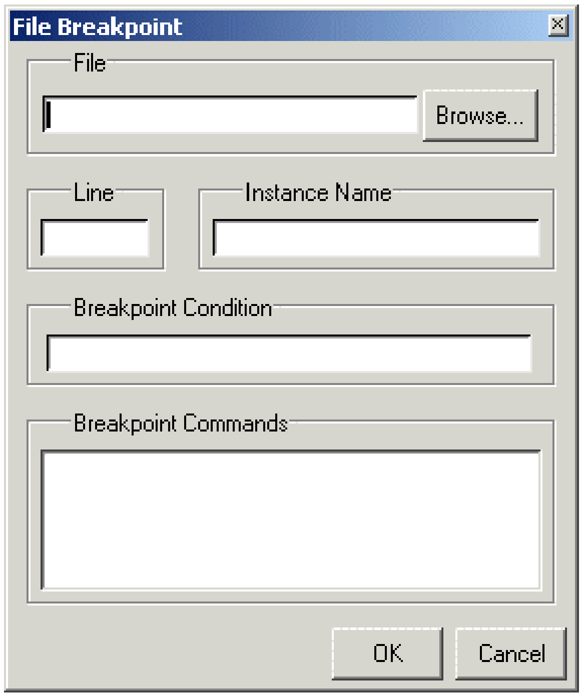
Choose whether to create a signal breakpoint or a file-line breakpoint and then select Next. Depending on which type of breakpoint you're creating, you'll see one of the two dialogs below. These are the same dialogs you'll see if you modify an exiting breakpoint.



The **Signals Breakpoint** dialog includes these options:

- **Breakpoint Label**
  Specify an optional text label for the breakpoint.
- **Breakpoint Condition**
  Specify condition(s) to be met for the command(s) to be executed. See the **when** command (CR-375) for more information on creating the condition statement.

- **Breakpoint Commands**
  Specify command(s) to be executed when the condition is met. Any ModelSim or Tcl command or series of commands are valid, with one exception – the **run** command (CR-246) cannot be used.



The **File Breakpoint** dialog includes these options:

- **File**
  Specify the file in which to set the breakpoint.
- **Line**
  Specify the line number on which to set the breakpoint. Note that breakpoints can be set only on executable lines.
- **Instance Name**
  Specify a region in which to apply the breakpoint. If left blank the breakpoint affects every instance in the design.
- **Breakpoint Condition**
  Specify a condition that determines whether the breakpoint is hit.
- **Breakpoint Commands**
  Specify command(s) to be executed when the breakpoint is hit. Any ModelSim or Tcl command or series of commands is valid, with one exception – the **run** command (CR-246) cannot be used.

Miscellaneous tools and add-ons

Several miscellaneous tools and add-ons are available from ModelSim menus. Follow the links below for more information.

- The GUI Expression Builder (UM-395)
  Edit > **Search** > **Search for Expression** > **Builder** (List or Wave window)
  Helps you build logical expressions for use in Wave and List window searches and several simulator commands. For expression format syntax see "GUI_expression_format" (CR-23).
- HDL language templates (UM-397)
  View > **Show language templates** (Source window)
  Helps you write VHDL or Verilog code.
- The Button Adder (UM-400)
  Window > **Customize** (any window)
  Allows you to add a temporary function button or toolbar to any window.
- The Macro Helper (UM-401)
  Tools > **Macro Helper** (Main window)
  Creates macros by recording mouse movements and key strokes. UNIX only (excluding Linux).
- The Tcl Debugger (UM-402)
  Tools > **Tcl Debugger** (Main window)
  Helps you debug your Tcl procedures.
- **Debug Detective**™
  Debug Detective is an add-on tool that lets you view any level of your design as block diagrams, Interface-Based Design™ (IBD™) tables, state machines, or flow charts. Enhanced debugging features include graphical breakpoints, signal probing, graphics to text source cross-highlighting, animation, and cause analysis.

  The tool is accessed directly from within ModelSim. Assuming you have purchased and installed Debug Detective, a new menu and toolbar button will appear in ModelSim when you load a design. Complete documentation for Debug Detective is available from the **Start Menu** once the product is installed. Please see www.mentor.com/debugdetective for more information.

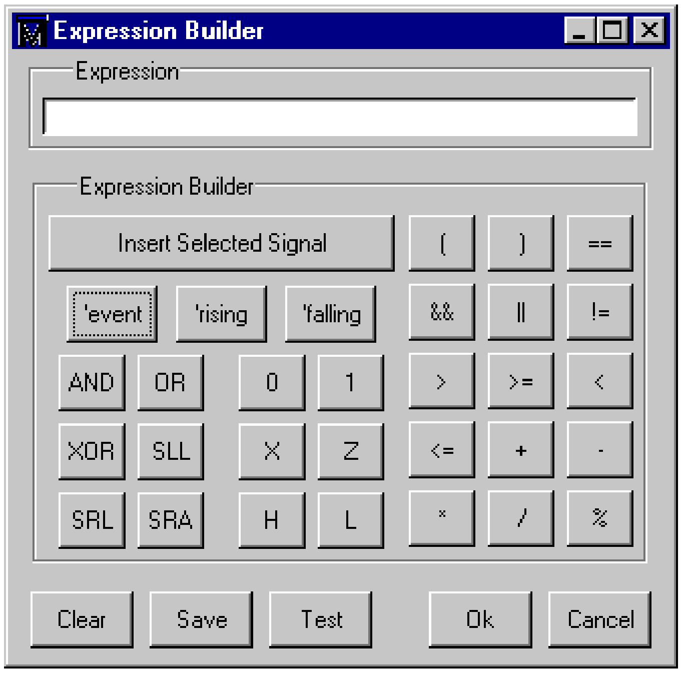
The GUI Expression Builder

The GUI Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the "GUI_expression_format" (CR-23).

To locate the Builder:

- select **Edit** > **Search** (List or Wave window)
- select the **Search for Expression** option in the resulting dialog box

- select the **Builder** button



The Expression Builder dialog box provides an array of buttons that help you build a GUI expression. For instance, rather than typing in a signal name, you can select the signal in the associated Wave or List window and press Insert Reference Signal in the Expression Builder. The result will be the full signal name added to the expression field. All Expression Builder buttons correspond to the "Expression syntax" (CR-24).

To search for when a signal reaches a particular value

Select the signal in the Wave window and click **Insert Selected Signal** and ==. Then, click the value buttons or type a value.

To evaluate only on clock edges

Click the **&&** button to AND this condition with the rest of the expression. Then select the clock in the Wave window and click **Insert Selected Signal** and **'rising**. You can also select the falling edge or both edges.

Operators

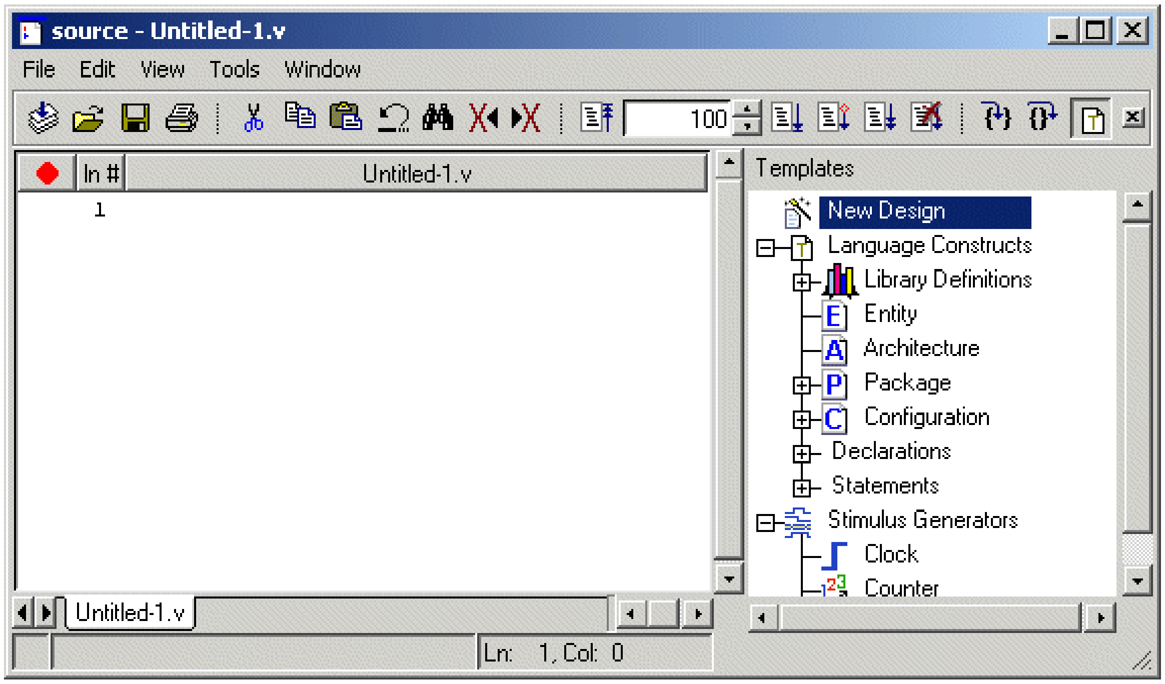
Other buttons will add operators of various kinds (see "Expression syntax" (CR-24)), or you can type them in.

HDL language templates

ModelSim language templates help you write VHDL or Verilog code. They are a collection of wizards, menus, and dialogs that produce code for new designs, language constructs, logic blocks, etc.

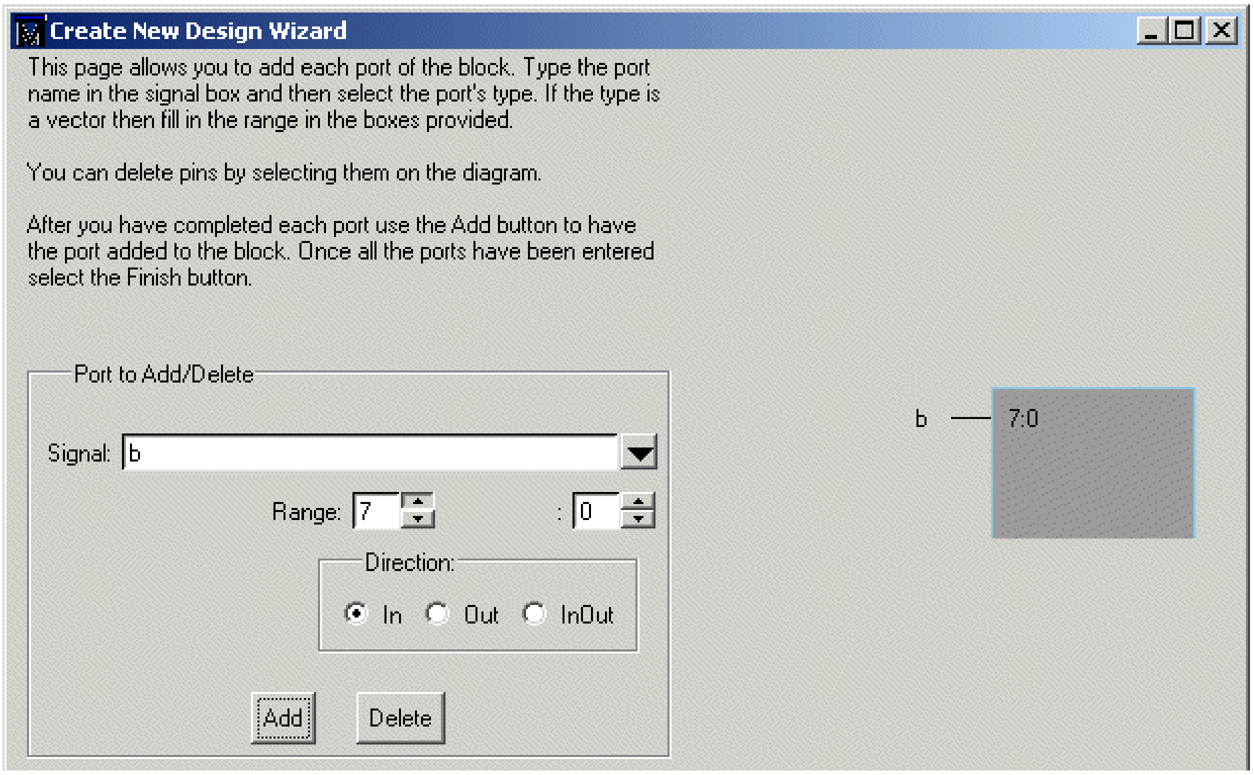
Important: The language templates are not intended to replace thorough knowledge of coding. They are intended as an interactive "reference" for creating small sections of code. If you are unfamiliar with VHDL or Verilog, you should attend a training class or consult one of the many books available on HDL languages.

To use the templates, either open an existing HDL file in the Source window (UM-325), or select **File > New** (Source window) to create a new file. Once the file is open, select **View > Show language templates**. This displays a pane that shows the available templates.



The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

Double-click an item in the list to begin creating code. Some of the items bring up wizards while others insert code into your HDL file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.



Create New Design Wizard

This page allows you to add each port of the block. Type the port name in the signal box and then select the port's type. If the type is a vector then fill in the range in the boxes provided.

You can delete pins by selecting them on the diagram.

After you have completed each port use the Add button to have the port added to the block. Once all the ports have been entered select the Finish button.

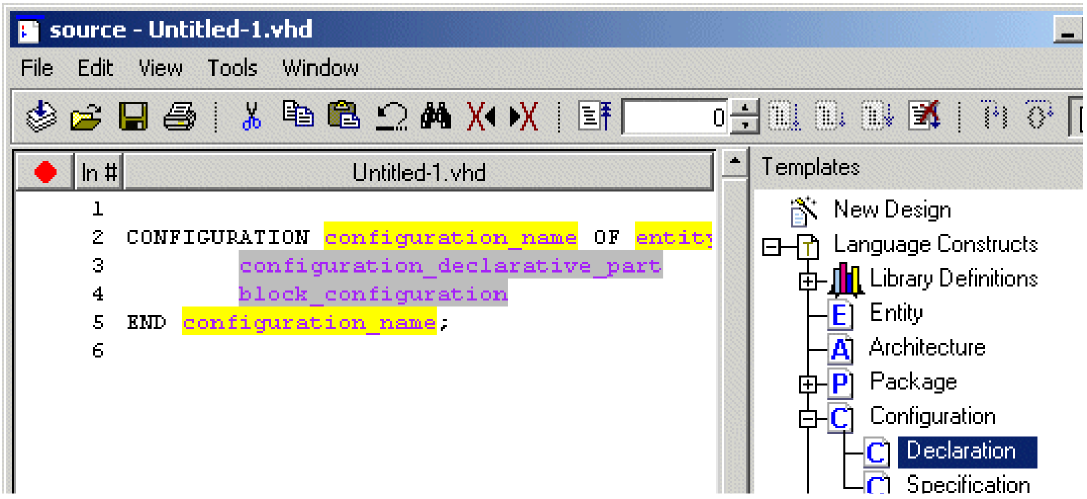
Port to Add/Delete

Signal: b

Range: 7 : 0

Direction: [✓ In] Out InOut

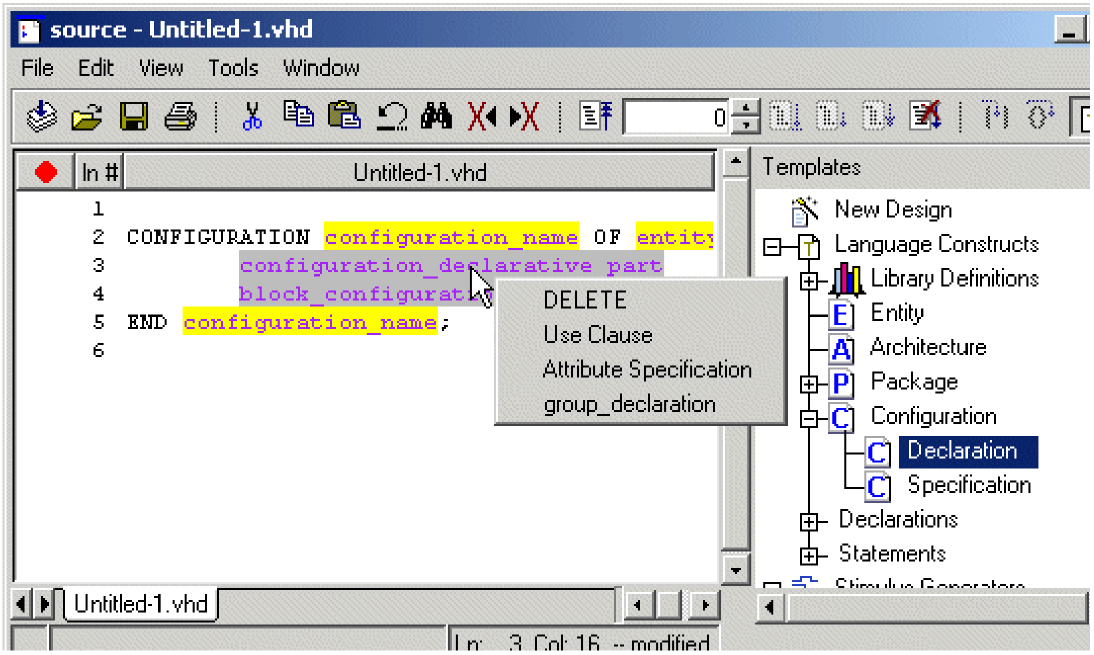
Code inserted into your source file may contain yellow or gray highlighted "fields". Yellow highlighting identifies an object that needs a name. Double-click the yellow object to enter a name. Note that all yellow objects with the same label (e.g., "configuration_name" below) will change to whatever name you enter. This ensures matching fields remain in synch.



source - Untitled-1.vhd

```
CONFIGURATION configuration_name OF entity
        configuration_declarative_part
        block_configuration
END configuration_name;

```

Gray highlighting indicates that a context menu with additional commands is available. In the example below, right-clicking "configuration_declarative_part" gives you three options for continuing the definition of the Configuration.



```
CONFIGURATION configuration_name OF entity
        configuration_declarative part
        block_configurat
END configuration_name;
```

The first menu item is always "DELETE." This allows you to remove unwanted objects from the HDL code, such as optional fields.

Keyboard shortcut

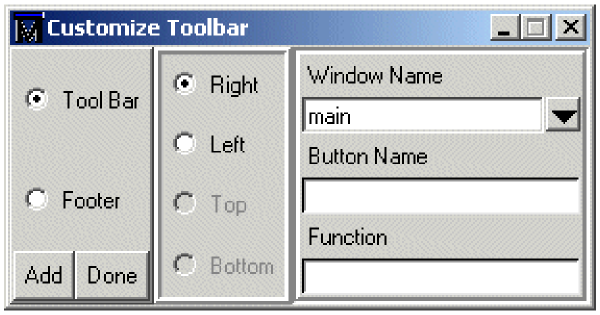
<control - p> edits a yellow field and expands a gray field at the current cursor location.

The Button Adder

The **Button Adder** creates a single button or a combined button and toolbar in any currently opened ModelSim window. The button exists only until you close the window unless you add the button code to the window's user hook variable (see "Making the button persistent" (UM-400) below).

Invoke the Button Adder from any ModelSim window menu: **Window > Customize**.

You have the following options for adding a button:



- **Window Name** is the name of the window to which you want to add the button.
- **Button Name** is the button's label.
- **Function** can be any command or macro you might execute from the ModelSim command line. For example, you might want to add a **Run** or **Step** button to the Wave window.

Locate the button within the window with these selections:

- **Tool Bar** places the button on a new toolbar.
- **Footer** adds the button to the window's status bar.

Justify the button within the toolbar/footer with these selections:

- **Right** places the button on the right side of the toolbar/footer.
- **Left** adds the button on the left side of the toolbar/footer.
- **Top** places the button at the top/center of the toolbar/footer.
- **Bottom** places the button at the bottom/center of the toolbar/footer.

Making the button persistent

When you create a button with the Button Adder, the underlying commands are echoed in the transcript. You can use these commands to make the button appear every time you invoke the window. Follow these steps:

1. Create a button using the Button Adder.
2. Copy the commands from the transcript into a Tcl procedure in the *modelsim.tcl* file. If you don't have a *modelsim.tcl* file already, create a new text file with that name and set the MODELSIM_TCL environment variable to the full path of the *modelsim.tcl* file.
3. Append the procedure name to the window's user_hook Tcl variable. See "Preference variables located in Tcl files" (UM-631) for more information on Tcl preference variables.

An example will help clarify. Say you create a button in the Wave window that adds all signals from the selected region to the Wave window. The button code will look something like this:

```
_add_menu .wave controls right SystemButtonFace black AddWaves {add wave *}
```

You would insert that code into a Tcl procedure in the *modelsim.tcl* file and then append the procedure to the PrefWave(user_hook) variable. The entire entry in the *modelsim.tcl* file would look as follows:

```
proc AddWaves winname {
_add_menu .wave controls right SystemButtonFace black AddWaves {add wave *}
}

lappend PrefWave(user_hook) AddWaves
```

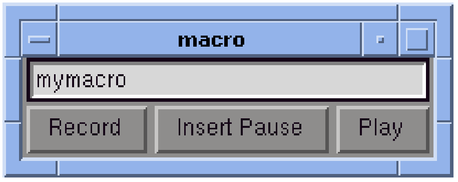
Now, any time you start ModelSim and open the Wave window, it will have a button labeled "AddWaves" that executes the command "add wave *".

The Macro Helper

This tool is available for UNIX only (excluding Linux).

The purpose of the Macro Helper is to aid macro creation by recording a simple series of mouse movements and key strokes. The resulting file can be called from a more complex macro by using the **play** (CR-214) command. Actions recorded by the Macro Helper can only take place within the ModelSim GUI (window sizing and repositioning are not recorded because they are handled by your operating system's window manager). In addition, the **run** (CR-246) commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.

Select **Tools > Macro Helper** (Main window) to access the Macro Helper.



- **Record a macro**
  by typing a new macro file name into the field provided, then press **Record**. Use the **Pause** and **Stop** buttons as shown in the table below.
- **Play a macro**
  by entering the file name of a Macro Helper file into the field and pressing **Play**.

Files created by the Macro Helper can be viewed with the **notepad** (CR-207).

Button	Description
Record/Stop	Record begins recording and toggles to Stop once a recording begins
Insert Pause	inserts a .5 second pause into the macro file; press the button more than once to add more pause time; the pause time can subsequently be edited in the macro file
Play	plays the Macro Helper file specified in the file name field

See the **macro_option** command (CR-191) for playback speed, delay, and debugging options for completed macro files.

The Tcl Debugger

We would like to thank Gregor Schmid for making TDebug available for use in the public domain.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of FITNESS FOR A PARTICULAR PURPOSE.

Starting the debugger

Select **Tools > Tcl Debugger** (Main window) to run the debugger. Make sure you use the ModelSim and TDebug menu selections to invoke and close the debugger. If you would like more information on the configuration of TDebug see **Help > Technotes > tdebug**.

The following text is an edited summary of the README file distributed with TDebug.

How it works

TDebug works by parsing and redefining Tcl/Tk-procedures, inserting calls to `td_eval' at certain points, which takes care of the display, stepping, breakpoints, variables etc. The advantages are that TDebug knows which statement in which procedure is currently being executed and can give visual feedback by highlighting it. All currently accessible variables and their values are displayed as well. Code can be evaluated in the context of the current procedure. Breakpoints can be set and deleted with the mouse.

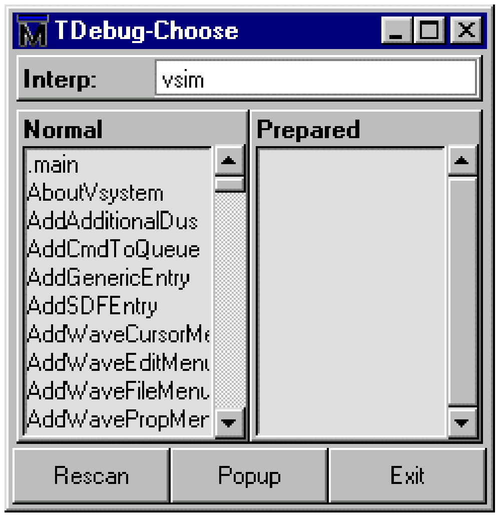
Unfortunately there are drawbacks to this approach. Preparation of large procedures is slow and due to Tcl's dynamic nature there is no guarantee that a procedure can be prepared at all. This problem has been alleviated somewhat with the introduction of partial preparation of procedures. There is still no possibility to get at code running in the global context.

The Chooser

Select **Tools > Tcl Debugger** (Main window) to open the TDebug chooser.

The TDebug chooser has three parts. At the top the current interpreter, *vsim.op_*, is shown. In the main section there are two list boxes. All currently defined procedures are shown in the left list box. By clicking the left mouse button on a procedure name, the procedure gets prepared for debugging and its name is moved to the right list box. Clicking a name in the right list box returns a procedure to its normal state.

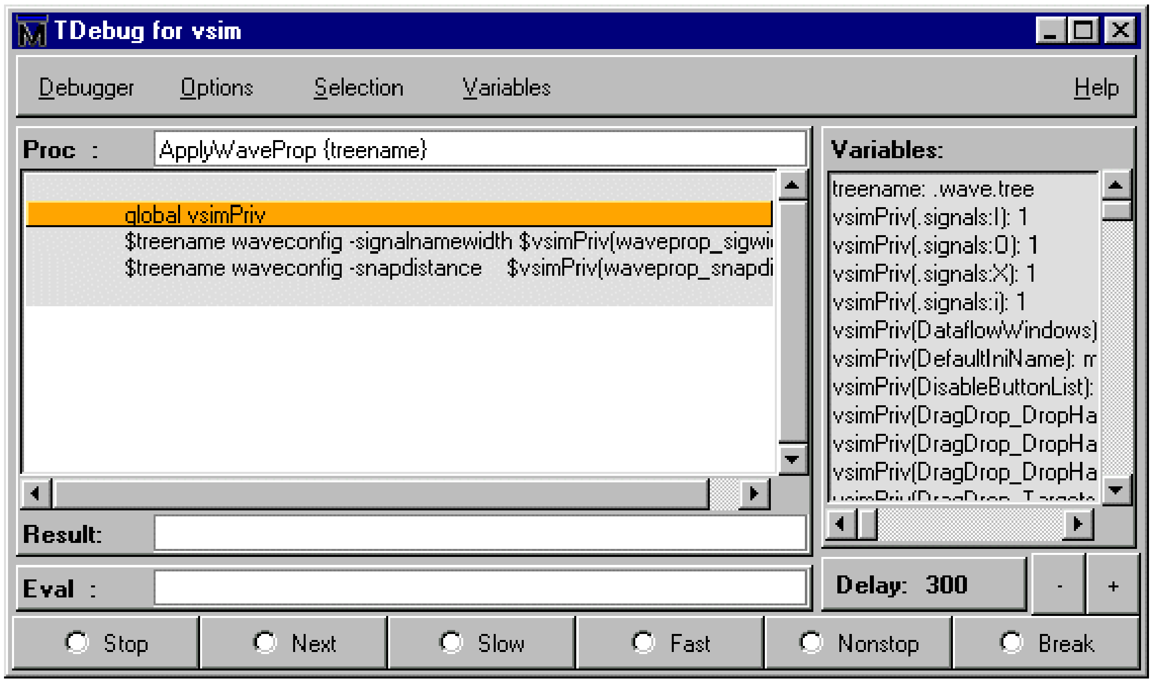
Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.



The three buttons at the bottom let you force a **Rescan** of the available procedures, **Popup** the debugger window or **Exit** TDebug. Exiting from TDebug doesn't terminate ModelSim, it merely detaches from *vsim.op_*, restoring all prepared procedures to their unmodified state.

The Debugger

Select the **Popup** button in the Chooser to open the debugger window.



Proc : ApplyWaveProp {treename}

```
        global vsimPriv
        $treename waveconfig -signalnamewidth $vsimPriv(waveprop_sigwi
        $treename waveconfig -snapdistance    $vsimPriv(waveprop_snapdi
```

```
treename: .wave.tree
vsimPriv(.signals:I): 1
vsimPriv(.signals:O): 1
vsimPriv(.signals:X): 1
vsimPriv(.signals:i): 1
vsimPriv(DataflowWindows)
vsimPriv(DefaultIniName): m
vsimPriv(DisableButtonList):
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
```

The debugger window is divided into the main region with the name of the current procedure (**Proc**), a listing in which the expression just executed is highlighted, the **Result** of this execution and the currently available **Variables** and their values, an entry to **Eval** expressions in the context of the current procedure, and some button controls for the state of the debugger.

A procedure listing displayed in the main region will have a darker background on all lines that have been prepared. You can prepare or restore additional lines by selecting a region (<Button-1>, standard selection) and choosing **Selection > Prepare Proc** or **Selection > Restore Proc** from the debugger menu (or by pressing ^P or ^R).

When using `Prepare' and `Restore', try to be smart about what you intend to do. If you select just a single word (plus some optional white space) it will be interpreted as the name of a procedure to prepare or restore. Otherwise, if the selection is owned by the listing, the corresponding lines will be used.

Be careful with partial prepare or restore! If you prepare random lines inside a `switch' or `bind' expression, you may get surprising results on execution, because the parser doesn't know about the surrounding expression and can't try to prevent problems.

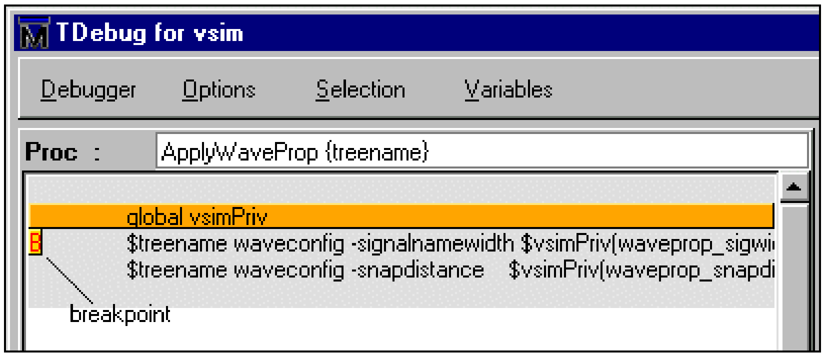
There are seven possible debugger states, one for each button and an `idle' or `waiting' state when no button is active. The button-activated states are:

Button	Description
Stop	stop after next expression, used to get out of slow/fast/nonstop mode
Next	execute one expression, then revert to idle
Slow	execute until end of procedure, stopping at breakpoints or when the state changes to stop; after each execution, stop for ’delay’ milliseconds; the delay can be changed with the ’+’ and ’-’ buttons
Fast	execute until end of procedure, stopping at breakpoints
Nonstop	execute until end of procedure without stopping at breakpoints or updating the display
Break	terminate execution of current procedure

Closing the debugger doesn't quit it, it only does `wm withdraw'. The debugger window will pop up the next time a prepared procedure is called. Make sure you close the debugger with **Debugger > Close**.

Breakpoints

To set/unset a breakpoint, double-click inside the listing. The breakpoint will be set at the innermost available expression that contains the position of the click. Conditional or counted breakpoints aren’t supported.

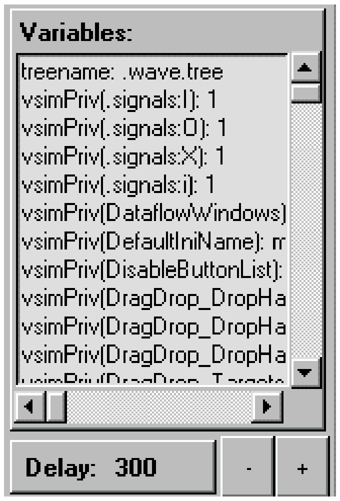


ApplyWaveProp {treename}

```
global vsimPriv
$treename waveconfig -signalnamewidth $vsimPriv(waveprop_sigwi
$treename waveconfig -snapdistance   $vsimPriv(waveprop_snapdi
```

The **Eval** entry supports a simple history mechanism available via the <Up_arrow> and <Down_arrow> keys. If you evaluate a command while stepping through a procedure, the command will be evaluated in the context of the procedure; otherwise it will be evaluated at the global level. The result will be displayed in the result field. This entry is useful for a lot of things, but especially to get access to variables outside the current scope.

Try entering the line `global td_priv' and watch the **Variables** box (with global and array variables enabled of course).



```
treename: .wave.tree
vsimPriv(.signals:I): 1
vsimPriv(.signals:O): 1
vsimPriv(.signals:X): 1
vsimPriv(.signals:i): 1
vsimPriv(DataflowWindows)
vsimPriv(DefaultIniName): m
vsimPriv(DisableButtonList):
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
```

Configuration

You can customize TDebug by setting up a file named .tdebugrc in your home directory. See the TDebug README at **Help** > **Technotes** > **tdebug** for more information on the configuration of TDebug.

TclPro Debugger

The Tools menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger and any available documentation can be acquired from Scriptics. Once acquired, do the following steps to use the TclPro Debugger:

1 Make sure the TclPro bin directory is in your PATH.

2 In TclPro Debugger, create a new project with Remote Debugging enabled.

3 Start ModelSim and select **Tools** > **TclPro Debugger** (Main window)

4 Press the Stop button in the debugger in order to set breakpoints, etc.

Note: TclPro Debugger version 1.4 does not work with ModelSim.

11 - Performance Analyzer

Chapter contents

You can use the Performance Analyzer to easily identify areas in your simulation where performance can be improved. The Performance Analyzer can be used at all levels of design simulation – Functional, RTL, and Gate Level – and has the potential to save hours of regression test time. In addition, ASIC and FPGA design flows benefit from the use of this tool.

Note: If you need to run the Performance Analyzer under Windows on a design that contains FLI/PLI/VPI code, add these two switches to the compiling/linking command:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* file that the profiler can use in its report.

Introducing Performance Analysis

The Performance Analyzer provides an interactive graphical representation of where ModelSim is spending its time while running your design. This feature enables you to quickly determine what is impacting the design environment's simulation performance. Those familiar with the design and validation environment will be able to find first-level improvements in a matter of minutes.

For example, the Performance Analyzer might show some or all of the following

- A non-accelerated VITAL library cell is impacting simulation run time
- A process is consuming more time than necessary because of non-required items in its sensitivity list
- A testbench process is active even though it is not needed
- A C module is inefficient
- A random number process is consuming simulation resources when in a testbench that is running in non-random mode

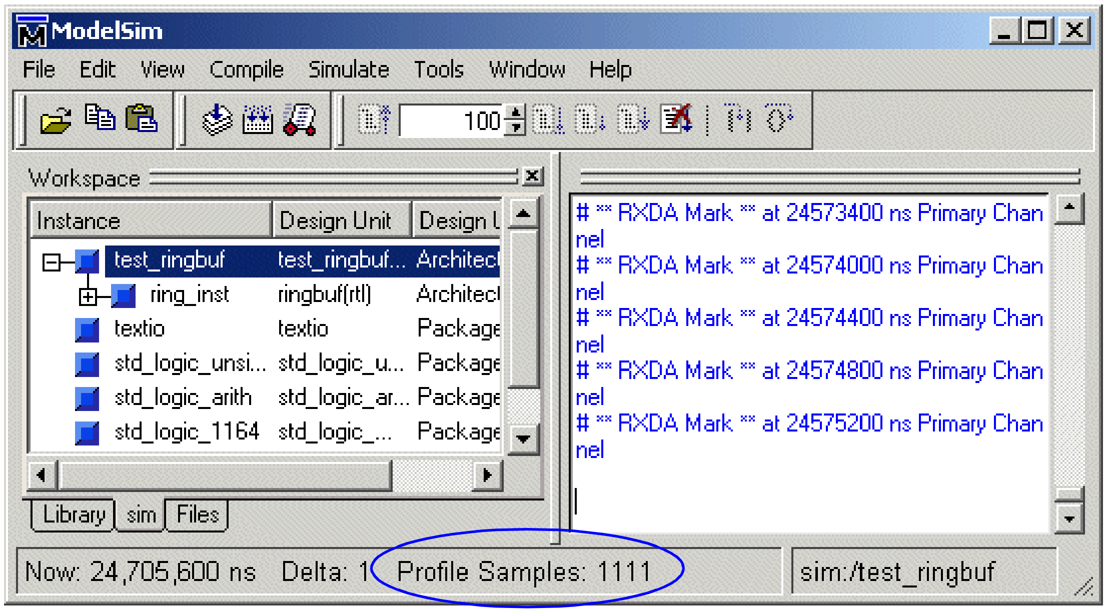
With this information, you can make changes to the VHDL or Verilog source code that will speed up the simulation.

A statistical sampling profiler

The Performance Analyzer is a statistical sampling profiler. It periodically samples the current simulation at a user-determined rate and records what is executing in the simulation. The advantage of statistical analysis is that an entire simulation may not have to be run to get good information from the Performance Analyzer. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

The Performance Analyzer reports only on the samples that it can attribute to user code. For example, if you used the **-nodebug** argument to **vcom** (CR-303) or **vlog** (CR-345), it could not report sample results.

During sampling, the Samples field in the footer of the Main window displays the number of profiling samples collected, and each sample becomes one data point in the simulation profile.



Instance	Design Unit	Design U
test_ringbuf	test_ringbuf...	Architect
ring_inst	ringbuf(rtl)	Architect
textio	textio	Package
std_logic_unsi...	std_logic_u...	Package
std_logic_arith	std_logic_ar...	Package
std_logic_1164	std_logic_...	Package

```
# ** RXDA Mark ** at 24573400 ns Primary Chan
nel
# ** RXDA Mark ** at 24574000 ns Primary Chan
nel
# ** RXDA Mark ** at 24574400 ns Primary Chan
nel
# ** RXDA Mark ** at 24574800 ns Primary Chan
nel
# ** RXDA Mark ** at 24575200 ns Primary Chan
nel
```

Now: 24,705,600 ns Delta: 1 Profile Samples: 1111 sim:/test_ringbuf

Getting started

Performance analysis occurs during the ModelSim **run** command To enable the Performance Analyzer, select **Tools > Profile > Profile On** (Main window). After this command is executed, all subsequent **run** commands will have profiling statistics gathered for them. With the Performance Analyzer enabled and a **run** command initiated, the simulator will provide a message indicating that profiling has started.

You can turn off the Performance Analyzer by selecting **Tools > Profile > Profile Off** (Main window). Any ModelSim **run** commands that follow will not be profiled.

Profiling results are cumulative. Therefore, each **run** command performed with profiling ON will add new information to the data being gathered. To clear this data, select **Tools > Profile > Clear Profile Data** (Main window).

Interpreting the data

The Performance Analyzer helps most in cases where a high percentage of simulation time is spent in one module/entity. For example, say Performance Analyzer shows the simulation is spending 60% of its time in module X. This information can be used to find where module X was implemented poorly and to implement a change that runs faster.

More commonly the Performance Analyzer will tell you, for example, that 30% of simulation time was spent in model X, 25% in model Y, and 20% in model Z. In such situations, careful examination and improvement of each model may result in overall speed improvement.

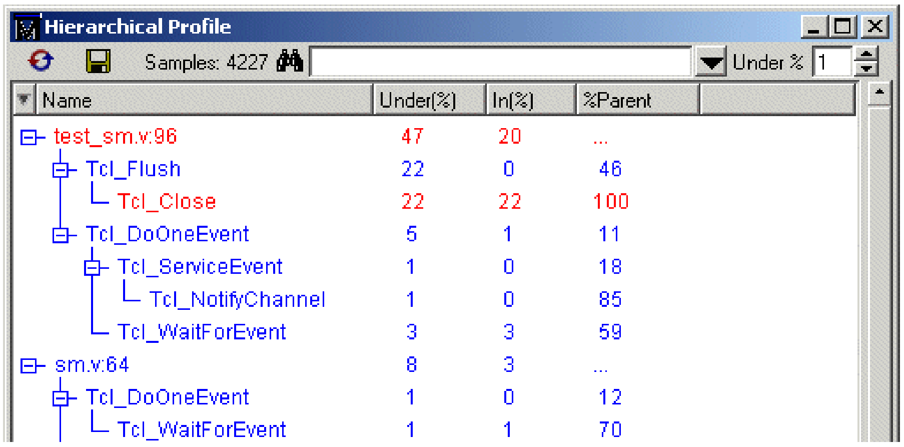
There are times, however, when the Performance Analyzer tells you nothing better than that the simulation has executed in several hundred different models and has spent less 1% or 2% of its time in any one of them. In such situations, the Performance Analyzer provides little helpful information and simulation improvement must come from a higher level examination of how the design can be changed or optimized.

Viewing Performance Analyzer results

The Performance Analyzer provides two views of the collected data – a *hierarchical* and a *ranked* view. The hierarchical view is accessed by selecting **Tools** > **Profile** > **View hierarchical profile** (Main window) or by typing **view_profile** at the VSIM prompt. The ranked view is accessed by selecting **Tools** > **Profile** > **View ranked profile** or by typing **view_profile_ranked** at the VSIM prompt.

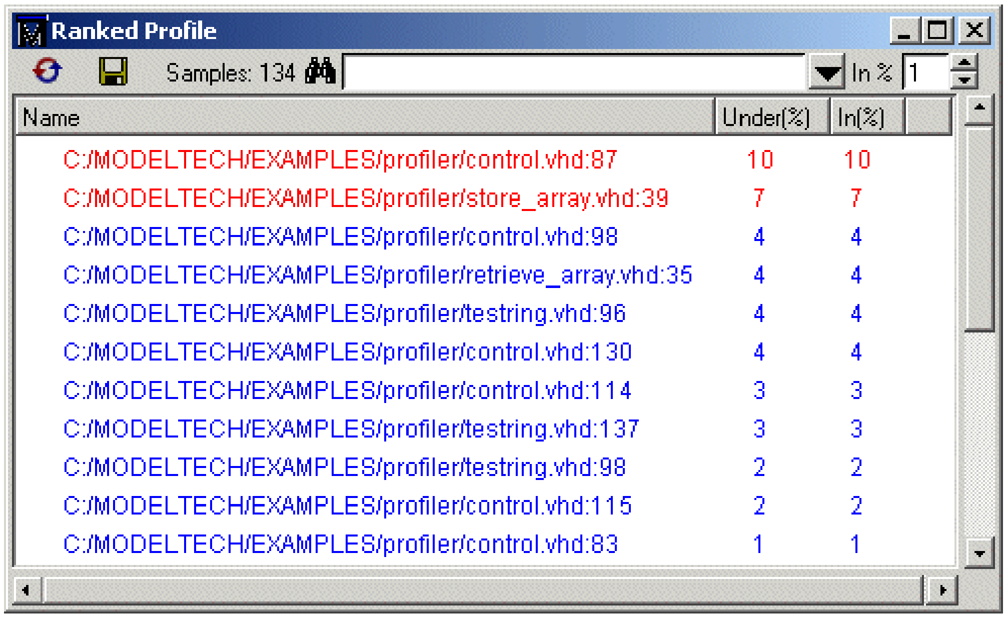
In the Hierarchical Profile window, you can expand and collapse various levels to hide data that is not useful and/or is cluttering the data display. Click on the '-' box to collapse all levels beneath the entry. Click on the '+' box to expand an entry. By default, all levels are fully expanded.

In the hierarchical view below, *test_sm.v:96* is taking the majority of the simulation time.



Name	Under(%)	In(%)	%Parent
test_sm.v:96	47	20	...
Tcl_Flush	22	0	46
Tcl_Close	22	22	100
Tcl_DoOneEvent	5	1	11
Tcl_ServiceEvent	1	0	18
Tcl_NotifyChannel	1	0	85
Tcl_WaitForEvent	3	3	59
sm.v:64	8	3	...
Tcl_DoOneEvent	1	0	12
Tcl_WaitForEvent	1	1	70

In the Ranked Profile view the modules and code lines are ranked in order of the amount of simulation time used.



Name	Under(%)	In(%)
C:/MODELTECH/EXAMPLES/profiler/control.vhd:87	10	10
C:/MODELTECH/EXAMPLES/profiler/store_array.vhd:39	7	7
C:/MODELTECH/EXAMPLES/profiler/control.vhd:98	4	4
C:/MODELTECH/EXAMPLES/profiler/retrieve_array.vhd:35	4	4
C:/MODELTECH/EXAMPLES/profiler/testring.vhd:96	4	4
C:/MODELTECH/EXAMPLES/profiler/control.vhd:130	4	4
C:/MODELTECH/EXAMPLES/profiler/control.vhd:114	3	3
C:/MODELTECH/EXAMPLES/profiler/testring.vhd:137	3	3
C:/MODELTECH/EXAMPLES/profiler/testring.vhd:98	2	2
C:/MODELTECH/EXAMPLES/profiler/control.vhd:115	2	2
C:/MODELTECH/EXAMPLES/profiler/control.vhd:83	1	1

The Hierarchical and Ranked profile windows share a similar toolbar. The table below describes the icons.

Button	Function
🔍	Provides access to a search function that can be used to search for a given string in the window. Type text in the entry box and then press Return or click the binocular icon.
Under % 1	Specifies a cutoff percentage for displaying the data. By default, every entry in the profiling data that has spent at least 1% of the simulation time under that entry will be displayed. In the Ranked view, the value is for the In%. See "Interpreting the Under(%) and In(%) fields" (UM-413) for more information. The **hierCutoff** and **rankCutoff** variables provide a similar function. See "Performance Analyzer preference variables" (UM-417)
↻	Causes the data to be reloaded from the simulator. If you change the cutoff percentage or do an additional simulation run, the Ranked and Hierarchical Profile windows are not updated automatically. You should click on this button to update the data being displayed in these windows.
💾	Allows the data to be saved to disk. You will be prompted for the output file name. The **profile report** command (CR-226) provides another way to save profile data.

Interpreting the Name field

The *Name*, *Under(%)*, and *In(%)* fields appear in both the ranked and hierarchical views. These fields are interpreted identically in both views. Typically a Name consists of an HDL file and line number pair. Most useful names consist of a line of VHDL or Verilog source code. If you use a PLI/VPI or FLI routine, then the name of the C function that implements that routine can also appear in the name field.

vsim is a stripped executable file, so that any functions inside of it will be credited to the line of code that uses the function.

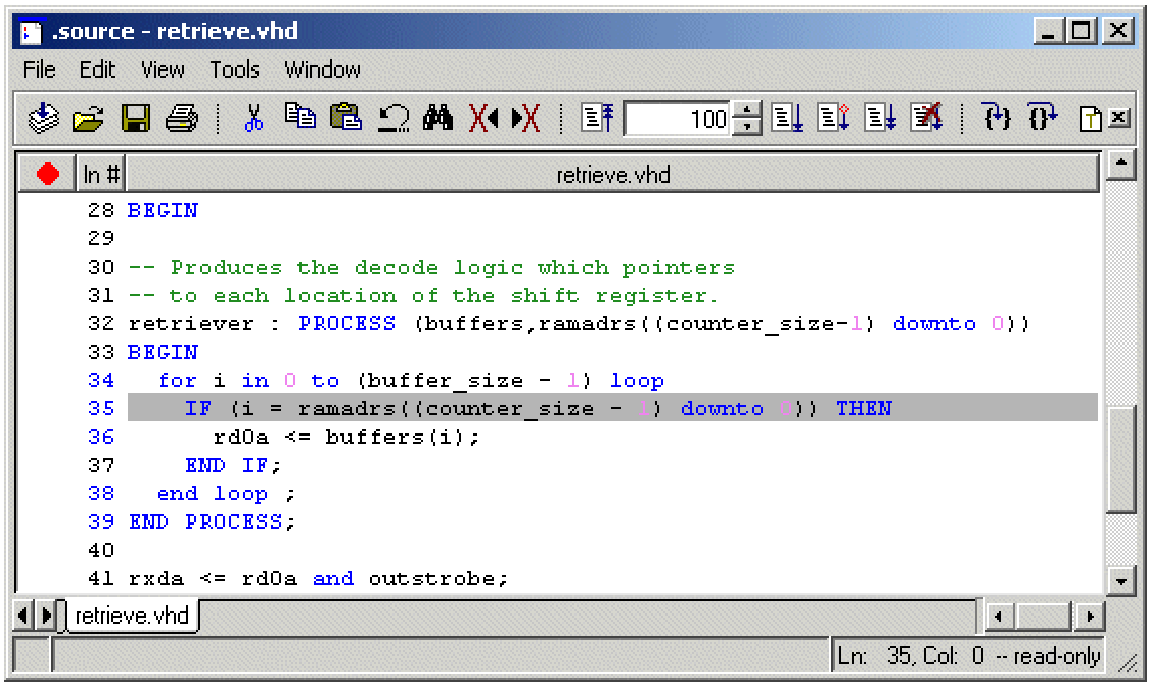
The *hierarchical view* opens with all levels displayed. You can collapse the hierarchical view by clicking the boxes next to the high-level names. At this time, the *hierarchical* view will not remember which levels are opened or closed when data is reloaded. By default, hierarchical levels are opened every time data is reloaded.

Interpreting the Under(%) and In(%) fields

The *In(%)* and *Under(%)* columns describe the percentage of the total simulation time spent in and under a function listed in the Name field.

The distinction between *In(%)* and *Under(%)* is subtle but important. *In(%)* shows that *x*% of the total simulation time was actually spent executing this one line of HDL code. *Under(%)* shows that a particular line and all support routines it needed took *x*% of total simulation time.

In the body of the Hierarchical Profile or Ranked Profile windows, you can double-click on any VHDL/Verilog file and line-number pair to bring up that file in the Source window with the selected line highlighted.



```
BEGIN

-- Produces the decode logic which pointers
-- to each location of the shift register.
retriever : PROCESS (buffers,ramadrs((counter_size-1) downto 0))
BEGIN
  for i in 0 to (buffer_size - 1) loop
    IF (i = ramadrs((counter_size - 1) downto 0)) THEN
      rd0a <= buffers(i);
    END IF;
  end loop ;
END PROCESS;

rxda <= rd0a and outstrobe;
```

Differences in the ranked and hierarchical views

The hierarchical view differs from the ranked view in two important respects.

- Entries in the Name column of the hierarchical view are indented in order to show which functions or routines call which others.
- A *%Parent* column in the hierarchical view allows you to see what percentage of a parent routine's simulation time is used in which subroutines.

Indentation in the Name column of the Hierarchical Profile window indicates which line is calling a function.

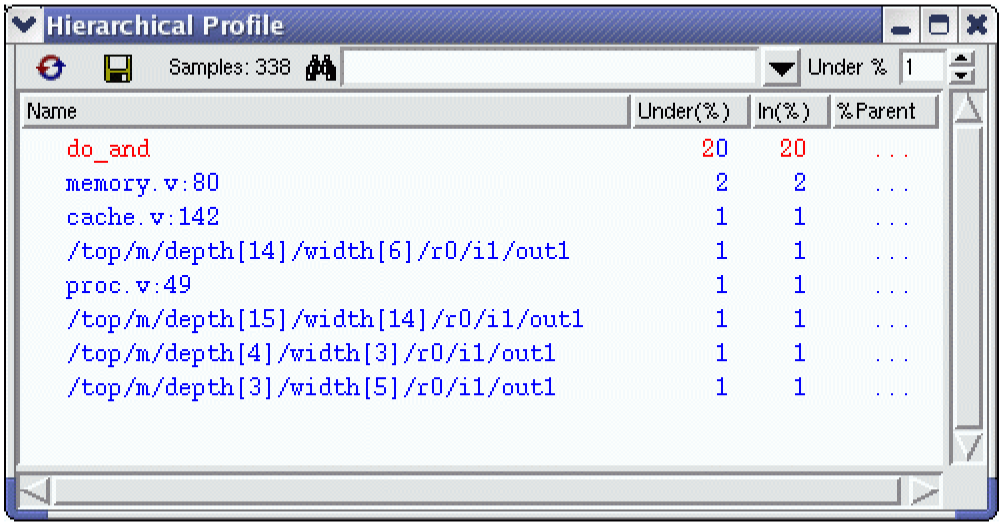
The hierarchical view presents data in a call-graph style format that provides more context than does the ranked view about where simulation time is spent . For example, your models may contain several instances of a utility function that computes the maximum of 3-delay values. A ranked view might reveal that the simulation spent 60% of its time in this utility function, but would not tell you which routine or routines were making the most use of it. The hierarchical view will reveal which line is calling the function most frequently. Using this information, you might decide that instead of calling the function every time to compute the maximum of the 3-delays, this spot in your VHDL code can be used to compute it just once. You can then store the maximum delay value in a local variable.

The *%Parent* column provides the percent of simulation time a given entry used of its parent's total simulation time. From this column, you can calculate the percentage of total simulation time taken up by any function. For example, if a particular parent entry used 10% of the total simulation time, and it called a routine that used 80% of its simulation time, then the percentage of total simulation time spent in that routine would be 80% of 10%, or 8%.

In addition to these differences, the ranked view displays any particular function only once, regardless of where it was used. In the hierarchical view, the function can appear multiple times – each time in the context of where it was used.

Analyzing C code performance

You can include C code in your design via SystemC, the Verilog PLI/VPI, or the ModelSim FLI. The Performance Analyzer can be used to determine the impact of these C modules on simulator performance. For example, in the illustration below, the *do_and* C module is using the majority of simulation time.



Name	Under(%)	In(%)	%Parent
do_and	20	20	...
memory.v:80	2	2	...
cache.v:142	1	1	...
/top/m/depth[14]/width[6]/r0/i1/out1	1	1	...
proc.v:49	1	1	...
/top/m/depth[15]/width[14]/r0/i1/out1	1	1	...
/top/m/depth[4]/width[3]/r0/i1/out1	1	1	...
/top/m/depth[3]/width[5]/r0/i1/out1	1	1	...

Factors that can affect simulator performance when a design includes C code are as follows:

- PLI/FLI applications with large sensitivity lists
- Calling operating system functions from C code
- Calling the simulator's command interpreter from C code
- Inefficient C code

In addition, the Verilog PLI/VPI requires maintenance of the simulator's internal data structures as well as the PLI/VPI data structures for portability. (VHDL does not have this problem in ModelSim because the FLI gets information directly from the simulator.)

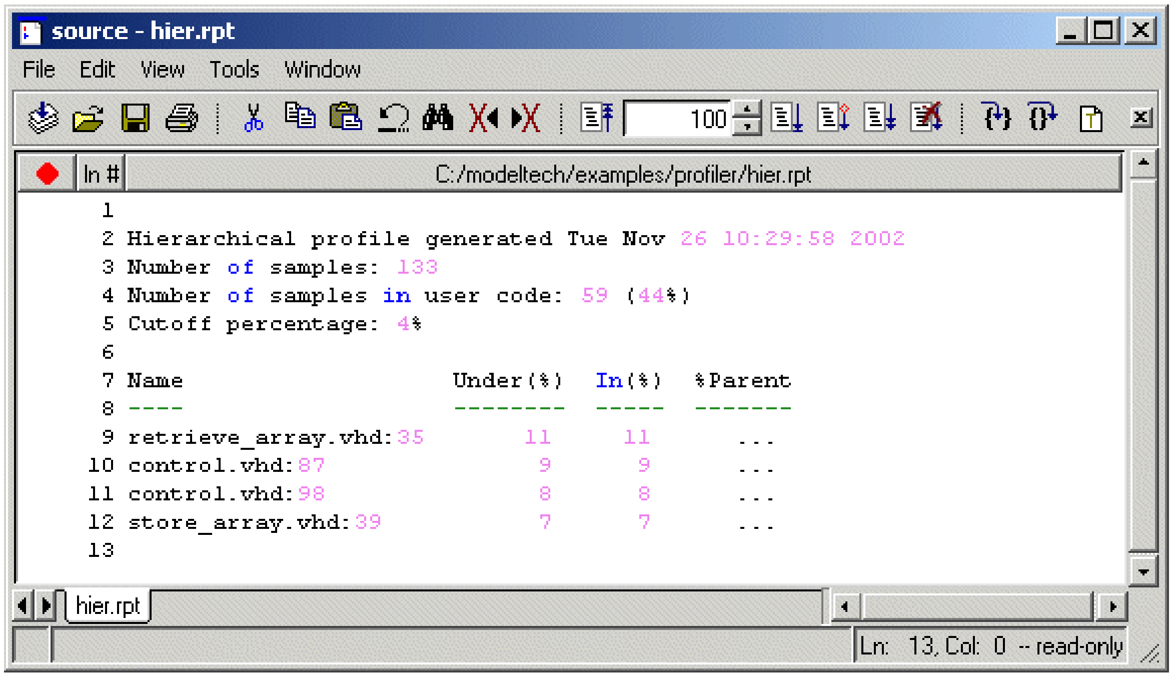
Reporting results

Either click the save icon on the toolbar or use the **profile report** command (CR-226) to save the Performance Analyzer results.

For example, the command

```
profile report -hierarchical -file hier.rpt -cutoff 4
```

will produce a profile report in a text file called *hier.rpt*, as shown here.


source - hier.rpt
File
Edit
View
Tools
Window
100
ln #
C:/modeltech/examples/profiler/hier.rpt
hier.rpt
Ln: 13, Col: 0 -- read-only

```

Hierarchical profile generated Tue Nov 26 10:29:58 2002
Number of samples: 133
Number of samples in user code: 59 (44%)
Cutoff percentage: 4%

Name                         Under(%)  In(%)  %Parent
----                         --------- -----  --------
retrieve_array.vhd:35            11     11       ...
control.vhd:87                    9      9       ...
control.vhd:98                    8      8       ...
store_array.vhd:39                7      7       ...

```

Profile menu

The following commands are available from the **Tools > Profile** menu (Main window).

Profile On	turn on the Performance Analyzer
Profile Off	turn off the Performance Analyzer
View hierarchical profile	view a hierarchical report of simulation performance; see "Interpreting the data" (UM-411)
View ranked profile	view a ranked report of simulation performance; see "Interpreting the data" (UM-411)
Clear Profile Data	clear current profile data

Performance Analyzer commands

The table below provides a brief description of the profile commands. See the *ModelSim Command Reference* for complete command details.

Command	**Description**
profile clear (CR-221)	clears any data that has been gathered during previous **run** commands; after this command is executed, all profiling data will be reset
profile interval (CR-222)	selects the frequency with which the profiler collects samples during a run command
profile off (CR-223)	disables runtime profiling
profile on (CR-224)	enables runtime analysis of where your simulation is spending its time
profile option (CR-225)	changes various profiling options
profile report (CR-226)	produces textual output of the profiling statistics that have been gathered up to the point at which you execute the command

Performance Analyzer preference variables

Various Tcl variables control how the Hierarchical Profile and Ranked Profile windows are displayed.You can set these preference variables by selecting **Tools > Edit Preferences > By Name > Profile** (Main window). Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory. See "Preference variables located in Tcl files" (UM-631) for more information.

12 - Code Coverage

Chapter contents

Introduction

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

With coverage enabled, ModelSim counts how many times each executable statement, branch, condition, expression, and logic node in each instance is executed during simulation. Statement coverage counts the execution of each statement on a line individually, even if there are multiple statements in a line. Branch coverage counts the execution of each conditional "if/then/else" and "case" statement and indicates when a true or false condition has not executed. Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. Expression coverage analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage. And toggle coverage counts each time a logic node transitions from one state to another.

Coverage statistics are displayed in the Main, Signals, and Source windows and also can be output in different text reports (see "Reporting coverage data" (UM-446)). Raw coverage data can be saved and recalled, or merged with coverage data from the current simulation (see "Saving and reloading coverage data" (UM-450)).

ModelSim Code Coverage offers these benefits:

- It is totally non-intrusive because it's integrated into the ModelSim engine – it doesn't require instrumented HDL code as do third-party coverage products.
- It has very little impact on simulation performance (typically 5 to 10 percent).
- It allows you to merge sets of coverage data without requiring elaboration of the design or a simulation license.

Usage flow for Code Coverage

The following is an overview of the usage flow for simulating with Code Coverage. More detailed instructions are presented in the sections that follow.

1. Compile the design using the **-cover bcest** argument to **vcom** (CR-303) or **vlog** (CR-345).
2. Simulate the design using the **-coverage** argument to **vsim** (CR-357).
3. Run the design.
4. Analyze coverage statistics in the Main, Signals, and Source windows.
5. Edit the source code to improve coverage.
6. Re-compile, re-simulate, and re-analyze the statistics and design.

Supported types

Code Coverage supports only certain data types.

VHDL

Supported types are scalar std_ulogic/std_logic. The tool doesn't currently support bit or boolean.

Vector and integer and real are not supported directly. However, subexpressions that involve an unsupported type and a relational operator and produce a boolean result are supported. These types of subexpressions are treated as an external expression that is first evaluated and then used as a boolean input to the full condition. The subexpression needs to look like:

```
(var <relop> const)
```

where "var" could be of type std_logic_vector, integer, or real; "<relop>" is a relational operator (e.g., <, >, >=); and "const" is a constant of the appropriate type. The tool doesn't currently support (var1 <relop> var2).

Verilog

Supported types are net and one-bit register, but subexpressions of the form:

```
(var1 <relop> var2)
```

are supported, where the variables may be multiple-bit registers or integer or real.

Important notes about coverage statistics

You should be aware of the following special circumstances related to calculating coverage statistics:

- When ModelSim optimizes a design, it "removes" unnecessary lines of code (e.g., code in a procedure that is never called). The lines that are optimized away aren't counted in the coverage data, and this may cause misleading results. As a result, when you compile with coverage enabled, ModelSim disables certain optimizations depending on which coverage types you choose. This produces more accurate statistics but also may slow simulation.

  The table below shows the coverage types and what ModelSim does to optimizations.

Coverage type	Effect on optimizations
statement	optimizations not disabled automatically; specify -O0 to get most accurate statistics
branch	case statement optimizations are disabled automatically
condition	optimizations not disabled automatically
expression	all optimizations disabled automatically
toggle	optimizations not disabled automatically

- Package bodies are not instance-specific: ModelSim sums the counts for all invocations no matter who the caller is. If you want separate statistics on each package, place them in separate files rather than mixing them with entities or architectures. Also, all standard and accelerated packages are ignored for coverage statistics calculation.

Enabling Code Coverage

Enabling Code Coverage is a two-step process:

1. Use the **-cover** argument to **vcom** or **vlog** when you compile your design. This argument tells ModelSim which coverage statistics to collect. For example:

   ```
   vlog top.v proc.v cache.v -cover bcesx
   ```

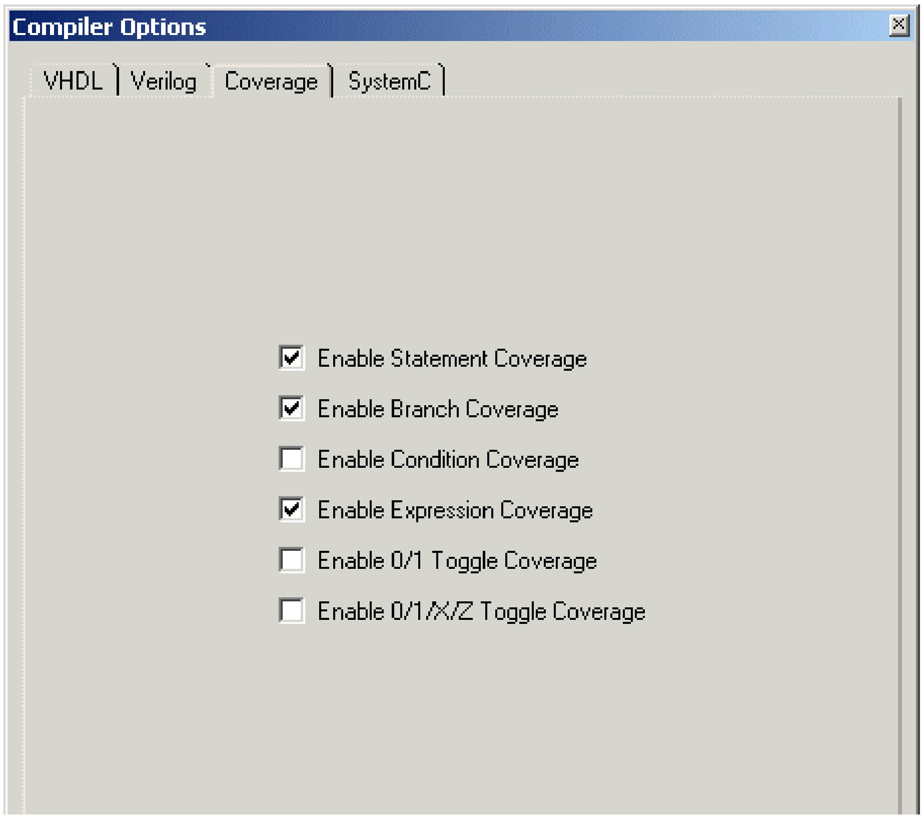
   Each character after the **-cover** argument identifies a type of coverage statistic: "**b**" indicates branch, "**c**" indicates condition, "**e**" indicates expression, "**s**" indicates statement, "**t**" indicates 2-transition toggle, and "**x**" indicates extended 6-transition toggle coverage (t and x are mutually exclusive). See "Enabling Toggle coverage" (UM-437) for details on two other methods for enabling toggle coverage.

2. Use the **-coverage** argument to **vsim** when you simulate your design. For example:

   ```
   vsim -coverage work.top
   ```

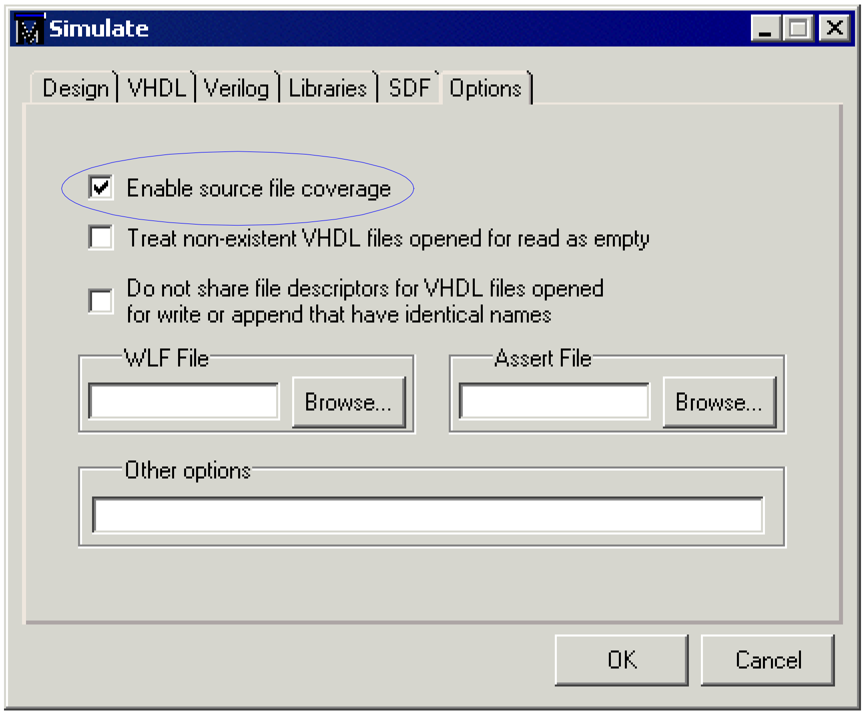
In ModelSim versions prior to 5.8, you didn't have to enable coverage at compile time. Code Coverage metrics (statement and branch coverage) were turned on just by using the **-coverage** argument to **vsim**. For backwards compatibility, ModelSim will still display statement statistics if you simulate with coverage enabled, even if you don't use the **-cover** argument when you compile the design.

To enable coverage from the graphic interface, first select **Compile > Compile Options** (Main window) and select the *Coverage* tab. Alternatively, if you are using a project, right-click on a selected design item (or items) and select **Properties**.



- [✓] Enable Statement Coverage
- [✓] Enable Branch Coverage
- [ ] Enable Condition Coverage
- [✓] Enable Expression Coverage
- [ ] Enable 0/1 Toggle Coverage
- [ ] Enable 0/1/X/Z Toggle Coverage

Next, select **Simulate > Simulate** (Main window) and check **Enable source file coverage** on the *Options* tab.



☑ Enable source file coverage

☐ Treat non-existent VHDL files opened for read as empty

☐ Do not share file descriptors for VHDL files opened for write or append that have identical names

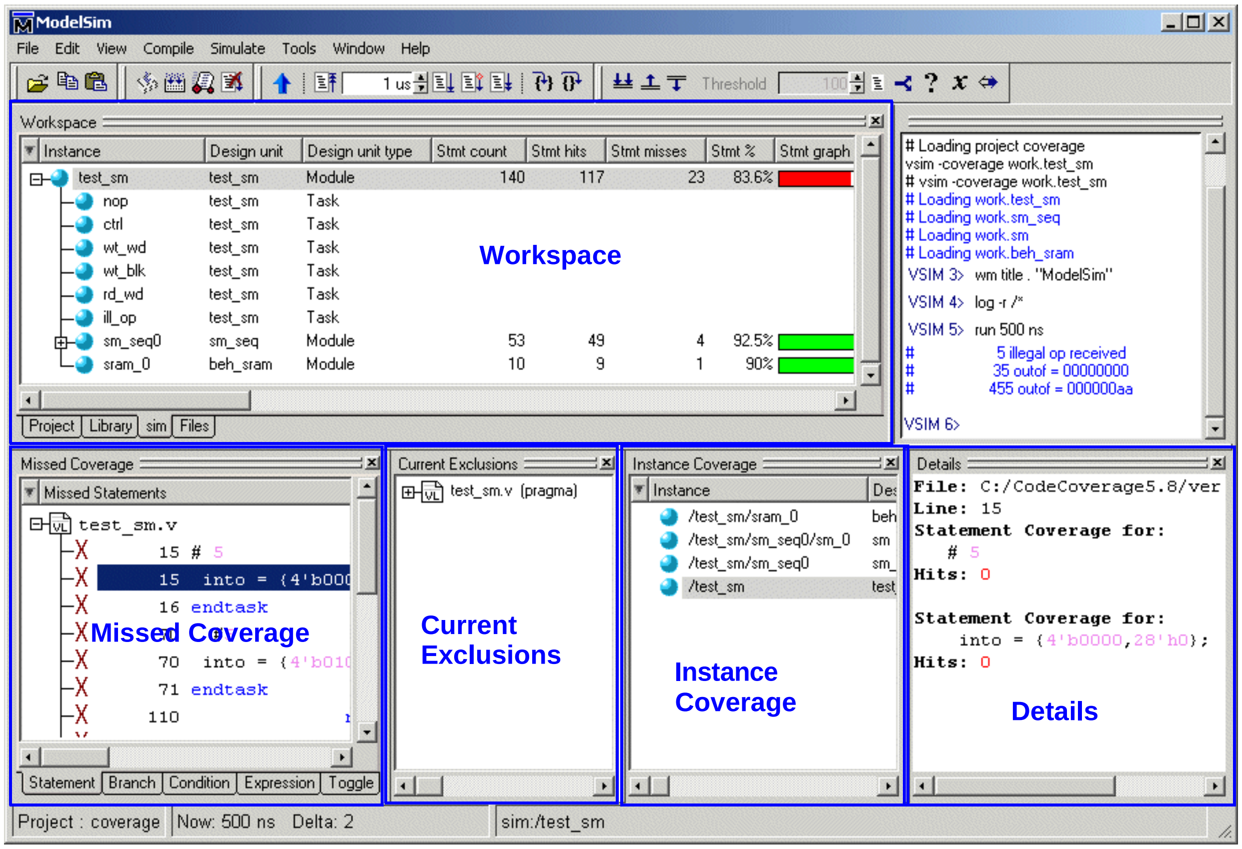
WLF File

Assert File

Other options

Viewing coverage data in the Main window

When you simulate a design with Code Coverage enabled, coverage data is displayed in the Main, Source, and Signals windows. In the Main window, coverage data displays in five window panes: Workspace, Missed Coverage, Current Exclusions, Instance Coverage, and Details.



Instance	Design unit	Design unit type	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt graph
test_sm	test_sm	Module	140	117	23	83.6%	🟥
nop	test_sm	Task					
ctrl	test_sm	Task					
wt_wd	test_sm	Task					
wt_blk	test_sm	Task					
rd_wd	test_sm	Task					
ill_op	test_sm	Task					
sm_seq0	sm_seq	Module	53	49	4	92.5%	🟩
sram_0	beh_sram	Module	10	9	1	90%	🟩

```
# Loading project coverage
vsim -coverage work.test_sm
# vsim -coverage work.test_sm
# Loading work.test_sm
# Loading work.sm_seq
# Loading work.sm
# Loading work.beh_sram
VSIM 3> wm title . "ModelSim"
VSIM 4> log -r /*
VSIM 5> run 500 ns
#              5 illegal op received
#             35 outof = 00000000
#            455 outof = 000000aa
VSIM 6>
```

```
test_sm.v
X     15 # 5
X     15  into = {4'b000
X     16 endtask
X     69
X     70  into = {4'b010
X     71 endtask
X    110
```

```
test_sm.v (pragma)
```

Instance	Des
/test_sm/sram_0	beh
/test_sm/sm_seq0/sm_0	sm
/test_sm/sm_seq0	sm_
/test_sm	test

```
File: C:/CodeCoverage5.8/ver
Line: 15
Statement Coverage for:
   # 5
Hits: 0

Statement Coverage for:
     into = {4'b0000,28'h0};
Hits: 0
```

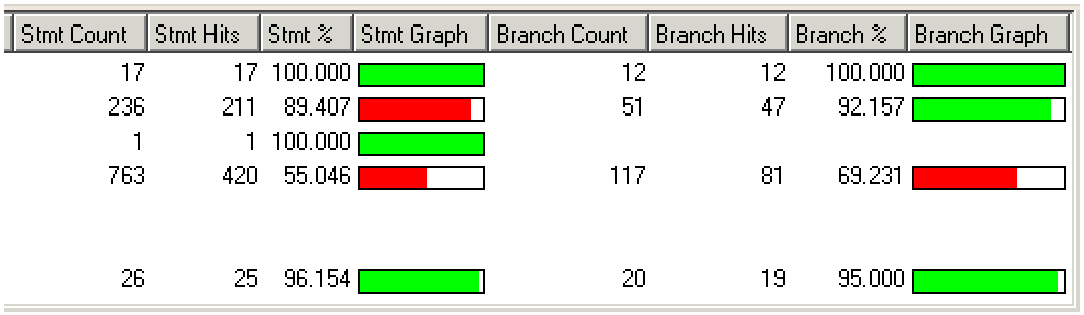
Workspace pane

The Workspace pane displays code coverage information in the Files tab and in the tabs displaying structure for any datasets being simulated (e.g., the *sim* tab). When coverage is invoked, several columns for displaying coverage data are added to the Workspace pane. You can toggle columns on/off by right-clicking on a column name and selecting from the context menu that appears. The following columns are relevant to the Workspace pane:

Column name	Description
Design unit	the name of the design unit
Design unit type	the type (e.g., Module, Entity, etc.) of the design unit
Stmt count	the number of executable statements in each file
Stmt hits	the number of executable statements that have been executed in the current simulation
Stmt misses	the number of executable statements that were not executed in the current simulation
Stmt %	the current ratio of Stmt hits to Stmt count
Stmt graph	a bar chart displaying the Stmt %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Branch count	the number of executable branches in each file
Branch hits	the number of executable branches that have been executed in the current simulation
Branch misses	the number of executable branches that were not executed in the current simulation
Branch %	the current ratio of Branch hits to Branch count
Branch graph	a bar chart displaying the Branch %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Condition rows	the number of conditions in each file
Condition hits	the number of times the conditions in a file have been executed
Condition misses	the number of conditions in a file that were not executed
Condition %	the current ratio of Condition hits to Condition rows
Condition graph	a bar chart displaying the Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable

Column name	Description
Expression rows	the number of executable expressions in each file
Expression hits	the number of times expressions in a file have been executed
Expression misses	the number of executable expressions in a file that were not executed
Expression %	the current ratio of Expression hits to Expression rows
Expression graph	a bar chart displaying the Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Toggle nodes	the number of points in each instance where the logic will transition from one state to another
Toggle hits	the number of nodes in each instance that have transitioned at least once
Toggle misses	the number of nodes in each instance that have not transitioned at least once
Toggle %	the current ratio of Toggle hits to Toggle nodes
Toggle graph	a bar chart displaying the Toggle %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable

The diagram below show a portion of the Workspace window pane with code coverage data displayed.

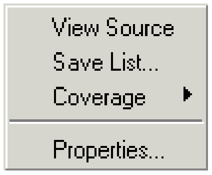


Stmt Count	Stmt Hits	Stmt %	Stmt Graph	Branch Count	Branch Hits	Branch %	Branch Graph
17	17	100.000	🟩	12	12	100.000	🟩
236	211	89.407	🟥	51	47	92.157	🟩
1	1	100.000	🟩				
763	420	55.046	🟥	117	81	69.231	🟥
26	25	96.154	🟩	20	19	95.000	🟩

You can sort code coverage information for any column by clicking the column heading. Clicking the column heading again will reverse the order.

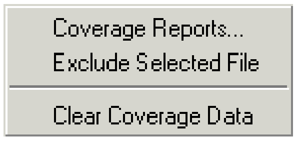
Workspace context menu

When you right-click in the Files tab of the Workspace pane, you open the following context menu.



The menu includes the following options:

- **View Source**
  Allows you to view the selected file in the Source window.
- **Save List**
  Opens the Save File List dialog and allows you to save the coverage statistics for the selected file in a text file.
- **Coverage**
  Opens a submenu that allows you to generate coverage reports, exclude the selected file from the coverage statistics (this selection will cause the file to appear in the Current Exclusions pane), or clear coverage data.

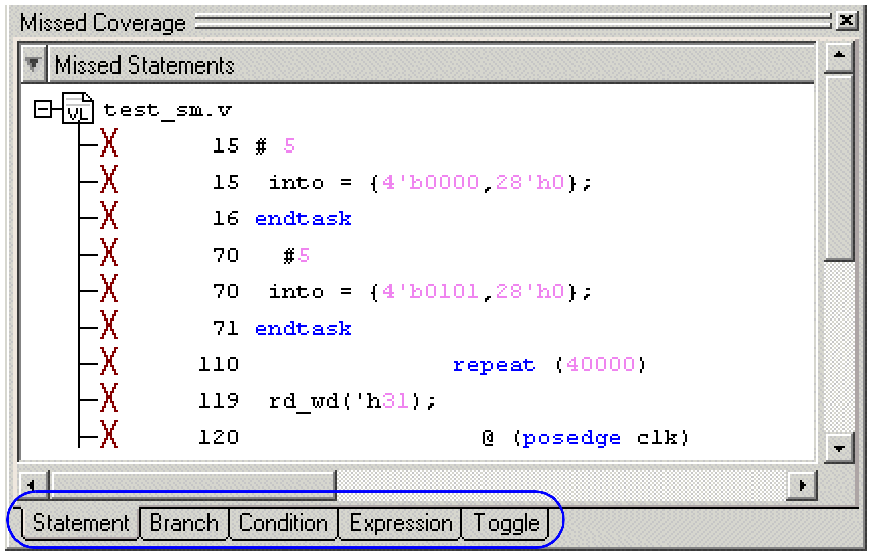


- **Properties**
  Opens the File Properties dialog box, which displays the file name, location, MS-DOS name (full pathname), file size, the last time the file was modified, and file attributes.

Coverage information in the Workspace pane is dynamically linked to the Missed Coverage pane and the Current Exclusions pane. Click the left mouse button on any file in the Workspace pane to display that file’s un-executed statements, branches, conditions, expressions, and toggles in the Missed Coverage pane. Lines from the selected file that are excluded from coverage statistics are displayed in the Current Exclusions pane.

Missed Coverage pane

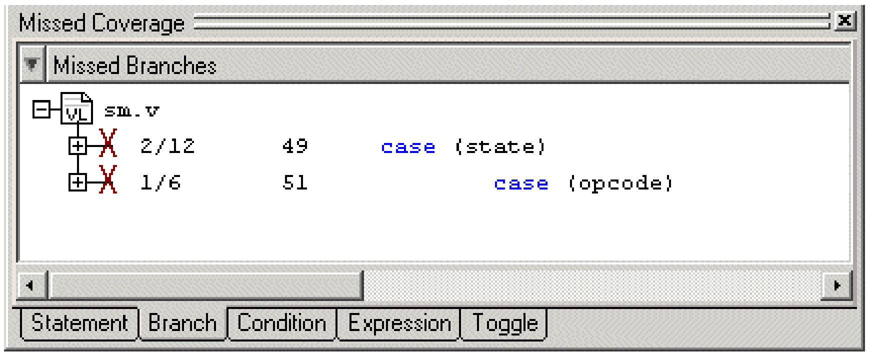
When you select a file in the Workspace pane, the Missed Coverage pane displays that file's un-executed statements, branches, conditions, and expressions and signals that haven't toggled. The pane includes a tab for each item, as shown below.



```
test_sm.v
X        15 # 5
X        15  into = {4'b0000,28'h0};
X        16 endtask
X        70   #5
X        70  into = {4'b0101,28'h0};
X        71 endtask
X       110                  repeat (40000)
X       119  rd_wd('h31);
X       120                     @ (posedge clk)
```

Each tab includes a column for the line number and a column for statement, branch, condition, expression, or toggle on that line. The "X" indicates the item was not executed.

The Branch tab also includes a column for branch code (conditional "if/then/else" and "case" statements). "X_T" indicates that only the true condition of the branch was not executed. "X_F" indicates that only the false condition of the branch was not executed. Fractional numbers indicate how many case statement labels were not executed. For example, if only one of four case labels executed, the Branch tab would indicate "X 1/4."

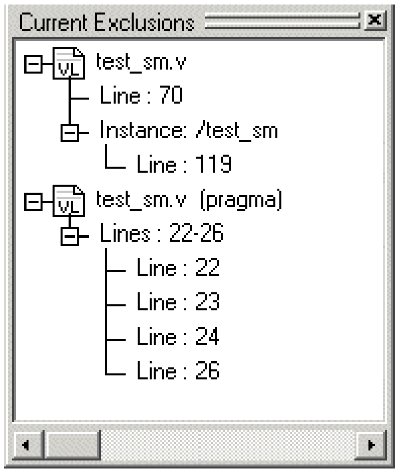


```
sm.v
X  2/12      49      case (state)
X  1/6       51              case (opcode)
```

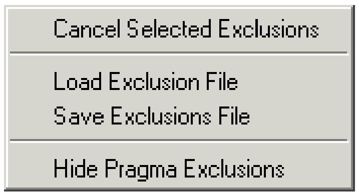
When you right-click any item in the Statement, Branch, Condition, or Expression tabs an **Exclude Selection** button will pop up, allowing you to exclude the item from coverage statistics and make it appear in the Current Exclusions pane.

Current Exclusions pane

The Current Exclusions pane lists all files and lines that are excluded from coverage statistics. See "Excluding items from coverage" (UM-443) for more details.



The **Current Exclusions** pane offers a pop-up menu with commands for controlling exclusions. Right-click anywhere in the pane to access the following commands:

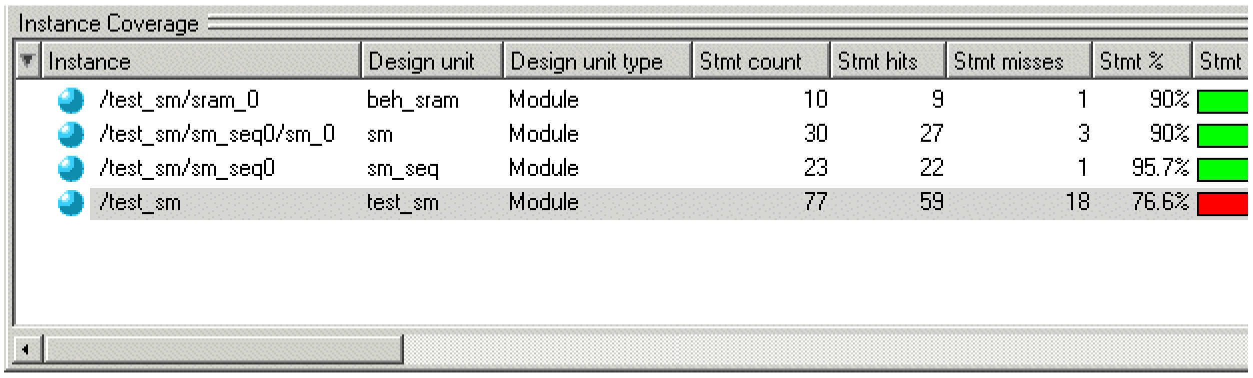


- **Cancel Selected Exclusions**
  Cancels exclusion filtering for selected lines or files and places them back into the coverage statistics.
- **Load Exclusion File**
  Opens the Load Exclusion File dialog, allowing you to select a saved exclusion file for loading. Eliminates the need to create a new exclusion file for every simulation run.
- **Save Exclusions File**
  Opens the Save Current Exclusions dialog, allowing you to name and save all current exclusions in a single file for later recall. Loading a saved exclusion file eliminates the need to create a new exclusion file for every simulation run.
- **Hide (Show) Pragma Exclusions**
  Toggles the Current Exclusions pane to hide or show VHDL and Verilog pragma exclusions.

Instance Coverage pane

The Instance Coverage pane displays coverage statistics for each instance in a flat, non-hierarchical view. The Instance Coverage pane contains the same code coverage statistics columns as in the "Workspace pane" (UM-427)

A partial view of the Instance Coverage pane is shown below.

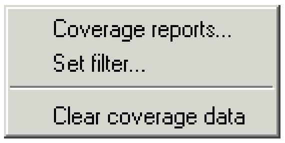


Instance Coverage

Instance	Design unit	Design unit type	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt
/test_sm/sram_0	beh_sram	Module	10	9	1	90%	🟩
/test_sm/sm_seq0/sm_0	sm	Module	30	27	3	90%	🟩
/test_sm/sm_seq0	sm_seq	Module	23	22	1	95.7%	🟩
/test_sm	test_sm	Module	77	59	18	76.6%	🟥

Instance coverage pane context menu

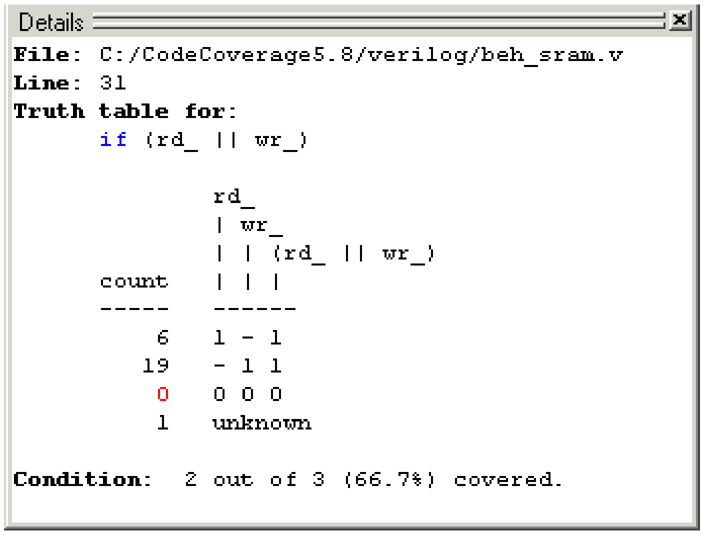
Right-click any item in the Instance Coverage pane to open a pop-up menu that allows you to create reports, set a display filter, or clear coverage data for the design.



- **Coverage reports** opens the Coverage Report dialog, which allows you to create a number of different code coverage reports (see "Reporting coverage data" (UM-446) for details).
- **Set filter** opens the Filter instance list dialog, which allows you to filter coverage statistics (see "Filtering coverage data" (UM-441) for details).
- **Clear coverage data** clears all coverage statistics for every item in the design.

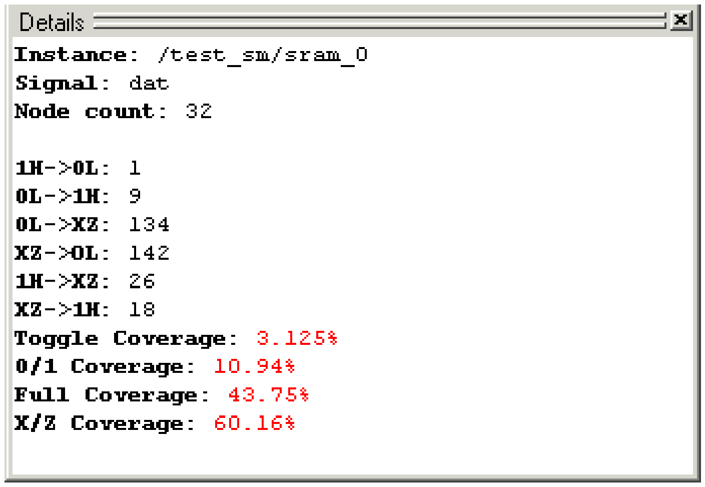
Details pane

After code coverage is invoked and the simulation is loaded and run you can turn on the Details pane by selecting **View** > **Coverage** > **Details** in the Main window. The Details pane shows the details of missed coverage. When an item is selected in the Missed Coverage pane, the details of that coverage are displayed in the Details pane. Truth tables will be displayed for condition and expression coverage, as shown here.



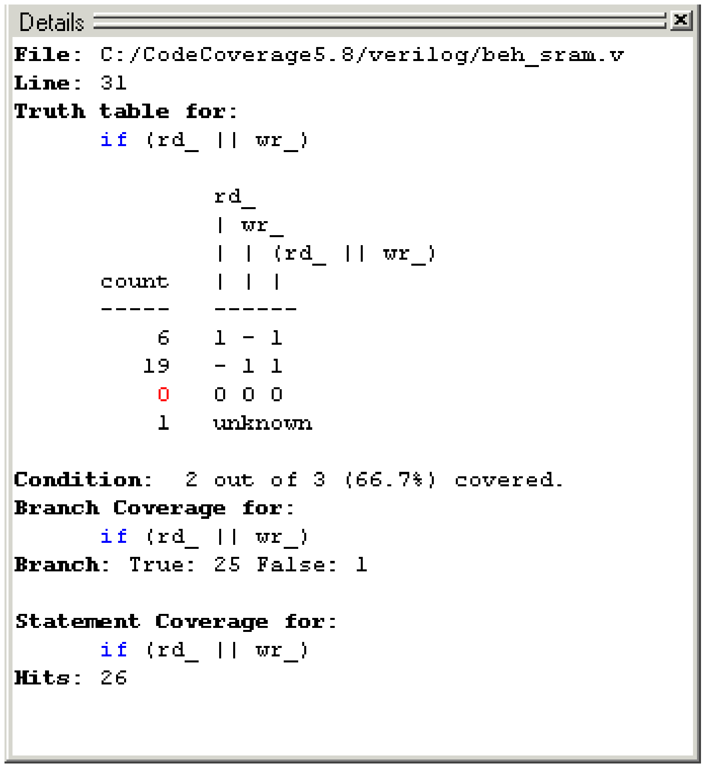
```
Details
File: C:/CodeCoverage5.8/verilog/beh_sram.v
Line: 31
Truth table for:
      if (rd_ || wr_)

              rd_
              | wr_
              | | (rd_ || wr_)
      count   | | |
      -----   ------
          6   1 - 1
         19   - 1 1
          0   0 0 0
          1   unknown

Condition:  2 out of 3 (66.7%) covered.
```

Toggle details are displayed as follows:



```
Details
Instance: /test_sm/sram_0
Signal: dat
Node count: 32

1H->0L: 1
0L->1H: 9
0L->XZ: 134
XZ->0L: 142
1H->XZ: 26
XZ->1H: 18
Toggle Coverage: 3.125%
0/1 Coverage: 10.94%
Full Coverage: 43.75%
X/Z Coverage: 60.16%
```

By clicking the left mouse button on the statement Hits column in the Source window, all coverage information for that line will be displayed in the Details pane as shown here:

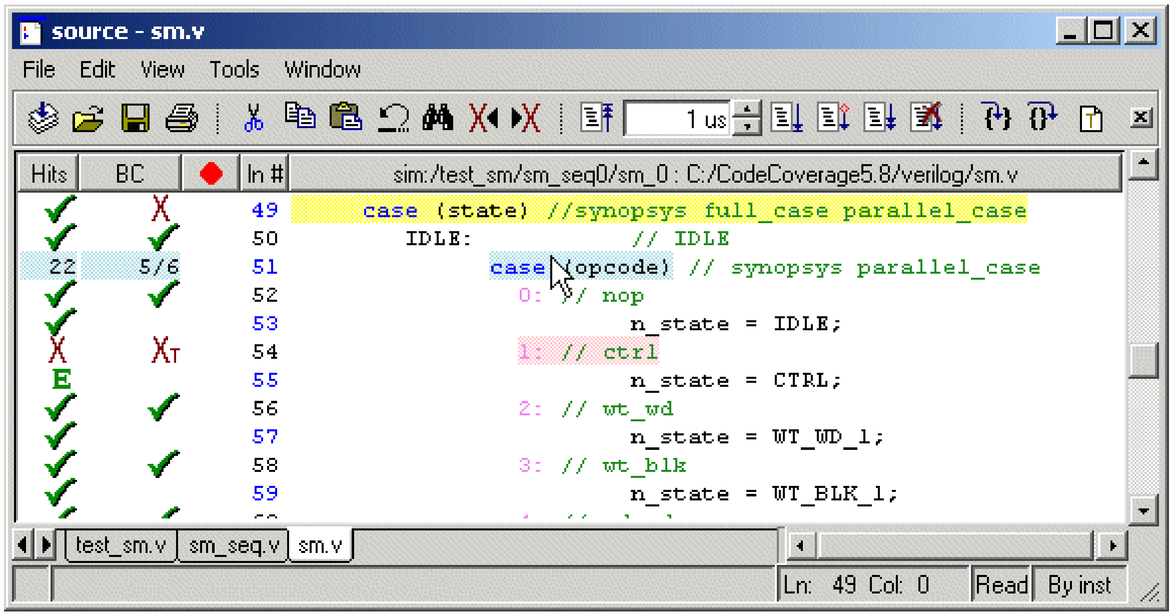


```
File: C:/CodeCoverage5.8/verilog/beh_sram.v
Line: 31
Truth table for:
      if (rd_ || wr_)

              rd_
              | wr_
              | | (rd_ || wr_)
      count   | | |
      -----   ------
          6   1 - 1
         19   - 1 1
          0   0 0 0
          1   unknown

Condition:  2 out of 3 (66.7%) covered.
Branch Coverage for:
      if (rd_ || wr_)
Branch: True: 25 False: 1

Statement Coverage for:
      if (rd_ || wr_)
Hits: 26
```

Viewing coverage data in the Source window

The Source window (UM-325) includes two columns for code coverage statistics – the Hits column and the BC (Branch Coverage) column. These columns provide an immediate visual indication about how your source code is executing. The default code coverage indicators are check marks and Xs.

- A green check mark indicates that the statements and/or branches in a particular line have executed.
- A red X indicates that a statement or branch was not executed.
- An X_T indicates the true branch of an conditional statement was not executed.
- An X_F indicates the false branch was not executed.
- A green "E" indicates a line of code that has been excluded from code coverage statistics.

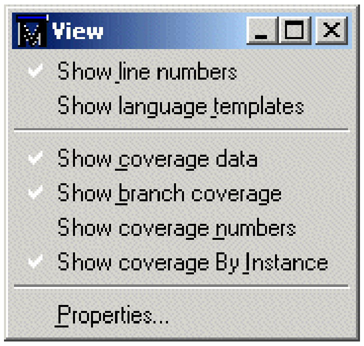


```
Hits  BC   ●  ln #  sim:/test_sm/sm_seq0/sm_0 : C:/CodeCoverage5.8/verilog/sm.v
 ✓    X       49      case (state) //synopsys full_case parallel_case
 ✓    ✓       50        IDLE:              // IDLE
 22   5/6     51              case (opcode) // synopsys parallel_case
 ✓    ✓       52                0: // nop
 ✓            53                       n_state = IDLE;
 X    X_T     54                1: // ctrl
 E            55                       n_state = CTRL;
 ✓    ✓       56                2: // wt_wd
 ✓            57                       n_state = WT_WD_1;
 ✓    ✓       58                3: // wt_blk
 ✓            59                       n_state = WT_BLK_1;
```

When you hover the cursor over a line of code (see line 51 in the illustration above), the number of statement and branch executions, or "hits," will be displayed in place of the check marks and Xs. Notice, in this illustration, five of six conditions have been executed.

Also, when you click in either the Hits or BC column, the Details pane in the Main window updates to display information on that line.

The Source window **View** menu provides five options for displaying coverage statistics:



- **Show line numbers** toggles the *ln #* column off and on.
- **Show coverage data** toggles the *Hits* column off and on.
- **Show branch coverage** toggles the *BC* column off and on.
- **Show coverage numbers** displays the number of executions in the *Hits* and *BC* columns rather than checkmarks and Xs. When multiple statements occur on a single line an ellipsis ("...") replaces the Hits number. In such cases, hover the cursor over each statement to highlight it and display the number of executions for that statement.
- **Show coverage By Instance** displays only the number of executions for the currently selected instance (in the Main window workspace).

You can skip to "missed lines" three ways: select **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** from the menu bar; click the Previous zero hits and Next zero hits icons on the toolbar; or press <Shift> - <Tab> (previous miss) or Tab (next miss).

Toggle coverage

Toggle coverage is the ability to count and collect changes of state on specified nodes, including Verilog nets and registers and the following VHDL signal types: bit, bit_vector, std_logic, and std_logic_vector. Toggle coverage is integrated as a metric into the coverage tool so that the use model and reporting are the same as the other coverage metrics.

There are two modes of toggle coverage operation - standard and extended. Standard toggle coverage only counts Low or 0 <--> High or 1 transitions. Extended toggle coverage counts these transitions plus the following:

X or Z --> 1 or H

X or Z --> 0 or L

1 or H --> X or Z

0 or L --> X or Z

This extended coverage allows a more detailed view of testbench effectiveness and is especially useful for examining coverage of tri-state signals. It helps to ensure, for example, that a bus has toggled from high 'Z' to a '1' or '0', and a '1' or '0' back to a high 'Z'.

Enabling Toggle coverage

In the Enabling Code Coverage (UM-423) section we explained that toggle coverage could be enabled during compile by using the ’t’ or ’x’ arguments with **vcom -cover** or **vlog -cover**. This section describes two other methods for enabling toggle coverage:

1. using the **toggle add** command (CR-271)

2. using the **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** selections in the Signals window menu.

Using the toggle add *command*

The **toggle add** command allows you to initiate toggle coverage at any time from the command line. (See the Command Reference (CR-271) for correct syntax and arguments.) Upon the next running of the simulation, toggle coverage data will be collected according to the arguments employed (i.e., the **-full** argument enables collection of extended toggle coverage statistics for the six transitions mentioned above).

Using the Signals window menu selections

You can enable toggle coverage by selecting **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** from the Signals window menu. These selections allow you to enable toggle coverage for Selected Signals, Signals in Region, or Signals in Design.

After making a selection, toggle coverage statistics will be captured the next time you run the simulation.

Excluding nodes from Toggle coverage

You can disable toggle coverage with the **toggle disable** command (CR-273). This command disables toggle statistics collection on the specified nodes and provides a method of implementing coverage exclusions for toggle coverage. It is intended to be used as follows:

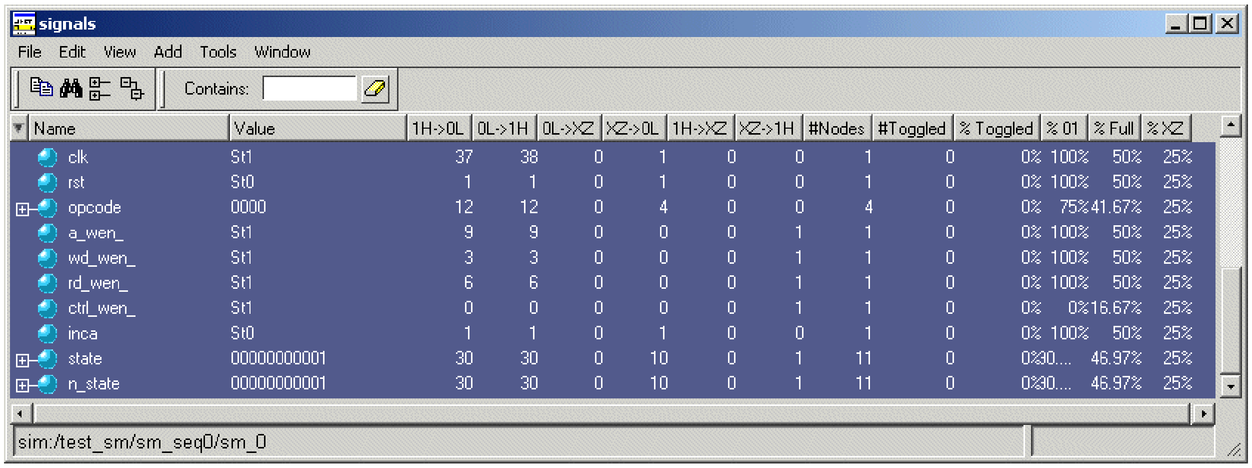
1. Enable toggle statistics collection for all signals using the **-cover t/x** argument to **vcom** or **vlog**.

2. Exclude certain signals by disabling them with the **toggle disable** command.

The **toggle enable** command (CR-274) re-enables toggle statistics collection on nodes whose toggle coverage has previously been disabled via the toggle disable command. (See the Command Reference for correct syntax.)

Viewing toggle coverage data in the Signals window

Toggle coverage data is displayed in the Signals window in multiple columns, as shown below. There is a column for each of the six transition types.



Name	Value	1H->0L	0L->1H	0L->XZ	XZ->0L	1H->XZ	XZ->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% XZ
clk	St1	37	38	0	1	0	0	1	0	0%	100%	50%	25%
rst	St0	1	1	0	1	0	0	1	0	0%	100%	50%	25%
opcode	0000	12	12	0	4	0	0	4	0	0%	75%	41.67%	25%
a_wen_	St1	9	9	0	0	0	1	1	0	0%	100%	50%	25%
wd_wen_	St1	3	3	0	0	0	1	1	0	0%	100%	50%	25%
rd_wen_	St1	6	6	0	0	0	1	1	0	0%	100%	50%	25%
ctrl_wen_	St1	0	0	0	0	0	1	1	0	0%	0%	16.67%	25%
inca	St0	1	1	0	1	0	0	1	0	0%	100%	50%	25%
state	00000000001	30	30	0	10	0	1	11	0	0%	30....	46.97%	25%
n_state	00000000001	30	30	0	10	0	1	11	0	0%	30....	46.97%	25%

sim:/test_sm/sm_seq0/sm_0

Right click any column name to toggle that column on or off.

The following table provides a description of the available columns:

Column name	Description
Name	the name of each signal in the current region
Value	the current value of each signal
1H -> 0L	the number of times each signal has transitioned from a 1 or a High state to a 0 or a Low state
0L -> 1H	the number of times each signal has transitioned from a 0 or a Low state to 1 or a High state
0L -> XZ	the number of times each signal has transitioned from a 0 or a Low state to an unknown (X) or a high impedance (Z) state
XZ -> 0L	the number of times each signal has transitioned from an unknown or high impedance state to a 0 or a Low state
1H -> XZ	the number of times each signal has transitioned from a 1 or a High state to an unknown or a high impedance state
XZ -> 1H	the number of times each signal has transitioned from an unknown or a high impedance state to 1 or a High state
# Nodes	the number of scalar bits in each signal
# Toggled	the number of nodes that have transitioned at least once
% Toggled	the current ration of the # Toggled to the # Nodes for each signal

Column name	Description
% 01	the percentage of **1H -> 0L** and **0L -> 1H** transitions that have occurred (transitions in the first two columns)
% Full	the percentage of all transitions that have occurred (all six columns)
% XZ	the percentage of **0L -> XZ**, **XZ -> 0L**, **1H -> XZ**, and **XZ -> 1H** transitions that have occurred (last four colmns)

Toggle coverage reporting

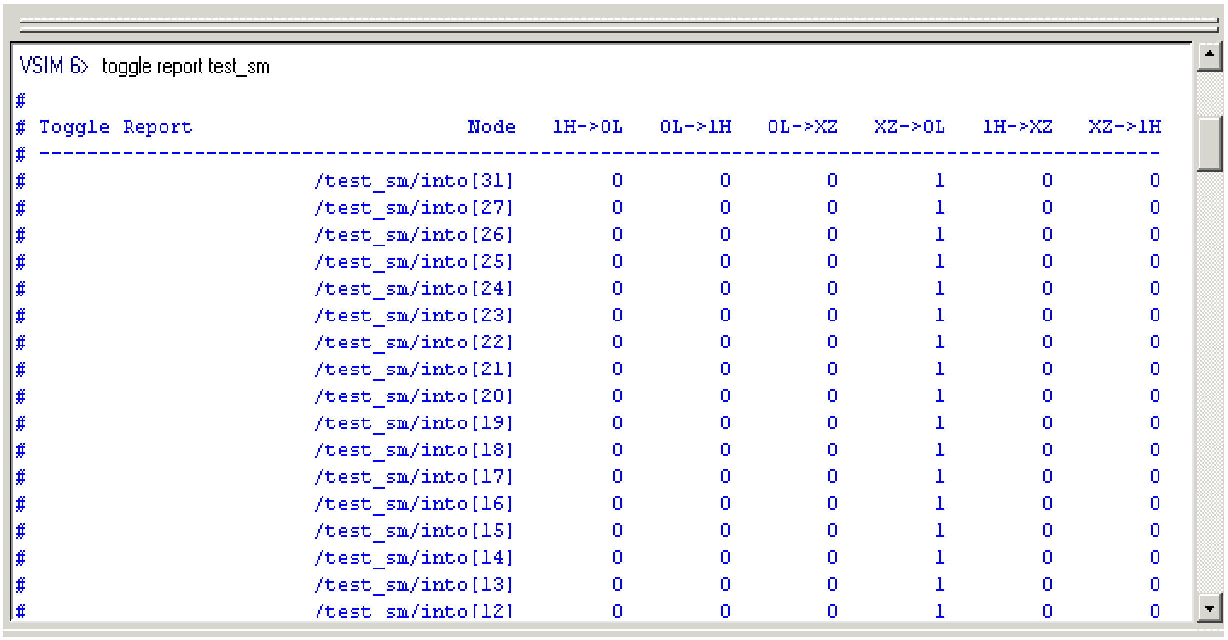
The **toggle report** command (CR-275) displays a list of all nodes that have not transitioned at least once. Also displayed is a summary of the number of nodes checked, the number that toggled, the number that didn't toggle, and a percentage that toggled.

The **toggle report** command is intended to be used as follows:

1 Enable statistics collection with the **toggle add** command (CR-271).

2 Run the simulation with the **run** command (CR-246).

3 Produce the report with the **toggle report** command..



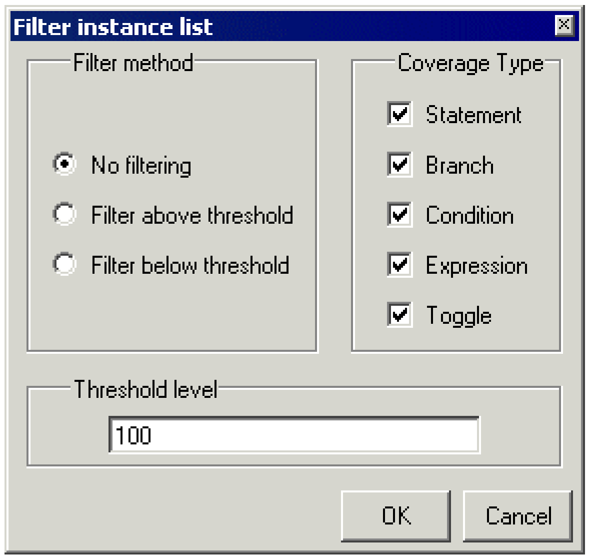
```
VSIM 6> toggle report test_sm
#
# Toggle Report                         Node   1H->0L   0L->1H   0L->XZ   XZ->0L   1H->XZ   XZ->1H
# -------------------------------------------------------------------------------------------------
#                       /test_sm/into[31]       0        0        0        1        0        0
#                       /test_sm/into[27]       0        0        0        1        0        0
#                       /test_sm/into[26]       0        0        0        1        0        0
#                       /test_sm/into[25]       0        0        0        1        0        0
#                       /test_sm/into[24]       0        0        0        1        0        0
#                       /test_sm/into[23]       0        0        0        1        0        0
#                       /test_sm/into[22]       0        0        0        1        0        0
#                       /test_sm/into[21]       0        0        0        1        0        0
#                       /test_sm/into[20]       0        0        0        1        0        0
#                       /test_sm/into[19]       0        0        0        1        0        0
#                       /test_sm/into[18]       0        0        0        1        0        0
#                       /test_sm/into[17]       0        0        0        1        0        0
#                       /test_sm/into[16]       0        0        0        1        0        0
#                       /test_sm/into[15]       0        0        0        1        0        0
#                       /test_sm/into[14]       0        0        0        1        0        0
#                       /test_sm/into[13]       0        0        0        1        0        0
#                       /test_sm/into[12]       0        0        0        1        0        0
```

You can produce this same information using the **coverage report** command (CR-137).

Filtering coverage data

You can specify a percentage above or below which you don't want to see coverage statistics. For example, you might set a threshhold of 85% such that only items with coverage below that percentage are displayed. Anything above that percentage is filtered.

You can set a filter using either a dialog or toolbar icons (see below). To access the dialog, right-click any item in the Instance Coverage pane and select **Set Filter**.


Filter instance list

Filter method
[● No filtering] ○ Filter above threshold ○ Filter below threshold

Coverage Type
☑ Statement
☑ Branch
☑ Condition
☑ Expression
☑ Toggle

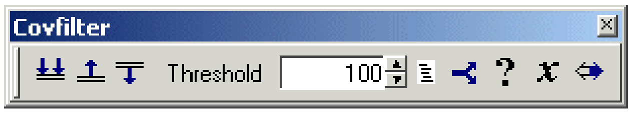
Threshold level: 100

The dialog has the following options:

- **Filter method**
  Specifies whether you want to filter items that exceed the threshold or fall below the threshold.
- **Coverage Type**
  Determines which coverage statistics you want to filter.
- **Threshold level**
  Specifies the percentage above or below which items are filtered.

Covfilter toolbar

When you simulate with Code Coverage enabled, the Covfilter toolbar is added to the Main window.



The toolbar has the following buttons:.

Covfilter toolbar buttons

Button	
	Enable Filtering enables display filtering of coverage statistics in the Workspace and Instance Coverage panes of the Main window
	Threshold above displays all coverage statistics above the Filter Threshold for selected columns
	Threshold below displays all coverage statistics below the Filter Threshold for selected columns
100	**Filter Threshold** specifies the display coverage percentage for the selected coverage columns
	Statement applies the display filter to all Statement coverage columns in the Workspace and Instance Coverage panes of the Main window
	Branch applies the display filter to all Branch coverage columns in the Workspace and Instance Coverage panes of the Main window
?	**Condition** applies the display filter to all Condition coverage columns in the Workspace and Instance Coverage panes of the Main window
	Expression applies the display filter to all Expression coverage columns in the Workspace and Instance Coverage panes of the Main window
	Toggle applies the display filter to all Toggle coverage columns in the Workspace and Instance Coverage panes of the Main window

Excluding items from coverage

You can exclude any number of lines or entire files so ModelSim doesn't collect statistics on them. The line exclusions can be instance-specific or they can apply to all instances in the enclosing design unit. You can also exclude nodes from toggle statistics collection using the **toggle disable** command (CR-273).

There are three methods for excluding lines and files:

- Use a popup menu command in the GUI
- Insert pragmas into your source code
- Create an exclusion filter file

Excluding lines/files via the GUI

There are several locations in the GUI where you can access commands to exclude lines or files:

- Right-click a file in the Main window Workspace pane and select **Coverage > Exclude Selected File** from the popup menu.
- Right-click an entry in the Main window Missed Coverage pane and select **Exclude Selection** or **Exclude Selection For Instance <inst_name>** from the popup menu.
- Right-click a line in the Hits column of the Source window and select **Exclude Coverage Line xxx**, **Exclude Coverage Line xxx For Instance <inst_name>**, or **Exclude Entire File**.

Excluding lines/files with pragmas

ModelSim also supports the use of source code pragmas to selectively turn coverage off and on. In Verilog, the pragmas are:

```
// coverage off
// coverage on
```

In VHDL, the pragmas are:

```
-- coverage off
-- coverage on
```

Bracket the line or lines you want to exclude with these pragmas.

▶ **Note:** Pragmas cannot be used to exclude specific conditions or expressions within lines.

Excluding lines/files with a filter file

Exclusion filter files specify files and line numbers that you wish to exclude from the coverage statistics. You can create the filter file in any text editor or save the current filter in the Source window by selecting **File > Save > Exclusion File** (Main window). To load the filter during a future analysis, select **File > Open > Exclusion File** (Main window).

Syntax

```
<filename>...
[[<range> ...] [<line#> ...]] | all

or

begin instance <instance_name>...
<inst_filename>...
[[<range> ...] [<line#> ...]] | all
end instance
```

Arguments

`<filename>`
The name of the file you want to exclude. Required if you are not specifying an instance. The filter file may include an unlimited number of filename entries, each on its own line. You may use environment variables in the pathname.

`begin instance <instance_name>`
The name of an instance for which you want to exclude lines. Required if you don't specify <filename>. The filter file may include an unlimited number of instances.

`<inst_filename>`
The name of the file(s) that compose the instance from which you are excluding lines. Optional.

`<range> ...`
A range of line numbers you want to exclude. Optional. Enter the range in "# - #" format. For example, 32 - 35. You can specify multiple ranges separated by spaces.

`<line#> ...`
A line number that you want to exclude. Optional. You can specify multiple line numbers separated by spaces.

`all`
Specifies that all lines in the file should be excluded. Required if a range or line number is not specified.

Example

```
control.vhd
  72 - 76 84 93
testring.vhd
  all
begin instance /test_delta/chip/bid01_inst
  src/delta/buffers.vhd
    45-46
end instance
```

Default filter file

The Tcl preference variable **PrefCoverage(pref_InitFilterFrom)** specifies a default filter file to read when a design is loaded with the **-coverage** switch. By default this variable is not set. See "Code Coverage preference variables" (UM-454) for details on changing this variable.

A file named *workingExclude.cov* appears in the design directory when you specify exclusions in the GUI. This file remains after quitting simulation.

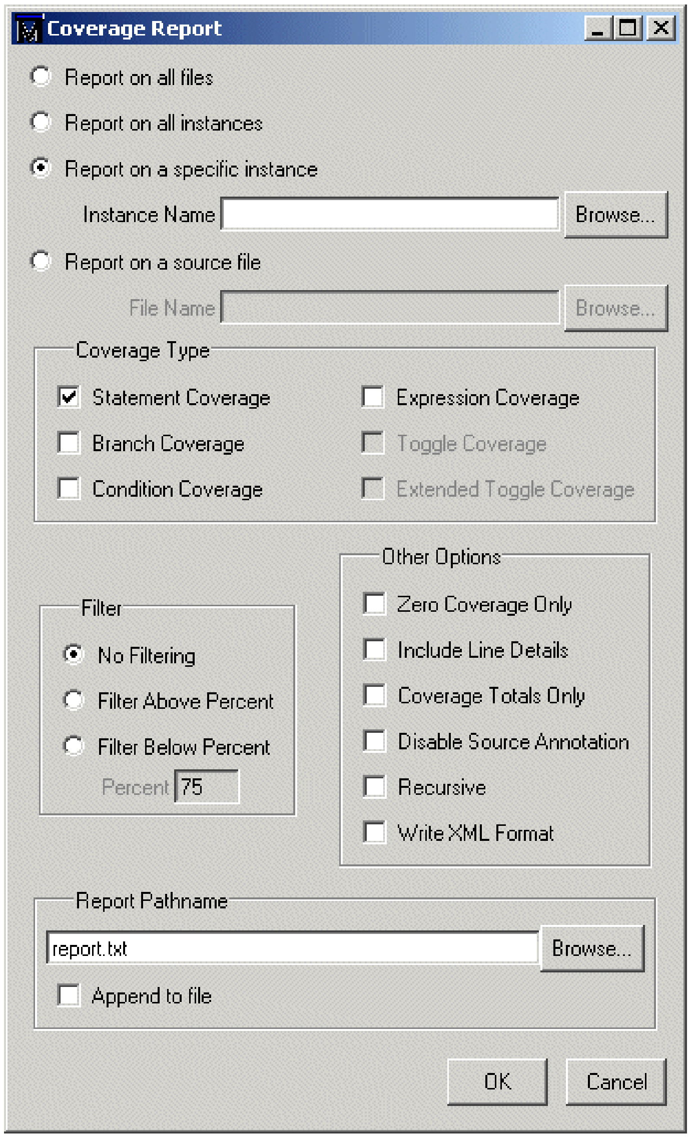
Excluding nodes from toggle statistics

To exclude nodes from toggle statistics collection, use the **toggle disable** command (CR-273).

Reporting coverage data

To create reports on coverage statistics, use either the **coverage report** command (CR-137), the **toggle report** command (CR-275) (see Toggle coverage reporting (UM-440) in this chapter), or the Coverage Report dialog.

To access the Coverage Report dialog, right-click any item in the *Files* tab of the Workspace pane and select **Coverage > Coverage Reports**; or, select **Tools > Coverage > Reports** (Main window).



Report on all files
Report on all instances
[✓ Report on a specific instance]
Instance Name
Report on a source file
File Name

Coverage Type
[✓ Statement Coverage]
Branch Coverage
Condition Coverage
Expression Coverage
Toggle Coverage
Extended Toggle Coverage

Filter
[✓ No Filtering]
Filter Above Percent
Filter Below Percent
Percent 75

Other Options
Zero Coverage Only
Include Line Details
Coverage Totals Only
Disable Source Annotation
Recursive
Write XML Format

Report Pathname
report.txt
Append to file

The dialog contains these options:

- **Report on all files**
  Saves a textual summary for each file in the design.
- **Report on all instances**
  Saves a textual summary for each instance in the design.

- **Report on a specific instance**
  Saves a textual summary for the specified instance. The selected instance automatically appears in the *Instance Name* field. You can browse for other instances.
- **Report on a source file**
  Saves a textual summary for the specified source file. The selected file automatically appears in the *File Name* field. You can browse for other source files.

The **Coverage Type** section of the dialog allows you to select the type of coverage to be reported – statement, branch, condition, expression, toggle, and extended toggle coverage.

The Coverage Report dialog includes options for filtering report data according to coverage percent. The default is No Filtering.

- **Zero Coverage Only**
  Saves a textual summary of statement and branch coverage that includes columns for the number of statements and branches not executed.
- **Include Line Details**
  Saves a detailed textual report of the statement and branch coverage for every line of code.
- **Include Coverage Totals**
  Saves a text report of the coverage totals by files and by instances. Includes total hits and coverage percentages for all active statements and branches.
- **Disable Source Annotation**
  Removes source code from coverage reports.
- **Recursive**
  Reports on the specified instance, and all included instances, recursively.
- **Write XML format**
  Produces output in an XML-structured format. The following example is an abbreviated "By Instance" report that includes line details:

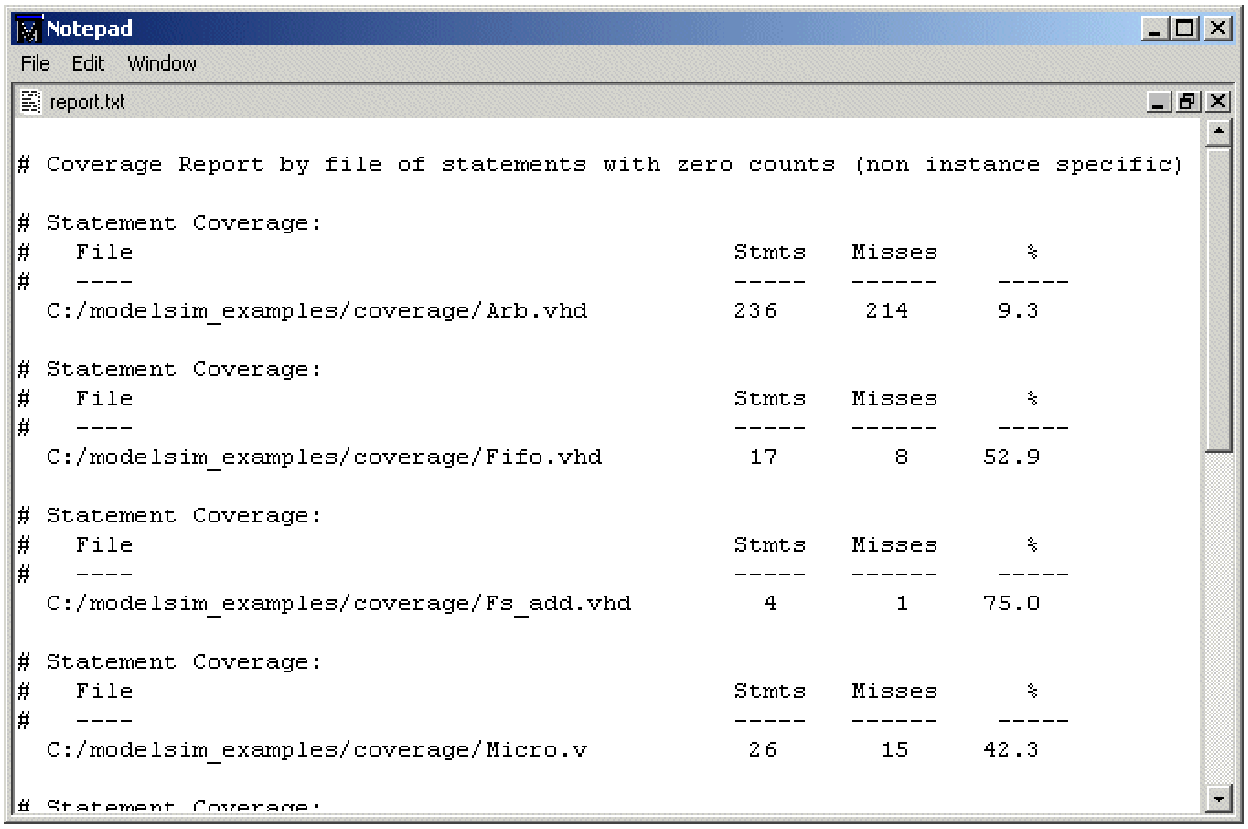
```
<?xml version="1.0"?>
<report xmlns="http://model.com/coverage"
lines="1"
byInstance="1">
<instance path="/test_delta/chip/control_126k_inst" du="mode_two_control">
<source_table files="1">
<file fn="0" path="C:/modelsim_examples/coverage/Modetwo.v"></file>
</source_table>
<statements active="30" hits="17" percent="56.7"> </statements>
<statement_data>
<stmt fn="0" ln="39" st="1" hits="82"> </stmt>
<stmt fn="0" ln="42" st="1" hits="82"> </stmt>
<stmt fn="0" ln="44" st="1" hits="82"> </stmt>
```

  "fn" stands for "filename", "ln" stands for "line number", and "st" stands for "statement.

Sample reports

Below are two abbreviated coverage reports with descriptions of select fields.

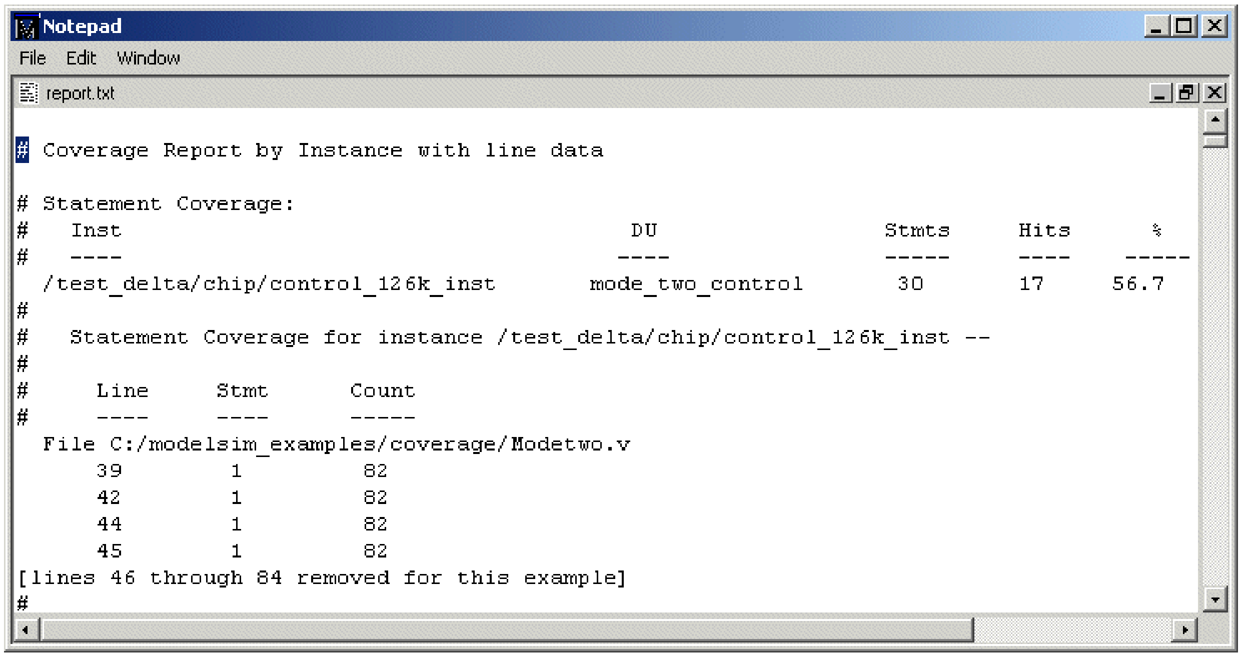
Zero counts report by file



```
# Coverage Report by file of statements with zero counts (non instance specific)

# Statement Coverage:
#   File                                          Stmts   Misses     %
#   ----                                          -----   ------   -----
  C:/modelsim_examples/coverage/Arb.vhd           236      214      9.3

# Statement Coverage:
#   File                                          Stmts   Misses     %
#   ----                                          -----   ------   -----
  C:/modelsim_examples/coverage/Fifo.vhd           17        8     52.9

# Statement Coverage:
#   File                                          Stmts   Misses     %
#   ----                                          -----   ------   -----
  C:/modelsim_examples/coverage/Fs_add.vhd          4        1     75.0

# Statement Coverage:
#   File                                          Stmts   Misses     %
#   ----                                          -----   ------   -----
  C:/modelsim_examples/coverage/Micro.v            26       15     42.3

# Statement Coverage:
```

The "%" field shows the percentage of statements in the file that had zero coverage.

Instance report with line details



```
# Coverage Report by Instance with line data

# Statement Coverage:
#   Inst                                   DU                   Stmts     Hits      %
#   ----                                  ----                  -----     ----    -----
  /test_delta/chip/control_126k_inst       mode_two_control       30        17      56.7
#
#   Statement Coverage for instance /test_delta/chip/control_126k_inst --
#
#     Line      Stmt       Count
#     ----      ----       -----
  File C:/modelsim_examples/coverage/Modetwo.v
      39          1          82
      42          1          82
      44          1          82
      45          1          82
[lines 46 through 84 removed for this example]
#
```

The "Stmt" field identifies the number of statements with zero coverage on that line.

Branch count report snippet

The following report snippet demonstrates two values that require explanation:

```
#    Branches with Zeros
#
#      Line        Stmt        True      False
#      ----        ----        ----      -----
         211          1         INF         0
         421          1         0           465,987,218
         665          1         -           0
```

Value	Meaning
INF	coverage value has exceeded ~4 billion (2^{32} -1)
-	the field is irrelevant to that particular line of code; for example, line 665 in the report above will never have an entry under the True column

Saving and reloading coverage data

Raw coverage data can be saved and then reloaded later. Saved data can also be merged with coverage statistics from the current simulation. You can perform these operations via the command line, the graphic interface, or the $coverage_save Verilog system task (see "ModelSim Verilog system tasks" (UM-149)).

From the command line

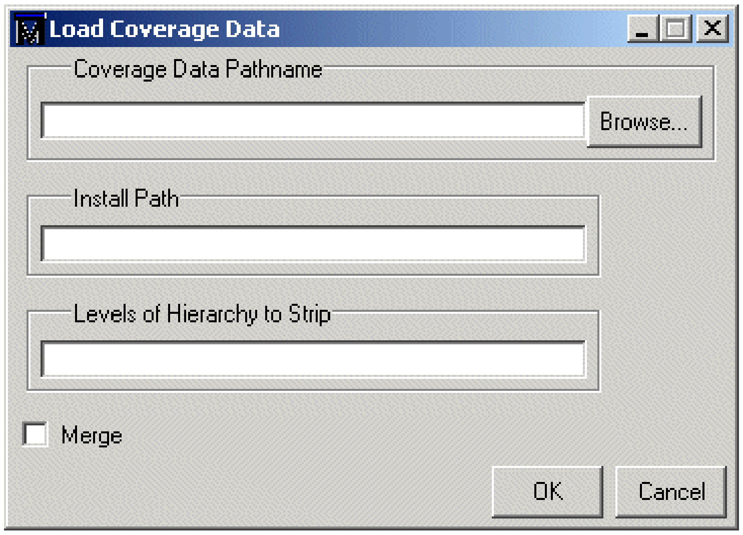
The **coverage save** command (CR-140) saves current coverage statistics to a file that can be reloaded later, preserving instance-specific information.

The **coverage reload** command (CR-136) seeds the coverage statistics of the current simulation with the output of a previous **coverage save** command. This allows you, for example, to gather statistics from multiple simulation runs into a single report.

From the graphic interface

To save raw coverage data, select **Tools** > **Coverage** > **Save** (Main window).

To reload previously saved coverage data, select **Tools** > **Coverage** > **Load**.



Loading a previously saved file clears all existing coverage data *unless* you check Merge. If you check **Merge** in this dialog, ModelSim merges the saved coverage data with coverage data in the current simulation.

Optionally, you can change the hierarchy of the file you are loading. Use the **Install Path** field to add hierarchy, and **Levels of Hierarchy to Strip** to delete hierarchy. This allows you to merge coverage results from simulations that have different hierarchies.

With the vcover utility

The merge utility, **vcover merge**, allows you to merge sets of coverage data without requiring elaboration of the design or a simulation license. It is a standard ModelSim utility that can be invoked from within the GUI or from the command line.

See the **vcover merge** command (CR-311) in the *ModelSim Command Reference* for further details.

Coverage statistics details

This section describes how condition and expression coverage statistics are calculated. In general, condition and expression coverage is limited to boolean and std_logic types. The coverage utility will analyze conditions and expressions of the form <integer variable> <op> <integer constant>. It will not, however, produce coverage results when, for example, two variables are being compared.

Condition coverage

Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. A truth table is constructed for the condition expression and counts are kept for each row of the truth table that occurs. For example, the following IF statement:

```
Line 180: IF (a or b) THEN x := 0; else x := 1; endif;
```

reflects this truth table.

Truth table for line 180

	counts	a	|b	||(a or b)
Row 1	5	1	-	1
Row 2	0	-	1	1
Row 3	8	0	0	0
unknown	0			

Row 1 indicates that (*a* or *b*) is true if *a* is true, no matter what *b* is. The "counts" column indicates that this combination has executed 5 times. The '-' character means "don't care." Likewise, row 2 indicates that the result is true if *b* is true no matter what *a* is, and this combination has executed zero times. Finally, row 3 indicates that the result is always zero when *a* is zero and *b* is zero, and that this combination has executed 8 times.

The truth table body only deals with boolean values. If any inputs are unknown, the result is set to unknown and is counted.

Values that are vectors are treated as subexpressions external to the table until they resolve to a boolean result. For example, take the IF statement:

```
Line 38:IF ((e = '1') AND (bus = "0111")) ...
```

A truth table will be generated in which bus = "0111" is evaluated as a subexpression and the result, which is boolean, becomes an input to the truth table. The truth table looks as follows:

Truth table for line 38

	counts	e	|(bus="0111")	||e=’1’) AND (bus = "0111")
Row 1	0	0	-	0
Row 2	10	-	0	0

Truth table for line 38

	counts	e	|(bus="0111")	||e='1') AND (bus = "0111")
Row 3	1	1	1	1
unknown	0	0		

Index expressions also serve as inputs to the table. Conditions containing function calls cannot be handled and will be ignored for condition coverage.

If a line contains a condition that is uncovered - some part of its truth table was not encountered - that line will appear in the Missed Coverage pane under the Conditions tab. When that line is selected, the condition truth table will appear in the Details pane and the line will be highlighted in the Source window.

Condition coverage truth tables are printed in coverage reports when the Condition Coverage type is selected in the Coverage Reports dialog (see "Reporting coverage data" (UM-446)) or when the **-lines** argument is specified in the **coverage report** command and one or more of the rows has a zero hit count.

Expression coverage

Expression coverage analyzes the expressions on the right hand side of assignment statements and counts when these expressions are executed. For expressions that involve boolean operators, a truth table is constructed and counts are tabulated for conditions matching rows in the truth table.

For example, take the statement:

```
Line 236: x <= a xor (not b(0));
```

This results are in the following truth table, with associated counts.

Truth table 236

	counts	a	|b(0)	|(a xor (not b(0)))	|||(not b(0))
Row 1	1	0	0	1	1
Row 2	0	0	1	0	0
Row 3	2	1	0	0	1
Row 4	0	1	1	1	0
unknown	0				

If a line contains an expression that is uncovered - some part of its truth table was not encountered - that line will appear in the Missed Coverage pane under the Expressions tab. When that line is selected, the expression truth table will appear in the Details pane and the line will be highlighted in the Source window.

As with condition coverage, expression coverage truth tables are printed in coverage reports when the Expression Coverage type is selected in the Coverage Reports dialog (see

"Reporting coverage data" (UM-446)) or when the **-lines** argument is specified in the **coverage report** command and one or more of the rows has a zero hit count.

Code Coverage preference variables

Various Tcl variables control how the coverage data is displayed. You can set these preference variables by selecting **Tools > Edit Preferences** in the Main window; then, in the Preferences dialog box select the **By Name** tab and expand the **Coverage** hierarchy. Select a property and click the **Change Value** button to change values. Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory.

13 - Waveform Compare

Chapter contents

Introduction

The ModelSim Waveform Compare feature allows you to compare the current live simulation against a reference dataset (.wlf file), compare two datasets, or compare different parts of the current live simulation. You can view the results of these comparisons in the Wave and List windows and generate a text file of the results in the Main window.

With the Waveform Compare feature you can:

- specify the signals or regions to be compared
- define tolerances for timing differences
- set a start time and end time for the comparison
- limit the comparison to a specific number of timing differences
- step through a succession of timing differences via buttons in the Wave window

All differences encountered in the comparison are summarized and listed in the transcript area of the Main window. Waveform differences are also displayed in the Wave and List windows (see "Wave window display" (UM-466) and "Compare objects in the List window" (UM-470)). You can also write a list of the differences to a file using the **compare info** command (CR-112).

Two modes of comparison

The Waveform Compare feature provides two modes of comparison: continuous and clocked.

Continuous Compare

In the continuous mode, a test signal (or a group of test signals within a region) is compared to a reference signal (or a group of reference signals within a region) at each transition of the reference. Timing differences between the test and reference signals are highlighted with rectangular red difference markers in the Wave window and yellow markers in the List window.

The continuous compare mode allows you to specify two edge tolerances for timing differences. The leading edge tolerance specifies how much earlier the test signal edge may occur before the reference signal edge. The trailing edge tolerance specifies how much later the test signal edge may occur after the reference signal edge. The default value for both tolerances is zero. In addition, these tolerances may be specified differently for each signal compared.

Clocked Compare

Clocked comparisons allow you to make a comparison only at or just after an edge on some signal. In this mode, you define one or more clocks. The test signal is compared to a reference signal and both are sampled relative to the defined clock. The clock can be defined as the rising or falling edge (or either edge) of a particular signal plus a user-specified delay. The design need not have any events occurring at the specified clock time.

Differences between the test signal(s) and clock are highlighted with red diamonds in the Wave window.

Comparing hierarchical and flattened designs

If you are comparing a hierarchical RTL design simulation against a flattened synthesized design simulation, you may have different hierarchies, different signal names, and the buses may be broken down into one-bit signals in the gate-level design. All of these differences can be handled by ModelSim's Waveform Compare feature.

- If the test design is hierarchical but the hierarchy is different from the hierarchy of the reference design, you can use the **compare add** command (CR-100) to specify which region path in the test design corresponds to that in the reference design.
- If the test design is flattened and test signal names are different from reference signal names, the **compare add** command (CR-100) allows you to specify which signal in the test design will be compared to which signal in the reference design.
- If, in addition, buses have been dismantled, or "bit-blasted", you can use the **-rebuild** option of the **compare add** command (CR-100) to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

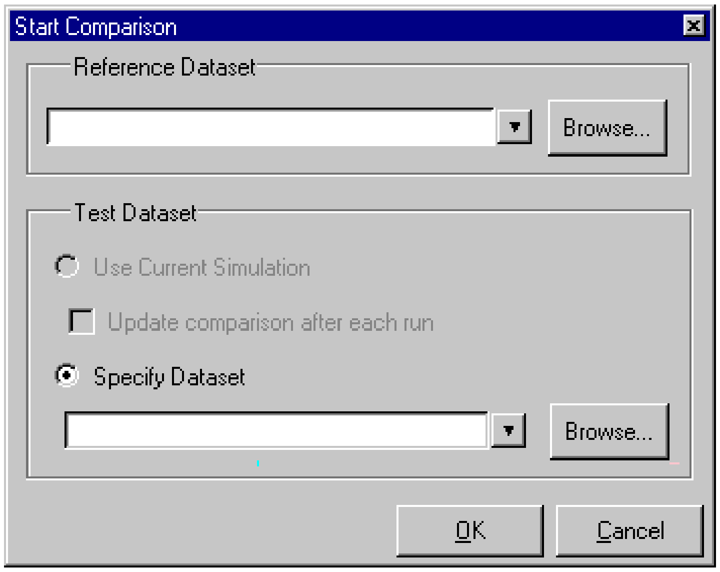
If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Compare feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Compare will let you know.

Graphic interface to Waveform Compare

Waveform Compare is initiated from either the Main or Wave window by selecting **Tools >Waveform Compare > Start Comparison**.

Opening dataset comparison

The Start Comparison dialog box allows you define the Reference and Test datasets.



Reference Dataset

The Reference Dataset is the .wlf file that the test dataset will be compared to. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

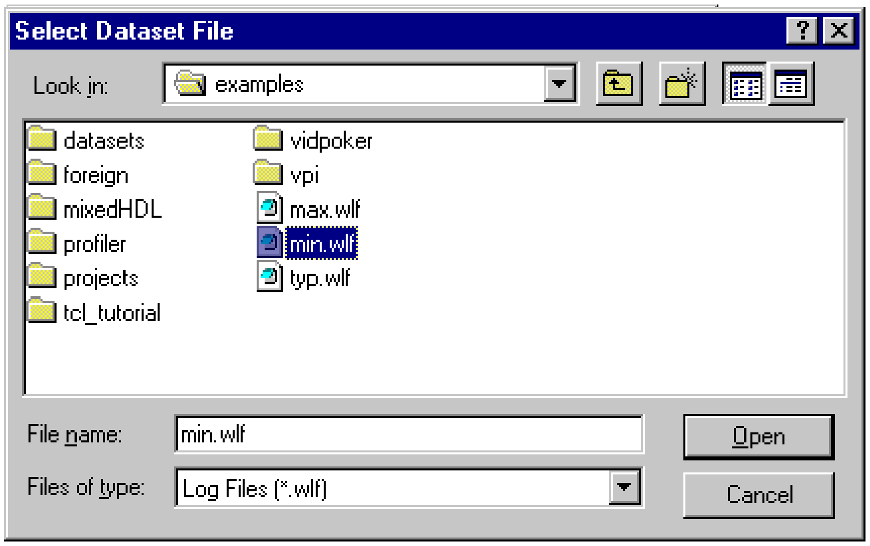
Test Dataset

The Test Dataset is the .wlf file that will be compared against the Reference Dataset. Like the Reference Dataset, it can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

- **Use Current Simulation**
  Selects the current simulation to be used as the Test Dataset. Provides for an optional update on the comparison after each simulation run.
- **Specify Dataset**
  Allows you to select any saved .wlf file to be used as the Test Dataset.

You can specify either dataset by typing in a dataset name, by selecting a dataset from a drop-down history of past dataset selections, or by clicking either of the Browse buttons.

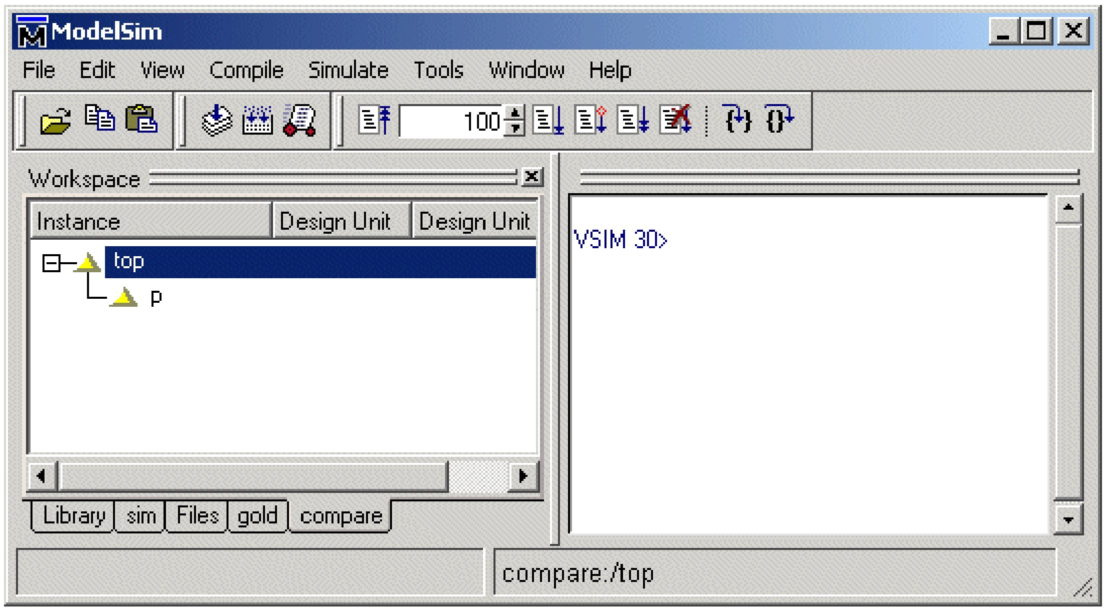
Both Browse buttons take you to the Select Dataset File dialog where you can browse for the dataset you want.



File name: min.wlf

Files of type: Log Files (*.wlf)

Once the Reference and Test Datasets have been specified, clicking "OK" in the Compare Dataset dialog box will place a Compare tab in the project pane of the Main window. After adding the signals, regions, and/or clocks you want to use in the comparison (see "Adding signals, regions, and clocks" (UM-461)) you'll be able to drag compare objects from this project tab into the Wave and List windows.



```
VSIM 30>
```

compare:/top

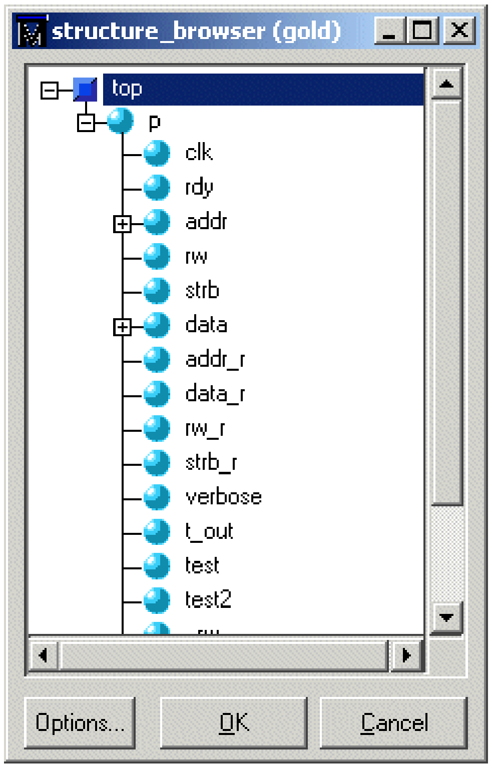
Adding signals, regions, and clocks

To designate the signals, regions and/or clocks to be used in the comparison, click **Tools > Waveform Compare > Add** in the Main or Wave window, then make a selection (Compare by Signal (UM-461), Compare by Region (UM-462), Clocks) from the popup menu.

Compare by signal

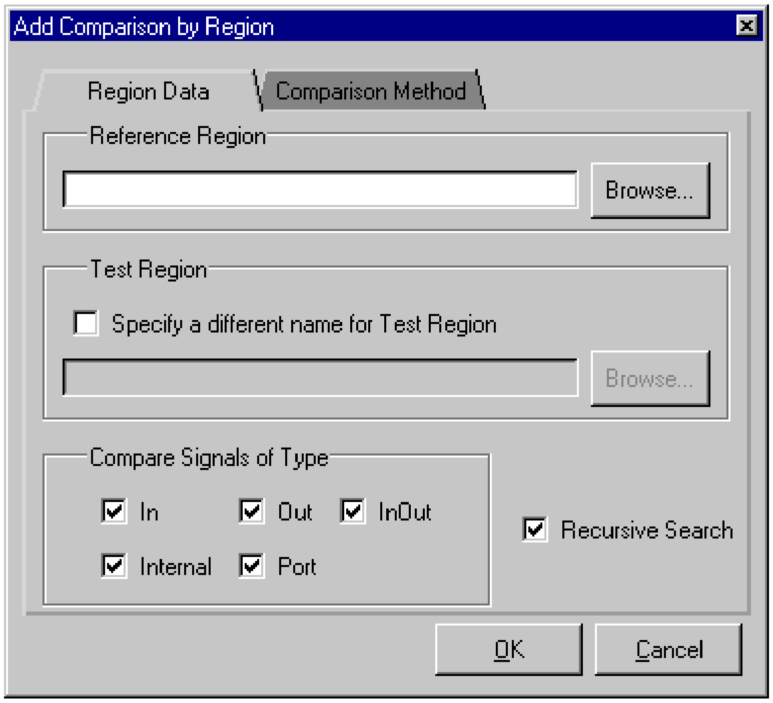
Clicking **Tools > Waveform Compare > Add > Compare by Signal** in the Wave window opens the structure_browser window, where you can specify signals to be used in the comparison.

You can also set signal options by clicking the Options button. See "Comparison Method tab" (UM-463) for details.



Compare by region

Clicking **Tools > Waveform Compare > Add > Compare by Region** in the Wave window opens the Add Comparison by Region window, where you can specify signals to be used in the comparison.

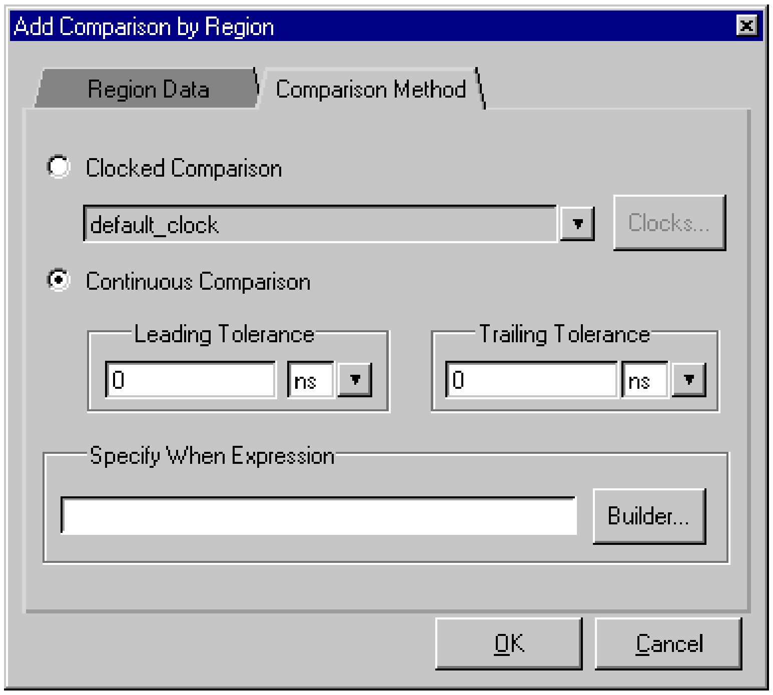


Region Data tab

- **Reference Region**
  Allows you to specify the reference region that will be used in the comparison.
- **Test Region**
  Allows you to specify a test region that might have a different name from that of the reference region.
- **Compare Signals of Type**
  Allows you to specify that All Types of signals will be used in the comparison or only Selected Types (In, Out, InOut, Internal, or Port).
- **Recursive Search**
  Specifies whether to search for signals in the hierarchy below the selected region.

Comparison Method tab

Allows you to select clocked or continuous comparison, and provides the capability to specify a "When" expression.

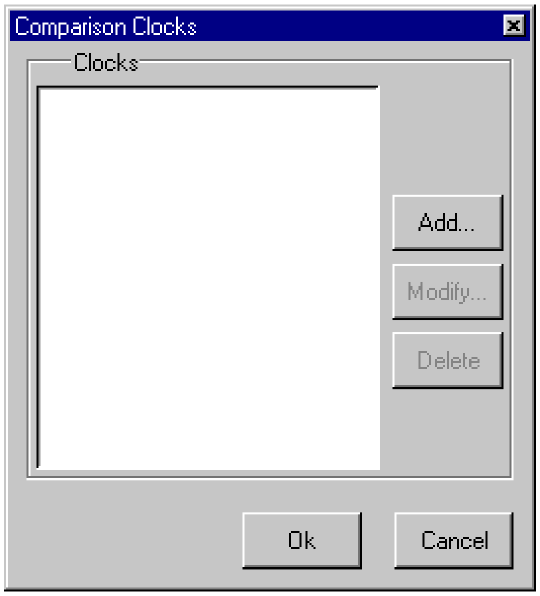


Clocked Comparison: default_clock
[✓ Continuous Comparison]
Leading Tolerance: 0 ns
Trailing Tolerance: 0 ns

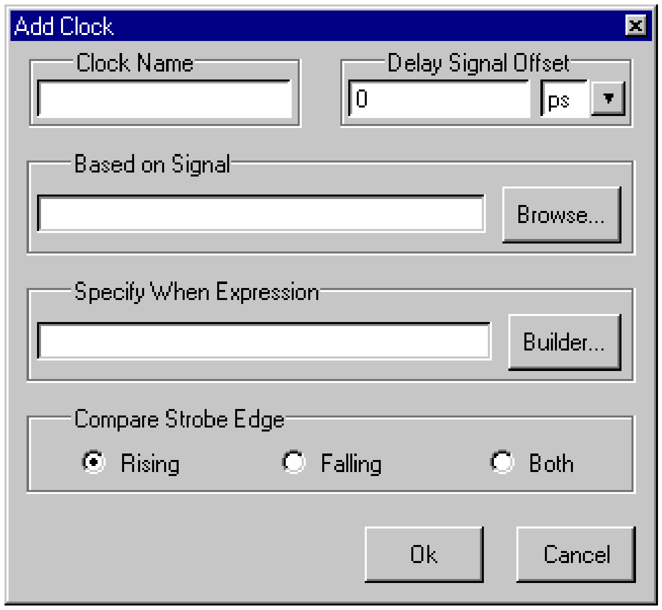
- **Clocked comparison**
  Allows you to select a clock from the drop-down history of past clock selections. Or, you can click the Clocks button to add a new clock.

  Clicking the Clocks button opens the Comparison Clocks dialog box.

  To add a signal, click the Add button to open the Add Clock dialog box, where you can define a clock signal name, a delay signal offset, the signal upon which the clock will be based, and whether the compare strobe edge will be the rising or falling edge or both. You can also use the Expression Builder to specify

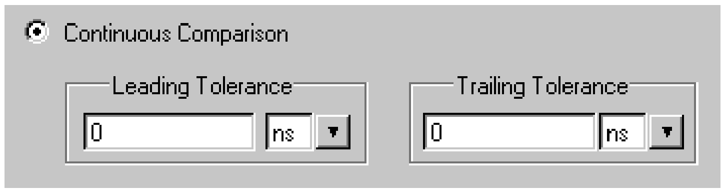


a when expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.

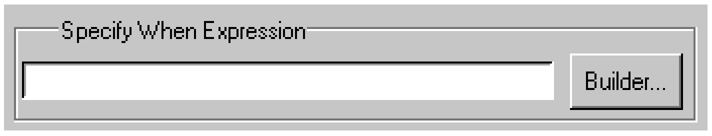


Compare Strobe Edge: [✓ Rising] Falling Both

- **Continuous comparison**
  With the Continuous Comparison method you can set leading and trailing edge tolerances. The leading edge tolerance specifies how much earlier the test signal edge may occur before the reference signal edge. The trailing edge tolerance specifies how much later the test signal edge may occur after the reference signal edge. The default value for both tolerances is zero. In addition, these tolerances may be specified differently for each signal compared.

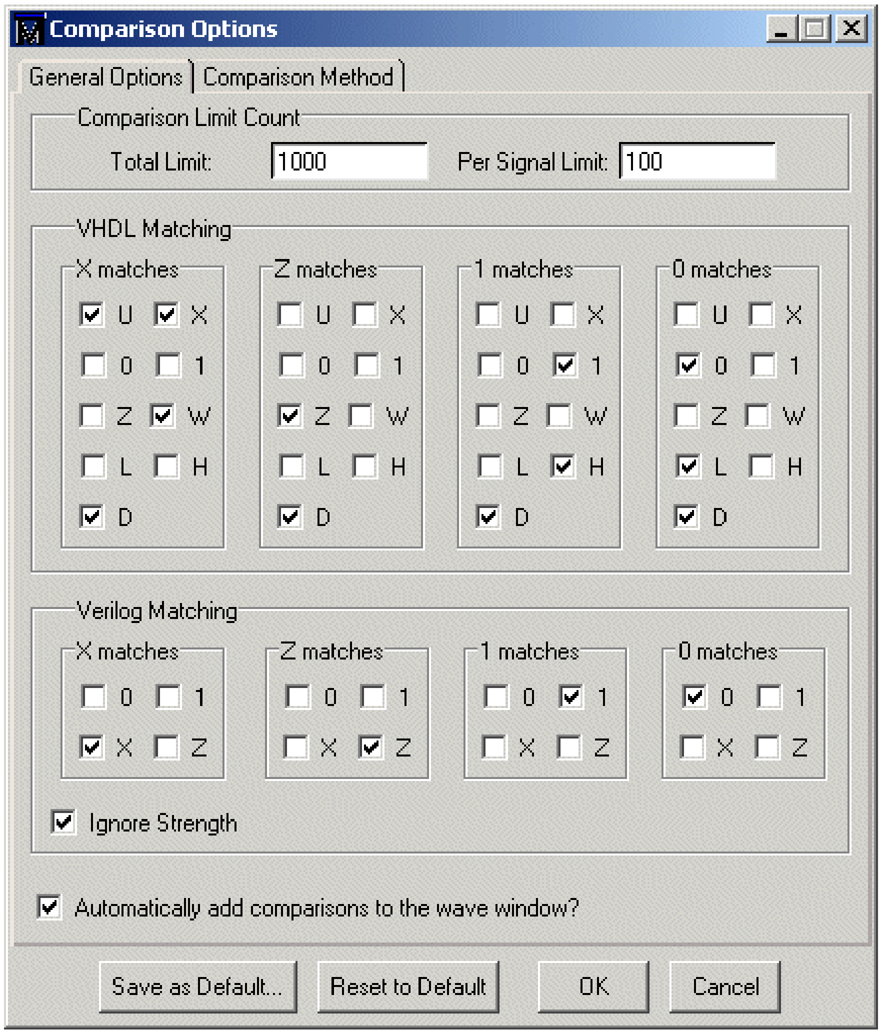


- **Specify When Expression**
  Allows you to use "The GUI Expression Builder" (UM-395) to specify a when expression that must evaluate to "true" or 1 at the signal edge for the comparison to become effective.



Setting compare options

Selecting **Tools** > **Waveform Compare** > **Options** in either the Main or Wave windows provides access to the **Comparison Options** dialog box. This dialog is divided into two tabs – the **General Options** tab and the **Comparison Method** tab (see "Comparison Method tab" (UM-463) for a description).



Comparison Options

General Options | Comparison Method

Comparison Limit Count

Total Limit: 1000   Per Signal Limit: 100

VHDL Matching

X matches: [✓] U [✓] X [ ] 0 [ ] 1 [ ] Z [✓] W [ ] L [ ] H [✓] D

Z matches: [ ] U [ ] X [ ] 0 [ ] 1 [✓] Z [ ] W [ ] L [ ] H [✓] D

1 matches: [ ] U [ ] X [ ] 0 [✓] 1 [ ] Z [ ] W [ ] L [✓] H [✓] D

0 matches: [ ] U [ ] X [✓] 0 [ ] 1 [ ] Z [ ] W [✓] L [ ] H [✓] D

Verilog Matching

X matches: [ ] 0 [ ] 1 [✓] X [ ] Z

Z matches: [ ] 0 [ ] 1 [ ] X [✓] Z

1 matches: [ ] 0 [✓] 1 [ ] X [ ] Z

0 matches: [✓] 0 [ ] 1 [ ] X [ ] Z

[✓] Ignore Strength

[✓] Automatically add comparisons to the wave window?

Save as Default... | Reset to Default | OK | Cancel

Comparison Limit Count — Allows you to limit the comparison to a specific number of total differences and/or a specific number of differences per signal.

VHDL Matching — Allows you to designate which VHDL signal values will match X, Z, 1, and 0 values.

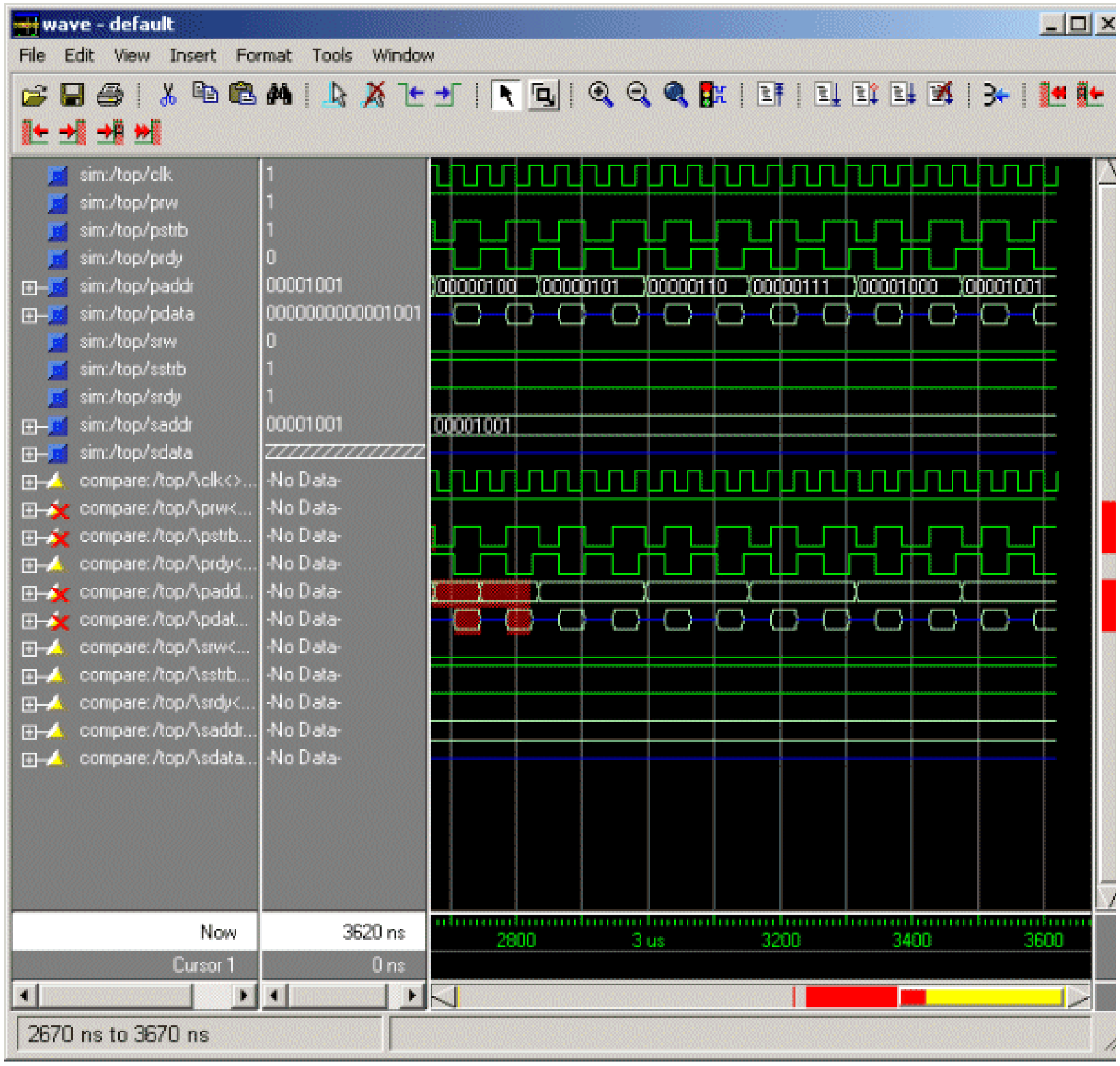
Verilog Matching — Allows you to designate which Verilog signal values will match X, Z, 1, and 0 values. It also allows you to ignore the strength of the Verilog signal and consider only logic values.

Automatically add comparisons to the wave window?— Specifies whether new signal comparison objects are added automatically to the Wave window.

Save as Default — Allows you to save all changes as the new default settings for subsequent comparisons.

Reset to Default — Resets all settings to original default values.

Wave window display



sim:/top/clk	1
sim:/top/prw	1
sim:/top/pstrb	1
sim:/top/prdy	0
sim:/top/paddr	00001001
sim:/top/pdata	0000000000001001
sim:/top/srw	0
sim:/top/sstrb	1
sim:/top/srdy	1
sim:/top/saddr	00001001
sim:/top/sdata	
compare:/top/\clk<>...	-No Data-
compare:/top/\prw<...	-No Data-
compare:/top/\pstrb...	-No Data-
compare:/top/\prdy<...	-No Data-
compare:/top/\padd...	-No Data-
compare:/top/\pdat...	-No Data-
compare:/top/\srw<...	-No Data-
compare:/top/\sstrb...	-No Data-
compare:/top/\srdy<...	-No Data-
compare:/top/\saddr...	-No Data-
compare:/top/\sdata...	-No Data-
Now	3620 ns
Cursor 1	0 ns

The Wave window provides a graphic display of comparison results. Pathnames of all test signals included in the comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.

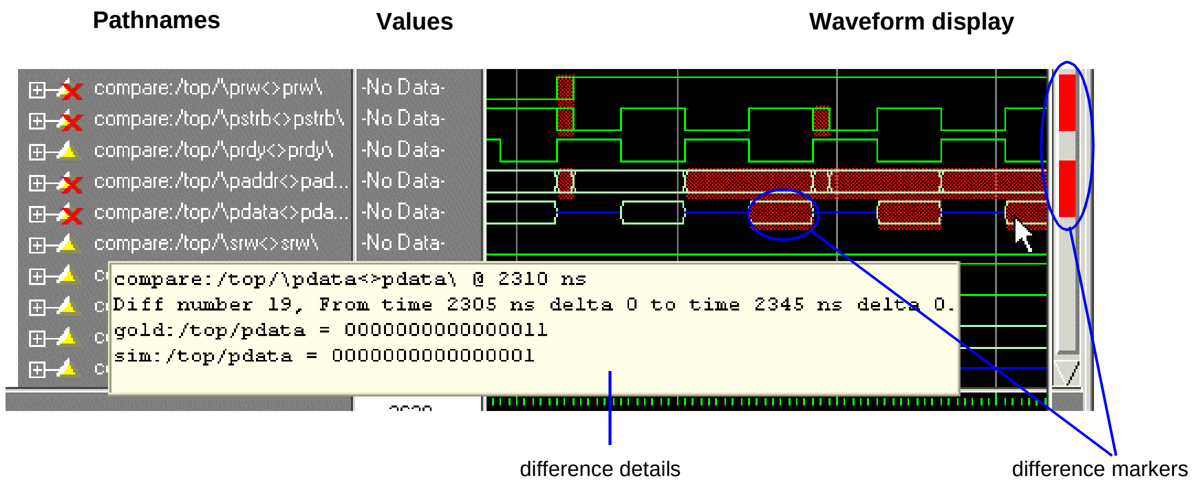
The names of the comparison items take the form

```
<path>/\refSignalName<>testSignalName\
```

If you compare two signals from different regions, the signal names include the uncommon part of the path.

In comparisons of signals with multiple bits, you may display them in "buswise" or "bitwise" format. Buswise format lists the busses under the compare item whereas bitwise format lists each individual bit under the compare item. To select one format or the other, click your right mouse button on the plus sign ('+') next to a compare item.

Timing differences are also indicated by red bars in the vertical and horizontal scroll bars of the waveform display, and by red difference markers on the waveforms themselves. Rectangular difference markers denote continuous differences. Diamond difference markers denote clocked differences. Placing your mouse cursor over any difference marker will initiate a popup display that provides timing details for that difference. You can toggle this popup on and off in the **Wave Window Properties** dialog (see "Setting Wave window display properties" (UM-352)).



compare:/top/\prw<>prw\ -No Data-
compare:/top/\pstrb<>pstrb\ -No Data-
compare:/top/\prdy<>prdy\ -No Data-
compare:/top/\paddr<>pad... -No Data-
compare:/top/\pdata<>pda... -No Data-
compare:/top/\srw<>srw\ -No Data-

```
compare:/top/\pdata<>pdata\ @ 2310 ns
Diff number 19, From time 2305 ns delta 0 to time 2345 ns delta 0.
gold:/top/pdata = 0000000000000011
sim:/top/pdata = 0000000000000001
```

The values column of the Wave window displays the words "match" or "diff" for every test signal, depending on the location of the selected cursor. "Match" indicates that the value of the test signal matches the value of the reference signal at the time of the selected cursor. "Diff" indicates a difference between the test and reference signal values at the selected cursor.

Annotating differences

You can tag differences with textual notes that are included in the difference details popup and comparison reports. Click a difference with the right mouse button, and select **Annotate Diff**. Or, use the **compare annotate** (CR-104) command.

Compare icons

The Wave window includes six comparison icons that let you quickly jump between differences. From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.

These buttons cycle through differences on all signals. To view differences for just the selected signal, press <tab> and <shift - tab> on your keyboard.

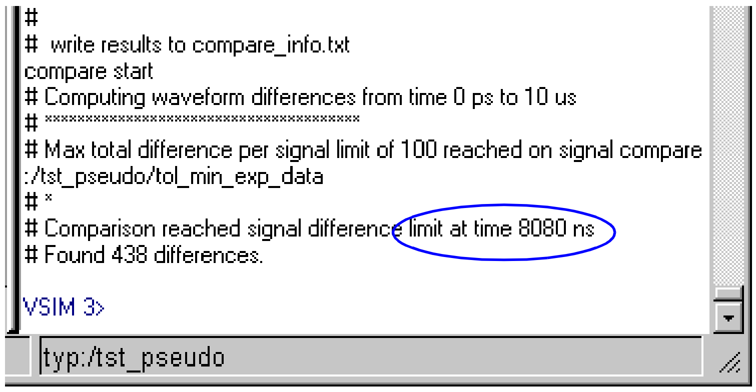
Note: If you have differences on individual bits of a bus, the compare icons will stop on those differences but <tab> and <shift - tab> will not.

A comparison is independent from any window in which you view it. As a result, if you have two Wave windows displayed, each containing different comparison objects, the compare icons will cycle through the differences displayed in both windows.

Waveform Compare menu

The **Compare** menu provides a number of options for controlling waveform comparisons.

- **Start Comparison**
  Opens the **Compare Dataset** dialog box where you can enter reference and test dataset names.
- **Comparison Wizard**
  Gives step-by-step assistance while you create a waveform comparison.
- **Run Comparison**
  Computes the number of differences from time zero to the end of the simulation run, from time zero until the maximum total number of differences per signal limit is reached, or from time zero until the maximum total number of differences for all signals compared is reached. This information is posted to the Main window transcript. It is equivalent to the **compare run** (CR-120) command:



```
#
# write results to compare_info.txt
compare start
# Computing waveform differences from time 0 ps to 10 us
# ****************************************************
# Max total difference per signal limit of 100 reached on signal compare
:/tst_pseudo/tol_min_exp_data
# *
# Comparison reached signal difference limit at time 8080 ns
# Found 438 differences.

VSIM 3>
```

- **End Comparison**
  Stops difference computation and closes the currently open comparison.

- **Add**

  Compare by Signal — Opens the **structure_browser** dialog box and allows you to designate signals for comparison.

  Compare by Region — Opens the Add Comparison by Region dialog box and allows you to designate a reference region for comparison. Also allows you to designate a test region of a different name.

  Clocks — Opens the Comparison Clocks dialog box and allows you to define clocks to be used in the comparison.

- **Options**
  Opens the Comparison Options dialog box, which allows you to define a number of waveform comparison options.

- **Differences**

  Clear — Clears all differences from the Wave window and resets the waveform comparison function. It is equivalent to the **compare reset** command (CR-119).

  Show — Displays the differences in text format in the transcript area of the Main window. It is equivalent to the **compare info** command (CR-112).

  Save — Opens the Specify Differences File dialog box where you can save the differences to a file that can be reloaded later in ModelSim. The default file name is "compare.dif".

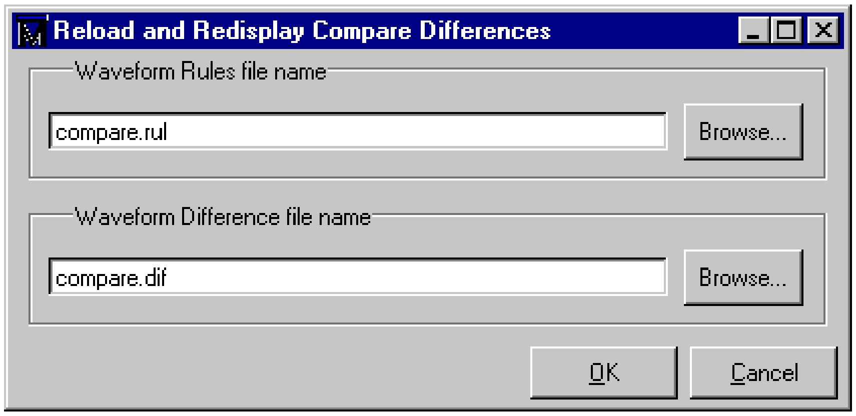
  Write Report— Saves a report of the differences to a text file that you can view.

- **Rules**

  Show — Displays the rules or instructions used to set up the waveform compare. It is equivalent to the **compare list** command (CR-113).

  Save — Opens the Specify Rule File dialog box and allows you to assign a name to the file that will contain all rules for making the comparison. The default file name is "compare.rul."

- **Reload**
  Opens the Reload and Redisplay Compare Differences dialog box and allows you to enter or browse for waveform rules and difference file names.


Reload and Redisplay Compare Differences
Waveform Rules file name
Browse...
Waveform Difference file name
Browse...
OK
Cancel

Waveform Rules file name: compare.rul

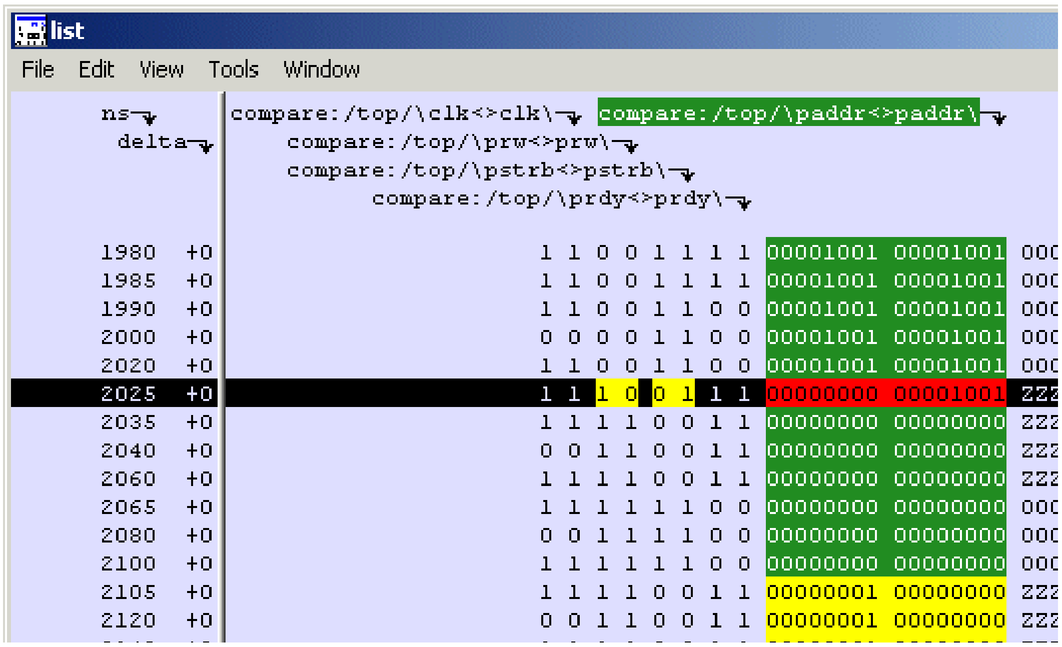
Waveform Difference file name: compare.dif

Printing compare differences

You can print the compare differences shown in the Wave window either to a printer or to a Postscript file. See "Printing and saving waveforms" (UM-363) for details.

Compare objects in the List window

Compare objects can be displayed in the List window too. Differences are highlighted with a yellow background. Tabbing on selected columns moves the selection to the next difference (actually difference edge). Shift-tabbing moves the selection backwards.



```
      ns     compare:/top/\clk<>clk\    compare:/top/\paddr<>paddr\
       delta     compare:/top/\prw<>prw\
                 compare:/top/\pstrb<>pstrb\
                       compare:/top/\prdy<>prdy\

      1980  +0                1 1 0 0 1 1 1 1 00001001 00001001 000
      1985  +0                1 1 0 0 1 1 1 1 00001001 00001001 000
      1990  +0                1 1 0 0 1 1 0 0 00001001 00001001 000
      2000  +0                0 0 0 0 1 1 0 0 00001001 00001001 000
      2020  +0                1 1 0 0 1 1 0 0 00001001 00001001 000
      2025  +0                1 1 1 0 0 1 1 1 00000000 00001001 ZZZ
      2035  +0                1 1 1 1 0 0 1 1 00000000 00000000 ZZZ
      2040  +0                0 0 1 1 0 0 1 1 00000000 00000000 ZZZ
      2060  +0                1 1 1 1 0 0 1 1 00000000 00000000 ZZZ
      2065  +0                1 1 1 1 1 1 0 0 00000000 00000000 000
      2080  +0                0 0 1 1 1 1 0 0 00000000 00000000 000
      2100  +0                1 1 1 1 1 1 0 0 00000000 00000000 000
      2105  +0                1 1 1 1 0 0 1 1 00000001 00000000 ZZZ
      2120  +0                0 0 1 1 0 0 1 1 00000001 00000000 ZZZ
```

Right-clicking on a yellow-highlighted difference gives you three options: **Diff Info**, **Annotate Diff**, and **Ignore/Noignore** diff. With these options you can elect to display difference information, you can ignore selected differences or turn off ignore, and you can annotate individual differences.

Waveform Compare commands

The table below provides a brief description of the compare commands. See the *ModelSim Command Reference* for complete command details.

Command	Description
compare add (CR-100)	defines a comparison between the signals in a specified reference design and the signals in a specified test design
compare annotate (CR-104)	annotates a difference with a textual note
compare clock (CR-105)	defines a clock for clocked comparison; or, if **-delete** is specified, deletes a previously-defined clock
compare configure (CR-107)	modifies options for compare signals or regions
compare continue (CR-109)	continues difference computation that had been suspended
compare delete (CR-110)	deletes a signal or region from the current open comparison
compare end (CR-111)	quits the comparison
compare info (CR-112)	writes out results of the comparison; writes to the transcript unless the **-write** option is specified
compare list (CR-113)	shows all the **compare add** commands currently in effect
compare options (CR-114)	sets values for various compare options on the Tcl parser side; when subsequent commands are called, these values become the defaults
compare reload (CR-118)	reloads comparison differences to allow viewing without recomputation
compare reset (CR-119)	clears the current compare differences, allowing another **compare start** to be executed
compare run (CR-120)	runs the difference computation on the signals selected for comparison; reports the total number of errors found
compare savediffs (CR-121)	saves the comparison result differences in a form that can be reloaded later
compare saverules (CR-122)	saves the comparison setup information (or "rules") to a file that can be re-executed later as a command file; saves compare options and all clock definitions and region and signal selections
compare see command (CR-123)	causes the specified compare difference to be made visible in the specified wave window, using whatever horizontal and vertical scrolling is necessary
compare start command (CR-125)	initializes internal data structures for waveform compare
compare stop command (CR-127)	used internally by the **compare stop** button to suspend comparison computations in progress
compare update command (CR-128)	used internally to update the comparison differences when comparing a live simulation against a .wlf file

Waveform Compare preference variables

Various Tcl variables control how the compare data is displayed. You can set these preference variables by selecting **Tools > Edit Preferences > By Name > Compare** (Main window). Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory.

14 - C Debug

Chapter contents

C Debug allows you to interactively debug FLI/PLI/VPI/SystemC C/C++ source code with the open-source gdb debugger. Even though C Debug doesn’t provide access to all gdb features, you may wish to read gdb documentation for additional information.

Please be aware of the following caveats before using C Debug:

- C Debug is an interface to the open-source gdb debugger. We have not customized gdb source code, and C Debug doesn’t remove any of the limitations or bugs of gdb.
- We assume that you are competent with C or C++ coding and C debugging in general.
- Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.
- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- Generally you should not have an existing *.gdbinit* file. If you do, make certain you haven’t done any of the following: defined your own commands or renamed existing commands; used 'set annotate...', 'set height...', 'set width...', or 'set print...'; set breakpoints or watchpoints.

Supported platforms and gdb versions

ModelSim ships with the gdb 6.0 debugger. Testing has shown this version to be the most reliable for SystemC applications. However, for FLI/PLI applications, you can also use a current installation of gdb if you prefer. C Debug has been tested on the these platforms with these versions of gdb:

Platform	**Required gdb version**
32-bit Solaris 2.6, 7, 8, 9	gdb-5.0-sol-2.6
32- and 64-bit HP-UX 11.0[a], 11.11	wdb version 3.3 or later
64-bit HP-UX B.11.22 on Itanium 2	wdb version 4.2
32-bit AIX 4.2, 4.3	gdb-5.1-aix-4.2
32-bit Redhat Linux 7.2 or later	*/usr/bin/gdb* 5.2 or later

a.You must install kernel patch PHKL_22568 (or a later patch that supercedes PHKL_22568) on HP-UX 11.0. If you do not, you will see the following error message when trying to enable C Debug:

```
# Unable to find dynamic library list.
# error from C debugger
```

To invoke C Debug, you must have the following:

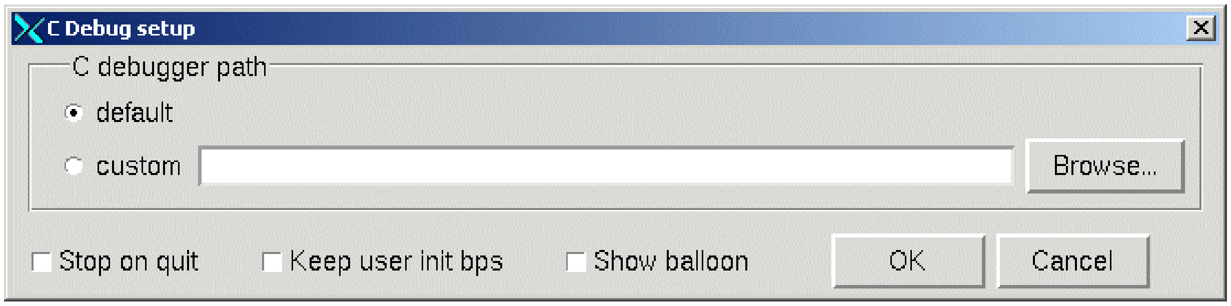
- A *cdebug* license feature; contact Model Technology sales for more information.
- The correct gdb debugger version for your platform.

Setting up C Debug

Before viewing your SystemC/C/C++ source code, you must set up the C Debug path and options. To set up C Debug, follow these steps:

1 Compile and link your C code with the **-g** switch (to create debug symbols) and without **-O** (or any other optimization switches you normally use). See *Chapter 7 - SystemC simulation* for information on compiling and linking SystemC code. See the *FLI Reference Manual* or *Chapter 6 - Verilog PLI / VPI* for information on compiling and linking C code.

2 Specify the path to the gdb debugger by selecting **Tools** > **C Debug** > **C Debug Setup**.



[✓ default] custom

Stop on quit
Keep user init bps
Show balloon

Select "default" to point at the Model Technology supplied version of gdb or "custom" to point at a separate installation.

3 Start the debugger by selecting **Tools** > **C Debug** > **Start C Debug**. ModelSim will start the debugger automatically if you set a breakpoint in a SystemC file.

4 If you are not using **gcc**, or otherwise haven't specified a source directory, specify a source directory for your C code with the following command:

```
ModelSim> gdb dir <srcdirpath1>[:<srcdirpath2>[...]]
```

Setting breakpoints

Breakpoints in C Debug work much like normal HDL breakpoints. You can set/delete and enable/disable them with ModelSim commands (**bp** (CR-81), **bd** (CR-76), **enablebp** (CR-163), **disablebp** (CR-153)) or via the Source window in the ModelSim GUI (see "Setting file-line breakpoints from the GUI" (UM-391)). Some differences do exist:

- The Breakpoints dialog in the ModelSim GUI doesn't list C breakpoints.
- C breakpoint id numbers require a "c." prefix when referenced in a command.
- When using the **bp** command (CR-81) to set a breakpoint in a C file, you must use the **-c** argument.

Here are some example commands:

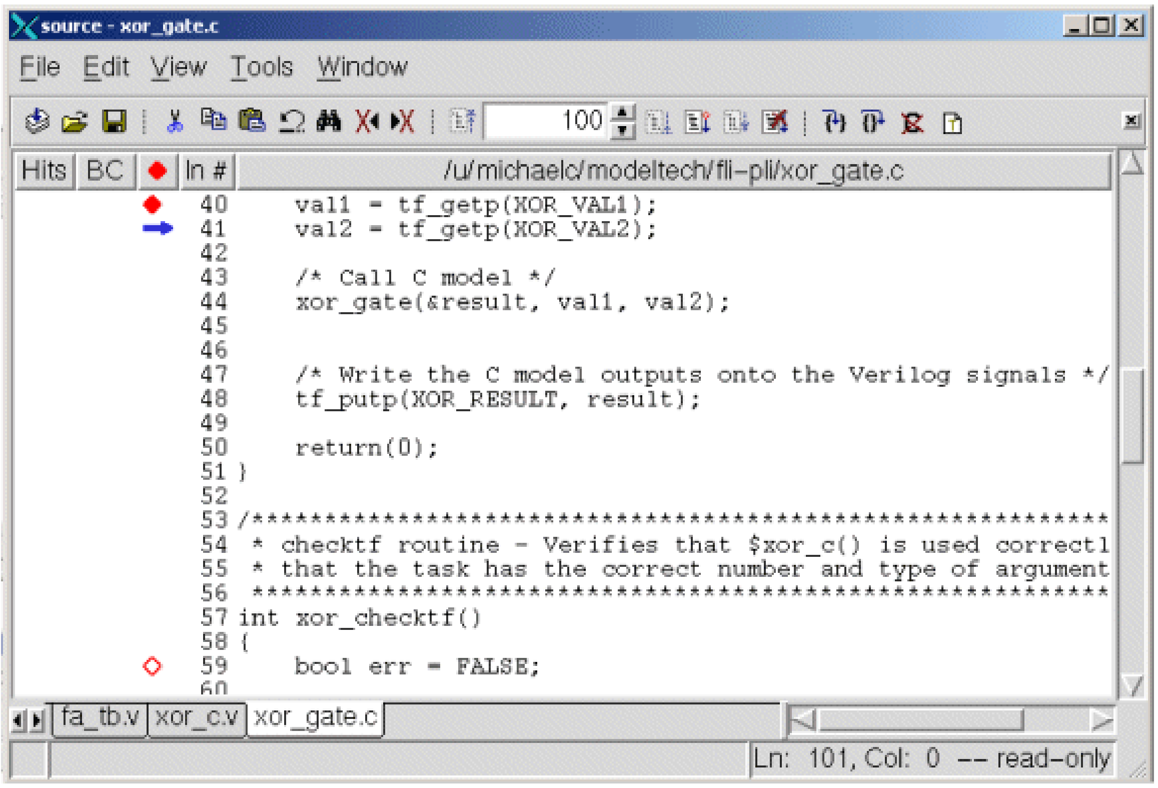
```
bp -c *0x400188d4
```
Sets a C breakpoint at the hex address 400188d4. Note the '*' prefix for the hex address.

```
bp -c or_checktf
```
Sets a C breakpoint at the entry to function **or_checktf**.

```
bp -c or.c 91
```
Sets a C breakpoint at line 91 of *or.c*.

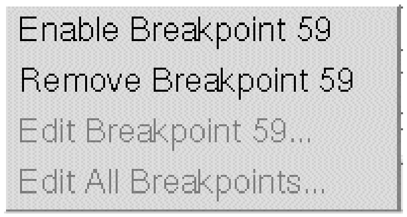
```
enablebp c.1
```
Enables C breakpoint number 1.

The graphic below shows a C file with one enabled breakpoint (on line 40) and one disabled breakpoint (on line 59).



```
40      val1 = tf_getp(XOR_VAL1);
41      val2 = tf_getp(XOR_VAL2);
42
43      /* Call C model */
44      xor_gate(&result, val1, val2);
45
46
47      /* Write the C model outputs onto the Verilog signals */
48      tf_putp(XOR_RESULT, result);
49
50      return(0);
51 }
52
53 /*************************************************************
54  * checktf routine - Verifies that $xor_c() is used correctl
55  * that the task has the correct number and type of argument
56  *************************************************************
57 int xor_checktf()
58 {
59      bool err = FALSE;
60
```

Clicking the red diamonds with your right (third) mouse button pops up a menu with commands for removing or enabling/disabling the breakpoints



Note: The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Do not set breakpoints in constructors of SystemC objects; it may crash the debugger.

Stepping in C Debug

Stepping in C Debug works much like you would expect. You use the same buttons and commands that you use when working with an HDL-only design.

Button	Menu equivalent	Other equivalents
Step steps the current simulation to the next statement; if the next statement is a call to a C function that was compiled with debug info, ModelSim will step into the function	Tools > C Debug > Run > Step	use the **step** command at the CDBG> prompt see: **step** (CR-264) command
Step Over statements are executed but treated as simple statements instead of entered and traced line-by-line; C functions are not stepped into unless you have an enabled breakpoint in the C file	Tools > C Debug > Run > Step -Over	use the **step -over** command at the CDBG> prompt see: **step** (CR-264) command
Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Tools > C Debug > Run > Continue	use the **run -continue** command at the CDBG> prompt see: **run** (CR-246)

Known problems with stepping in C Debug

The following are known limitations which relate to problems with gdb:

- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- With some platform and compiler versions, step may actually behave like run -continue when in a C file. This is a gdb quirk that results from not having any debugging information when in an internal function to VSIM (i.e., any FLI or VPI function). In these situations, use step -over to move line-by-line.

Finding function entry points with Auto find bp

ModelSim can automatically locate and set breakpoints at all currently known function entry points (i.e., PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Select **Tools > C Debug > Auto find bp** to invoke this feature.

The **Auto find bp** command provides a "snapshot" of your design when you invoke the command. If additional callbacks get registered later in the simulation, ModelSim will not identify these new function entry points *unless* you re-execute the **Auto find bp** command. If you want functions to be identified regardless of when they are registered, use "Identifying all registered function calls" (UM-480) instead.

The **Auto find bp** command sets breakpoints in an enabled state and doesn't toggle that state to account for step-over or run-continue commands. This may result in unexpected behavior. For example, say you have invoked the Auto find bp command and you are currently stopped on a line of code that calls a C function. If you execute a step -over or run -continue command, ModelSim will stop on the breakpoint set in the called C file.

Identifying all registered function calls

Auto step mode automatically identifies and sets breakpoints at registered function calls (i.e., PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Auto step mode is helpful when you are not entirely familiar with a design and its associated C routines. As you step through the design, ModelSim steps into and displays the associated C file when you hit a C function call in your HDL code. If you execute a **step -over** or **run -continue** command, ModelSim does not step into the C code.

When you first enable Auto step mode, ModelSim scans your design and sets enabled breakpoints at all currently known function entry points. As you step through the simulation, Auto step continues looking for newly registered callbacks and sets enabled breakpoints at any new entry points it identifies. Once you execute a step -over or run -continue command, Auto step disables the breakpoints it set, and the simulation continues running. The next time you execute a step command, the automatic breakpoints are re-enabled and Auto step sets breakpoints on any new entry points it identifies.

Note that Auto step does not disable user-set breakpoints.

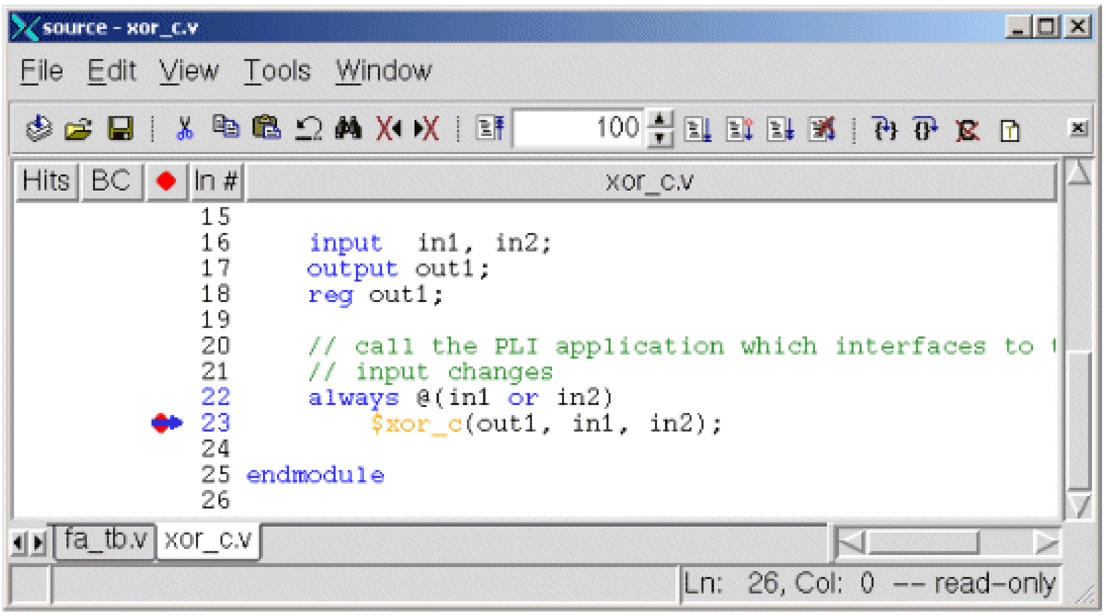
Enabling Auto step mode

To enable Auto step mode, follow these steps:

1. Configure C Debug as described in "Setting up C Debug" (UM-475).
2. Select **Tools** > **C Debug** > **Enable auto step** (Main window).
3. Load and run your design.

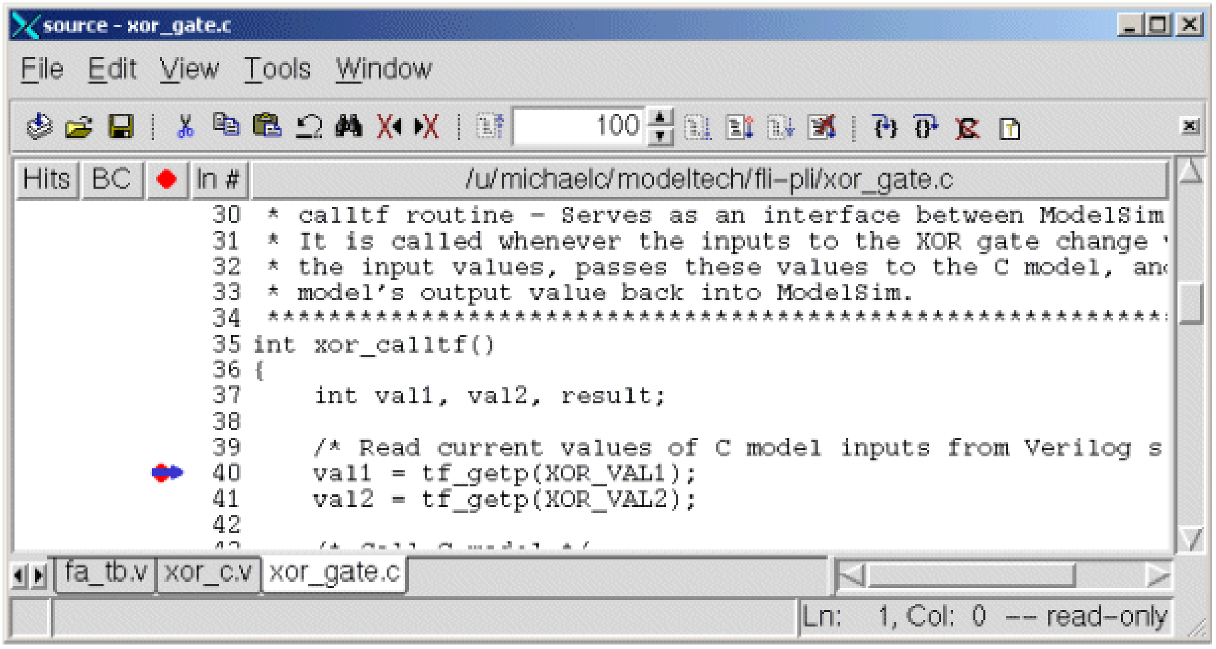
Example

The graphic below shows a simulation that has stopped at a user-set breakpoint on a PLI system task.



```
    input  in1, in2;
    output out1;
    reg out1;

    // call the PLI application which interfaces to t
    // input changes
    always @(in1 or in2)
        $xor_c(out1, in1, in2);

endmodule
```

Because Auto step mode is enabled, ModelSim automatically sets a breakpoint in the underlying *xor_gate.c* file. If you click the step button at this point, ModelSim will step into that file.


source - xor_gate.c
File Edit View Tools Window
100
Hits BC ln # /u/michaelc/modeltech/fli-pli/xor_gate.c
fa_tb.v xor_c.v xor_gate.c
Ln: 1, Col: 0 -- read-only

```
 * calltf routine - Serves as an interface between ModelSim
 * It is called whenever the inputs to the XOR gate change '
 * the input values, passes these values to the C model, an
 * model's output value back into ModelSim.
 *************************************************************
int xor_calltf()
{
    int val1, val2, result;

    /* Read current values of C model inputs from Verilog s
    val1 = tf_getp(XOR_VAL1);
    val2 = tf_getp(XOR_VAL2);
```

Auto find bp versus Auto step mode

As noted in "Finding function entry points with Auto find bp" (UM-479), the **Auto find bp** command also locates and sets breakpoints at function entry points. Note the following differences between Auto find bp and Auto step mode:

- Auto find bp provides a "snapshot" of currently known function entry points at the time you invoke the command. Auto step mode continues to locate and set automatic breakpoints in newly registered function calls as the simulation continues. In other words, Auto find bp is static while Auto step mode is dynamic.
- Auto find bp sets automatic breakpoints in an enabled state and doesn't change that state to account for step-over or run-continue commands. Auto step mode enables and disables automatic breakpoints depending on how you step through the design. In cases where you invoke both features, Auto step mode takes precedence over Auto find bp. In other words, even if Auto find bp has set enabled breakpoints, if you then invoke Auto step mode, it will toggle those breakpoints to account for step-over and run-continue commands.

Debugging functions during elaboration

Initialization mode allows you to examine and debug functions that are called during elaboration (i.e., while your design is in the process of loading). When you select this mode, ModelSim sets special breakpoints for foreign architectures and PLI/VPI modules that allow you to set breakpoints in the initialization functions. When the design finishes loading, the special breakpoints are automatically deleted, and any breakpoints that you set are disabled (unless you specify **Keep user init bps** in the C debug setup dialog).

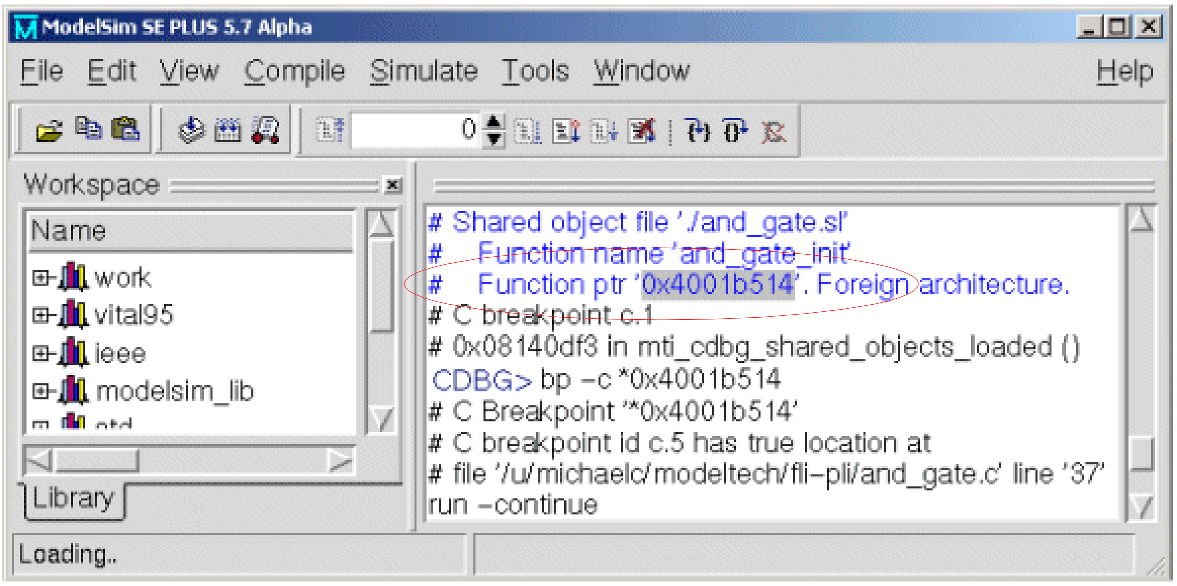
To run C Debug in initialization mode, follow these steps:

1 Start C Debug by selecting **Tools > C Debug > Start C Debug** *before* loading your design.

2 Select **Tools > C Debug > Init mode**.

3 Load your design.

As the design loads, ModelSim prints to the Transcript the names and/or hex addresses of called functions. For example the Transcript below shows a function pointer to a foreign architecture:



```
# Shared object file './and_gate.sl'
#    Function name 'and_gate_init'
#    Function ptr '0x4001b514'. Foreign architecture.
# C breakpoint c.1
# 0x08140df3 in mti_cdbg_shared_objects_loaded ()
CDBG> bp -c *0x4001b514
# C Breakpoint '*0x4001b514'
# C breakpoint id c.5 has true location at
# file '/u/michaelc/modeltech/fli-pli/and_gate.c' line '37'
run -continue
```

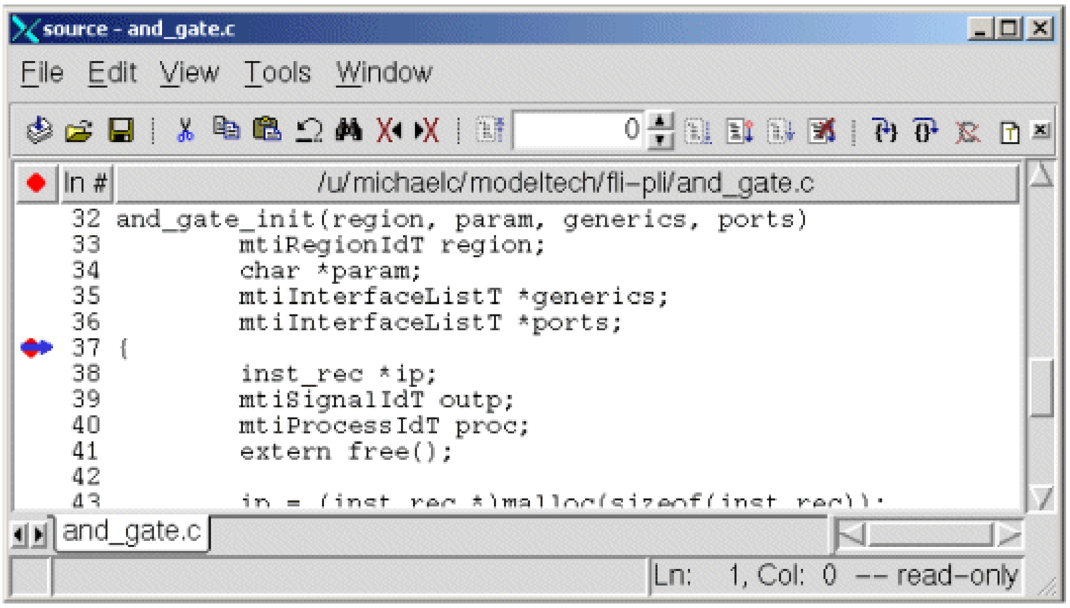
To set a breakpoint on that function, you would type:

```
bp -c *0x4001b514
```

or

```
bp -c and_gate_init
```

ModelSim in turn reports that it has set a breakpoint at line 37 of the *and_gate.c* file. As you continue through the design load using **run -continue**, ModelSim hits that breakpoint and displays the file and associated line in the Source window.


source - and_gate.c
File Edit View Tools Window
ln #
/u/michaelc/modeltech/fli-pli/and_gate.c
and_gate.c
Ln: 1, Col: 0 -- read-only

```
32 and_gate_init(region, param, generics, ports)
33         mtiRegionIdT region;
34         char *param;
35         mtiInterfaceListT *generics;
36         mtiInterfaceListT *ports;
37 {
38         inst_rec *ip;
39         mtiSignalIdT outp;
40         mtiProcessIdT proc;
41         extern free();
42
43         ip = (inst_rec *)malloc(sizeof(inst_rec));
```

FLI functions in initialization mode

There are two kinds of FLI functions that you may encounter in initialization mode. The first is a foreign architecture which was shown above. The second is a foreign function. ModelSim produces a Transcript message like the following when it encounters a foreign function during initialization:

```
# Shared object file './all.sl'
#    Function name 'in_params'
#    Function ptr '0x4001a950'. Foreign function.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., bp -c in_params) or the function pointer (i.e., bp -c *0x4001a950). Note, however, that foreign functions aren't called during initialization. You would hit the breakpoint only during runtime and then only if you enabled the breakpoint after initialization was complete or had specified **Keep user init bps** in the C debug setup dialog.

PLI functions in initialization mode

There are two methods for registering callback functions in the PLI: 1) using a veriusertfs array to define all usertf entries; and 2) adding an init_usertfs function to explicitly register each usertfs entry (see "Registering PLI applications" (UM-155) for more details). The messages ModelSim produces in initialization mode vary depending on which method you use.

ModelSim produces a Transcript message like the following when it encounters a veriusertfs array during initialization:

```
# vsim -pli ./veriuser.sl mux_tb
# Loading ./veriuser.sl
# Shared object file './veriuser.sl'
#    veriusertfs array - registering calltf
#    Function ptr '0x40019518'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#    veriusertfs array - registering checktf
#    Function ptr '0x40019570'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#    veriusertfs array - registering sizetf
#    Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#    veriusertfs array - registering misctf
#    Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set breakpoints on non-null callbacks using the function pointer (e.g., bp -c *0x40019570). You cannot set breakpoints on null functions. The sizetf and misctf entries in the example above are null (the function pointer is '0x0').

ModelSim reports the entries in multiples of four with at least one entry each for calltf, checktf, sizetf, and misctf. Checktf and sizetf functions are called during initialization but calltf and misctf are not called until runtime.

The second registration method uses init_usertfs functions for each usertfs entry. ModelSim produces a Transcript message like the following when it encounters an init_usertfs function during initialization:

```
# Shared object file './veriuser.sl'
#    Function name 'init_usertfs'
#    Function ptr '0x40019bec'. Before first call of init_usertfs.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., bp -c init_usertfs) or the function pointer (i.e., bp -c *0x40019bec). ModelSim will hit this breakpoint as you continue through initialization.

VPI functions in initialization mode

VPI functions are registered via routines placed in a table named vlog_startup_routines (see "Registering VPI applications" (UM-157) for more details). ModelSim produces a Transcript message like the following when it encounters a vlog_startup_routines table during initialization:

```
# Shared object file './vpi_test.sl'
#    vlog_startup_routines array
#    Function ptr '0x4001d310'. Before first call using function pointer.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using the function pointer (i.e., bp -c *0x4001d310). ModelSim will hit this breakpoint as you continue through initialization.

Completing design load

If you are through looking at the initialization code you can select **Tools** > **C Debug** > **Complete load** at any time, and ModelSim will continue loading the design without stopping. The one exception to this is if you have set a breakpoint in a LoadDone callback and also specified **Keep user init bps** in the C Debug Setup dialog (see "C Debug dialog reference" (UM-490)).

Debugging functions when quitting simulation

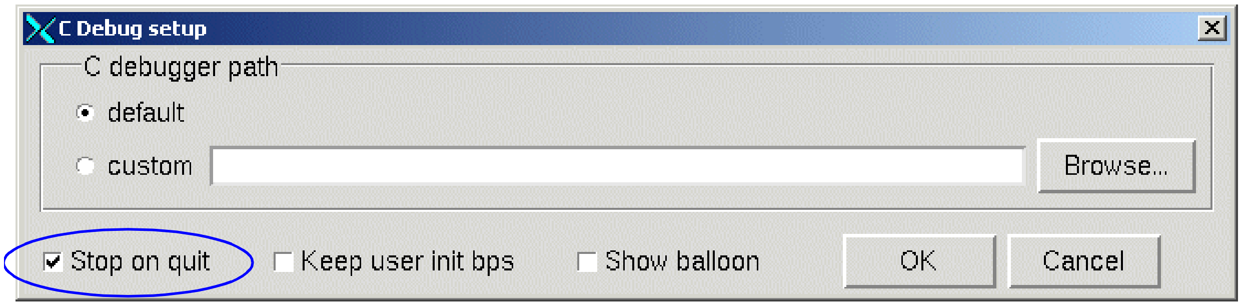
Stop on quit mode allows you to debug functions that are called when the simulator exits. Such functions include those referenced by an **mti_AddQuitCB** function in FLI code, **misctf** functions called by a quit or $finish in PLI code, or **cbEndofSimulation** functions called by a quit or $finish in VPI code.

To enable Stop on quit mode, follow these steps:

1 Start C Debug by selecting **Tools** > **C Debug** > **Start C Debug**.

2 Select **Tools** > **C Debug** > **C Debug Setup**.

3 Select **Stop on quit** in the C Debug setup dialog.


C Debug setup

C debugger path

[✓ default] custom

[✓ Stop on quit] Keep user init bps Show balloon

With this mode enabled, if you have set a breakpoint in a quit callback function, C Debug will stop at the breakpoint after you issue the quit command in ModelSim. This allows you to step and examine the code in the quit callback function.

Invoke **run -continue** when you are done looking at the C code.

Note that whether or not a C breakpoint was hit, when you return to the VSIM> prompt, you'll need to quit C Debug by selecting **Tools** > **C Debug** > **Quit C Debug** before finally quitting the simulation.

C Debug menu reference

The following commands are available from the **Tools > C Debug** menu.

Start C Debug	turns on C Debug so you can set breakpoints and step through C code
C Debug setup	specifies the location of your gdb installation
Enable auto step	configures C Debug to run in "Identifying all registered function calls" (UM-480)
Run	provides access to step, step-over, run-continue, and run-finish commands
Quit C Debug	turns off C Debug; do this before exiting ModelSim
Init mode	configures C Debug to run in "Debugging functions during elaboration" (UM-483)
Complete load	cancels "Debugging functions during elaboration" (UM-483) and completes loading the rest of your design
Auto find bp	sets breakpoints at all the FLI/PLI/VPI function entry points that are known (registered) when you make this menu selection
Info bp	lists all currently set breakpoints including the source file names, line numbers, and breakpoint ids
Show	shows the values of the local variables and arguments of the current C function
Traceback	if known, identifies the HDL source line from which the C function was called; when running in "Debugging functions during elaboration" (UM-483), no HDL information is available, and this command will list only the gdb traceback stack
C Interrupt	"re-activates" the C debugger when you are stopped in HDL code
Command entry	opens a command prompt dialog so you can enter commands even if the GUI prompt is inaccessible; the GUI prompt may become inaccessible in certain situations (e.g., when debugging FLI LoadDone callback functions)
Refresh	reopens a C source file if you close the Source window inadvertently while stopped in the C debugger

C Debug command reference

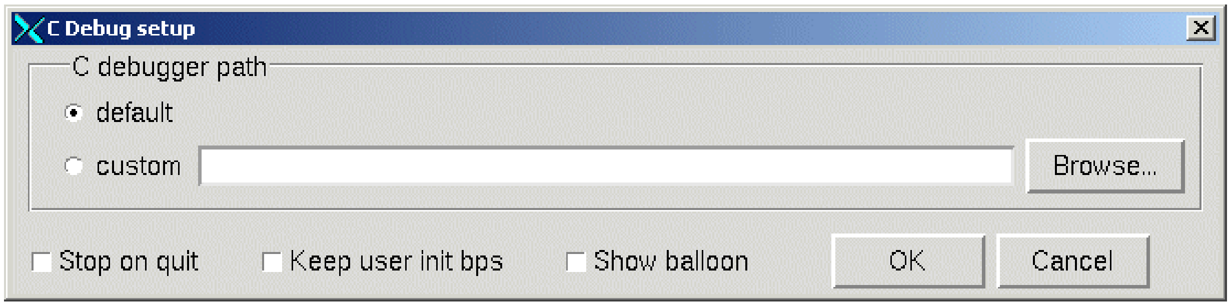
The table below provides a brief description of the commands that can be invoked when C Debug is running. Follow the links to the *ModelSim SE Command Reference* for complete command syntax.

Command	Description	Corresponding menu command
bd (CR-76)	deletes a previously set C breakpoint	right click breakpoint in Source window and select Remove Breakpoint
bp (CR-81) -c	sets a C breakpoint	click the desired line number in the Source window
change (CR-87)	changes the value of a C variable	none
describe (CR-152)	prints the type information of a C variable	select the C variable name in the Source window and select Tools > Describe or right click and select Describe.
disablebp (CR-153)	disables a previously set C breakpoint	right click breakpoint in Source window and select Disable Breakpoint
enablebp (CR-163)	enables a previously disabled C breakpoint	right click breakpoint in Source window and select Enable Breakpoint
examine (CR-167)	prints the value of a C variable	select the C variable name in the Source window and select Tools > Examine or right click and select Examine
gdb dir (CR-179)	sets the source directory search path for the C debugger	none
pop (CR-215)	moves the specified number of call frames up the C callstack	none
push (CR-231)	moves the specified number of call frames down the C callstack	none
run (CR-246) -continue	continues running the simulation after stopping	click the run -continue button on the Main or Source window toolbar
run (CR-246) -finish	continues running the simulation until control returns to the calling function	Tools > C Debug > Run > Finish
show (CR-260)	displays the names and types of the local variables and arguments of the current C function	Tools > C Debug > Show
step (CR-264)	single step in the C debugger to the next executable line of C code; **step** goes into function calls, whereas **step -over** does not	click the step or step -over button on the Main or Source window toolbar
tb (CR-266)	displays a stack trace of the C call stack	Tools > C Debug > Traceback

C Debug dialog reference

This section describes C Debug dialogs.

C Debug setup dialog



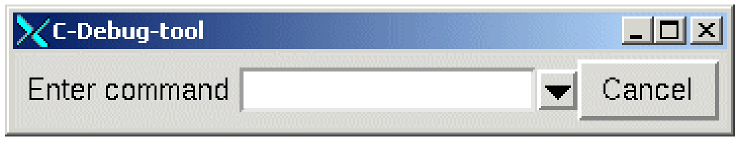
Usage

Configuring C Debug

Field descriptions

- **C debugger path**
  Specifies the path to the installed copy of **gdb**. Select "default" to point at the Model Technology supplied gdb or "custom" to point at another installation of gdb. See "Supported platforms and gdb versions" (UM-474) for the supported versions.
- **Stop on quit**
  Allows you to debug functions that get called when the simulator is exiting. See "Debugging functions when quitting simulation" (UM-487) for details.
- **Keep user init bps**
  Leaves enabled any breakpoints you set while running in initialization mode (see "Debugging functions during elaboration" (UM-483)). Normally breakpoints set during initialization mode are disabled once the design is finished loading.
- **Show source balloon**
  Enables name/value popup in the Source window when you hover your mouse pointer over a variable name.

Command entry dialog



Usage

Entering debugging commands when the CDBG> prompt in the Main window is unavailable

Field descriptions

- **Enter command**
  Specify the debugging command to execute.

15 - PSL Assertions

Chapter contents

What are assertions?

Assertions have been around for a long time but have recently garnered heightened attention due to the increasing importance of verification in most design flows. Additionally, the recent introduction of new languages such as PSL have made assertions more powerful than they have been in the past.

Definition

An assertion is a design property that is evaluated by a tool. A property is a statement about a design that evaluates to true or false. Properties tell a tool what the design should do, what it should not do, or what limits exist on its behavior. In effect we are saying, *assert* that this property is true; if it is false, tell me.

Types of assertions

Broadly speaking there are three types of assertions: interface/system level assertions, internal architecture assertions, and functional coverage assessment.

Interface/system-level assertions

Sometimes referred to as "black-box," these types of assertions are high-level properties of a design that describe only the inputs of a module or system. The interfaces are generally between major blocks of a design that are owned by different designers. The assertions are typically placed in an external file and then attached to a design unit.

Verification engineers typically apply this use model. Many organizations prohibit the verification team from touching synthesizable RTL code. Therefore, they cannot embed assertions. Also, assertions that are defined in a separate file are easier to reuse at multiple abstraction levels (architectural, RTL and gate) as the design objects that they reference are very likely to exist at all levels.

Internal architecture assertions

Called "white-box" or "clear-box," these types of assertions are specific to the internals of a module. Internal assertions are typically written directly in the HDL code, and the property verification occurs as the simulation proceeds. The is the most typical use of assertions and is done for block/module-level verification. Designers typically apply this use model as it is easy and natural for them to include PSL assertions directly in the HDL code as the code is being written.

The advantage to internal assertions is errors can be identified very early in a simulation.

PSL assertion language

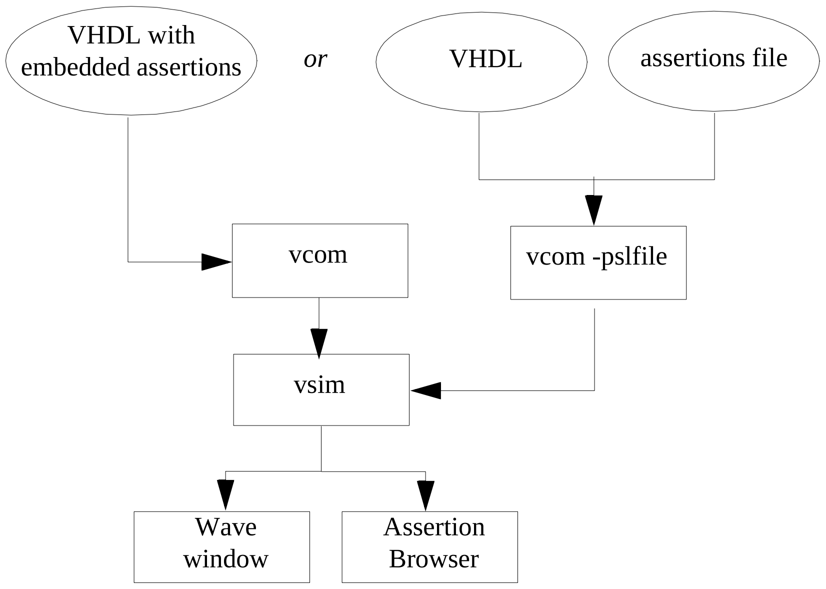
ModelSim currently supports PSL assertions. PSL is an Accellera standard that was born out of the Sugar language created at IBM. The syntax and semantics of PSL are described in the *Property Specification Language Reference Manual*, Version 1.01, published April 25, 2003. We strongly encourage you to get a copy of this specification.

In the current implementation, ModelSim supports only the simple subset of PSL (refer to Section 4.4.5, pg 25 of PSL LRM 1.01 for a description of this subset).

Using assertions in ModelSim

Assertion flow

The following diagram gives a visual depiction of using assertions in ModelSim.



ModelSim lets you embed assertions within your VHDL code or supply them in a separate file. If the assertions are embedded, **vcom** will compile them automatically. If the assertions are in a separate file, you add the **-pslfile** argument to **vcom**. Once compilation is complete, you invoke the simulator **vsim** on the design. The simulator automatically handles any assertions that are present in the design. From there you run the simulation and debug any assertion failures.

Limitations

The current release has some limitations. Most of these features will be added in future releases.

- Only the simple subset of PSL is supported except ’within’ constructs. The PSL LRM defines the simple subset in section 4.4.5 Pg 25 of PSL LRM 1.01.
- There is no Verilog support. PSL assertions can only be embedded inside VHDL code, and external assertions can only be bound to a VHDL architecture.
- Vunits can only be bound to an entity or an architecture.
- A separate PSL file must be compiled with the entity or architecture to which it is bound.
- There is no support for verification unit inheritance–vunits cannot be derived from other vunits.
- Embedded assertions cannot be placed inside VHDL generate statements.
- There is no support for replicated properties (i.e., PSL "forall" syntax).
- There is no support for endpoints.

- There is no support for parameterized named sequences and properties. For example, 'sequence s0(boolean rb, clock; const n) = ...' is illegal.
- PSL limits vunits to a single default clock declaration. However, there are no restrictions on the number of default clock declarations embedded within the HDL source.
- The @ clock expression operator is supported but can only be used for a single clock. Multi-clock support is not yet available.
- There is no support for %for and %if preprocessor commands.
- There is no support for integer, structures, and union in the modeling layer. The only PSL built-in functions currently supported are rose() and fell().
- Only "assert" and "assume" assertion directives are supported. "Assume" directives are treated functionally the same as "assert" directives.
- There is no support for post-simulation run of assertions (i.e., users cannot run assertion engine in post simulation mode). The Assertion Browser is not active in post-simulation mode either.
- Checkpoint/restore isn’t currently supported with PSL assertions.
- Vprop and vmode in the PSL modeling layer are not supported.

Embedding assertions in your code

One way of looking at assertions is as design documentation. In other words, anywhere you would normally write a comment to capture pre-conditions, constraints or other assumptions as well as to document the proper functionality of a module, process or subprogram, use assertions to capture the information instead.

Syntax

PSL assertions are embedded using metacomments prefixed with 'psl'. For example:

```
-- psl sequence s0 is {b0; b1; b2};
```

The PSL statement can be multiline. For example:

```
-- psl sequence s0 is
-- {b0; b1; b2};
```

Note that the second line did not require a 'psl' prefix. Once in PSL context, the parser will remain there until a PSL statement is terminated with a semicolon (';').

Restrictions

Embedded assertions have the following restriction as to where they can be embedded:

- Assertions can be embedded only in declarative and statement regions of an entity or architecture body.
- Assertions cannot be embedded in generate statements.
- In a statement region, assertions can appear at places where concurrent statements may appear. If they appear in a sequential statement, ModelSim will generate an error.
- Assertions cannot be embedded in VHDL procedures and functions.

Example

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;
  use WORK.constants.all;
entity dram_control is
  generic ( BUG : Boolean := TRUE );
  port ( clk     : IN     std_logic;
         reset_n : IN     std_logic;
         as_n    : IN     std_logic;
         addr_in : IN     std_logic_vector(AIN-1 downto 0);
         addr_out: OUT    std_logic_vector(AOUT-1 downto 0);
         rw      : IN     std_logic;  -- 1 to read; 0 to write
         we_n    : OUT    std_logic;
         ras_n   : OUT    std_logic;
         cas_n   : OUT    std_logic;
         ack     : OUT    std_logic );
end entity dram_control;

architecture RTL of dram_control is

  type memory_state is (IDLE, MEM_ACCESS, SWITCH, RAS_CAS, OP_ACK, REF1,
REF2);
```

```
  signal mem_state : memory_state := IDLE;

  signal col_out   : std_logic; -- Output column address
                                -- = 1 for column address
                                -- = 0 for row address

  signal count     : natural range 0 to 2;        -- Cycle counter
  signal ref_count : natural range 0 to REF_CNT;  -- Refresh counter
  signal refresh   : std_logic;                   -- Refresh request

--psl default clock is rising_edge(clk);
-- Check the write cycle
-- psl property check_write is always {fell(as_n) and not rw} |=> {
--  [*0 to 5];
--  (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(7 downto 4)));
--  (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(3 downto 0)))[*2];
--  (ras_n = '0' and cas_n = '0')[*2];
--  ack};

--psl assert check_write;

begin
.
.
.
```

Writing assertions in an external file

Assertions in an external file are grouped in vunits and bound to an architecture. The external PSL statements are interpreted as if the text of the statement was inserted in the architecture, immediately before the end of the architecture.

Syntax

```
vunit name (entity_name[(<arch_name>)])
{
    default clock is <clock_decl>;
    <assertions>;
    ...
}
```

name – The name of the vunit.

entity_name – The hierarchical path to the associated entity.

<arch_name> – The associated architecture.

<clock_decl> – The default clock declaration for the vunit.

<assertions> – Any number of verification directives or PSL statements.

Restrictions

The following restrictions exist when providing assertions in a separate file.

- The vunits can be bound only to an entity or architecture.
- The PSL file and its corresponding VHDL file must be compiled together.

Example

The following is an example with three assertions in one vunit.

```
vunit check_dram_controller(dram_control(RTL))
{
  default clock is rising_edge(clk);

  -- declare refresh sequence
  sequence refresh_sequence  is
    {not cas_n and ras_n and we_n; [*1]; (not cas_n and not ras_n and
      we_n)[*2]; cas_n and ras_n};

  -- Make sure the first refresh happened in 24 cycles period after reset
  property check_refresh_rate  is always {
    (not reset_n)[+];    -- reset_n active for one or more
    rose(reset_n);     -- reset_n deactivates
    (rose(refresh))[->1 to inf]}      -- wait for next refresh_start
  |->
    {[*18 to 32]; refresh_sequence};

  assert check_refresh_rate;


  -- Check the write cycle
  property check_write  is always {fell(as_n) and not rw} |=> {
    [*0 to 5];
    (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(7 downto 4)));
    (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(3 downto
      0)))[*2];
    (ras_n = '0' and cas_n = '0')[*2];
    ack};

  assert check_write;

  -- check the read cycle
  property check_read is always {fell(as_n) and rw} |=> {
    [*0 to 5];
    (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(7 downto 4)));
    (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(3 downto
      0)))[*2];
    (ras_n = '0' and cas_n = '0')[*3];
    ack};

  assert check_read;
}
```

Understanding clock declarations

All assertions in ModelSim must be associated with one and only one clock. Unclocked and multiple clocked assertions are not currently supported.

Default clock

Any assertion that is not individually clocked will be clocked by the default clock. For example:

```
default clock is rose(clk);
assert always sigb@rose(clk1)
assert always siga;
```

The first assertion is sensitive to *clk1*. The second assertion is sensitive to *clk* (the default clock).

When using embedded assertions, if you declare an unclocked assertion before defining default clock, ModelSim produces an error. For example, the following code will produce an error, assuming there is no other default clock statement above the assertion:

```
assert always siga;
default clock is rose(clk);
```

This is not true in the case of assertions located in an external file. The default clock applies to all unclocked statements regardless of their order within the file.

As noted earlier in "Limitations" (UM-496), default clock declarations are associated with directives *not* with named properties or sequences. For example:

```
default clock is clk1
property p0 is always a->b
default clock is clk2
assert p0
```

The property *p0* is evaluated at every *clk2*.

Partially clocked properties

The default clock also applies to partially clocked properties. For example:

```
default clock is rose(clk);
assert always (b0 |->  (b1@rose(clk1)))
```

In this case, only the RHS of the implication(|->) expression is clocked. The outermost property is unclocked, so default clock applies to this assertion. However, that makes the property multiple clocked, which we do not currently support. ModelSim emits a warning in such cases and only considers the outermost clock. So ModelSim will behave as if the property was written like this:

```
assert always (b0 |-> b1)@rose(clk)
```

The warning produced will look something like this:

```
** Warning: [11] ./src/multiclk/test.vhd(34): The PSL expression possibly
contains multiple clock domains. Multiple clock domains are not supported.
The outermost clock domain will overide the inner clock domains
```

Also, the complete assertion property must be clocked. For example, if you have the following assertion:

```
assert always (b0 |->  (b1@rose(clk1)))
```

and no default clock preceding it, then since part of the property is unclocked, ModelSim will produce an error.

Understanding assertion names

PSL does not provide a method for naming directives. ModelSim must generate an assertion name for reporting information about the assertion. If you want ModelSim to generate predictable names, you should always assert on a named property. For example:

```
property p0 is always a -> b;
assert p0;
```

The name generated for this assertion statement will be *assert__p0*. Generically, the syntax of the generated name is:

```
assert__<property name>.
```

However, if you write the same assertion in this manner:

```
assert always a -> b;
```

there is no property name, so ModelSim will generate a name like *assert__0* (i.e., a number appended to "assert__").

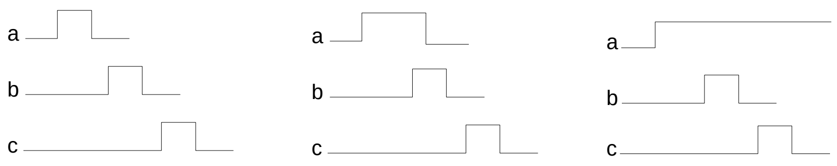
General assertion writing guidelines

Assertion writing can become complicated and confusing. If not written correctly, assertions can also impact simulator performance. This section offers suggestions for how to write assertions that are easy to debug and don't slow down your simulation unduly.

- Keep directives simple. Create named assertions that you then reference from the directive (e.g., assert check1).
- Keep properties and sequences simple too. Build complex assertions out of simple, short assertions/sequences.
- Do not use implication with never directives. You will rarely get what you want if you use implication with a never.
- Create named sequences so you can reuse them in multiple assertions.
- Be aware of "unexpected matches." For example, the following assertion:

  ```
  assert always a->next(b)->next(c);
  ```

  will match all of the following conditions (as well as others):



- Keep time ranges specified in sequences as short as possible according to the actual design property being specified. Avoid long time ranges as this increases the number of concurrent 'in-flight' checks of the same property and thereby impacts performance.

Understanding operator precedence and curly braces

VHDL and PSL have conflicting operator precedence rules that necessitate the use of curly braces in some cases.

Rule

In general, whenever a PSL operation expects a PSL sequence as an operand, that PSL sequence must appear within curly braces.

Exceptions

The following are exceptions to the above rule:

- The always, never, and eventually! operators can accept a named sequence as an operand without the requisite curly-braces (e.g., "always myseq;" is equivalent to "always {myseq};").
- The "trigger" operand of the within* operators can also accept a named sequence as an operand without the requisite curly-braces (e.g., "within(myseq, bool){myseq};" is equivalent to "within({myseq}, bool){myseq};").

Compiling and simulating assertions

Embedded assertions

Embedded assertions are compiled automatically by default. If you have embedded assertions that you don't want to compile, use the **-nopsl** argument to the **vcom** command (CR-303).

External assertions file

To compile assertions in an external file, invoke the compiler with the **-pslfile** argument and specify the assertions file name. For example:

```
vcom tadder.vhd adder.vhd -pslfile adder.psl
```

The design and its associated assertions file must be compiled in the same invocation.

Making changes to assertions

After making any changes to embedded assertions, you need to re-compile the design unit. After making changes in separate file assertions, you need to compile both the separate file and the design unit file to which the vunit binds in the same **vcom** invocation.

Simulating assertions

If any assertions were compiled, the **vsim** command (CR-357) automatically invokes the assertion engine at runtime. If you do not want to simulate the compiled assertions, use the **-nopsl** argument.

VHDL code inside PSL statements

VHDL statements may be placed in either embedded PSL metacomments or in external vunits. For example, the following code is legal:

```
--psl process(reg)
--psl ...
--psl end process
--psl assert always p0;
```

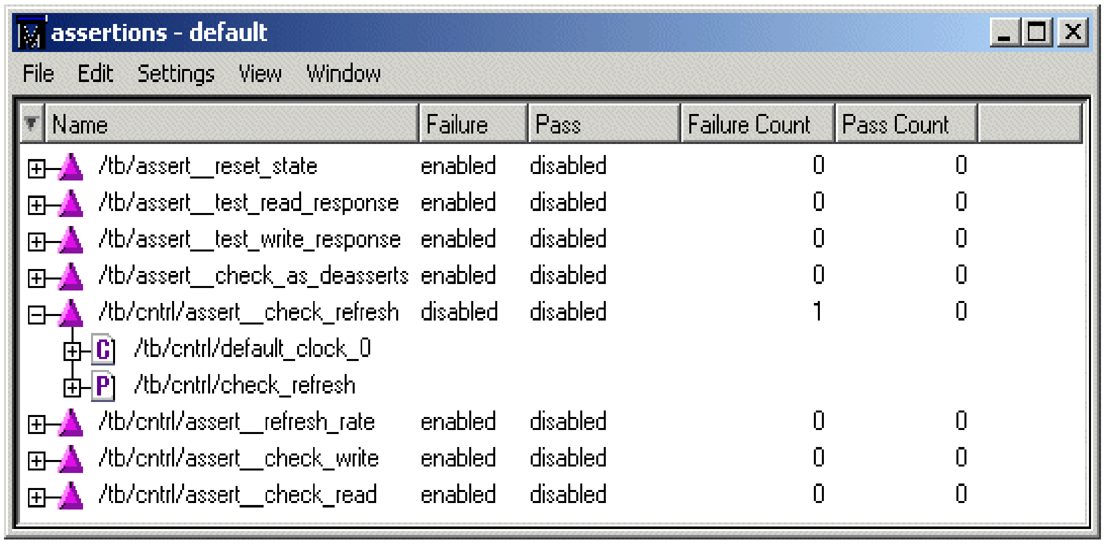
The VHDL statements are parsed along with the PSL statements when you compile the design with **vcom**. If you compile the design using **vcom -nopsl**, then neither the VHDL statements nor the PSL statements are parsed.

Managing assertions

You can manage your assertions via the GUI or by entering commands at the VSIM> prompt.

Viewing assertions in the Assertion Browser

The Assertion Browser provides a convenient interface to all of the assertions in the current simulation. To open the Assertion Browser, select **View > Assertion Browser.**



Name	Failure	Pass	Failure Count	Pass Count
/tb/assert__reset_state	enabled	disabled	0	0
/tb/assert__test_read_response	enabled	disabled	0	0
/tb/assert__test_write_response	enabled	disabled	0	0
/tb/assert__check_as_deasserts	enabled	disabled	0	0
/tb/cntrl/assert__check_refresh	disabled	disabled	1	0
/tb/cntrl/default_clock_0				
/tb/cntrl/check_refresh				
/tb/cntrl/assert__refresh_rate	enabled	disabled	0	0
/tb/cntrl/assert__check_write	enabled	disabled	0	0
/tb/cntrl/assert__check_read	enabled	disabled	0	0

The Assertions Browser lists all embedded and external assertions that were successfully compiled and simulated during the current session. The plus sign ('+') to the left of the Name field lets you expand the assertion hierarchy to show its elements (properties, sequences, clocks, and HDL signals).

The window displays five fields by default, as detailed below. See "Hiding/showing fields in the Assertion Browser" (UM-509) for details on how to hide or show fields.

The Assertions Browser includes the following fields:

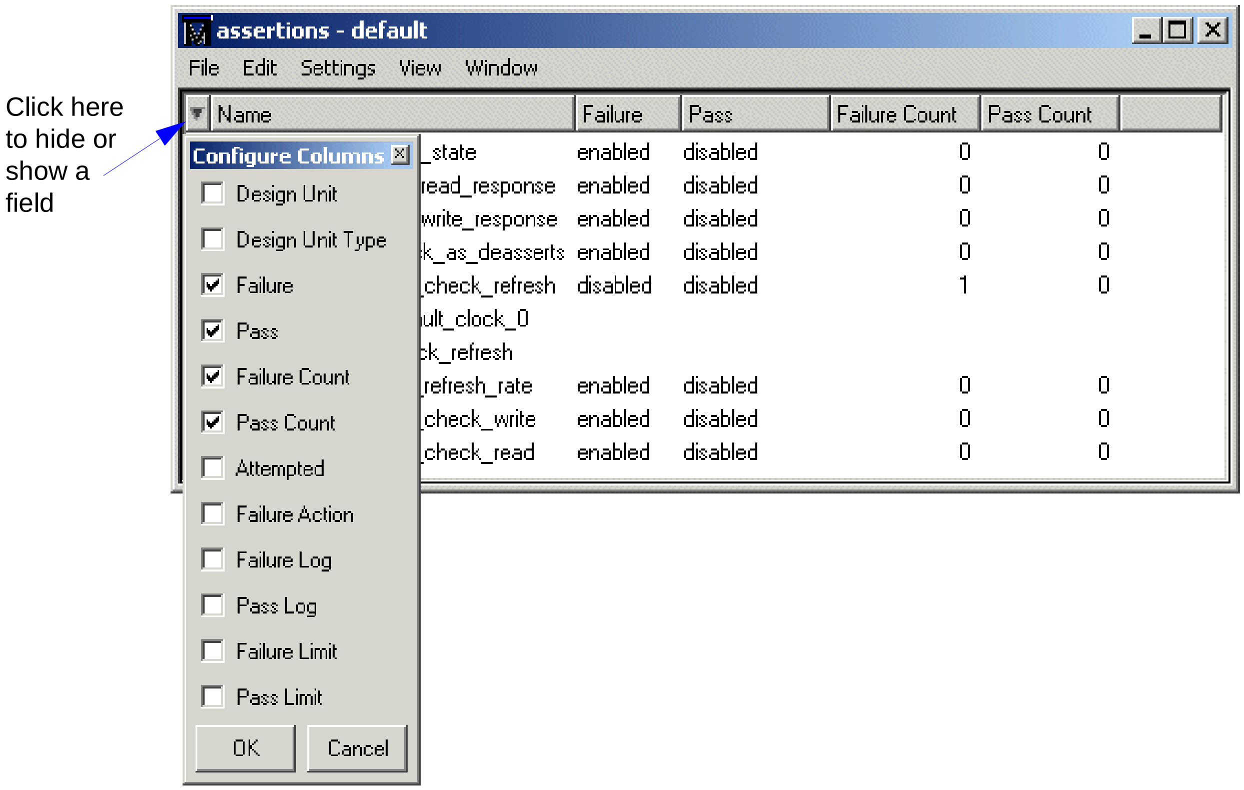
- The **Name** field lists the PSL statement or vunit name you specified in the assertion code. For vunits the individual assertion names are listed under the vunit name. Also, any signal referenced in an assertion will be part of the hierarchy as well. See "Understanding assertion names" (UM-504) for more details on assertion names.
- The **Design Unit** field identifies the design unit to which the assertion is bound. Not displayed by default.
- The **Design Unit Type** field lists the HDL type of the design unit. Not displayed by default.
- The **Failure** field shows "enabled" when failure checking is enabled on the assertion. If the field shows "disabled", ModelSim isn't checking that assertion's failures.
- The **Pass** field shows "enabled" when pass checking is enabled on the assertion. If the field shows disabled, ModelSim isn't tracking that assertion's checking.

- The **Failure Count** field counts the total number of times the assertion has failed in the current simulation. These counts are maintained between runs unless you reset the count for the assertion.
- The **Pass Count** field counts the total number of times the assertions has passed in the current simulation. These counts are maintained between runs unless you reset the count for the assertion.
- The **Attempted** field shows a green checkmark when an assertion has triggered and a red 'X' when it has not triggered. Not displayed by default.
- The **Failure Action** field lists the action that ModelSim takes when the assertion passes or fails. Not displayed by default.
- The **Failure Log** field shows "enabled" when failure messages will be logged to the transcript. The field shows "disabled" when failure messages will not be logged to the transcript. Not displayed by default.
- The **Pass Log** field shows "enabled" when pass messages will be logged to the transcript. The field shows "disabled" when pass messages will not be logged to the transcript. Not displayed by default.
- The **Failure Limit** field shows the number of times ModelSim will respond to a failure event on an assertion. Not displayed by default.
- The **Pass Limit** field shows the number of times ModelSim will respond to a failure event on an assertion. Not displayed by default.

You can also view this same information in textual format using the **assertion report** command (CR-73).

Hiding/showing fields in the Assertion Browser

You can hide or show any of the fields in the Assertion Browser. Click the drop-down arrow on the left-hand side of the dialog and select a field name.


Click here to hide or show a field
assertions - default
File Edit Settings View Window
Configure Columns
OK
Cancel

Name	Failure	Pass	Failure Count	Pass Count
_state	enabled	disabled	0	0
read_response	enabled	disabled	0	0
write_response	enabled	disabled	0	0
k_as_deasserts	enabled	disabled	0	0
_check_refresh	disabled	disabled	1	0
ult_clock_0				
ck_refresh				
_refresh_rate	enabled	disabled	0	0
_check_write	enabled	disabled	0	0
_check_read	enabled	disabled	0	0

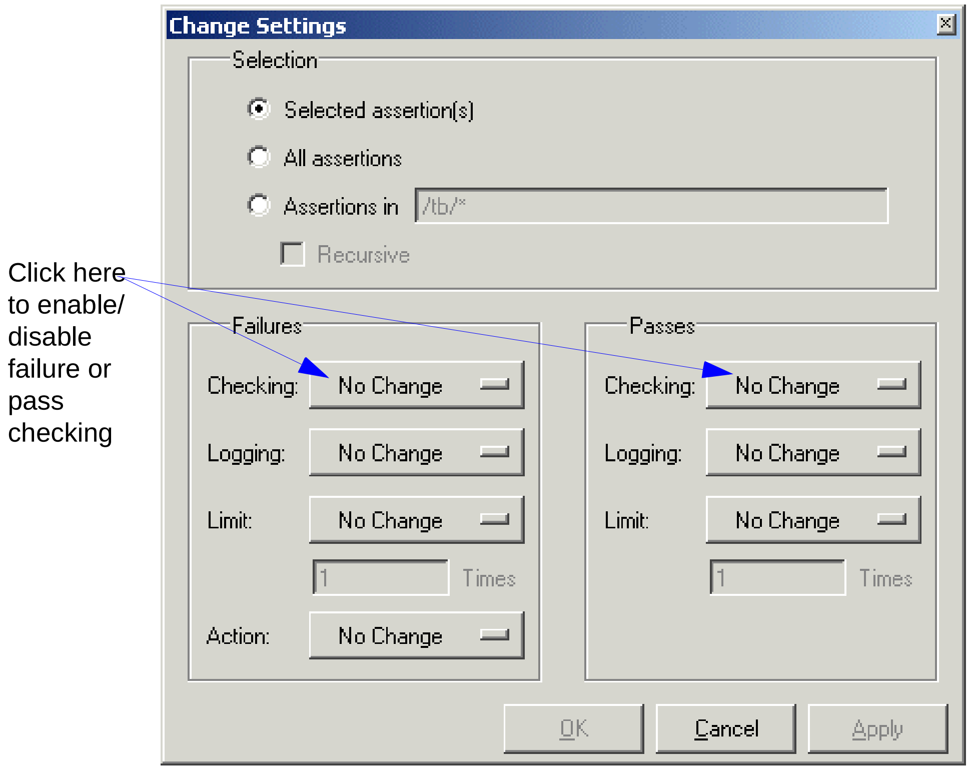
- ☐ Design Unit
- ☐ Design Unit Type
- ☑ Failure
- ☑ Pass
- ☑ Failure Count
- ☑ Pass Count
- ☐ Attempted
- ☐ Failure Action
- ☐ Failure Log
- ☐ Pass Log
- ☐ Failure Limit
- ☐ Pass Limit

The selection acts as a toggle. Select it once to hide a field; select it again to show the field.

Enabling/disabling failure and pass checking

To enable or disable an assertion's failure or pass checking from the GUI, right-click an assertion in the Assertion Browser and select **Failure Checking** or **Pass Checking** (or use the **Settings** menu on the menu bar). The selection acts as a toggle.

To gain greater control over enabling and disabling, right-click an assertion and select **Change**. This opens the Change Settings dialog.



Selection
- [✓ Selected assertion(s)]
- All assertions
- Assertions in /tb/*
- Recursive [ ]

Failures		Passes	
Checking:	No Change	Checking:	No Change
Logging:	No Change	Logging:	No Change
Limit:	No Change	Limit:	No Change
	1 Times		1 Times
Action:	No Change		

In this dialog, you can enable/disable failure or pass checking for the selected assertion, all assertions, or the assertions in a particular instance. Select **Recursive** when enabling/ disabling by instance to search for assertions in subregions of the instance.

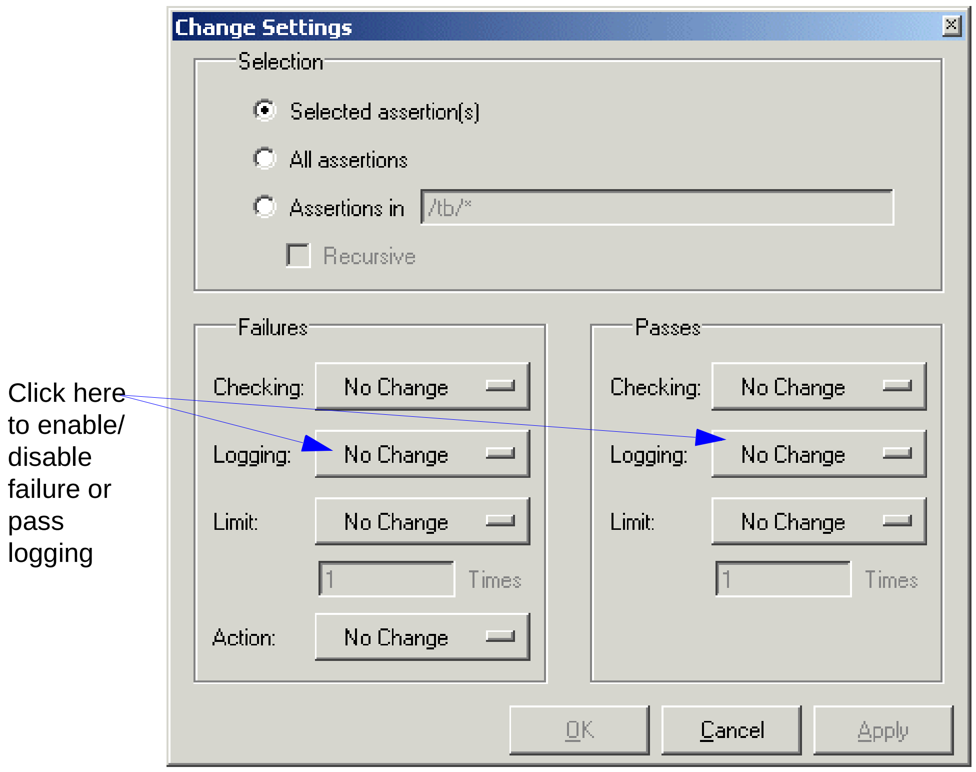
Select **Enable** or **Disable** from the **Checking** drop downs in the middle of the dialog and then click OK.

You can also enable or disable failure and pass checking using the **assertion fail** command (CR-69) or the **assertion pass** command (CR-71), respectively.

Enabling/disabling failure and pass logging

To enable or disable an assertion’s failure or pass logging from the GUI, right-click an assertion in the Assertion Browser and select **Failure Log** or **Pass Log** (or use the **Settings** menu on the menu bar). The selection acts as a toggle.

To gain greater control over logging, right-click an assertion and select **Change**. This opens the Change Settings dialog.



Selection
[✓ Selected assertion(s)]
All assertions
Assertions in /tb/*
Recursive

Failures
Checking: No Change
Logging: No Change
Limit: No Change
1 Times
Action: No Change

Passes
Checking: No Change
Logging: No Change
Limit: No Change
1 Times

In this dialog, you can enable/disable failure or pass logging for the selected assertion, all assertions, or the assertions in a particular instance. Select **Recursive** when enabling/ disabling by instance to search for assertions in subregions of the instance.

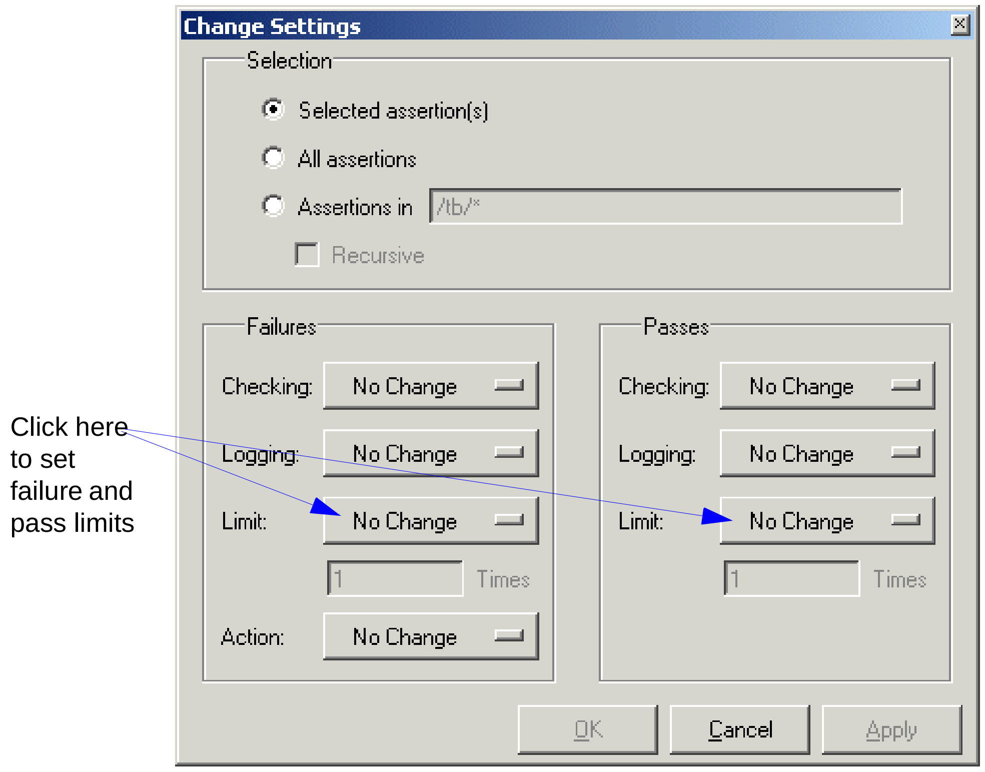
Select **Enable** or **Disable** from the **Logging** drop downs in the middle of the dialog and then click OK.

You can also enable or disable failure and pass logging using the **assertion fail** command (CR-69) or the **assertion pass** command (CR-71), respectively.

Setting failure and pass limits

The failure and pass limits determines how many times ModelSim processes an assertion before disabling it for the duration of the simulation. By default the number is one for both failure and pass limits. In other words, once an assertion passes or fails, ModelSim disables for the duration of the simulation.

If you want to see more than one assertion failure or pass, right-click the assertion in the Assertion Browser and select **Change**. This opens the Change Settings dialog.



Selection

[✓ Selected assertion(s)]

All assertions

Assertions in /tb/*

Recursive

Failures		Passes	
Checking:	No Change	Checking:	No Change
Logging:	No Change	Logging:	No Change
Limit:	No Change	Limit:	No Change
	1 Times		1 Times
Action:	No Change		

You can set the failure and pass limits for the selected assertion, all assertions, or the assertions in a particular instance. Select **Recursive** when setting by instance to search for assertions in subregions of the instance.

Select **Limited** or **Unlimited** from the **Limit** drop downs at the bottom of the dialog. If you select Limited, enter an integer in the field below the drop down and then click **OK**.

Once the limit is reached, ModelSim disables that assertion. ModelSim continues to respond to others if their limit has not been reached. The limit applies to the entire simulation session and not to any single simulation run command.

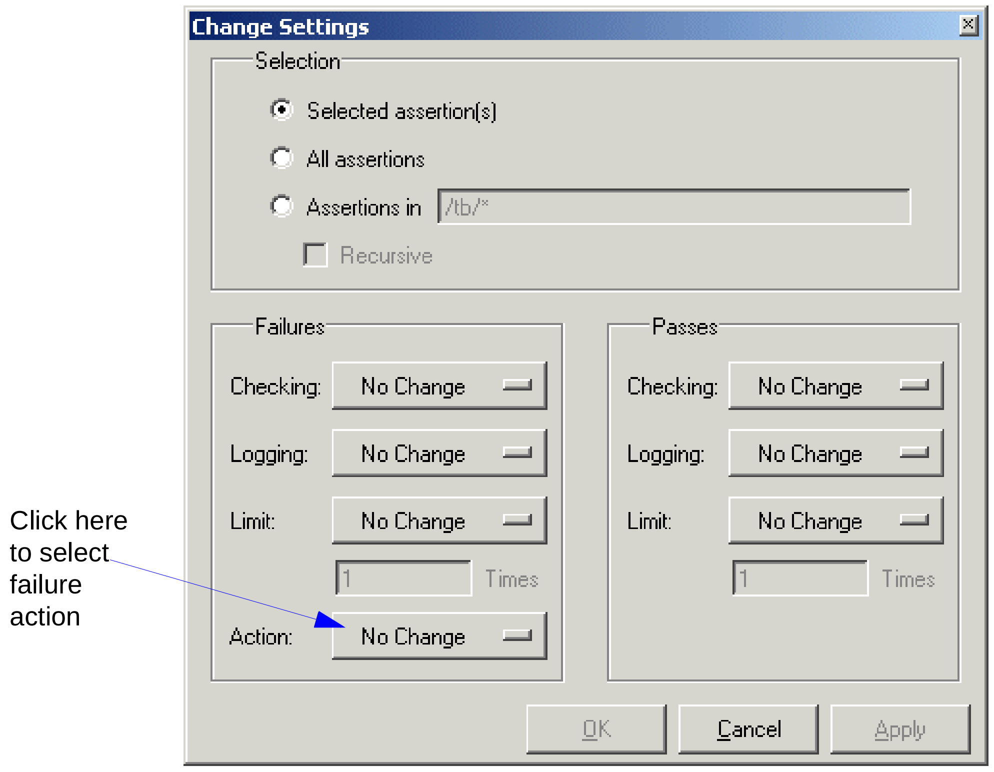
You can also set failure and pass limits using the **assertion fail** command (CR-69) or the **assertion pass** command (CR-71), respectively.

Setting failure action

ModelSim can take one of three actions when an assertion fails: it can log the failure in the transcript and continue the simulation; it can break (pause) the simulation; or it can stop and exit the simulation. By default the failure action is "continue."

To set assertion action in the GUI, right-click an assertion in the Assertion Browser and select **Failure Action** and then Continue, Break, or Exit (or use the **Settings** menu on the menu bar).

To gain greater control over setting failure action, right-click an assertion and select **Change**. This opens the Change Settings dialog.



Change Settings

Selection
- (●) Selected assertion(s)
- ( ) All assertions
- ( ) Assertions in /tb/*
- [ ] Recursive

Failures		Passes	
Checking:	No Change	Checking:	No Change
Logging:	No Change	Logging:	No Change
Limit:	No Change	Limit:	No Change
	1 Times		1 Times
Action:	No Change		

OK Cancel Apply

You can set the action for the selected assertion, all assertions, or the assertions in a particular instance. Select **Recursive** when setting by instance to search for assertions in subregions of the instance.

Select **Continue**, **Break**, or **Exit** from the **Action** drop down in the bottom left corner of the dialog and then click OK.

You can also set failure action using the **assertion fail** command (CR-69).

Reporting on assertions

You can use the **assertion report** command (CR-73) to print to the transcript a variety of information about assertions in the current design.

Specifying an alternative output file for assertion messages

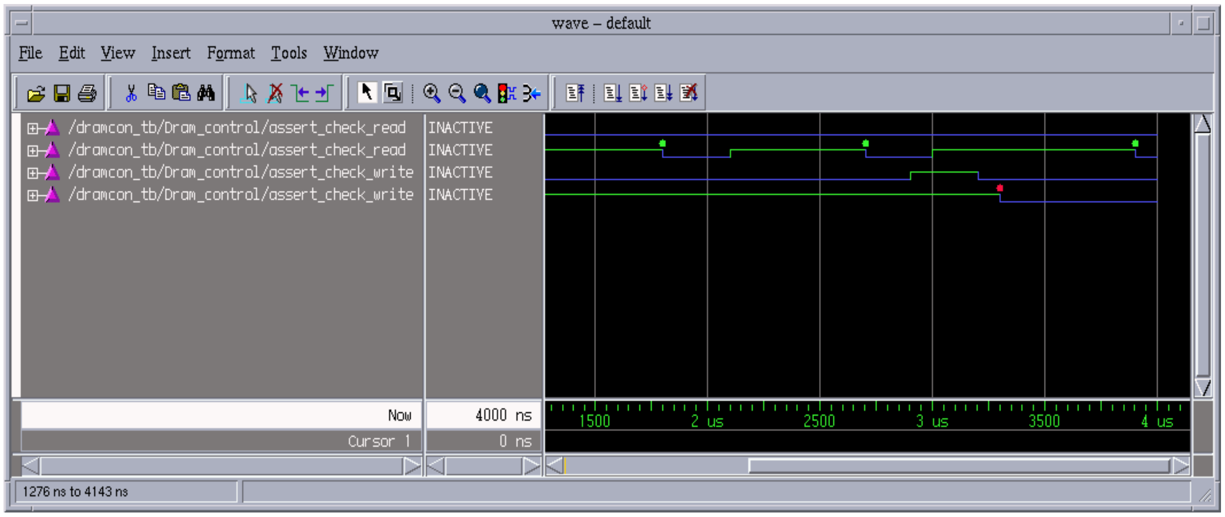
You can specify an alternative output file for recording assertion messages. To do this, invoke **vsim** with the **-assertfile <filename>** argument. By default assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file. You can set a permanent default for the alternative output file using the AssertFile (UM-621) variable in the *modelsim.ini* file.

Viewing assertions in the Wave window

You can view assertions in the Wave window just like any other signal in your design. Simply drag an assertion from the Assertion Browser and drop it in the Wave window or right-click an assertion in the Assertion Browser and select **Add Wave**.

Assertion ’signals’

ModelSim represents assertions as waveforms in the Wave window. The picture below shows several assertions in a Wave window.



```
/dramcon_tb/Dram_control/assert_check_read   INACTIVE
/dramcon_tb/Dram_control/assert_check_read   INACTIVE
/dramcon_tb/Dram_control/assert_check_write  INACTIVE
/dramcon_tb/Dram_control/assert_check_write  INACTIVE
Now       4000 ns
Cursor 1     0 ns
1276 ns to 4143 ns
```

Assertion items are represented by a magenta triangle. The name of each assertion comes from the assertion code. The plus sign (’+’) to the left of the name indicates that an assertion is a composite trace and can be expanded to show its elements (properties, sequences, clocks, and HDL signals).

The value in the value pane is determined by the active cursor in the waveform pane. The value will be one of "ACTIVE", "INACTIVE", "PASS" or "FAIL".

The waveform for an assertion represents both continuous and instantaneous information. The continuous information is whether or not the assertion is active. The assertion is active anytime it matches the first element in the directive. When active, the trace is raised and painted green; when inactive it is lowered and painted blue. The instantaneous information is a pass or fail event on the assertion. These are shown as filled circles above the trace at the time of the event. A pass is a green circle and a fail is a red circle.

Graphic element	Meaning
blue line	assertion is inactive
green line	assertion is active
green dot	assertion passed
red dot	assertion failed

Example debugging session

The following example shows a typical debugging session for an assertion failure. The example is based on a DRAM controller with a DRAM behavioral model and a self-checking testbench (i.e., it writes to memory addresses and reads back the values to compare to what was written). The design has a bug somewhere that we need to locate.

The files for this example are included in *<install_dir>/modeltech/examples/psl*.

How would you debug without assertions?

If you didn't add assertions to the design, the first indication of a problem would come when the testbench found a difference between a write and a read. In the example design, this error occurs at time 267,400 ns. Either a wrong value was written to memory or the memory location was corrupted after it was written.

To debug the error, you might first examine simulation waveforms and look for all writes to the memory location and check the data on the bus and the actual memory contents at the location after each write. If that didn't identify the problem, you might then check all refresh cycles to determine if a refresh corrupted the memory location.

Quite possibly, both debugging activities would be required depending on one's skill (or luck) in determining the most likely cause of the error. Anyway you look at it, it's a tedious exercise.

The example assertions file

Adding assertions to the design can ease the debugging task significantly. Here is the assertions file that we will compile and simulate with the design:

```
vunit check_top_unit(tb)
{
  default clock is rose(clk);

  // Check for reset
  sequence reset_state is {[*]; fell(reset); ras_n and cas_n and not ack};
  assert always reset_state;

  // check for memory response
  sequence test_read_response is {[*]; rose(as) and rw; [*5 to 12]; ack};
  sequence test_write_response is {[*]; rose(as) and not rw; [*5 to 12];
ack};

  assert always test_read_response;
  assert always test_write_response;

  // Check if address strobe is deasserted after acknowledge from memory
  sequence check_as_deasserts is {[*]; rose(as); [*5 to 12]; ack; not as};
  assert always check_as_deasserts;

}

vunit check_dram_controller(tb.cntrl)
{
  default clock is rose(clk);

  // declare refresh sequence
  sequence refresh_sequence is
```

```
    {not cas_n and ras_n and we_n; [*1]; (not cas_n and not ras_n and /
    we_n)[*2]; cas_n and ras_n};

  // Check if the refresh_sequence repeats in 24 to 30 cycles
  property check_refresh_rate is {
     (not reset_n)[+];                     // reset_n active for one or more
      rose(reset_n);                        // reset_n deactivates
      (rose(not cas_n and ras_n and we_n))[->0..]} // wait for next
refresh_start
    |->
      {[*18..32]; refresh_sequence};

  assert always check_refresh_rate;

  // Check the write cycle
    property check_write is {fell(as_n) and not rw} |=> {
          [*0..5];
          not ras_n and cas_n and addr_out = addr_in[7 downto 4];
          (not ras_n and cas_n and (addr_out = addr_in[3 downto 0]))[*2];
          (not ras_n and not cas_n)[*2];
          ack};

  assert always check_write;

  // check the read cycle
  property check_read is {fell(as_n) and rw} |=> {
          [*0..5];
          not ras_n and cas_n and addr_out = addr_in[7 downto 4];
          (not ras_n and cas_n and (addr_out = addr_in[3 downto 0]))[*2];
          (not ras_n and not cas_n)[*3];
          ack};
  assert always check_read;
}
```

Debugging the assertion failure

Here are the steps to debug the assertion failure:

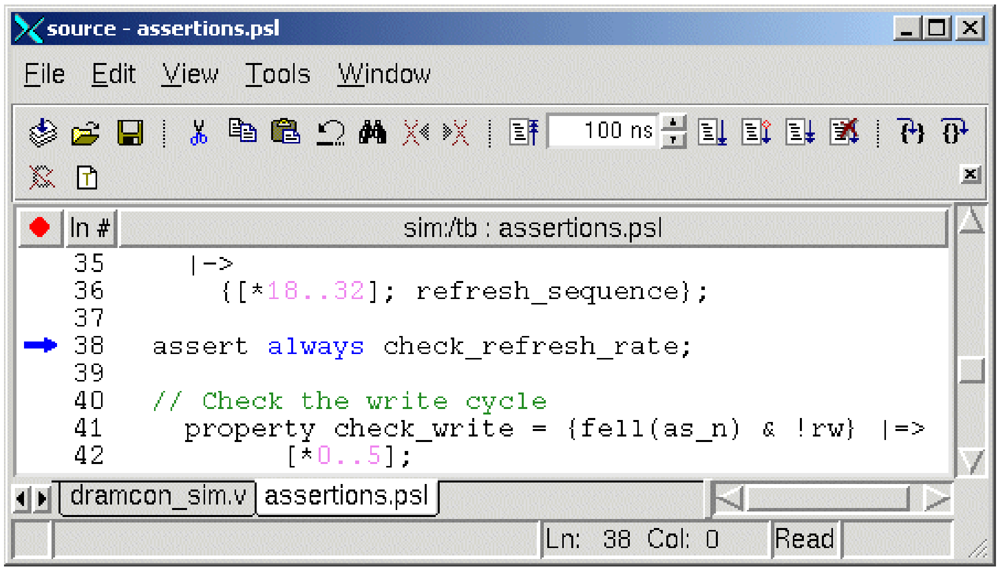
1 Once you compile and simulate the design with the assertions file, select **Setting** > **Action** from the Assertion Browser and set the assertion action to break on failures.

2 Execute **run -all** and observe the error message in the transcript:

```
VSIM> run -all
# ** Error: Assertion tb.cntrl.assert_check_refresh_rate
(File:assertions.psl Line:38) failed at 3400 for startTime 100
#
#    Time: 3400 ns  Iteration: 1  Region: /tb  File: dramcon_sim.v
# Simulation stop requested.
```

Notice that we caught the problem much earlier in the simulation than when we simulated without assertions. We also know that the problem has something to do with the refresh rate.

3 When the failure occurs, the assertions file automatically opens in the Source window with the line marker on the failed assertion.



```
35      |->
36        {[*18..32]; refresh_sequence};
37
38    assert always check_refresh_rate;
39
40    // Check the write cycle
41      property check_write = {fell(as_n) & !rw} |=>
42             [*0..5];
```

Looking at the property definition in the Source window (lines 31-36), we see that if reset has completed and a refresh cycle has been detected, then refresh must successfully complete every 18 to 32 clock cycles.

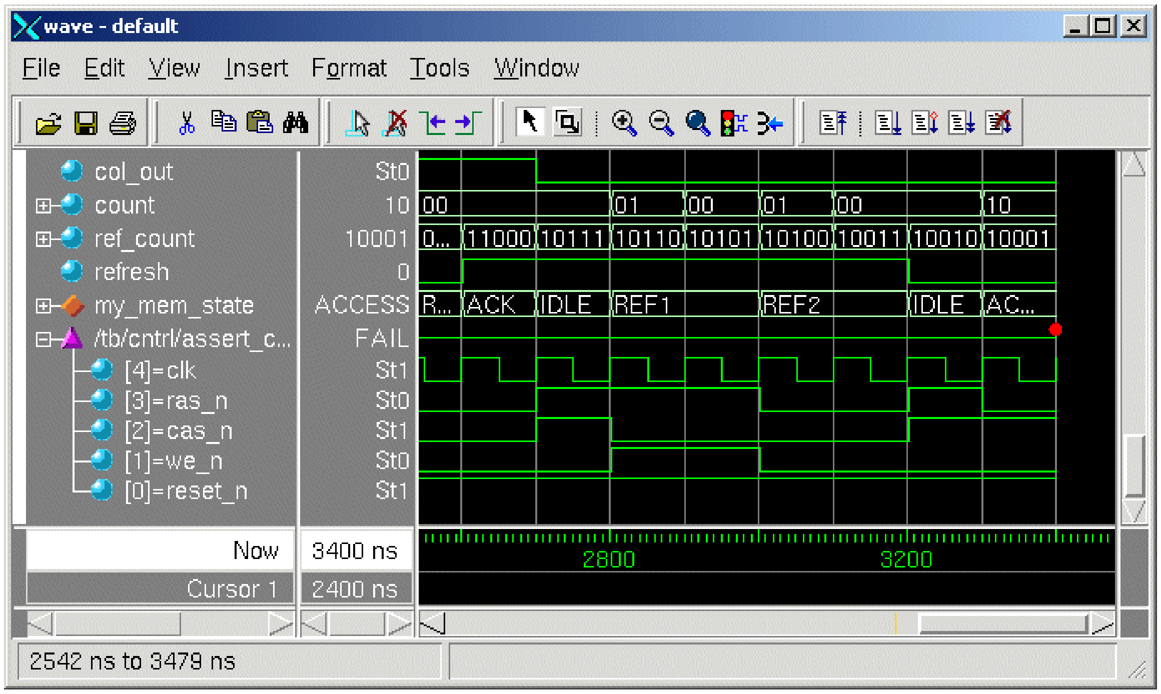
```
property check_refresh_rate is {
     (not reset_n)[+];                    // reset_n active for one or more
      rose(reset_n);                        // reset_n deactivates
      (rose(not cas_n and ras_n and we_n))[->0..]} // wait for next
refresh_start
    |->
      {[*18 to 32]; refresh_sequence};
```

The *refresh_sequence* (the last line of the property) is defined on lines 27 and 28:

```
// declare refresh sequence
sequence refresh_sequence is
{not cas_n and ras_n and we_n; [*1]; (not cas_n and not ras_n and we_n)[*2];
cas_n and ras_n};
```

The key part of the refresh protocol is that *we_n* must be held high (write enable not active) for the entire refresh cycle. Let's check *we_n* in the Wave window to see if it actually holds for the entire cycle.

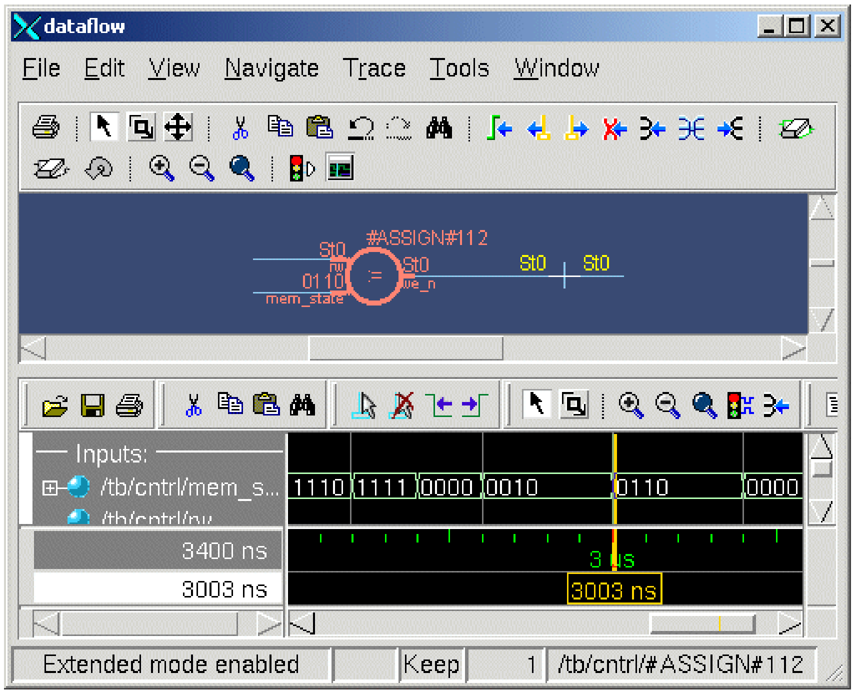
4 Right click the *check_refresh_rate* assertion in the Assertions Browser and select **Add Wave**.



Virtual signal *my_mem_state* shows the refresh cycle. By dragging that signal so it is next to the assertion, we can easily see that *we_n* is high during REF1 only and not REF2. Because *we_n* is supposed to be high through the entire refresh cycle, the assertion failed.

Next we need to access the source code for *we_n* to fix the problem. The easiest way to do this is via the Dataflow window.

5 Double-click on the *we_n* wave to open the Dataflow window.



6 Scroll if necessary to find the component in the Dataflow pane and select the signal assignment. The source code is now displayed in the Source window.

The bug is that the logic assigning *we_n* is wrong as it does not account for the REF2 state. If we fix the logic to account for REF2, the assertion will pass.

ModelSim assertion commands

The table below provides a brief description of the compare commands. See the *ModelSim Command Reference* for complete command details.

Command	Description
assertion fail (CR-69)	configures simulator response to an assertion failure
assertion pass (CR-71)	configures simulator response to an assertion pass
assertion report (CR-73)	produces textual summary of assertion results from a simulation

16 - Signal Spy

Chapter contents

This chapter describes the Signal Spy^{TM} procedures and system tasks. These allow you to monitor, drive, force, and release hierarchical items in VHDL or mixed designs.

Introduction

The Verilog language allows access to any signal from any other hierarchical block without having to route it via the interface. This means you can use hierarchical notation to either assign or determine the value of a signal in the design hierarchy from a testbench. This capability fails when a Verilog testbench attempts to reference a signal in a VHDL block or reference a signal in a Verilog block through a VHDL level of hierarchy.

This limitation exists because VHDL does not allow hierarchical notation. In order to reference internal hierarchical signals, you have to resort to defining signals in a global package and then utilize those signals in the hierarchical blocks in question. But, this requires that you keep making changes depending on the signals that you want to reference.

The Signal Spy procedures and system tasks overcome the aforementioned limitations. They allow you to monitor (spy), drive, force, or release hierarchical objects in a VHDL or mixed design.

The VHDL procedures are provided via the "Util package" (UM-94) within the *modelsim_lib* library. To access the procedures you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

The Verilog tasks are available as built-in "System tasks" (UM-144). The table below shows the VHDL procedures and their corresponding Verilog system tasks.

VHDL procedures	Verilog system tasks
init_signal_driver (UM-525)	$init_signal_driver (UM-534)
init_signal_spy (UM-528)	$init_signal_spy (UM-537)
signal_force (UM-530)	$signal_force (UM-539)
signal_release (UM-532)	$signal_release (UM-541)

Designed for testbenches

Signal Spy limits the portability of your code. HDL code with Signal Spy procedures or tasks works only in ModelSim, not other simulators. We therefore recommend using Signal Spy only in testbenches, where portability is less of a concern, and the need for such a tool is more applicable.

init_signal_driver

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net (called the src_object) onto an existing VHDL signal or Verilog net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

The init_signal_driver procedure drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the init_signal_driver value in the resolution of the signal.

Call only once

The init_signal_driver procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_driver only once for a particular pair of signals. Once init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all init_signal_driver calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_driver calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

Syntax

```
init_signal_driver(src_object, dest_object, delay, delay_type, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
delay	time	Optional. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
delay_type	del_mode	Optional. Specifies the type of delay that will be applied. The value must be either mti_inertial or mti_transport. The default is mti_inertial.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related procedures

init_signal_spy (UM-528), signal_force (UM-530), signal_release (UM-532)

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to *mti_transport*, the setting will be ignored and the delay type will be *mti_inertial*.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.

Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
  signal clk0 : std_logic;

begin

  gen_clk0 : process
  begin
    clk0 <= '1' after 0 ps, '0' after 20 ps;
    wait for 40 ps;
  end process gen_clk0;

  drive_sig_process : process
  begin
    init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
    init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps,
                        mti_transport);
    wait;
  end process drive_sig_process;

  ...

end;
```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay_type while setting the verbose parameter to a 1. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps.

init_signal_spy

The init_signal_spy() procedure mirrors the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

The init_signal_spy procedure only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value that was set by init_signal_spy.

Call only once

The init_signal_spy procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_spy once for a particular pair of signals. Once init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation.

Thus, we recommend that you place all init_signal_spy calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_spy calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

Syntax

```
init_signal_spy(src_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	**Type**	**Description**
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object's value is mirrored onto the dest_object. Default is 0, no message.

Related functions

init_signal_driver (UM-525), signal_force (UM-530), signal_release (UM-532)

Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

Example

```
library ieee, modelsim_lib;
use ieee.std_logic_1164.all
use modelsim_lib.util.all;
entity top is
end;

architecture only of top is
  signal top_sig1 : std_logic;
begin
  ...
  spy_process : process
  begin
    init_signal_spy("/top/uut/inst1/sig1","/top_sig1",1);
    wait;
  end process spy_process;
  ...
end;
```

In this example, the value of */top/uut/inst1/sig1* will be mirrored onto */top_sig1*.

signal_force

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_force works the same as the **force** command (CR-176) with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal_release, a force or release command, or a subsequent signal_force is issued. Signal_force can be called concurrently or sequentially in a process.

Syntax

```
signal_force( dest_object, value, rel_time, force_type, cancel_period,
verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	forcetype	Optional. Specifies the type of force that will be applied. The value must be one of the following; default, deposit, drive, or freeze. The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the **force** command (CR-176) for further details on force type.

Name	**Type**	**Description**
cancel_period	time	Optional. Cancels the signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1 ms. A negative value means that the force will not be cancelled.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related functions

init_signal_driver (UM-525), init_signal_spy (UM-528), signal_release (UM-532)

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
begin

  force_process : process
  begin
    signal_force("/testbench/uut/blk1/reset", "1", 0 ns, freeze, open, 1);
    signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms, 1);
    wait;
  end process force_process;

  ...

end;
```

The above example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 2 ms after the second signal_force call was executed.

If you want to skip parameters so that you can specify subsequent parameters, you need to use the keyword "open" as a placeholder for the skipped parameter(s). The first signal_force procedure illustrates this, where an "open" for the cancel_period parameter means that the default value of -1 ms is used.

signal_release

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_release works the same as the **noforce** command (CR-204). Signal_release can be called concurrently or sequentially in a process.

Syntax

```
signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related functions

init_signal_driver (UM-525), init_signal_spy (UM-528), signal_force (UM-530)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is

  signal release_flag : std_logic;

begin

  stim_design : process
  begin
    ...
    wait until release_flag = '1';
    signal_release("/testbench/dut/blk1/data", 1);
    signal_release("/testbench/dut/blk1/clk", 1);
    ...
  end process stim_design;

  ...

end;
```

The above example releases any forces on the signals data and *clk* when the signal *release_flag* is a "1". Both calls will send a message to the transcript stating which signal was released and when.

$init_signal_driver

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_driver system task drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the $init_signal_driver value in the resolution of the signal.

Call only once

The $init_signal_driver system task creates a persistent relationship between the source and destination signals. Hence, you need to call $init_signal_driver only once for a particular pair of signals. Once $init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all $init_signal_driver calls in a Verilog initial block. See the example below.

Syntax

```
$init_signal_driver(src_object, dest_object, delay, delay_type, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
delay	integer, real, or time	Optional. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
delay_type	integer	Optional. Specifies the type of delay that will be applied. The value must be either 0 (inertial) or 1 (transport). The default is 0.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related procedures

$init_signal_spy (UM-537), $signal_force (UM-539), $signal_release (UM-541)

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to 1 (transport), the setting will be ignored, and the delay type will be inertial.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.
- Verilog memories (arrays of registers) are not supported.

Example

```
`timescale 1 ps / 1 ps

module testbench;

reg clk0;

initial begin
  clk0 = 1;
  forever begin
   #20 clk0 = ~clk0;
  end
end

initial begin
   $init_signal_driver("clk0", "/testbench/uut/blk1/clk", , , 1);
   $init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100, 1);
end

  ...

endmodule
```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps. For the second call to work, the *.../blk2/clk* must be a VHDL based signal, because if it were a Verilog net a 100 ps inertial delay would consume the 40 ps clock period. Verilog nets are limited to only inertial delays and thus the setting of 1 (transport delay) would be ignored.

$init_signal_spy

The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_spy system task only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value set by $init_signal_spy.

Call only once

The $init_signal_spy system task creates a persistent relationship between the source and the destination signal. Hence, you need to call $init_signal_spy only once for a particular pair of signals. Once $init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation. Thus, we recommend that you place all $init_signal_spy calls in a Verilog initial block. See the example below.

Syntax

```
$init_signal_spy(src_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a Verilog register or VHDL signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object's value is mirrored onto the dest_object. Default is 0, no message.

Related tasks

$init_signal_driver (UM-534), $signal_force (UM-539), $signal_release (UM-541)

Limitations

- When mirroring the value of a VHDL signal onto a Verilog register, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

Example

```
module testbench;
...
reg top_sig1;
...
initial
  begin
    $init_signal_spy("/top/uut/inst1/sig1","/top_sig1", 1);
  end
...
endmodule
```

In this example, the value of */top/uut/inst1/sig1* will be mirrored onto */top_sig1*.

$signal_force

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_force works the same as the **force** command (CR-176) with the exception that you cannot issue a repeating force. The force will remain on the signal until a $signal_release, a force or release command, or a subsequent $signal_force is issued. $signal_force can be called concurrently or sequentially in a process.

Syntax

```
$signal_force( dest_object, value, rel_time, force_type, cancel_period,
verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	integer, real, or time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	integer	Optional. Specifies the type of force that will be applied. The value must be one of the following; 0 (default), 1 (deposit), 2 (drive), or 3 (freeze). The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the **force** command (CR-176) for further details on force type.

Name	Type	Description
cancel_period	integer, real, time	Optional. Cancels the $signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related functions

$init_signal_driver (UM-534), $init_signal_spy (UM-537), $signal_release (UM-541)

Limitations

- You cannot force bits or slices of a register; you can force only the entire register.
- Verilog memories (arrays of registers) are not supported.

Example

```
`timescale 1 ns / 1 ns

module testbench;

initial
  begin
   $signal_force("/testbench/uut/blk1/reset", "1", 0, 3, , 1);
   $signal_force("/testbench/uut/blk1/reset", "0", 40, 3, 200000, 1);
  end

...

endmodule
```

The above example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 200000 ns after the second $signal_force call was executed.

$signal_release

The $signal_release() system task releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_release works the same as the **noforce** command (CR-204). $signal_release can be called concurrently or sequentially in a process.

Syntax

```
$signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related functions

$init_signal_driver (UM-534), $init_signal_spy (UM-537), $signal_force (UM-539)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

Example

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
  $signal_release("/testbench/dut/blk1/data", 1);
  $signal_release("/testbench/dut/blk1/clk", 1);
end

...

endmodule
```

The above example releases any forces on the signals *data* and *clk* when the register *release_flag* transitions to a "1". Both calls will send a message to the transcript stating which signal was released and when.

17 - Standard Delay Format (SDF) Timing Annotation

Chapter contents

This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator. ASIC and FPGA vendors usually provide tools that create SDF files for use with their cell libraries. Refer to your vendor's documentation for details on creating SDF files for your library. Many vendors also provide instructions on using their SDF files and libraries with ModelSim.

The SDF specification was originally created for Verilog designs, but it has also been adopted for VHDL VITAL designs. In general, the designer does not need to be familiar with the details of the SDF specification because the cell library provider has already supplied tools that create SDF files that match their libraries.

▶ **Note:** ModelSim will read SDF files that were compressed using gzip. Other compression formats (e.g., Unix zip) are not supported.

Specifying SDF files for simulation

ModelSim supports SDF versions 1.0 through 3.0. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** (CR-357) command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>
-sdftyp [<instance>=]<filename>
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

Instance specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

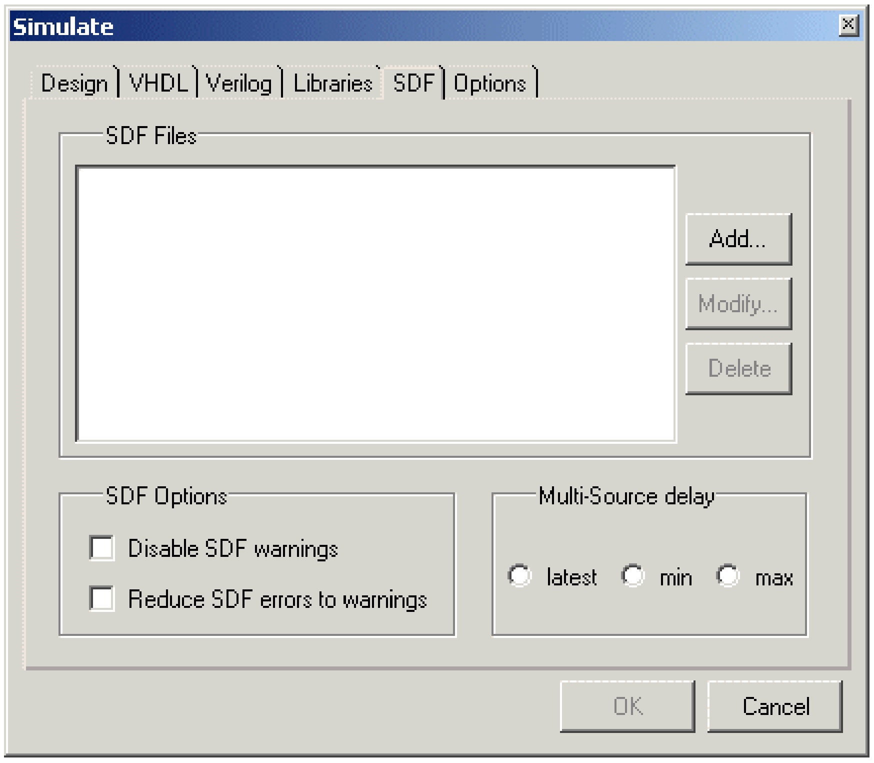
```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

SDF specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Simulate** dialog box under the SDF tab.



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Simulate** (Main window). See the GUI chapter for a description of this dialog.

For Verilog designs, you can also specify SDF files by using the **$sdf_annotate** system task. See "The $sdf_annotate system task" (UM-548) for more details.

Errors and warnings

Errors issued by the SDF annotator while loading the design prevent the simulation from continuing, whereas warnings do not. Use the **-sdfnoerror** option with **vsim** (CR-357) to change SDF errors to warnings so that the simulation can continue. Warning messages can be suppressed by using **vsim** with either the **-sdfnowarn** or **+nosdfwarn** options.

Another option is to use the **SDF** tab from the **Simulate** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn +nosdfwarn) to disable warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See "Troubleshooting" (UM-556) for more information on errors and warnings and how to avoid them.

VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see "VITAL specification and source code" (UM-91).

SDF to VHDL generic matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

SDF construct	Matching VHDL generic name
(IOPATH a y (3))	tpd_a_y
(IOPATH (posedge clk) q (1) (2))	tpd_clk_q_posedge
(INTERCONNECT u1/y u2/a (5))	tipd_a
(SETUP d (posedge clk) (5))	tsetup_d_clk_noedge_posedge
(HOLD (negedge d) (posedge clk) (5))	thold_d_clk_negedge_posedge
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk
(WIDTH (COND (reset==1’b0) clk) (5))	tpw_clk_reset_eq_0

Resolving errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
** Error (vsim-SDF-3240) myasic.sdf(18):
Instance '/testbench/dut/u1' does not have a generic named 'tpd_a_y'
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files provided by the cell library vendor.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don't match the names expected by the annotator, then there are several possibilities:

- The vendor's tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor's library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke **vsim** (CR-357) with the **-vital2.2b** option:

  ```
  vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
  ```

For more information on resolving errors see "Troubleshooting" (UM-556).

Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the **$sdf_annotate** system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The **$sdf_annotate** task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

The $sdf_annotate system task

The syntax for **$sdf_annotate** is:

Syntax

```
$sdf_annotate
  (["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"],
  ["<mtm_spec>"], ["<scale_factor>"], ["<scale_type>"]);
```

Arguments

`"<sdffile>"`
String that specifies the SDF file. Required.

`<instance>`
Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the $sdf_annotate call is made.

`"<config_file>"`
String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

`"<log_file>"`
String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

`"<mtm_spec>"`
String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is "tool_control". The "tool_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).

`"<scale_factor>"`
String that specifies delay scaling factors. Optional. The format is "<min_mult>:<typ_mult>:<max_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.

`"<scale_type>"`
String that overrides the **<mtm_spec>** delay selection. Optional. The **<mtm_spec>** delay selection is always used to select the delay scaling factor, but if a **<scale_type>** is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the **<mtm_spec>** value.

Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

SDF to Verilog construct matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows:

IOPATH is matched to specify path delays or primitives:

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

The IOPATH construct usually annotates path delays. If the module contains no path delays, then all primitives that drive the specified output port are annotated.

INTERCONNECT and **PORT** are matched to input ports:

SDF	Verilog
(INTERCONNECT u1.y u2.a (5))	input a;
(PORT u2.a (5))	inout a;

Both of these constructs identify a module input or inout port and create an internal net that is a delayed version of the port. This is called a Module Input Port Delay (MIPD). All primitives, specify path delays, and specify timing checks connected to the original port are reconnected to the new MIPD net.

PATHPULSE and **GLOBALPATHPULSE** are matched to specify path delays:

SDF	Verilog
(PATHPULSE a y (5) (10))	(a => y) = 0;
(GLOBALPATHPULSE a y (30) (60))	(a => y) = 0;

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

DEVICE is matched to primitives or specify path delays:

SDF	Verilog
(DEVICE y (5))	and u1(y, a, b);
(DEVICE y (5))	(a => y) = 0; (b => y) = 0;

If the SDF cell instance is a primitive instance, then that primitive's delay is annotated. If it is a module instance, then all specify path delays are annotated that drive the output port specified in the DEVICE construct (all path delays are annotated if the output port is omitted). If the module contains no path delays, then all primitives that drive the specified output port are annotated (or all primitives that drive any output port if the output port is omitted).

SETUP is matched to $setup and $setuphold:

SDF	Verilog
(SETUP d (posedge clk) (5))	$setup(d, posedge clk, 0);
(SETUP d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

HOLD is matched to $hold and $setuphold:

SDF	Verilog
(HOLD d (posedge clk) (5))	$hold(posedge clk, d, 0);
(HOLD d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

SETUPHOLD is matched to $setup, $hold, and $setuphold:

SDF	Verilog
(SETUPHOLD d (posedge clk) (5) (5))	$setup(d, posedge clk, 0);
(SETUPHOLD d (posedge clk) (5) (5))	$hold(posedge clk, d, 0);
(SETUPHOLD d (posedge clk) (5) (5))	$setuphold(posedge clk, d, 0, 0);

RECOVERY is matched to $recovery:

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	$recovery(negedge reset, posedge clk, 0);

REMOVAL is matched to $removal:

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	$removal(negedge reset, posedge clk, 0);

RECREM is matched to $recovery, $removal, and $recrem:

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$recrem(negedge reset, posedge clk, 0);

SKEW is matched to $skew:

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	$skew(posedge clk1, posedge clk2, 0);

WIDTH is matched to $width:

SDF	Verilog
(WIDTH (posedge clk) (5))	$width(posedge clk, 0);

PERIOD is matched to $period:

SDF	Verilog
(PERIOD (posedge clk) (5))	$period(posedge clk, 0);

NOCHANGE is matched to $nochange:

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	$nochange(negedge write, addr, 0, 0);

Optional edge specifications

Timing check ports and path delay input ports can have optional edge specifications. The annotator uses the following rules to match edges:

- A match occurs if the SDF port does not have an edge.
- A match occurs if the specify port does not have an edge.
- A match occurs if the SDF port edge is identical to the specify port edge.
- A match occurs if explicit edge transitions in the specify port edge overlap with the SDF port edge.

These rules allow SDF annotation to take place even if there is a difference between the number of edge-specific constructs in the SDF file and the Verilog specify block. For example, the Verilog specify block may contain separate setup timing checks for a falling and rising edge on data with respect to clock, while the SDF file may contain only a single setup check for both edges:

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(posedge data, posedge clk, 0);
(SETUP data (posedge clock) (5))	$setup(negedge data, posedge clk, 0);

In this case, the cell accommodates more accurate data than can be supplied by the tool that created the SDF file, and both timing checks correctly receive the same value.

Likewise, the SDF file may contain more accurate data than the model can accommodate.

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	$setup(data, posedge clk, 0);

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers can also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(data, edge[01, 0x] clk, 0);

The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port.

Optional conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

SDF	Verilog
(SETUP data (COND (reset!=1) (posedge clock)) (5))	$setup(data, posedge clk &&& (reset==0), 0);

The conditions are semantically equivalent and a match occurs. In contrast, path delay conditions may be complicated and semantically equivalent conditions may not match. For example,

SDF	Verilog
(COND (r1 || r2) (IOPATH clk q (5)))	if (r1 || r2) (clk => q) = 5; // matches
(COND (r1 || r2) (IOPATH clk q (5)))	if (r2 || r1) (clk => q) = 5; // does not match

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

Rounded timing values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF TIMESCALE is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps (from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

SDF for mixed VHDL and Verilog designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells can be annotated from the same SDF file. This flexibility is available only by using the simulator's SDF command-line options. The Verilog $sdf_annotate system task can annotate Verilog cells only. See the **vsim** command (CR-357) for more information on SDF command-line options.

Interconnect delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the **vsim** command for more information on the relevant command-line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Disabling timing checks

ModelSim offers a number of options for disabling timing checks on a "global" or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the *ModelSim Command Reference* for more details.

Command and argument	Effect
tcheck_set (CR-267)	modifies reporting or X generation status on one or more timing checks
tcheck_status (CR-269)	prints to the Transcript the current status of one or more timing checks
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	disables Verilog and VITAL timing checks for all instances in the specified design

Troubleshooting

Specifying the wrong instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See "Instance specification" (UM-544) for an example.

A common example for both VHDL and Verilog testbenches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL testbench

```
entity testbench is end;

architecture only of testbench is
    component myasic
    end component;
begin
    dut : myasic;
end;
```

Verilog testbench

```
module testbench;
    myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you can leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, open the structure window, navigate to the model instance, select it, and enter the **environment** command (CR-166). This command displays the instance name that should be used in the SDF command-line option.

Mistaking a component or module name for an instance label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/myasic’.
```

Forgetting to specify the instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

```
vsim -sdfmax myasic.sdf testbench
```

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u1’

** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u2’

** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u3’

** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u4’

** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u5’

** Warning (vsim-SDF-3432) myasic.sdf:
This file is probably applied to the wrong instance.

** Warning (vsim-SDF-3432) myasic.sdf:
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:
Failed to find any of the 358 instances from this file.

** Warning (vsim-SDF-3442) myasic.sdf:
Try instance ’/testbench/dut’. It contains all instance paths from this
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see "Resolving errors" (UM-547) for specific VHDL VITAL SDF troubleshooting.

18 - Value Change Dump (VCD) Files

Chapter contents

This chapter describes how to use VCD files in ModelSim. The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes. VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides command equivalents for these system tasks and extends VCD support to VHDL designs. The ModelSim commands can be used on VHDL, Verilog, or mixed designs.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

Creating a VCD file

There are two flows in ModelSim for creating a VCD file. One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information; the other produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

Flow for four-state VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name with the **vcd file** command (CR-294) and add items to the file with the **vcd add** command (CR-284):

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be a VCD file in the working directory.

Flow for extended VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name and items to add with the **vcd dumpports** command (CR-287):

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be an extended VCD file in the working directory.

Case sensitivity

VHDL is not case sensitive so ModelSim converts all signal names to lower case when it produces a VCD file. Conversely, Verilog designs are case sensitive so ModelSim maintains case when it produces a VCD file.

Checkpoint/restore and writing VCD files

ModelSim versions 5.7d and later support checkpoint/restore while reading or writing a VCD file. If a checkpoint occurs while ModelSim is writing a VCD file, the entire VCD file is copied into the checkpoint file. Since VCD files can be very large, it is possible that disk space problems may occur. Consequently, ModelSim issues a warning in this situation.

Using extended VCD as stimulus

You can use an extended VCD file as stimulus to re-simulate your design. There are two ways to do this: 1) simulate the top level of a design unit with the input values from an extended VCD file; and 2) specify one or more instances in a design to be replaced with the output values from the associated VCD file.

Simulating with input values from a VCD file

When simulating with inputs from an extended VCD file, you can simulate only one design unit at a time. In other words, you can apply the VCD file inputs only to the top level of the design unit for which you captured port data.

The general procedure includes two steps:

1. Create a VCD file for a single design unit using the **vcd dumpports** command (CR-287).

2. Resimulate the single design unit using the **-vcdstim** argument to **vsim** (CR-357). Note that **-vcdstim** works only with VCD files that were created by a ModelSim simulation.

Example 1 — Verilog counter

First, create the VCD file for the single instance using **vcd dumpports**:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim counter.vcd counter
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 2 — VHDL adder

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench2
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 3 — Mixed-HDL design

First, create three VCD files, one for each module:

```
% cd ~/modeltech/examples/mixedHDL
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

Replacing instances with output values from a VCD file

Replacing instances with output values from a VCD file lets you simulate without the instance's source or even the compiled object. The general procedure includes two steps:

1. Create VCD files for one or more instances in your design using the **vcd dumpports** command (CR-287). If necessary, use the **-vcdstim** switch to handle port order problems (see below).

2. Re-simulate your design using the **-vcdstim <instance>=<filename>** argument to **vsim** (CR-357). Note that this works only with VCD files that were created by a ModelSim simulation.

Example

In the following example, the three instances */top/p*, */top/c*, and */top/m* are replaced in simulation by the output values found in the corresponding VCD files.

First, create VCD files for all instances you want to replace:

```
vcd dumpports -vcdstim -file proc.vcd /top/p/*
vcd dumpports -vcdstim -file cache.vcd /top/c/*
vcd dumpports -vcdstim -file memory.vcd /top/m/*
run 1000
```

Next, simulate your design and map the instances to the VCD files you created:

```
vsim top -vcdstim /top/p=proc.vcd -vcdstim /top/c=cache.vcd
-vcdstim /top/m=memory.vcd
```

Port order issues

The **-vcdstim** argument to the **vcd dumpports** command ensures the order that port names appear in the VCD file matches the order that they are declared in the instance's module or entity declaration. Consider the following module declaration:

```
module proc(clk, addr, data, rw, strb, rdy);
  input  clk, rdy;
  output addr, rw, strb;
  inout  data;
```

The order of the ports in the module line (`clk, addr, data, ...`) does not match the order of those ports in the input, output, and inout lines (`clk, rdy, addr, ...`). In this case the **-vcdstim** argument to the **vcd dumpports** command needs to be used.

In cases where the order is the same, you do not need to use the **-vcdstim** argument to **vcd dumpports**. Also, module declarations of the form:

```
module proc(input clk, output addr, inout data, ...)
```

do not require use of the argument.

ModelSim VCD commands and VCD tasks

ModelSim VCD commands map to IEEE Std 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below maps the VCD commands to their associated tasks.

VCD commands	VCD system tasks
vcd add (CR-284)	$dumpvars
vcd checkpoint (CR-285)	$dumpall
vcd file (CR-294)▲	$dumpfile
vcd flush (CR-298)	$dumpflush
vcd limit (CR-299)	$dumplimit
vcd off (CR-300)	$dumpoff
vcd on (CR-301)	$dumpon

ModelSim versions 5.5 and later also support extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

VCD dumpports commands	VCD system tasks
vcd dumpports (CR-287)	$dumpports
vcd dumpportsall (CR-289)	$dumpportsall
vcd dumpportsflush (CR-290)	$dumpportsflush
vcd dumpportslimit (CR-291)	$dumpportslimit
vcd dumpportsoff (CR-292)	$dumpportsoff
vcd dumpportson (CR-293)	$dumpportson

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as $dumpfile, $dumpvar, etc. The difference is that $fdumpfile can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

VCD commands	VCD system tasks
vcd add (CR-284) `-file <filename>`	$fdumpvars
vcd checkpoint (CR-285) `<filename>`	$fdumpall
vcd files (CR-296) `<filename>`▲	$fdumpfile
vcd flush (CR-298) `<filename>`	$fdumpflush

VCD commands	VCD system tasks
vcd limit (CR-299) `<filename>`	$fdumplimit
vcd off (CR-300) `<filename>`	$fdumpoff
vcd on (CR-301) `<filename>`	$fdumpon

▲ **Important:** Note that two commands (**vcd file** and **vcd files**) are available to specify a filename and state mapping for a VCD file. **Vcd file** allows for only one VCD file and exists for backwards compatibility with ModelSim versions prior to 5.5. **Vcd files** allows for creation of multiple VCD files and is the preferred command to use in ModelSim versions 5.5 and later.

Compressing files with VCD tasks

ModelSim can produce compressed VCD files using the **gzip** compression algorithm. Since we cannot change the syntax of the system tasks, we act on the extension of the output file name. If you specify a *.gz* extension on the filename, ModelSim will compress the output.

A VCD file from source to output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL source code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
    port (CLK, RESET, data_in   : IN STD_LOGIC;
        Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
    process (CLK,RESET)
    begin
        if (RESET = '1') then
            Q <= (others => '0') ;
        elsif (CLK'event and CLK = '1') then
            Q <= Q(Q'left - 1 downto 0) & data_in ;
        end if ;
    end process ;
end ;
```

VCD simulator commands

At simulator time zero, the designer executes the following commands:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
quit -sim
```

VCD output

The VCD file created as a result of the preceding scenario would be called *output.vcd.* The following pages show how it would look.

VCD output

```
$date
    Thu Sep 18 11:07:43 2003
$end
$version
    ModelSim Version 5.8
$end
$timescale
    1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ' q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumpoff
x!
x"
x#
x$
x%
x&
x'
x(
x)
x*
x+
x,
$end
#300
$dumpon
1!
0"
1#
0$
0%
0&
0'
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1'
#850
0!
#900
1!
1&
#950
0!
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
```

```
0!
1"
0,
0+
0*
0)
0(
0'
0&
0%
0$
#1200
1!
$dumpall
1!
1"
1#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
```

Capturing port driver data

Some ASIC vendors' toolkits read a VCD file format that provides details on port drivers. This information can be used, for example, to drive a tester. See the ASIC vendor's documentation for toolkit specific information.

In ModelSim use the **vcd dumpports** command (CR-287) to create a VCD file that captures port driver data.

Port driver direction information is captured as TSSI states in the VCD file. Each time an external or internal port driver changes values, a new value change is recorded in the VCD file with the following format:

```
p<TSSI state> <0 strength> <1 strength> <identifier_code>
```

Supported TSSI states

The supported <TSSI states> are:

Input (testfixture)	Output (dut)
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state
d low (two or more drivers active)	l low (two or more drivers active)
u high (two or more drivers active)	h high (two or more drivers active)

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
F three-state (input and output unconnected)
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
B unknown (input driving high and output driving low)
b unknown (input driving high and output driving unknown)
C unknown (input driving unknown and output driving low)
c unknown (input driving unknown and output driving high)

Unknown direction
f unknown (input and output three-stated)

Strength values

The <strength> values are based on Verilog strengths:

Strength	**VHDL std_logic mappings**
0 highz	’Z’
1 small	
2 medium	
3 weak	
4 large	
5 pull	’W’,’H’,’L’
6 strong	’U’,’X’,’0’,’1’,’-’
7 supply	

Port identifier code

The <identifier_code> is an integer preceded by < that starts at zero and is incremented for each port in the order the ports are specified. Also, the variable type recorded in the VCD header is "port".

Example VCD output from vcd dumpports

The following is an example VCD file created with the **vcd dumpports** command.

```
$comment
    File created using the following command:
        vcd file myvcdfile.vcd -dumpports
$end
$date
    Thu Sep 18 07:35:58 2003
$end
$version
    dumpports ModelSim Version 5.8
$end
$timescale
    1ns
$end
$scope module test_counter $end
$scope module dut $end
$var port 1 <0 count [7] $end
$var port 1 <1 count [6] $end
$var port 1 <2 count [5] $end
$var port 1 <3 count [4] $end
$var port 1 <4 count [3] $end
$var port 1 <5 count [2] $end
$var port 1 <6 count [1] $end
$var port 1 <7 count [0] $end
$var port 1 <8 clk $end
$var port 1 <9 reset $end
$upscope $end
$upscope $end
$enddefinitions $end
#0
$dumpports
pX 6 6 <7
pX 6 6 <6
pX 6 6 <5
pX 6 6 <4
pX 6 6 <3
pX 6 6 <2
pX 6 6 <1
pX 6 6 <0
pD 6 0 <9
pD 6 0 <8
$end
#5
pU 0 6 <9
#8
pL 6 0 <7
pL 6 0 <6
pL 6 0 <5
pL 6 0 <4
pL 6 0 <3
pL 6 0 <2
pL 6 0 <1
pL 6 0 <0
#9
pD 6 0 <9
#10
pU 0 6 <8
#12
```

```
pH 0 6 <7
#20
pD 6 0 <8
#30
pU 0 6 <8
#32
pL 6 0 <7
pH 0 6 <6
#40
pD 6 0 <8
#50
pU 0 6 <8
#52
pH 0 6 <7
#60
pD 6 0 <8
#70
pU 0 6 <8
#72
pL 6 0 <7
pL 6 0 <6
pH 0 6 <5
#80
pD 6 0 <8
#90
pU 0 6 <8
#92
pH 0 6 <7
#100
pD 6 0 <8
$vcdclose
#100
$end
```

19 - Logic Modeling SmartModels

Chapter contents

The Logic Modeling SWIFT-based SmartModel library can be used with ModelSim VHDL and Verilog. The SmartModel library is a collection of behavioral models supplied in binary form with a procedural interface that is accessed by the simulator. This chapter describes how to use the SmartModel library with ModelSim.

The SmartModel library must be obtained from Logic Modeling along with the documentation that describes how to use it. This chapter only describes the specifics of using the library with ModelSim.

A 32-bit SmartModel will not run with a 64-bit version of SE. When trying to load the operating system specific 32-bit library into the 64-bit executable, the pointer sizes will be incorrect.

VHDL SmartModel interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

The ModelSim software locates the SmartModel interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **sm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
libsm = $MODEL_TECH/libsm.sl
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software (Windows
NT)
; libsm = $MODEL_TECH/libsm.dll
;  Logic Modeling's SmartModel SWIFT software (HP 9000 Series 700)
; libswift = $LMC_HOME/lib/hp700.lib/libswift.sl
;  Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
;  Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
;  Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
;  Logic Modeling's SmartModel SWIFT software (Linux)
; libswift = $LMC_HOME/lib/x86_linux.lib/libswift.so
```

The **libsm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the SmartModel software. The **libswift** entry points to the Logic Modeling dynamic link library software that accesses the SmartModels. The simulator automatically loads both the **libsm** and **libswift** libraries when it elaborates a SmartModel foreign architecture.

By default, the **libsm** entry points to the *libsm.sl* supplied in the ModelSim installation directory indicated by the **MODEL_TECH** environment variable. ModelSim automatically sets the **MODEL_TECH** environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libsm** entry (precede the line with the ";" character) and uncomment the **libsm** entry for the Windows operating system.

Uncomment the appropriate **libswift** entry for your operating system. The **LMC_HOME** environment variable must be set to the root of the SmartModel library installation directory. Consult Logic Modeling's documentation for details.

Creating foreign architectures with sm_entity

The ModelSim **sm_entity** tool automatically creates entities and foreign architectures for SmartModels. Its usage is as follows:

Syntax

```
sm_entity
  [-] [-xe] [-xa] [-c] [-all] [-v] [-93] [<SmartModelName>...]
```

Arguments

`-`
Read SmartModel names from standard input.

`-xe`
Do not generate entity declarations.

`-xa`
Do not generate architecture bodies.

`-c`
Generate component declarations.

`-all`
Select all models installed in the SmartModel library.

`-v`
Display progress messages.

`-93`
Use extended identifiers where needed.

`<SmartModelName>`
Name of a SmartModel (see the SmartModel library documentation for details on SmartModel names).

By default, the **sm_entity** tool writes an entity and foreign architecture to stdout for each SmartModel name listed on the command line. Optionally, you can include the component declaration (**-c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

The simplest way to prepare SmartModels for use with ModelSim VHDL is to generate the entities and foreign architectures for all installed SmartModels, and compile them into a library named **lmc**. This is easily accomplished with the following commands:

```
% sm_entity -all > sml.vhd
% vlib lmc
% vcom -work lmc sml.vhd
```

To instantiate the SmartModels in your VHDL design, you also need to generate component declarations for the SmartModels. Add these component declarations to a package named **sml** (for example), and compile the package into the **lmc** library:

```
% sm_entity -all -c -xe -xa > smlcomp.vhd
```

Edit the resulting *smlcomp.vhd* file to turn it into a package of SmartModel component declarations as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package sml is
    <component declarations go here>
end sml;
```

Compile the package into the **lmc** library:

```
% vcom -work lmc smlcomp.vhd
```

The SmartModels can now be referenced in your design by adding the following **library** and **use** clauses to your code:

```
library lmc;
use lmc.sml.all;
```

The following is an example of an entity and foreign architecture created by **sm_entity** for the cy7c285 SmartModel.

```
library ieee;
use ieee.std_logic_1164.all;

entity cy7c285 is
    generic (TimingVersion : STRING := "CY7C285-65";
        DelayRange : STRING := "Max";
        MemoryFile : STRING := "memory" );
    port ( A0 : in std_logic;
        A1 : in std_logic;
        A2 : in std_logic;
        A3 : in std_logic;
        A4 : in std_logic;
        A5 : in std_logic;
        A6 : in std_logic;
        A7 : in std_logic;
        A8 : in std_logic;
        A9 : in std_logic;
        A10 : in std_logic;
        A11 : in std_logic;
        A12 : in std_logic;
        A13 : in std_logic;
        A14 : in std_logic;
        A15 : in std_logic;
        CS : in std_logic;
        O0 : out std_logic;
        O1 : out std_logic;
        O2 : out std_logic;
        O3 : out std_logic;
        O4 : out std_logic;
        O5 : out std_logic;
        O6 : out std_logic;
        O7 : out std_logic;
        WAIT_PORT : inout std_logic );
end;

architecture SmartModel of cy7c285 is
    attribute FOREIGN : STRING;
    attribute FOREIGN of SmartModel : architecture is
        "sm_init $MODEL_TECH/libsm.sl ; cy7c285";
begin
end SmartModel;
```

Entity details

- The entity name is the SmartModel name (you can manually change this name if you like).
- The port names are the same as the SmartModel port names *(these names must not be changed*). If the SmartModel port name is not a valid VHDL identifier, then **sm_entity** automatically converts it to a valid name. If **sm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. If the **-93** option had been specified in the example above, then *WAIT* would have been converted to \WAIT\\. Note that in this example the port *WAIT* was converted to *WAIT_PORT* because **wait** is a VHDL reserved word.
- The port types are **std_logic**. This data type supports the full range of SmartModel logic states.
- The *DelayRange*, *TimingVersion*, and *MemoryFil*e generics represent the SmartModel attributes of the same name. Consult your SmartModel library documentation for a description of these attributes (and others). **Sm_entity** creates a generic for each attribute of the particular SmartModel. The default generic value is the default attribute value that the SmartModel has supplied to **sm_entity**.

Architecture details

- The first part of the foreign attribute string (sm_init) is the same for all SmartModels.
- The second part (*$MODEL_TECH/libsm.sl*) is taken from the **libsm** entry in the initialization file, *modelsim.ini*.
- The third part (cy7c285) is the SmartModel name. This name correlates the architecture with the SmartModel at elaboration.

Vector ports

The entities generated by **sm_entity** only contain single-bit ports, never vectored ports. This is necessary because ModelSim correlates entity ports with the SmartModel SWIFT interface by name. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```
component cy7c285
    generic ( TimingVersion : STRING := "CY7C285-65";
        DelayRange : STRING := "Max";
        MemoryFile : STRING := "memory" );
    port ( A : in std_logic_vector (15 downto 0);
        CS : in std_logic;
        O : out std_logic_vector (7 downto 0);
        WAIT_PORT : inout std_logic );
end component;


for all: cy7c285
    use entity work.cy7c285
    port map (A0 => A(0),
        A1 => A(1),
        A2 => A(2),
        A3 => A(3),
        A4 => A(4),
        A5 => A(5),
        A6 => A(6),
        A7 => A(7),
        A8 => A(8),
        A9 => A(9),
        A10 => A(10),
        A11 => A(11),
        A12 => A(12),
        A13 => A(13),
        A14 => A(14),
        A15 => A(15),
        CS => CS,
        O0 => O(0),
        O1 => O(1),
        O2 => O(2),
        O3 => O(3),
        O4 => O(4),
        O5 => O(5),
        O6 => O(6),
        O7 => O(7),
        WAIT_PORT => WAIT_PORT );
```

Command channel

The command channel is a SmartModel feature that lets you invoke SmartModel specific commands. These commands are documented in the SmartModel library documentation from Synopsys. ModelSim provides access to the Command Channel from the command line. The form of a SmartModel command is:

```
lmc <instance_name>|-all "<SmartModel command>"
```

The **instance_name** argument is either a full hierarchical name or a relative name of a SmartModel instance. A relative name is relative to the current environment setting (see **environment** command (CR-166)). For example, to turn timing checks off for SmartModel */top/u1*:

```
lmc /top/u1 "SetConstraints Off"
```

Use **-all** to apply the command to all SmartModel instances. For example, to turn timing checks off for all SmartModel instances:

```
lmc -all "SetConstraints Off"
```

There are also some SmartModel commands that apply globally to the current simulation session rather than to models. The form of a SmartModel session command is:

```
lmcsession "<SmartModel session command>"
```

SmartModel Windows

Some models in the SmartModel library provide access to internal registers with a feature called SmartModel Windows. Refer to Logic Modeling's SmartModel library documentation (available on Synopsys' web site) for details on this feature. The simulator interface to this feature is described below.

Window names that are not valid VHDL or Verilog identifiers are converted to VHDL extended identifiers. For example, with a window named z1I10.GSR.OR, ModelSim will treat the name as \z1I10.GSR.OR\ (for all commands including lmcwin, add wave, and examine). You must then use that name in all commands. For example,

```
add wave /top/swift_model/\z1I10.GSR.OR\
```

Extended identifiers are case sensitive.

ReportStatus

The **ReportStatus** command displays model information, including the names of window registers. For example,

```
lmc /top/u1 ReportStatus
```

SmartModel Windows description:

```
WA "Read-Only (Read Only)"
WB "1-bit"
WC "64-bit"
```

This model contains window registers named *wa*, *wb*, and *wc*. These names can be used in subsequent window (**lmcwin**) commands.

SmartModel lmcwin commands

The following window commands are supported:

- **lmcwin read** <window_instance> [-<radix>]
- **lmcwin write** <window_instance> <value>
- **lmcwin enable** <window_instance>
- **lmcwin disable** <window_instance>
- **lmcwin release** <window_instance>

Each command requires a window instance argument that identifies a specific model instance and window name. For example, */top/u1/wa* refers to window *wa* in model instance */top/u1*.

lmcwin read

The **lmcwin read** command displays the current value of a window. The optional radix argument is **-binary**, **-decimal**, or **-hexadecimal** (these names can be abbreviated). The default is to display the value using the **std_logic** characters. For example, the following command displays the 64-bit window *wc* in hexadecimal:

```
lmcwin read /top/u1/wc -h
```

lmcwin write

The **lmcwin write** command writes a value into a window. The format of the value argument is the same as used in other simulator commands that take value arguments. For example, to write 1 to window *wb*, and all 1's to window *wc*:

```
lmcwin write /top/u1/wb 1
lmcwin write /top/u1/wc X"FFFFFFFFFFFFFFFF"
```

lmcwin enable

The **lmcwin enable** command enables continuous monitoring of a window. The specified window is added to the model instance as a signal (with the same name as the window) of type **std_logic** or **std_logic_vector**. This signal's values can then be referenced in simulator commands that read signal values, such as the **add list** command (CR-55) shown below. The window signal is continuously updated to reflect the value in the model. For example, to list window *wa*:

```
lmcwin enable /top/u1/wa
add list /top/u1/wa
```

lmcwin disable

The **lmcwin disable** command disables continuous monitoring of a window. The window signal is not deleted, but it no longer is updated when the model's window register changes value. For example, to disable continuous monitoring of window *wa*:

```
lmcwin disable /top/u1/wa
```

lmcwin release

Some windows are actually nets, and the **lmcwin write** command behaves more like a continuous force on the net. The **lmcwin release** command disables the effect of a previous **lmcwin write** command on a window net.

Memory arrays

A memory model usually makes the entire register array available as a window. In this case, the window commands operate only on a single element at a time. The element is selected as an array reference in the window instance specification. For example, to read element 5 from the window memory *mem*:

```
lmcwin read /top/u2/mem(5)
```

Omitting the element specification defaults to element 0. Also, continuous monitoring is limited to a single array element. The associated window signal is updated with the most recently enabled element for continuous monitoring.

Verilog SmartModel interface

The SWIFT SmartModel library, beginning with release r40b, provides an optional library of Verilog modules and a PLI application that communicates between a simulator's PLI and the SWIFT simulator interface. The Logic Modeling documentation refers to this as the Logic Models to Verilog (LMTV) interface. To install this option, you must select the simulator type "Verilog" when you run Logic Modeling’s SmartInstall program.

Linking the LMTV interface to the simulator

Synopsys provides a dynamically loadable library that links ModelSim to the LMTV interface. See chapter 5, "Using MTI Verilog with Synopsys Models," in the "Simulator Configuration Guide for Synopsys Models" (available on Synopsys’ web site) for directions on how to link to this library.

20 - Logic Modeling hardware models

Chapter contents

Logic Modeling hardware models can be used with ModelSim VHDL and Verilog. A hardware model allows simulation of a device using the actual silicon installed as a hardware model in one of Logic Modeling's hardware modeling systems. The hardware modeling system is a network resource with a procedural interface that is accessed by the simulator. This chapter describes how to use Logic Modeling hardware models with ModelSim.

▶ **Note:** Please refer to Logic Modeling documentation from Synopsys for details on using the hardware modeler. This chapter only describes the specifics of using hardware models with ModelSim SE.

VHDL hardware model interface

ModelSim VHDL interfaces to a hardware model through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific hardware model with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the hardware modeler software and establishes communication with the specific hardware model.

The ModelSim software locates the hardware modeler interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **hm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
libhm = $MODEL_TECH/libhm.sl
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
(Windows NT)
; libhm = $MODEL_TECH/libhm.dll
;  Logic Modeling's hardware modeler SFI software (HP 9000 Series 700)
; libsfi = <sfi_dir>/lib/hp700/libsfi.sl
;  Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsfi = <sfi_dir>/lib/rs6000/libsfi.a
;  Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsfi = <sfi_dir>/lib/sun4.solaris/libsfi.so
;  Logic Modeling's hardware modeler SFI software (Window NT)
; libsfi = <sfi_dir>/lib/pcnt/lm_sfi.dll
;  Logic Modeling's hardware modeler SFI software (Linux)
; libsfi = <sfi_dir>/lib/linux/libsfi.so
```

The **libhm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the hardware modeler software. The **libsfi** entry points to the Logic Modeling dynamic link library software that accesses the hardware modeler. The simulator automatically loads both the **libhm** and **libsfi** libraries when it elaborates a hardware model foreign architecture.

By default, the **libhm** entry points to the *libhm.sl* supplied in the ModelSim installation directory indicated by the MODEL_TECH environment variable. ModelSim automatically sets the MODEL_TECH environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libhm** entry (precede the line with the ";" character) and uncomment the **libhm** entry for the Windows operating system.

Uncomment the appropriate **libsfi** entry for your operating system, and replace **<sfi_dir>** with the path to the hardware modeler software installation directory. In addition, you must set the **LM_LIB** and **LM_DIR** environment variables as described in Logic Modeling documentation from Synopsys.

Creating foreign architectures with hm_entity

The ModelSim **hm_entity** tool automatically creates entities and foreign architectures for hardware models. Its usage is as follows:

Syntax

```
hm_entity
  [-xe] [-xa] [-c] [-93] <shell software filename>
```

Arguments

`-xe`
Do not generate entity declarations.

`-xa`
Do not generate architecture bodies.

`-c`
Generate component declarations.

`-93`
Use extended identifiers where needed.

`<shell software filename>`
Hardware model shell software filename (see Logic Modeling documentation from Synopsys for details on shell software files)

By default, the **hm_entity** tool writes an entity and foreign architecture to stdout for the hardware model. Optionally, you can include the component declaration (**-c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

Once you have created the entity and foreign architecture, you must compile it into a library. For example, the following commands compile the entity and foreign architecture for a hardware model named **LMTEST**:

```
% hm_entity LMTEST.MDL > lmtest.vhd
% vlib lmc
% vcom -work lmc lmtest.vhd
```

To instantiate the hardware model in your VHDL design, you will also need to generate a component declaration. If you have multiple hardware models, you may want to add all of their component declarations to a package so that you can easily reference them in your design. The following command writes the component declaration to stdout for the **LMTEST** hardware model.

```
% hm_entity -c -xe -xa LMTEST.MDL
```

Paste the resulting component declaration into the appropriate place in your design or into a package.

The following is an example of the entity and foreign architecture created by **hm_entity** for the CY7C285 hardware model:

```
library ieee;
use ieee.std_logic_1164.all;

entity cy7c285 is
    generic ( DelayRange : STRING := "Max" );
    port ( A0 : in std_logic;
        A1 : in std_logic;
        A2 : in std_logic;
        A3 : in std_logic;
        A4 : in std_logic;
        A5 : in std_logic;
        A6 : in std_logic;
        A7 : in std_logic;
        A8 : in std_logic;
        A9 : in std_logic;
        A10 : in std_logic;
        A11 : in std_logic;
        A12 : in std_logic;
        A13 : in std_logic;
        A14 : in std_logic;
        A15 : in std_logic;
        CS : in std_logic;
        O0 : out std_logic;
        O1 : out std_logic;
        O2 : out std_logic;
        O3 : out std_logic;
        O4 : out std_logic;
        O5 : out std_logic;
        O6 : out std_logic;
        O7 : out std_logic;
        W : inout std_logic );
end;

architecture Hardware of cy7c285 is
    attribute FOREIGN : STRING;
    attribute FOREIGN of Hardware : architecture is
        "hm_init $MODEL_TECH/libhm.sl ; CY7C285.MDL";
begin
end Hardware;
```

Entity details

- The entity name is the hardware model name (you can manually change this name if you like).
- The port names are the same as the hardware model port names *(these names must not be changed*). If the hardware model port name is not a valid VHDL identifier, then **hm_entity** issues an error message. If **hm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. Another option is to create a pin-name mapping file. Consult the Logic Modeling documentation from Synopsys for details.
- The port types are **std_logic**. This data type supports the full range of hardware model logic states.
- The *DelayRange* generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".

Architecture details

- The first part of the foreign attribute string (hm_init) is the same for all hardware models.
- The second part (*$MODEL_TECH/libhm.sl*) is taken from the **libhm** entry in the initialization file, *modelsim.ini*.
- The third part (CY7C285.MDL) is the shell software filename. This name correlates the architecture with the hardware model at elaboration.

Vector ports

The entities generated by **hm_entity** only contain single-bit ports, never vectored ports. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 hardware model:

```
component cy7c285
    generic ( DelayRange : STRING := "Max");
    port ( A : in std_logic_vector (15 downto 0);
        CS : in std_logic;
        O : out std_logic_vector (7 downto 0);
        WAIT_PORT : inout std_logic );
end component;


for all: cy7c285
    use entity work.cy7c285
    port map (A0 => A(0),
        A1 => A(1),
        A2 => A(2),
        A3 => A(3),
        A4 => A(4),
        A5 => A(5),
        A6 => A(6),
        A7 => A(7),
        A8 => A(8),
        A9 => A(9),
        A10 => A(10),
        A11 => A(11),
        A12 => A(12),
        A13 => A(13),
        A14 => A(14),
        A15 => A(15),
        CS => CS,
        O0 => O(0),
        O1 => O(1),
        O2 => O(2),
        O3 => O(3),
        O4 => O(4),
        O5 => O(5),
        O6 => O(6),
        O7 => O(7),
        WAIT_PORT => W );
```

Hardware model commands

The following simulator commands are available for hardware models. Refer to the Logic Modeling documentation from Synopsys for details on these operations.

lm_vectors on|off <instance_name> [<filename>]

Enable/disable test vector logging for the specified hardware model.

lm_measure_timing on|off <instance_name> [<filename>]

Enable/disable timing measurement for the specified hardware model.

lm_timing_checks on|off <instance_name>

Enable/disable timing checks for the specified hardware model.

lm_loop_patterns on|off <instance_name>

Enable/disable pattern looping for the specified hardware model.

lm_unknowns on|off <instance_name>

Enable/disable unknown propagation for the specified hardware model.

21 - Tcl and macros (DO files)

Chapter contents

This chapter provides an overview of Tcl (tool command language) as used with ModelSim. Macros in ModelSim are simply Tcl scripts that contain ModelSim and, optionally, Tcl commands.

Tcl is a scripting language for controlling and extending ModelSim. Within ModelSim you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts on the fly without stopping to recompile or restart ModelSim. In addition, if ModelSim does not provide the command you need, you can use Tcl to create your own commands.

Tcl features within ModelSim

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

Tcl References

Two books about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk by* Brent Welch published by Prentice Hall. You can also consult the following online references:

- Select **Help > Tcl Man Pages** (Main window).
- The Model Technology web site lists a variety of Tcl resources: www.model.com/resources/tcltk.asp

Tcl commands

For complete information on Tcl commands, select **Help > Tcl Man Pages** (Main window). Also see "Preference variables located in Tcl files" (UM-631) for information on Tcl variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Previous ModelSim command	Command changed to (or replaced by)
continue	**run** (CR-246) with the **-continue** option
format list | wave	**write format** (CR-389) with either list or wave specified
if	replaced by the Tcl **if** command, see "if command syntax" (UM-596) for more information
list	**add list** (CR-55)
nolist | nowave	**delete** (CR-151) with either list or wave specified
set	replaced by the Tcl **set** command, see "set command syntax" (UM-597) for more information
source	**vsource** (CR-374)
wave	**add wave** (CR-64)

Tcl command syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on if command syntax (UM-596) and set command syntax (UM-597) follow.

1 A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.

2 A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

3 Words of a command are separated by white space (except for newlines, which are command separators).

4 If the first character of a word is a double-quote (""") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

5 If the first character of a word is an open brace ("{") then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

6 If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

7 If a word contains a dollar-sign ("$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

```
$name
```

Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

```
$name(index)
```

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

```
${name}
```

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

8 If a backslash ("\") appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

`\a`	Audible alert (bell) (0x7).
`\b`	Backspace (0x8).
`\f`	Form feed (0xc).
`\n`	Newline (0xa).
`\r`	Carriage-return (0xd).
`\t`	Tab (0x9).
`\v`	Vertical tab (0xb).
`\<newline>whiteSpace`	A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.
`\\`	Backslash ("\").
`\`*`ooo`*	The digits ooo (one, two, or three of them) give the octal value of the character.

`\x`*`hh`*	The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

9 If a hash character ("#") appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

10 Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

11 Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

if command syntax

The Tcl **if** command executes scripts conditionally. Note that in the syntax below the "?" indicates an optional argument.

Syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything else is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

set command syntax

The Tcl **set** command reads and writes variables. Note that in the syntax below the "?" indicates an optional argument.

Syntax

```
set varName ?value?
```

Description

Returns the value of variable *varName*. If *value* is specified, then sets the value of *varName* to value, creating a new variable if one doesn't already exist, and returns its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the global command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare varName to be a namespace variable.

Command substitution

Placing a command in square brackets [ ] will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

Command separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

Multiple-line commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace '{' is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if { [exa sig_a] == "0011ZZ"}  {
    echo "Signal value matches"
    do macro_1.do
} else {
    echo "Signal value fails"
    do macro_2.do
}
```

Evaluation order

An important thing to remember when using Tcl is that anything put in curly brackets {} is not evaluated immediately. This is important for if-then-else statements, procedures, loops, and so forth.

Tcl relational expression evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

  ```
  if {[exa var_1] == 345}...
  ```

  The following will also work:

  ```
  if {[exa var_1] == "345"}...
  ```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

  ```
  if {[exa var_2] == 001Z}...
  ```

  will give an error.

  ```
  if {[exa var_2] == "001Z"}...
  ```

  will work okay.

- Don't quote single characters in single quotes:

  ```
  if {[exa var_3] == 'X'}...
  ```

  will give an error

  ```
  if {[exa var_3] == "X"}...
  ```

  will work okay.

- For the equal operator, you must use the C operator "==". For not-equal, you must use the C operator "!=".

Variable substitution

When a $<var_name> is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

▶ **Note:** Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See "Simulator state variables" (UM-634) for more information about ModelSim-defined variables.

System commands

To pass commands to the UNIX shell or DOS window, use the Tcl **exec** command:

```
echo The date is [exec date]
```

List processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, etc. to list var_name
lindex list_name index	returns the index-th element of list_name; the first element is 0
linsert list_name index val1 val2 ...	inserts val1, val2, etc. just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, etc.
llength list_name	returns the number of elements in list_name
lrange list_name first last	returns a sublist of list_name, from index first to index last; first or last may be "end", which refers to the last element in the list
lreplace list_name first last val1, val2, ...	replaces elements first through last with val1, val2, etc.

Two other commands, **lsearch** and **lsort,** are also available for list manipulation. See the Tcl man pages (**Help > Tcl Man Pages**) for more information on these commands.

See also the ModelSim Tcl command: **lecho** (CR-184)

ModelSim Tcl commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided here; for more information and command syntax see the *ModelSim Command Reference*.

Command	Description
alias (CR-68)	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find (CR-172)	locates incrTcl classes and objects
lecho (CR-184)	takes one or more Tcl lists as arguments and pretty-prints them to the Main window
lshift (CR-189)	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist (CR-190)	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv (CR-220)	echoes to the Main window the current names and values of all environment variables

ModelSim Tcl time commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

Conversions

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Command	Description
eqTime <time> <time>	evaluates for equal
neqTime <time> <time>	evaluates for not equal
gtTime <time> <time>	evaluates for greater than
gteTime <time> <time>	evaluates for greater than or equal
ltTime <time> <time>	evaluates for less than
lteTime <time> <time>	evaluates for less than or equal

All relation operations return 1 or 0 for true or false respectively and are suitable return values for TCL conditional expressions. For example,

```
if {[eqTime $Now 1750ns]} {
    ...
}
```

Arithmetic

Command	Description
addTime <time> <time>	add time
divTime <time> <time>	64-bit integer divide
mulTime <time> <time>	64-bit integer multiply
subTime <time> <time>	subtract time

Tcl examples

This is an example of using the Tcl **while** loop to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

```
set b [list]
set i [expr {[llength $a] - 1}]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

This example uses the Tcl **for** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

```
set b [list]
for {set i [expr {[llength $a] - 1}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

This example uses the Tcl **foreach** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way (the foreach command iterates over all of the elements of a list):

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

This example shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

This example is a list reversal that skips a particular element by using the Tcl **continue** command:

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

The next example works in UNIX only. In a Windows environment, the Tcl **exec** command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

(in VHDL source):

```
    signal datime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in macro):

```
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [clock format [clock seconds]]
    force -deposit datime $s
    if {do_the_echo} {
        echo "New time is [examine -value datime]"
    }
}

bp src/waveadd.vhd 133 {set_date; continue}
    --sets the breakpoint to call set_date
```

This next example shows a complete Tcl script that restores multiple Wave windows to their state in a previous simulation, including signals listed, geometry, and screen position. It also adds buttons to the Main window toolbar to ease management of the wave files.

```
##  This file contains procedures to manage multiple wave files.
##  Source this file from the command line or as a startup script.
##  source <path>/wave_mgr.tcl

##  add_wave_buttons
##      Add wave management buttons to the main toolbar (new, save and load)

##  new_wave
##      Dialog box creates a new wave window with the user provided name

##  named_wave <name>
##      Creates a new wave window with the specified title

##  save_wave <file-root>
##      Saves name, window location and contents for all open windows

##  wave windows
##      Creates <file-root><n>.do file for each window where <n> is 1
##      to the number of windows. Default file-root is "wave". Also
##       creates windowSet.do file that contains title and geometry info.

##  load_wave <file-root>
##      Opens and loads wave windows for all files matching <file-root><n>.do
##      where <n> are the numbers from 1-9. Default <file-root> is "wave".
##       Also runs windowSet.do file if it exists.

## Add wave management buttons to the main toolbar

proc add_wave_buttons {} {
_add_menu main controls right SystemMenu SystemWindowFrame {Load Waves} \
load_wave
_add_menu main controls right SystemMenu SystemWindowFrame {Save Waves} \
save_wave
_add_menu main controls right SystemMenu SystemWindowFrame {New Wave} \
new_wave
}
## Simple Dialog requests name of new wave window. Defaults to Wave<n>

proc new_wave {} {
    global vsimPriv
    set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
    set windowName [GetValue . "Create Named Wave Window:" $defaultName ]
    if {$windowName == ""} {
        # Dialog canceled
        # abort operation
        return
    }
    ## Debug
    puts "Window name: $windowName\n"
    if {$windowName == "{}"} {
        set windowName ""
    }
    if {$windowName != ""} {
        named_wave $windowName
    } else {
        named_wave $defaultName
    }
}

## Creates a new wave window with the provided name (defaults to "Wave")

proc named_wave {{name "Wave"}} {
    set newWave [view -new wave]
    if {[string length $name] > 0} {
        wm title $newWave $name
    }
}

## Writes out format of all wave windows, stores geometry and title info in
## windowSet.do file. Removes any extra files with the same fileroot.
## Default file name is wave<n> starting from 1.

proc save_wave {{fileroot "wave"}} {
    global vsimPriv
    set n 1
    if {[catch {open windowSet_$fileroot.do w 755} fileId]} {
        error "Open failure for $fileroot ($fileId)"
    }
    foreach w $vsimPriv(WaveWindows) {
        echo "Saving: [wm title $w]"
        set filename $fileroot$n.do
        if {[file exists $filename]} {
            # Use different file
            set n2 0
            while {[file exists ${fileroot}${n}${n2}.do]} {
                incr n2
            }
            set filename ${fileroot}${n}${n2}.do
        }
        write format wave -window $w $filename
        puts $fileId "wm title $w \"[wm title $w]\""
        puts $fileId "wm geometry $w [wm geometry $w]"
        puts $fileId "mtiGrid_colconfig $w.grid name -width \
            [mtiGrid_colcget $w.grid name -width]"
        puts $fileId "mtiGrid_colconfig $w.grid value -width \
            [mtiGrid_colcget $w.grid value -width]"
        flush $fileId
        incr n
    }

    foreach f [lsort [glob -nocomplain $fileroot\[$n-9\].do]] {
            echo "Removing: $f"
            exec rm $f
        }
    }
}

## Provide file root argument and load_wave restores all saved windows.
## Default file root is "wave".

proc load_wave {{fileroot "wave"}} {
    foreach f [lsort [glob -nocomplain $fileroot\[1-9\].do]] {
        echo "Loading: $f"
        view -new wave
        do $f
    }
    if {[file exists windowSet_$fileroot.do]} {
        do windowSet_$fileroot.do
    }
}

...
```

This next example specifies the compiler arguments and lets you compile any number of files.

```
set Files [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    set lappend Files $1
    shift
}
eval vcom -93 -explicit -noaccel $Files
```

This example is an enhanced version of the last one. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a *.vhd* file extension.

```
set vhdFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    if {[string match *.vhd $1]} {
        lappend vhdFiles $1
    } else {
        lappend vFiles $1
    }
    shift
}
if {[llength $vhdFiles] > 0} {
    eval vcom -93 -explicit -noaccel $vhdFiles
}
if {[llength $vFiles] > 0} {
    eval vlog $vFiles
}
```

Macros (DO files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools > Execute Macro** (Main window) menu selection or the **do** command (CR-156).

Creating DO files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the Main window transcript as a DO file (see "Saving the Main window transcript file" (UM-264)).

The following is a simple DO file that was saved from the Main window transcript. It is used in the dataset exercise in the ModelSim Tutorial. This DO file adds several signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

```
add wave ld
add wave rst
add wave clk
add wave d
add wave q
force -freeze clk 0 0, 1 {50 ns} -r 100
force rst 1
force rst 0 10
force ld 0
force d 1010
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters $1 through $9 in the macro file. For example say the macro "*testfile*" contains the line **bp** $1 $2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do  testfile  design.vhd  127
```

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the **shift** command (CR-259) to see the other parameters.

Making macro parameters optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the argc (UM-634) simulator state variable. The **argc** simulator state variable returns the number of parameters passed. The examples below show several ways of using **argc**.

Example 1

This macro specifies the files to compile and handles 0-2 compiler arguments as parameters. If you supply more arguments, ModelSim generates a message.

```
switch $argc {
  0 {vcom file1.vhd file2.vhd file3.vhd }
  1 {vcom $1 file1.vhd file2.vhd file3.vhd }
  2 {vcom $1 $2 file1.vhd file2.vhd file3.vhd }
  default {echo Too many arguments. The macro accepts 0-2 args.  }
}
```

Example 2

This macro specifies the compiler arguments and lets you compile any number of files.

```
variable Files ""
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set Files [concat $Files $1]
  shift
}
eval vcom -93 -explicit -noaccel $Files
```

Example 3

This macro is an enhanced version of the one shown in example 2. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a .vhd file extension.

```
variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist   0
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    set vhdFiles [concat $vhdFiles $1]
    set vhdFilesExist 1
  } else {
    set vFiles [concat $vFiles $1]
    set vFilesExist 1
  }
  shift
}
if {$vhdFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
  eval vlog -fast -forcecode $vFiles
}
```

Useful commands for handling breakpoints and errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The following commands may be useful for handling such events. (Any other legal command may be executed as well.)

command	result
run (CR-246) **-continue**	continue as if the breakpoint had not been executed, completes the **run** (CR-246) that was interrupted
resume (CR-243)	continue running the macro
onbreak (CR-210)	specify a command to run when you hit a breakpoint within a macro
onElabError (CR-211)	specify a command to run when an error is encountered during elaboration
onerror (CR-212)	specify a command to run when an error is encountered within a macro
status (CR-263)	get a traceback of nested macro calls when a macro is interrupted
abort (CR-51)	terminate a macro once the macro has been interrupted or paused
pause (CR-213)	cause the macro to be interrupted; the macro can be resumed by entering a **resume** command (CR-243) via the command line
transcript (CR-278)	control echoing of macro commands to the Main window transcript

Note: You can also set the OnErrorDefaultAction Tcl variable (see "Preference variables located in Tcl files" (UM-631)) in the *pref.tcl* file to dictate what action ModelSim takes when an error occurs.

Error action in DO files

If a command in a macro returns an error, ModelSim does the following:

1 If an **onerror** (CR-212) command has been set in the macro script, ModelSim executes that command.

2 If no **onerror** command has been specified in the script, ModelSim checks the **OnErrorDefaultAction** Tcl variable. If the variable is defined, it's action will be invoked.

3 If neither 1 or 2 is true, the macro aborts.

Using the Tcl source command with DO files

Either the **do** command or Tcl **source** command can execute a DO file, but they behave differently.

With the **source** command, the DO file is executed exactly as if the commands in it were typed in by hand at the prompt. Each time a breakpoint is hit the Source window is updated to show the breakpoint. This behavior could be inconvenient with a large DO file containing many breakpoints.

When a **do** command is interrupted by an error or breakpoint, it does not update any windows, and keeps the DO file "locked". This keeps the Source window from flashing, scrolling, and moving the arrow when a complex DO file is executed. Typically an **onbreak resume** command is used to keep the macro running as it hits breakpoints. Add an **onbreak abort** command to the DO file if you want to exit the macro and update the Source window.

A - ModelSim variables

Appendix contents

This appendix documents the following types of ModelSim variables:

- **environment variables**
  Variables referenced and set according to operating system conventions. Environment variables prepare the ModelSim environment prior to simulation.
- **ModelSim preference variables**
  Variables used to control compiler or simulator functions and modify the appearance of the ModelSim GUI.
- **simulator state variable**s
  Variables that provide feedback on the state of the current simulation.

Variable settings report

The **report** command (CR-238) returns a list of current settings for either the simulator state, or simulator control variables. Use the following commands at either the ModelSim or VSIM prompt:

```
report simulator state
report simulator control
```

The simulator control variables reported by the **report simulator control** command can be set interactively using the Tcl **set** command (UM-597).

Personal preferences

There are several preferences stored by ModelSim on a personal basis, independent of *modelsim.ini* or *modelsim.tcl* files. These preferences are stored in $(HOME)/.modelsim on UNIX and in the Windows Registry under HKEY_CURRENT_USER\Software\Model Technology Incorporated\ModelSim. Among these preferences are:

- **mti_ask_LBViewTypes**, **mti_ask_LBViewPath**, **mti_ask_LBViewLoadable**
  Settings for the **Customize Library View** dialog. Determine the view of the Library tab in the Main window workspace.
- **mti_pane_cnt**, **mti_pane_size**, **pane_#**, **pane_percent**
  Determine layout of various panes in the Main window.
- **open_workspace**
  Setting for whether or not to display the Main window workspace.
- **pinit**
  Project Initialization state (one of: Welcome | OpenLast | NoWelcome). This determines whether the Welcome To ModelSim dialog box appears when you invoke the tool.
- **project_history**
  Project History.
- **printersetup**
  All setup parameters related to Printing (i.e., current printer, etc.).
- **transcriptpercent**
  The size of the Main window transcript pane. Expressed as a percentage of the width of the Main window.

The HKEY_CURRENT_USER key is unique for each user Login on Windows NT.

Returning to the original ModelSim defaults

If you would like to return ModelSim's interface to its original state, simply rename or delete the existing *modelsim.tcl* and *modelsim.ini* files. ModelSim will use *pref.tcl* for GUI preferences and make a copy of *<install_dir>/modeltech/modelsim.ini* to use the next time ModelSim is invoked without an existing project (if you start a new project the new MPF file will use the settings in the new *modelsim.ini* file).

Environment variables

Before compiling or simulating, several environment variables may be set to provide the functions described in the table below. The variables are in the *autoexec.bat* file on Windows 98/Me machines, and set through the System control panel on NT/2000/XP machines. For UNIX, the variables are typically found in the *.login* script. The LM_LICENSE_FILE variable is required; all others are optional.

Variable	Description
DOPATH	used by ModelSim to search for DO files (macros); consists of a colon-separated (semi-colon for Windows) list of paths to directories; this environment variable can be overridden by the DOPATH Tcl preference variable The DOPATH environment variable isn't accessible when you invoke **vsim** from a Unix shell or from a Windows command prompt. It is accessible once ModelSim or **vsim** is invoked. If you need to invoke from a shell or command line and use the DOPATH environment variable, use the following syntax: `vsim -do "do <dofile_name>" <design_unit>`
EDITOR	specifies the editor to invoke with the **edit** command (CR-162)
HOME	used by ModelSim to look for an optional graphical preference file and optional location map file; see: "Preference variables located in INI files" (UM-617)
LM_LICENSE_FILE	used by the ModelSim license file manager to find the location of the license file; may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files; REQUIRED
MODEL_TECH	set by all ModelSim tools to the directory in which the binary executable resides; **DO NOT SET THIS VARIABLE!**
MODEL_TECH_TCL	used by ModelSim to find Tcl libraries for Tcl/Tk 8.3 and vsim; may also be used to specify a startup DO file; defaults to */modeltech/../tcl*; may be set to an alternate path
MGC_LOCATION_MAP	used by ModelSim tools to find source files based on easily reallocated "soft" paths; optional; see the Tcl variables: **SourceDir** and **SourceMap**

Variable	**Description**
MODELSIM	used by all ModelSim tools to find the *modelsim.ini* file; consists of a path including the file name. An alternative use of this variable is to set it to the path of a project file (*<Project_Root_Dir>/<Project_Name>.mpf*). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace *modelsim.ini* as the initialization file for all ModelSim tools.
MODELSIM_TCL	used by ModelSim to look for an optional graphical preference file; can be a colon-separated (UNIX) or semi-colon separated (Windows) list of file paths
MTI_COSIM_TRACE	creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.
MTI_TF_LIMIT	limits the size of the VSOUT temp file (generated by the ModelSim kernel); the value of the variable is the size of k-bytes; TMPDIR (below) controls the location of this file, STDOUT controls the name; default = 10, 0 = no limit; does *not* control the size of the transcript file
MTI_USELIB_DIR	specifies the directory into which object libraries are compiled when using the **-compile_uselibs** argument to the **vlog** command (CR-345)
NOMMAP	if set to 1, disables memory mapping in ModelSim; this should be used only when running on Linux 7.1; it will decrease the speed with which ModelSim reads files
PLIOBJS	used by ModelSim to search for PLI object files for loading; consists of a space-separated list of file or path names
STDOUT	the VSOUT temp file (generated by the simulator kernel) is deleted when the simulator exits; the file is not deleted if you specify a filename for VSOUT with STDOUT; specifying a name and location (use TMPDIR) for the VSOUT file will also help you locate and delete the file in event of a crash (an unnamed VSOUT file is not deleted after a crash either)
TMPDIR (Unix) TMP (Windows)	specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel

Creating environment variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

Using Windows 98/Me

Open and edit the *autoexec.bat* file by adding this line:

```
set MY_PATH=\temp\work
```

Restart Windows to initialize the new variable.

Using Windows NT/2000/XP

Right-click the **My Computer** icon and select **Properties**, then select the **Environment** tab (in Windows 2000/XP select the Advanced tab and then Environment Variables). Add the new variable with this data—Variable:*MY_PATH* and Value:*\temp\work*.

Click **Set** and **Apply** to initialize the variable.

Library mapping with environment variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command (CR-356) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from a DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you're using **vmap** from the ModelSim/VSIM prompt type:

```
vmap MY_VITAL \$MY_PATH
```

If you used DOS **vmap**, this line will be added to the *modelsim.ini*:

```
MY_VITAL = c:\temp\work
```

If **vmap** is used from the ModelSim/VSIM prompt, the *modelsim.ini* file will be modified with this line:

```
 MY_VITAL = $MY_PATH
```

You can easily add additional hierarchy to the path. For example,

```
vmap MORE_VITAL %MY_PATH%\more_path\and_more_path
vmap MORE_VITAL \$MY_PATH\more_path\and_more_path
```

The "$" character in the examples above is Tcl syntax that precedes a variable. The "\" character is an escape character that keeps the variable from being evaluated during the execution of **vmap**.

Referencing environment variables within ModelSim

There are two ways to reference environment variables within ModelSim. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
use std.textio.all;
entity test is end;
architecture only of test is
begin
    process
        FILE in_file : text is in "$ENV_VAR_NAME";
    begin
        wait;
    end process;
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

Note: Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

Removing temp files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the ModelSim GUI. In normal circumstances the file is deleted when the simulator exits. If ModelSim crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

Preference variables located in INI files

ModelSim initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings. The default initialization file is *modelsim.ini* and is located in your install directory.

To set these variables, edit the initialization file directly with any text editor. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

The following tables list the variables by section, and in order of their appearance within the INI file:

INI file sections
[Library] library path variables (UM-617)
[vlog] Verilog compiler control variables (UM-618)
[vcom] VHDL compiler control variables (UM-619)
[sccom] SystemC compiler control variables (UM-620)
[vsim] simulator control variables (UM-621)
[lmc] Logic Modeling variables (UM-627)

[Library] library path variables

Variable name	Value range	Purpose
ieee	any valid path; may include environment variables	sets the path to the library containing IEEE and Synopsys arithmetic packages; the default is $MODEL_TECH/../ieee
modelsim_lib	any valid path; may include environment variables	sets the path to the library containing Model Technology VHDL utilities such as Signal Spy; the default is $MODEL_TECH/../modelsim_lib
std	any valid path; may include environment variables	sets the path to the VHDL STD library; the default is $MODEL_TECH/../std
std_developerskit	any valid path; may include environment variables	sets the path to the libraries for MGC standard developer's kit; the default is $MODEL_TECH/../std_developerskit
synopsys	any valid path; may include environment variables	sets the path to the accelerated arithmetic packages; the default is $MODEL_TECH/../ synopsys

Variable name	Value range	Purpose
verilog	any valid path; may include environment variables	sets the path to the library containing VHDL/ Verilog type mappings; the default is $MODEL_TECH/../verilog
vital2000	any valid path; may include environment variables	sets the path to the VITAL 2000 library; the default is $MODEL_TECH/../vital2000
others	any valid path; may include environment variables	points to another *modelsim.ini* file whose library path variables will also be read; the pathname must include "modelsim.ini"; only one others variable can be specified in any *modelsim.ini* file.

[vlog] Verilog compiler control variables

Variable name	Value range	Purpose	Default
Hazard	0, 1	if 1, turns on Verilog hazard checking (order-dependent accessing of global variables)	off (0)
Incremental	0, 1	if 1, turns on incremental compilation of modules	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
Protect	0, 1	if 1, enables `protect directive processing; see "ModelSim compiler directives" (UM-152) for details	off (0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
Show_Lint	0, 1	if 1, turns on lint-style checking	off (0)
ScalarOpts	0, 1	if 1, activates optimizations on expressions that don’t involve signals, waits, or function/procedure/task invocations	off (0)
Show_BadOptionWarning	0, 1	if 1, generates a warning whenever an unknown plus argument is encountered	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
vlog95compat	0, 1	if 1, disables System Verilog and Verilog 2001 support and makes compiler compatible with IEEE Std 1364-1995	off (0)
UpCase	0, 1	if 1, turns on converting regular Verilog identifiers to uppercase. Allows case insensitivity for module names; see also "Verilog-XL compatible compiler arguments" (UM-113)	off (0)

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
CheckSynthesis	0, 1	if 1, turns on limited synthesis rule compliance checking; checks only signals used (read) by a process; also, understands only combinational logic, not clocked logic	off (0)
Explicit	0, 1	if 1, turns on resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration)	on (1)
IgnoreVitalErrors	0, 1	if 1, ignores VITAL compliance checking errors	off (0)
NoCaseStaticError	0, 1	if 1, changes case statement static errors to warnings	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
NoIndexCheck	0, 1	if 1, run time index checks are disabled	off (0)
NoOthersStaticError	0, 1	if 1, disables errors caused by aggregates that are not locally static	off (0)
NoRangeCheck	0, 1	if 1, disables run time range checking	off (0)
NoVital	0, 1	if 1, turns off acceleration of the VITAL packages	off (0)
NoVitalCheck	0, 1	if 1, turns off VITAL compliance checking	off (0)
Optimize_1164	0, 1	if 0, turns off optimization for the IEEE std_logic_1164 package	on (1)
PedanticErrors	0, 1	if 1, overrides NoCaseStaticError and NoOthersStaticError	off(0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
RequireConfigForAllDefault Binding	0, 1	if 1, instructs the compiler not to generate a default binding during compilation	off (0)
ScalarOpts	0, 1	if 1, activates optimizations on expressions that don't involve signals, waits, or function/procedure/ task invocations	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
Show_VitalChecksWarnings	0, 1	if 0, turns off VITAL compliance-check warnings	on (1)
Show_Warning1	0, 1	if 0, turns off unbound-component warnings	on (1)
Show_Warning2	0, 1	if 0, turns off process-without-a-wait-statement warnings	on (1)

Variable name	Value range	Purpose	Default
Show_Warning3	0, 1	if 0, turns off null-range warnings	on (1)
Show_Warning4	0, 1	if 0, turns off no-space-in-time-literal warnings	on (1)
Show_Warning5	0, 1	if 0, turns off multiple-drivers-on-unresolved-signal warnings	on (1)
VHDL93	0, 1, 2	if 0, enables support for VHDL-1987; if 1, enables support for VHDL-1993; if 2, enables support for VHDL-2002	2

[sccom] SystemC compiler control variables

Variable name	Value range	Purpose	Default
NoNameBind	0, 1	if 1, disables name binding during compilation; see "Name association (binding)" (UM-204) and "`-nonamebind`" (CR-249) for details	off (0)
UseScv	0, 1	if 1, turns on use of SCV include files and library; see"`-scv`" (CR-249) for details	off (0)
SccomVerbose	0, 1	if 1, turns on verbose messages from **sccom** (CR-248) : see "`-verbose`" (CR-249) for details	off (0)
CppOptions	any valid C+++ compiler options	adds any specified C++ compiler options to the **sccom** command line at the time of invocation	none
CppPath	C++ compiler path	If used, variables should point directly to the location of the g++ executable, such as: `% CppPath /usr/bin/g++` This variable is not required when running SystemC designs. By default, you should install and use the built-in g++ compiler that comes with ModelSim	none
SccomLogfile	0, 1	if 1, creates a logfile for sccom	off (0)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
AssertionPassEnable	0, 1	turns on pass tracking for PSL assertions	off (0)
AssertionFailEnable	0, 1	turns on failure tracking for PSL assertions	on (1)
AssertionPassLimit	Any positive integer and -1	sets limit for the number of times ModelSim will respond to a PSL assertion pass event; after the limit is reached on a particular assertion, that assertion is disabled; use -1 for infinity	1
AssertionFailLimit	Any positive integer and -1	sets limit for the number of times ModelSim will respond to a PSL assertion failure event; after the limit is reached on a particular assertion, that assertion is disabled; use -1 for infinity	1
AssertionPassLog	0, 1	turns on transcript logging for PSL assertion pass events	on (1)
AssertionFailLog	0, 1	turns on transcript logging for PSL assertion failure events	on (1)
AssertionFailAction	0, 1, 2	sets action for a PSL failure event; use 0 for continue, 1 for break, 2 for exit	continue (0)
AssertFile	any valid filename	alternative file for storing VHDL or PSL assertion messages	transcript
AssertionFormat	see next column	defines format of VHDL assertion messages; fields include: %S - severity level %R - report message %T - time of assertion %D - delta %I - instance or region pathname (if available) %i - instance pathname with process %O - process name %K - kind of item path points to; returns Instance, Signal, Process, or Unknown %P - instance or region path without leaf process %F - file %L - line number of assertion, or if from subprogram, line from which call is made %% - print '%' character	"** %S: %R\n Time: %T Iteration: %D%I\n"

Variable name	Value range	Purpose	Default
AssertionFormatBreak	see AssertionFormat above	defines format of messages for VHDL assertions that trigger a breakpoint; see AssertionFormat for options	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatNote	see AssertionFormat above	defines format of messages for VHDL Note assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D%I\n"
AssertionFormatWarning	see AssertionFormat above	defines format of messages for VHDL Warning assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D%I\n"
AssertionFormatError	see AssertionFormat above	defines format of messages for VHDL Error assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatFail	see AssertionFormat above	defines format of messages for VHDL Fail assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatFatal	see AssertionFormat above	defines format of messages for VHDL Fatal assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	"** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
BreakOnAssertion	0-4	defines severity of VHDL assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal); this variable can be set interactively with the Tcl **set** command (UM-597)	3
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format; this variable can be set interactively with the Tcl **set** command (UM-597)	on (1)

Variable name	Value range	Purpose	Default
CommandHistory	any valid filename	sets the name of a file in which to store the Main window command history	commented out (;)
ConcurrentFileLimit	any positive integer	controls the number of VHDL files open concurrently; this number should be less than the current limit setting for max file descriptors; 0 = unlimited	40
DatasetSeparator	any character except those with special meaning (i.e., \, {, }, etc.)	the dataset separator for fully-rooted contexts, for example sim:/top; must not be the same character as PathSeparator	:
DefaultForceKind	freeze, drive, or deposit	defines the kind of force used when not otherwise specified; this variable can be set interactively with the Tcl **set** command (UM-597)	drive for resolved signals; freeze for unresolved signals
DefaultRadix	symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii	a numeric radix may be specified as a name or number (i.e., binary can be specified as binary or 2; octal as octal or 8; etc.); this variable can be set interactively with the Tcl **set** command (UM-597)	symbolic
DefaultRestartOptions	one or more of: -force, -noassertions, -nobreakpoint, -nolist, -nolog, -nowave	sets default behavior for the restart command	commented out (;)
DelayFileOpen	0, 1	if 1, open VHDL87 files on first read or write, else open files when elaborated; this variable can be set interactively with the Tcl **set** command (UM-597)	off (0)
GenerateFormat	Any non-quoted string containing at a minimum a %s followed by a %d	controls the format of a generate statement label (don't quote it)	%s__%d
IgnoreError	0,1	if 1, ignore assertion errors; this variable can be set interactively with the Tcl **set** command (UM-597)	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures; this variable can be set interactively with the Tcl **set** command (UM-597)	off (0)

Variable name	Value range	Purpose	Default
IgnoreNote	0,1	if 1, ignore assertion notes; this variable can be set interactively with the Tcl **set** command (UM-597)	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings; this variable can be set interactively with the Tcl **set** command (UM-597)	off (0)
IterationLimit	positive integer	limit on simulation kernel iterations allowed without advancing time; this variable can be set interactively with the Tcl **set** command (UM-597)	5000
License	any single <license_option>	if set, controls ModelSim license file search; license options include: nomgc - excludes MGC licenses nomti - excludes MTI licenses noqueue - do not wait in license queue if no licenses are available plus - only use PLUS license vlog - only use VLOG license vhdl - only use VHDL license viewsim - accepts a simulation license rather than being queued for a viewer license see also the **vsim** command (CR-357) <license_option>	search all licenses
NumericStdNoWarnings	0, 1	if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed; this variable can be set interactively with the Tcl **set** command (UM-597)	off (0)
PathSeparator	any character except those with special meaning (i.e., \, {, }, etc.)	used for hierarchical pathnames; must not be the same character as DatasetSeparator; this variable can be set interactively with the Tcl **set** command (UM-597)	/
Resolution	fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100	simulator resolution; no space between value and units (i.e., 10fs, not 10 fs); overridden by the -t argument to **vsim** (CR-357); if your delays get truncated, set the resolution smaller; this value must be less than or equal to the UserTimeUnit (described below)	ns

Variable name	Value range	Purpose	Default
RunLength	positive integer	default simulation length in units specified by the UserTimeUnit variable; this variable can be set interactively with the Tcl **set** command (UM-597)	100
Show3DMem	0, 1	controls whether or not arrays of 3 or more dimensions are listed as memories in the Memory window; this variable can be set with the Tcl **set** command (UM-597)	on (1)
ShowIntMem	0, 1	controls whether or not integer arrays are listed as memories in the Memory window; this variable can be set with the Tcl **set** command (UM-597)	on (1)
ShowEnumMem	0, 1	controls whether or not enumerated type arrays (other than std_logic-based arrays) are listed as memories in the Memory window; this variable can be set with the Tcl **set** command (UM-597)	on (1)
Startup	= do <DO filename>; any valid macro (do) file	specifies the ModelSim startup macro; see the **do** command (CR-156)	commented out (;)
StdArithNoWarnings	0, 1	if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed; this variable can be set interactively with the Tcl **set** command	off (0)
TranscriptFile	any valid filename	file for saving command transcript; environment variables may be included in the pathname	transcript
UnbufferedOutput	0, 1	controls VHDL and Verilog files open for write; 0 = Buffered, 1 = Unbuffered	0
UseCsupV2	0, 1	instructs **vsim** to use */usr/lib/libCsup_v2.sl* for shared object loading; for use only on HP-UX 11.00 when you have compiled FLI/PLI/VPI C++ code with aCC's -AA option	off (0)

Variable name	**Value range**	**Purpose**	**Default**
UserTimeUnit	fs, ps, ns, us, ms, sec, or default	specifies scaling for the Wave window and the default time units to use for commands such as **force** (CR-176) and **run** (CR-246); should generally be set to default, in which case it takes the value of the Resolution variable; this variable can be set interactively with the Tcl **set** command (UM-597)	default
Veriuser	one or more valid shared object names	list of dynamically loadable objects for Verilog PLI/VPI applications; see *Chapter 6 - Verilog PLI / VPI*	commented out (;)
WaveSignalNameWidth	0, positive integer	controls the number of visible hierarchical regions of a signal name shown in the Wave window (UM-337); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy	0
WLFCompress	0, 1	turns WLF file compression on (1) or off (0)	1
WLFDeleteOnQuit	0, 1	specifies whether a WLF file should be deleted when the simulation ends; if set to 0, the file is not deleted; if set to 1, the file is deleted	0
WLFOptimize	0, 1	specifies whether the viewing of waveforms is optimized; default is enabled; WLF files created prior to ModelSim version 5.8 cannot take advantage of the optimization	1
WLFSaveAllRegions	0, 1	specifies whether to save all design hierarchy in the WLF file (1) or only regions containing logged signals (0)	0
WLFSizeLimit	0 - positive integer of MB	WLF file size limit; limits WLF file by size (as closely as possible) to the specified number of megabytes; if both size and time limits are specified the most restrictive is used; setting to 0 results in no limit	0
WLFTimeLimit	0 - positive integer, time unit is optional	WLF file time limit; limits WLF file by time (as closely as possible) to the specified amount of time. If both time and size limits are specified the most restrictive is used; setting to 0 results in no limit	0

[lmc] Logic Modeling variables

Logic Modeling SmartModels and hardware modeler interface

ModelSim's interface with Logic Modeling's SmartModels and hardware modeler are specified in the **[lmc]** section of the INI/MPF file; for more information see "VHDL SmartModel interface" (UM-576) and "VHDL hardware model interface" (UM-586) respectively.

Reading variable values from the INI file

You can read values from the *modelsim.ini* file with the following function:

`GetPrivateProfileString <section> <key> <defaultValue>`
Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetPrivateProfileString vsim
CheckpointCompressMode 1]

set PrefMain(file) [GetPrivateProfileString vsim TranscriptFile ""]
```

Commonly used INI variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

Environment variables

You can use environment variables in your initialization files. Use a dollar sign ($) before the environment variable name. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

There is one environment variable, MODEL_TECH, that you cannot — and should not — set. MODEL_TECH is a special variable set by Model Technology software. Its value is the name of the directory from which the VCOM or VLOG compilers or VSIM simulator was invoked. MODEL_TECH is used by the other Model Technology tools to find the libraries.

Hierarchical library mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search only the library section of the initialization file specified by the "others" clause. For example:

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

Creating a transcript file

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

Using a startup file

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the **do** command (CR-156) for additional information on creating do files.

Turning off assertion messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Turning off warnings from arithmetic packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

These variables can also be set interactively using the Tcl **set** command (UM-597). This capability provides an answer to a common question about disabling warnings at time 0. You might enter commands like the following in a DO file or at the ModelSim prompt:

```
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

Alternatively, you could use the **when** command (CR-375) to accomplish the same thing:

```
when {$now = @1ns } {set NumericStdNoWarnings 1}
run -all
```

Note that the time unit (ns in this case) would vary depending on your simulation resolution.

Force command defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. This is designed to provide compatibility with version 4.1 and earlier force files. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

Restart command defaults

The **restart** command has **-force**, **-nobreakpoint**, **-nolist**, **-nolog**, and **-nowave** options. You can set any of these as defaults by entering the following line in the *modelsim.ini* file:

```
DefaultRestartOptions = <options>
```

where `<options>` can be one or more of -force, -nobreakpoint, -nolist, -nolog, and -nowave.

Example: `DefaultRestartOptions = -nolog -force`

Note: You can also set these defaults in the *modelsim.tcl* file. The Tcl file settings will override the .ini file settings.

VHDL standard

You can specify which version of the 1076 Std ModelSim follows by default using the VHDL93 variable:

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

Opening VHDL files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the **DelayFileOpen** option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

Preference variables located in Tcl files

ModelSim Tcl preference variables give you control over fonts, colors, prompts, window positions and other simulator window characteristics. Preference files, which contain Tcl commands that set preference variables, are loaded before any windows are created, and so affect all windows.

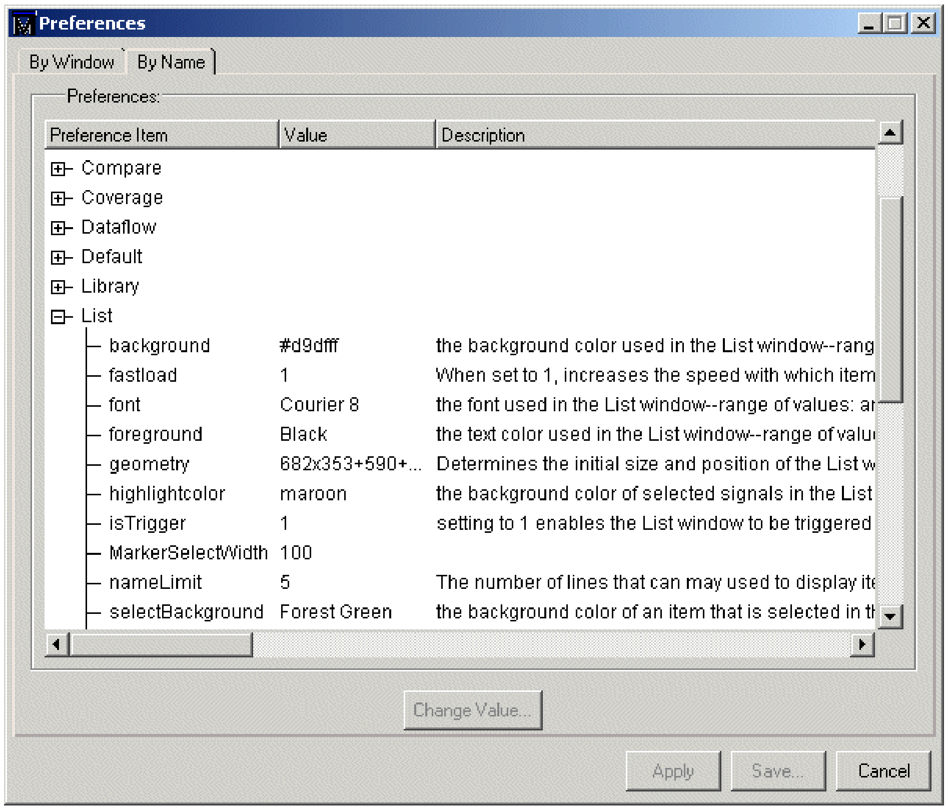
When you invoke ModelSim the first time, it loads default preferences from the *pref.tcl* file. You can customize the preference variables and save a file called *modelsim.tcl* file that ModelSim reads in lieu of *pref.tcl*. Once you have created a *modelsim.tcl* file, ModelSim attempts to load the file each time it starts up. ModelSim searches for the file as follows:

- use MODELSIM_TCL (UM-614) environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use $(HOME)/modelsim.tcl if it exists

Important: If your preference file is not named *modelsim.tcl*, or if the file is not located in the directories mentioned above, you must refer to it with the MODELSIM_TCL environment variable.

Setting variables from the GUI

Select **Tools > Edit Preferences** in the Main window to open the Preferences dialog box.



Preference Item	Value	Description
⊞ Compare		
⊞ Coverage		
⊞ Dataflow		
⊞ Default		
⊞ Library		
⊟ List		
background	#d9dfff	the background color used in the List window--rang
fastload	1	When set to 1, increases the speed with which item
font	Courier 8	the font used in the List window--range of values: ar
foreground	Black	the text color used in the List window--range of valu
geometry	682x353+590+...	Determines the initial size and position of the List w
highlightcolor	maroon	the background color of selected signals in the List
isTrigger	1	setting to 1 enables the List window to be triggered
MarkerSelectWidth	100	
nameLimit	5	The number of lines that can may used to display ite
selectBackground	Forest Green	the background color of an item that is selected in th

To change a setting, select the preference item and click **Change Value**. Click **Apply** to accept the settings for the current session only. Click **Save** to create a *modelsim.tcl* file with the current preference settings.

Setting variables from the command line

Use the Tcl **set** command (UM-597) to customize preference variables from the Main window command line:

```
set <variable name> <variable value>
```

This command establishes variable values for the current session only. To save the current preference settings to a *modelsim.tcl* file, use the **write preferences** command:

```
write preferences modelsim.tcl
```

User-defined variables

Temporary, user-defined variables can be created with the **set** command (UM-597). Like simulator variables, user-defined variables are preceded by a dollar sign when referenced. To create a variable with the **set** command:

```
set user1 7
```

You can use the variable in a command like:

```
echo "user1 = $user1"
```

More preferences

Additional compiler and simulator preferences may be set in the *modelsim.ini* file; see "Preference variables located in INI files" (UM-617).

Variable precedence

Note that some variables can be set in a .tcl file or a .ini file. A variable set in a .tcl file takes precedence over the same variable set in a .ini file. For example, assume you have the following line in your *modelsim.ini* file:

```
TranscriptFile = transcript
```

And assume you have the following line in your *modelsim.tcl* file:

```
set PrefMain(file) {}
```

In this case the setting in the *modelsim.tcl* file will override that in the *modelsim.ini* file, and a transcript file will not be produced.

Simulator state variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros). The variables are referenced in commands by prefixing the name with a dollar sign ($).

Variable	Result
argc	returns the total number of parameters passed to the current macro
architecture	returns the name of the top-level architecture currently being simulated; for an optimized Verilog module, returns architecture name; for a configuration or non-optimized Verilog module, this variable returns an empty string
configuration	returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration
delta	returns the number of the current simulator iteration
entity	returns the name of the top-level VHDL entity or Verilog module currently being simulated
library	returns the library name for the current region
MacroNestingLevel	returns the current depth of macro call nesting
n	represents a macro parameter, where n can be an integer in the range 1-9
Now	always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands
now	when time resolution is a unary unit (i.e., 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (i.e., 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands
resolution	returns the current simulation time resolution

Referencing simulator state variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign ($). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 ps 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \$now will not be interpreted as the current simulator time.

Special considerations for the now variable

For the **when** command (CR-375), special processing is performed on comparisons involving the **now** variable. If you specify "when {$now=100}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

```
if { [gtTime $now 2us] } {
.
.
.
```

See "ModelSim Tcl time commands" (UM-601) for details on 64-bit time operators.

B - ModelSim shortcuts

Appendix contents

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Command shortcuts

- You may abbreviate command syntax, but there’s a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work. For this reason ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.
- Multiple commands may be entered on one line if they are separated by semi-colons (;). For example:

  ```
  vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
  ```

  The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

  ```
  vsim -c -do "run 20 ; simstats ; quit -f" top
  ```

  You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

  ```
  vsim -do "run 20 ; echo [simstats]; quit -f" -c top
  ```

Command history shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the ModelSim/VSIM prompt:

Shortcut	Description
`!!`	repeats the last command
`!n`	repeats command number n; n is the VSIM prompt number (e.g., for this prompt: VSIM 12>, n =12)
`!abc`	repeats the most recent command starting with "abc"
`^xyz^ab^`	replaces "xyz" in the last command with "ab"
up and down arrows	scrolls through the command history with the keyboard arrows
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
`his or history`	shows the last few commands (up to 50 are kept)

Main and Source window mouse and keyboard shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Mouse - UNIX	Mouse - Windows	Result
< left-button - click >		move the insertion cursor
< left-button - press > + drag		select
< shift - left-button - press >		extend selection
< left-button - double-click >		select word
< left-button - double-click > + drag		select word + word
< control - left-button - click >		move insertion cursor without changing the selection
< left-button - click > on previous ModelSim or VSIM prompt		copy and paste previous command string to current prompt
< middle-button - click >		paste clipboard
< middle-button - press > + drag		scroll the window

Keystrokes - UNIX	Keystrokes - Windows	Result
< left | right arrow >		move cursor left | right one character
< control > < left | right arrow >		move cursor left | right one word
< shift > < left | right | up | down arrow >		extend selection of text
< control > < shift > < left | right arrow >		extend selection of text by word
< up | down arrow >		scroll through command history (in Source window, moves cursor one line up | down)
< control > < up | down >		moves cursor up | down one paragraph
< control > < home >		move cursor to the beginning of the text
< control > < end >		move cursor to the end of the text
< backspace >, < control-h >	< backspace >	delete character to the left
< delete >, < control-d >	< delete >	delete character to the right
none	esc	cancel

Keystrokes - UNIX	Keystrokes - Windows	Result
< alt >		activate or inactivate menu bar mode
< alt > < F4 >		close active window
< control - a >, < home >	< home >	move cursor to the beginning of the line
< control - b >	< left arrow >	move cursor left
< control - d >	< delete >	delete character to the right
< control - e >, < end >	< end >	move cursor to the end of the line
< control - f >	<right arrow>	move cursor right one character
< control - k >		delete to the end of line
< control - n >	< down arrow >	move cursor one line down (Source window only under Windows)
< control - o >	< enter >	insert a newline character at the cursor
< control - p >	< up arrow >	move cursor one line up (Source window only under Windows)
< control - s >	< control - f >	find
< F3 >		find next
< control - t >		reverse the order of the two characters on either side of the cursor
< control - u >		delete line
< control - v >, PageDn	PageDn	move cursor down one screen
< control - w >	< control - x >	cut the selection
< control - x >, < control - s>	< control - s >	save
< control - y >, F18	< control - v >	paste the selection
none	< control - a >	select the entire contents of the widget
< control - \ >		clear any selection in the widget
< control - ->, < control - / >	< control - Z >	undoes previous edits in the Source window
< meta - "<" >	none	move cursor to the beginning of the file
< meta - ">" >	none	move cursor to the end of the file
< meta - v >, PageUp	PageUp	move cursor up one screen
< Meta - w>	< control - c >	copy selection

Keystrokes - UNIX	Keystrokes - Windows	Result
< F8 >		search for the most recent command that matches the characters typed (Main window only)
< F9>		run simulation
< F10 >		continue simulation
	< F11 >	single-step
	< F12>	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Key	Action
<left arrow>	scroll listing left (selects and highlights the item to the left of the currently selected item)
<right arrow>	scroll listing right (selects and highlights the item to the right of the currently selected item)
<up arrow>	scroll listing up
<down arrow>	scroll listing down
<page up> <control-up arrow>	scroll listing up by page
<page down> <control-down arrow>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<shift-left arrow> <shift-right arrow>	extends selection left/right
<control-f> Windows <control-s> UNIX	opens the Find dialog box to find the specified item label within the list display

Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

Mouse action	Result
< control - left-button - drag down and right>[a]	zoom area (in)
< control - left-button - drag up and right>	zoom out
< control - left-button - drag up and left>	zoom fit
<left-button - drag> (Select mode) < middle-button - drag> (Zoom mode)	moves closest cursor
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom(vertical scroll) or far left or right (horizontal scroll)
< middle mouse-button - click in scroll bar trough> (UNIX only)	scrolls window to position of click

a. If you enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

Keystroke	Action
i I or +	zoom in (mouse pointer must be over the the cursor or waveform panes)
o O or -	zoom out (mouse pointer must be over the the cursor or waveform panes)
f or F	zoom full (mouse pointer must be over the the cursor or waveform panes)
l or L	zoom last (mouse pointer must be over the the cursor or waveform panes)
r or R	zoom range (mouse pointer must be over the the cursor or waveform panes)
<up arrow>/ <down arrow>	with mouse over waveform pane, scrolls entire window up/ down one line; with mouse over pathname or values pane, scrolls highlight up/down one line
<left arrow>	scroll pathname, values, or waveform pane left
<right arrow>	scroll pathname, values, or waveform pane right
<page up>	scroll waveform pane up by a page

Keystroke	**Action**
<page down>	scroll waveform pane down by a page
<tab>	search forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	search backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f> Windows <control-s> UNIX	open the find dialog box; searches within the specified field in the pathname or values pane for text strings
<control-left arrow>	scroll pathname, values, or waveform pane left by a page
<control-right arrow>	scroll pathname, values, or waveform pane right by a page

Right mouse button

The right mouse button provides shortcut menus in the most windows. See *Chapter 10 - Graphic interface* for menu descriptions.

C - ModelSim messages

Appendix contents

This appendix documents various status and warning messages that are produced by ModelSim.

ModelSim message system

The ModelSim message system helps you identify and troubleshoot problems while using the application. The messages display in a standard format in the Main window transcript. Accordingly, you can also access them from a saved transcript file (see "Saving the Main window transcript file" (UM-264) for more details).

Message format

The format for the messages is:

```
** <SEVERITY LEVEL>: ([<Tool>[-<Group>]]-<MsgNum>) <Message>
```

SEVERITY LEVEL may be one of the following:

severity level	meaning
Note	This is an informational message.
Warning	There may be a problem that will affect the accuracy of your results.
Error	The tool cannot complete the operation.
Fatal	The tool cannot complete execution.
INTERNAL ERROR	This is an unexpected error that should be reported to support@model.com.

Tool indicates which ModelSim tool was being executed when the message was generated. For example tool could be **vcom**, **vdel**, **vsim**, etc.

Group indicates the topic to which the problem is related. For example group could be FLI, PLI, VCD, etc.

Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few
arguments.
```

Getting more information

Each message is identified by a unique MsgNum id. You can access additional information about a message using the unique id and the **verror** (CR-317) command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

Suppressing warning messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

Suppressing VCOM warning messages

Use the `-nowarn <number>` argument to **vcom** (CR-303) to suppress a specific warning message. For example:

```
vcom -nowarn 1
```

Suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the *modelsim.ini* file (see "[vcom] VHDL compiler control variables" (UM-619)).

The warning message numbers are:

```
1 = unbound component
2 = process without a wait statement
3 = null range
4 = no space in time literal
5 = multiple drivers on unresolved signal
6 = compliance checks
7 = optimization messages
8 = lint checks
9 = signal value dependency at elaboration
10 = VHDL93 constructs in VHDL87 code
```

Suppressing VLOG warning messages

Use the `+nowarn<CODE>` argument to **vlog** (CR-345) to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

Suppresses decay warning messages.

Suppressing VSIM warning messages

Use the `+nowarn<CODE>` argument to **vsim** (CR-357) to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vsim +nowarnTFMPC
```

Suppresses warning messages about too few port connections.

Exit codes

The table below describes exit codes used by ModelSim tools.

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing
3	Cannot create a system process (execv, fork, spawn, etc.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
100	GUI Tcl initialization failure
101	GUI Tk initialization failure

Exit code	Description
102	GUI IncrTk initialization failure
111	X11 display error
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

Miscellaneous messages

This section describes miscellaneous messages which may be associated with ModelSim.

Empty port name warning

Message text

```
# ** WARNING: [8] <path/file_name>:
empty port name in port list.
```

Meaning

ModelSim reports these warnings if you use the **-lint** argument to **vlog** (CR-345). It reports the warning for any NULL module ports.

Suggested action

If you wish to ignore this warning, do not use the **-lint** argument.

Lock message

Message text

```
waiting for lock by user@user. Lockfile is <library_path>/_lock
```

Meaning

The *_lock* file is created in a library when you begin a compilation into that library, and it is removed when the compilation completes. This prevents simultaneous updates to the library. If a previous compile did not terminate properly, ModelSim may fail to remove the *_lock* file.

Suggested action

Manually remove the *_lock* file after making sure that no one else is actually using that library.

Metavalue detected warning

Message text

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

Meaning

This warning is an assertion being issued by the IEEE **numeric_std** package. It indicates that there is an 'X' in the comparison.

Suggested action

The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue arrow in the source window will be pointing at the line following the line with the comparison.

These messages can be turned off by setting the **NumericStdNoWarnings** variable to 1 from the command line or in the *modelsim.ini* file.

Sensitivity list warning

Message text

```
signal is read by the process but is not in the sensitivity list
```

Meaning

ModelSim outputs this message when you use the **-check_synthesis** argument to **vcom** (CR-303). It reports the warning for any signal that is read by the process but is not in the sensitivity list.

Suggested action

There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, you're only option as of version 5.7 is to not use the **-check_synthesis** argument. A more robust implementation of the argument may be added to a future version.

Tcl Initialization error 2

Message text

```
Tcl_Init Error 2 : Can't find a usable Init.tcl in the following directories :
   ./../tcl/tcl8.3 .
```

Meaning

This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

- modeltech-base.tar.gz
- modeltech-docs.tar.gz
- modeltech-<platform>.exe.gz

If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

This message could also occur if the file or directory was deleted or corrupted.

Suggested action

Reinstall ModelSim with all three files.

Too few port connections

Message text

```
# ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port connections.
Expected 2, found 1.
# Region: /foo/tb
```

Meaning

This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn't necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn't connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

Here are some examples of legal instantiations that will and will not cause the warning message.

Module definition:

```
module foo (a, b, c, d);
```

Instantiation that does not connect all pins but will not produce the warning:

`foo inst1(e, f, g, );` – positional association

`foo inst1(.a(e), .b(f), .c(g), .d());` – named association

Instantiation that does not connect all pins but will produce the warning:

`foo inst1(e, f, g);` – positional association

`foo inst1(.a(e), .b(f), .c(g));` – named association

Any instantiation above will leave pin *d* unconnected but the first example has a placeholder for the connection. Here's another example:

```
foo inst1(e, , g, h);

foo inst1(.a(e), .b(), .c(g), .d(h));
```

Suggested actions

- Check that there is not an extra comma at the end of the port list. (e.g., model(a,b,)). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
- If you are purposefully leaving pins unconnected, you can disable these messages using the **+nowarnTFMPC** argument to vsim.

VSIM license lost

Message text

```
Console output:
Signal 0 caught... Closing vsim vlm child.
vsim is exiting with code 4
FATAL ERROR in license manager

transcript/vsim output:
# ** Error: VSIM license lost; attempting to re-establish.
#    Time: 5027 ns  Iteration: 2
# ** Fatal: Unable to kill and restart license process.
#    Time: 5027 ns  Iteration: 2
```

Meaning

ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.

Suggested action

Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

D - System initialization

Appendix contents

ModelSim goes through numerous steps as it initializes the system during startup. It accesses various files and environment variables to determine library mappings, configure the GUI, check licensing, and so forth.

Files accessed during startup

The table below describes the files that are read during startup. They are listed in the order in which they are accessed.

File	**Purpose**
modelsim.ini	contains initial tool settings; see "Preference variables located in INI files" (UM-617) for specific details on the *modelsim.ini* file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is mgc_location_map
pref.tcl	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics; see "Preference variables located in Tcl files" (UM-631) for specific details on the *pref.tcl* file
modelsim.tcl	contains user-customized settings for fonts, colors, prompts, window positions, and other simulator window characteristics; see "Preference variables located in Tcl files" (UM-631) for more details on the *modelsim.tcl* file
<project_name>.mpf	if available, loads last project file which is specified in the registry (Windows) or *$(HOME)/.modelsim* (UNIX); see "What are projects?" (UM-32) for details on project settings

Environment variables accessed during startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see "Environment variables" (UM-613).

Environment variable	**Purpose**
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., *../modeltech/<platform>/*)
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies the pathname of the *modelsim.ini* file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the pathname of the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the pathname of all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the pathname of the MGC tool suite
TCL_LIBRARY	identifies the pathname of the Tcl library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the pathname of the Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the pathname of the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the pathname of the [incr]Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the pathname of the Tcl files that are used by ModelSim; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim
MODELSIM_TCL	identifies the pathname of user-customized GUI preferences (e.g., *C:\modeltech\modelsim.tcl*; this environment variable can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX)

Initialization sequence

The following list describes in detail ModelSim's initialization sequence. The sequence includes a number of conditional structures, the results of which are determined by the existence of certain files and the current settings of environment variables.

In the steps below, names in uppercase denote environment variables (except MTI_LIB_DIR which is a Tcl variable). Instances of *$(NAME)* denote paths that are determined by an environment variable (except *$(MTI_LIB_DIR)* which is determined by a Tcl variable).

1 Determines the path to the executable directory (../modeltech/<platform>/). Sets MODEL_TECH to this path, *unless* MODEL_TECH_OVERRIDE exists, in which case MODEL_TECH is set to the same value as MODEL_TECH_OVERRIDE.

2 Finds the *modelsim.ini* file by evaluating the following conditions:

- use MODELSIM if it exists; else
- use *$(MGC_WD)/modelsim.ini*; else
- use *./modelsim.ini*; else
- use *$(MODEL_TECH)/modelsim.ini*; else
- use *$(MODEL_TECH)/../modelsim.ini*; else
- use *$(MGC_HOME)/lib/modelsim.ini*; else
- set path to *./modelsim.ini* even though the file doesn't exist

3 Finds the location map file by evaluating the following conditions:

- use MGC_LOCATION_MAP if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else
- use *mgc_location_map* if it exists; else
- use *$(HOME)/mgc/mgc_location_map*; else
- use *$(HOME)/mgc_location_map*; else
- use *$(MGC_HOME)/etc/mgc_location_map*; else
- use *$(MGC_HOME)/shared/etc/mgc_location_map*; else
- use *$(MODEL_TECH)/mgc_location_map*; else
- use *$(MODEL_TECH)/../mgc_location_map*; else
- use no map

4 Reads various variables from the [vsim] section of the *modelsim.ini* file. See "[vsim] simulator control variables" (UM-621) for more details.

5 Parses any command line arguments that were included when you started ModelSim and reports any problems.

6 Defines the following environment variables:

- use MODEL_TECH_TCL if it exists; else

- set MODEL_TECH_TCL=*$(MODEL_TECH)/../tcl*
- set TCL_LIBRARY=*$(MODEL_TECH_TCL)/tcl8.3*
- set TK_LIBRARY=*$(MODEL_TECH_TCL)/tk8.3*
- set ITCL_LIBRARY=*$(MODEL_TECH_TCL)/itcl3.0*
- set ITK_LIBRARY=*$(MODEL_TECH_TCL)/itk3.0*
- set VSIM_LIBRARY=*$(MODEL_TECH_TCL)/vsim*

7 Initializes the simulator's Tcl interpreter.

8 Checks for a valid license (a license is not checked out unless specified by a *modelsim.ini* setting or command line option).

The next four steps relate to initializing the graphical user interface.

9 Sets Tcl variable MTI_LIB_DIR=$(MODEL_TECH_TCL)

10 Loads *$(MTI_LIB_DIR)/vsim/pref.tcl*.

11 Finds the *modelsim.tcl* file by evaluating the following conditions:

- use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use *$(HOME)/modelsim.tcl* if it exists

12 Loads last working directory, project file, and printer defaults from the registry (Windows) or *$(HOME)/.modelsim* (UNIX).

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler options dialog or use the **vmap** command, the tool updates the appropriate sections of the file.
- The *pref.tcl* file references the default .ini file via the [GetPrivateProfileString] Tcl command. The .ini file that is read will be the default file defined at the time *pref.tcl* is loaded.

Licensing Agreement

evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceives or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this subsection shall survive termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than employees and contractors, excluding Mentor Graphics' competitors, whose job performance requires access. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise (“attempted transfer”) without Mentor Graphics' prior written consent and payment of Mentor Graphics then-current applicable transfer charges. Any attempted transfer without Mentor Graphics’ prior written consent shall be a material breach of this Agreement and may. at Mentor graphics’ option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The provisions of this section 4 shall survive the termination or expiration of this Agreement.

5. **LIMITED WARRANTY.**

5.1. Mentor Graphics warrants that during the warranty period, Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH

IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED “AS IS.”

5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY.

8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USEOF SOFTWARE AS DESCRIBED IN SECTION 7.

9. INFRINGEMENT.

9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics’ obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.

9.4. THIS SECTION 9 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will automatically terminate if you fail to comply with any term or condition of this Agreement or if you fail to pay for the license when due and such failure to pay continues for a period of 30 days after written notice from Mentor Graphics. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth in this Agreement.

14. **AUDIT RIGHTS.** With reasonable prior notice, Mentor Graphics shall have the right to audit during your normal business hours all records and accounts as may contain information regarding your compliance with the terms of this Agreement. Mentor Graphics shall keep in confidence all information gained as a result of any audit. Mentor Graphics shall only use or disclose such information as necessary to enforce its rights under this Agreement.

15. **CONTROLLING LAW AND JURISDICTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF OREGON, USA, IF

YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH AND SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Dublin, Ireland when the laws of Ireland apply, or Wilsonville, Oregon when the laws of Oregon apply. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located.

16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

17. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions, except valid license agreements related to the subject matter of this Agreement (which are physically signed by you and an authorized agent of Mentor Graphics) either referenced in the purchase order or otherwise governing this subject matter. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and expenses.

Rev. 020826, Part Number 214231

Index

CR = Command Reference, UM = User's Manual

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z