ModelSim

SE

User's Manual

Version 5.5

Published: 22/Feb/01

The world's most popular HDL simulator

Model Technology Incorporated
10450 SW Nimbus Avenue / Bldg. R-B
Portland OR 97223-4347 USA

phone: 503-641-1340
fax: 503-526-5410
e-mail: support@model.com, sales@model.com
home page: http://www.model.com

Table of Contents

1 - Introduction (1-15)

2 - Projects and system initialization (2-25)

3 - Design libraries (3-41)

4 - VHDL Simulation (4-55)

5 - Verilog Simulation (5-73)

6 - Mixed VHDL and Verilog Designs (6-127)

7 - Datasets (saved simulations) and virtuals (7-137)

8 - ModelSim Graphic Interface (8-149)

9 - Performance Analyzer (9-281)

10 - Code Coverage (10-291)

11 - Waveform Comparison (11-301)

12 - Standard Delay Format (SDF) Timing Annotation (12-325)

13 - Value Change Dump (VCD) Files (13-341)

14 - Logic Modeling SmartModels (14-353)

15 - Logic Modeling Hardware Models (15-363)

16 - Tcl and ModelSim (16-369)

A - Technical Support, Updates, and Licensing (A-385)

B - ModelSim Variables (B-391)

C - ModelSim Shortcuts (C-409)

D - Using the FLEXlm License Manager (D-417)

E - Tips and Techniques (E-425)

1 - Introduction

Chapter contents

This documentation was written for Model*Sim* SE version 5.5 for UNIX and Microsoft Windows 95/98/ME/NT/2000 (see note below for exception). If the Model*Sim* software you are using is a later release, check the README file that accompanied the software. Any supplemental information will be there.

Although this document covers both VHDL and Verilog simulation, you will find it a useful reference for single HDL design work.

Performance tools included with ModelSim SE

All Model*Sim* SE versions include the following performance tools:

- Performance Analyzer (9-281)
  Identifies areas in your simulation where performance can be improved.

Note: Performance Analyzer *will not* operate on Windows 95.

- Code Coverage (10-291)
  Gives you graphical and report file feedback on how the source code is being executed.

ModelSim's graphic interface

While your operating system interface provides the window-management frame, Model*Sim* 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
- Linux (Red Hat v. 6.0 or later) with KDE or GNOME
- Microsoft Windows 95/98/ME/NT/2000

Because Model*Sim*'s graphic interface is based on Tcl/TK, you also have the tools to build your own simulation environment. Preference variables and configuration commands, "Preference variables located in INI files" (B-396), and "Graphic interface commands" (8-277) give you control over the use and placement of windows, menus, menu options and buttons. See "Tcl and ModelSim" (16-369) for more information on Tcl.

For an in-depth look at Model*Sim*'s graphic interface see, *Chapter 8 - ModelSim Graphic Interface*.

Standards supported

ModelSim VHDL supports both the IEEE 1076-1987 and 1076-1993 VHDL, 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 either IEEE Standard 1076-1987 or 1076-1993.

ModelSim Verilog is based on the IEEE Std 1364 *Standard Hardware Description Language Based on the Verilog Hardware Description Language*. 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.

Assumptions

We assume that you are familiar with the use of your operating system. You should also be familiar with the window management functions of your graphic interface: either OpenWindows, OSF/Motif, CDE, HP VUE, KDE, GNOME, or Microsoft Windows 95/98/ME/NT/2000.

We also assume that you have a working knowledge of VHDL and Verilog. Although ModelSim is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal. If you need more information about HDLs, check out our Online References - www.model.com (1-22).

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

For installation instructions please refer to the *Start Here for ModelSim* guide that was shipped with the ModelSim CD. *Start Here* may also be downloaded from our website: www.model.com.

Sections in this document

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

2 - Projects and system initialization (2-25)
This chapter provides a definition of a Model*Sim* "project" and discusses the use of a new file extension for project files.

3 - Design libraries (3-41)
To simulate an HDL design using Model*Sim*, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter.

4 - VHDL Simulation (4-55)
This chapter is an overview of compilation and simulation for VHDL within the Model*Sim* environment.

5 - Verilog Simulation (5-73)
This chapter is an overview of compilation and simulation for Verilog within the Model*Sim* environment.

6 - Mixed VHDL and Verilog Designs (6-127)
Model*Sim*/Plus single-kernel simulation (SKS) allows you to simulate designs that are written in VHDL and/or Verilog. This chapter outlines data mapping and the criteria established to instantiate design units between HDLs.

7 - Datasets (saved simulations) and virtuals (7-137)
This chapter describes datasets and virtuals - both methods for viewing and organizing simulation data in Model*Sim*.

8 - ModelSim Graphic Interface (8-149)
This chapter describes the graphic interface available while operating Model*Sim*. Model*Sim*'s graphic interface is designed to provide consistency throughout all operating system environments.

9 - Performance Analyzer (9-281)
This chapter describes how the Model*Sim* Performance Analyzer is used to easily identify areas in your simulation where performance can be improved.

10 - Code Coverage (10-291)
This chapter describes the Code Coverage feature. Code Coverage gives you graphical and report file feedback on how the source code is being executed.

11 - Waveform Comparison (11-301)
This chapter describes Waveform Comparison, a feature that lets you compare simulations.

12 - Standard Delay Format (SDF) Timing Annotation (12-325)
This chapter discusses Model*Sim*'s implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Command reference

The complete command reference for all Model*Sim* commands is located in the *ModelSim Command Reference*. Command Reference cross reference page numbers are prefixed with "CR" (e.g.,"ModelSim Commands" (CR-9)).

Text conventions

Text conventions used in this manual include:

italic text	provides emphasis and sets off filenames, path names, and design unit names
bold text	indicates commands, command options, menu choices, package and library logical names, as well as variables and dialog box selection
`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 > Save**
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 Model*Sim* (e.g., DO, WLF, INI, MPF, PDF, etc.)

What is an "HDL item"

Because Model*Sim* works with both VHDL and Verilog, “HDL” refers to either VHDL or Verilog when a specific language reference is not needed. Depending on the context, “HDL item” can refer to any of the following:

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

Where to find our documentation

Model*Sim* documentation is available from our website at model.com/support/documentation.asp or in the following formats and locations:

Document	Format	How to get it
Start Here for ModelSim SE (installation & support 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 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**
ModelSim Foreign Language Interface Reference	PDF, HTML	select **Main window > Help > SE Documentation**
Model*Sim* Command Help	ASCII	type `help [command name]` at the prompt in the Main window
Tcl Man Pages (Tcl manual)	HTML	select **Main window > Help > Tcl Man Pages**, or find *contents.htm* in *\modeltech\tcl_help_html*
technotes	ASCII	select **Main window > Help > Technotes**, or located in the *\modeltech\docs\technotes* directory

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 Model*Sim* CD. It is also available without cost from Adobe at http://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 key words much faster.

Online References - www.model.com

The Model Technology web site includes links to support, software downloads, and many EDA information sources. Check the links below for the most current information.

Latest version email

Place your name on our list for email notification of new releases and updates.
model.com/support/register_news_list.asp

News

Current news of Model Technology within the EDA industry.
model.com/news_events/default.asp

Partners

Model Technology's value added partners, OEM partners, FPGA partners, ASIC partners, and training partners.
model.com/partners/default.asp

Products

A complete collection of Model Technology product information.
model.com/products/default.asp

Technical Documents

Technical notes, application notes, FAQs.
model.com/resources/techdocs.asp

Sales

Locate Model*Sim* sales contacts anywhere in the world.
model.com/contact_us.asp

Support

Model Technology email support and software downloads.
model.com/support/default.asp

Comments

Comments and questions about this manual and Model*Sim* software are welcome. Call, write, fax or email:

Model Technology Incorporated
10450 SW Nimbus Avenue, Bldg. R-B
Portland, OR 97223-4347 USA

phone: 503-641-1340
fax: 503-526-5410
email: manuals@model.com
home page: http://www.model.com

2 - Projects and system initialization

Chapter contents

This chapter discusses Model*Sim* projects. Projects greatly simplify the process of compiling and simulating a design and are a great tool for getting started with Model*Sim*. This chapter also includes a section on Model*Sim* initialization.

Introduction

What are projects?

Projects are collection entities for HDL 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 consist of:

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

What are the benefits of projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with Model*Sim*; 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
- 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 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 .ini variable settings every time the project is opened; in previous versions you had to quit Model*Sim* and restart the program to read in a new .ini file

How do projects differ in version 5.5?

Projects have improved a great deal from earlier versions. Some of the key differences include:

- A new interface eliminates the need to write custom scripts.
- You don't have to copy files into a specific directory; you can establish references to files in any location.
- You don't have to specify compiler switches; the automatic defaults will work for many designs. However, if you do want to customize the settings, you do it through a dialog box rather than by writing a script.
- All metadata (compiler settings, compile order, file mappings) are stored in the project .mpf file.

Important: 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 version 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. For a discussion of more advanced project features, see "Customizing project settings" (2-34).

Step 1 — Create a new project (2-29)

This creates a .mpf file and a working library.

Step 2 — Add files to the project (2-31)

Projects can reference or include HDL source files 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 — Compile the files (2-32)

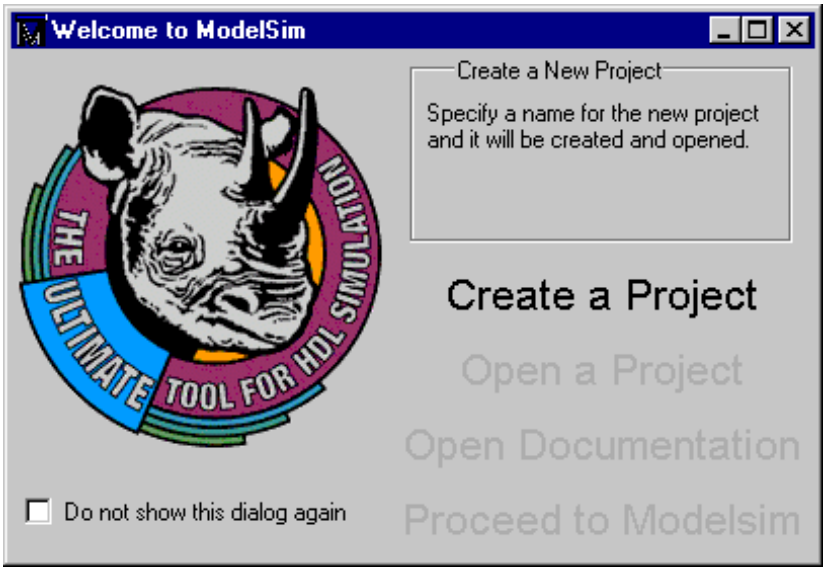
This checks syntax and semantics and creates the pseudo machine code Model*Sim* uses for simulation.

Step 4 — Simulate a design (2-33)

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

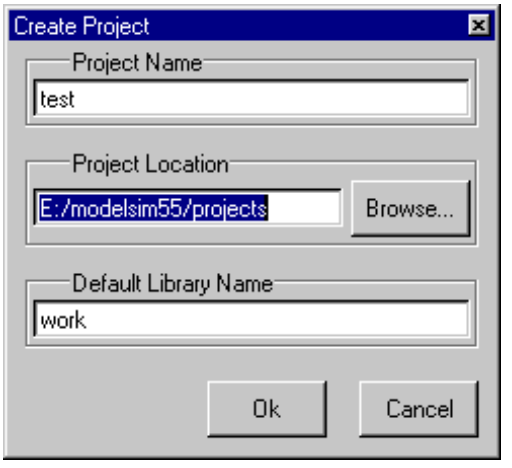
Step 1 — Create a new project

1 Select **Create a Project** from the Welcome to Model*Sim* screen that opens the first time you start Model*Sim*. If this screen is not available, you can enable it by selecting **Help > Enable Welcome** (Main window).



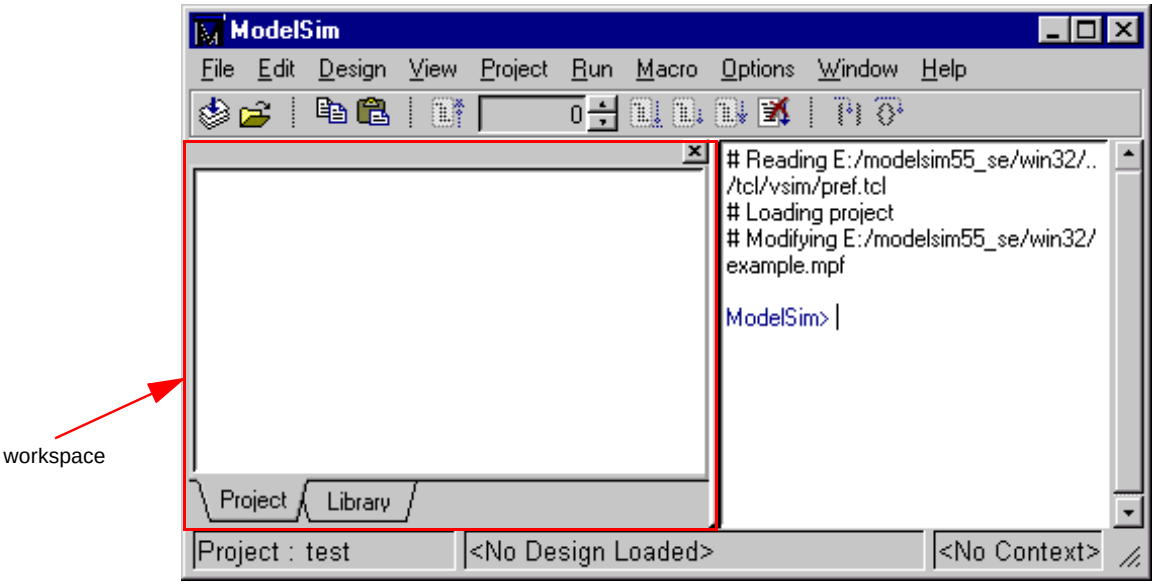
You can also use the **File > New > Project** (Main window) command to create a new project.

2 Clicking the **Create a Project** button opens the Create Project dialog box.



3 Specify a **Project Name** and **Project Location**. The location is where the project .mpf file and any copied source files will be stored. You can leave the Default Library Name set to "work," or specify a different name if desired. The name that is specified will be used to create a working library subdirectory within the Project Location.

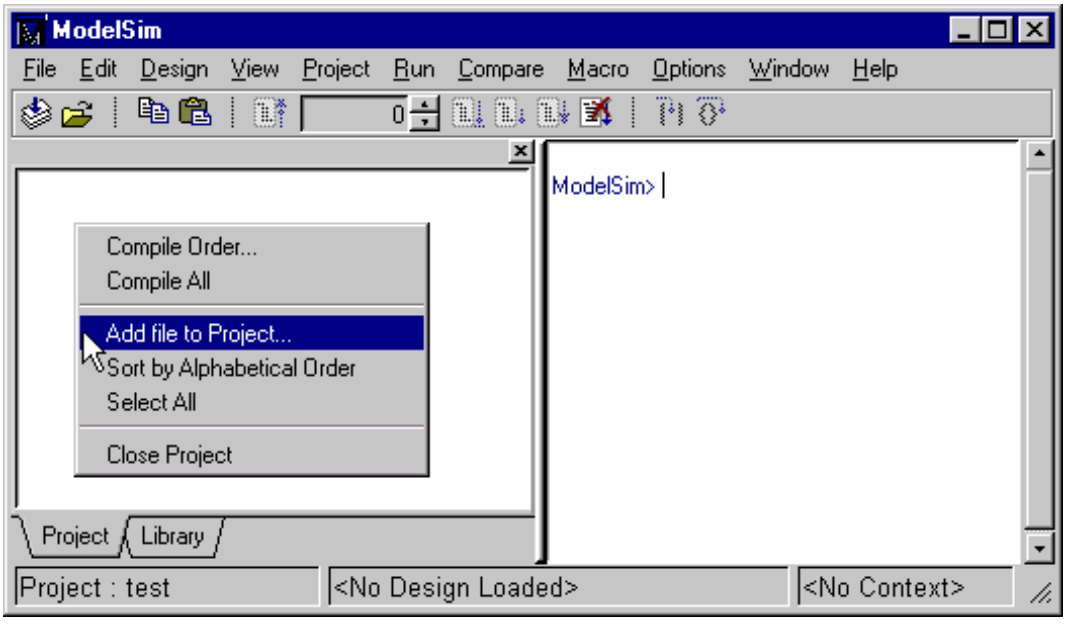
After selecting OK, you will see a blank Project page in the workspace area of the Main window. You can hide or show the workspace at any time using the **View > Hide/Show Workspace** command.



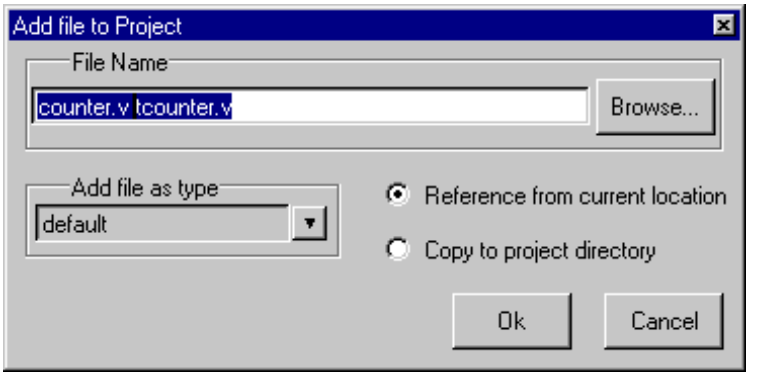
The name of the current project is shown at the bottom left corner of the Main window.

Step 2 — Add files to the project

Your right mouse button (2nd button in Windows; 3rd button in UNIX) gives quick access to project commands. When you right-click in the workspace, a context menu appears. The menu that appears depends on where you click in the workspace.



1 Right click in a blank area on the Project page and select **Add file to Project**. This opens the Add file to Project dialog. You can also select **Project > Add file to Project** from the menu bar.

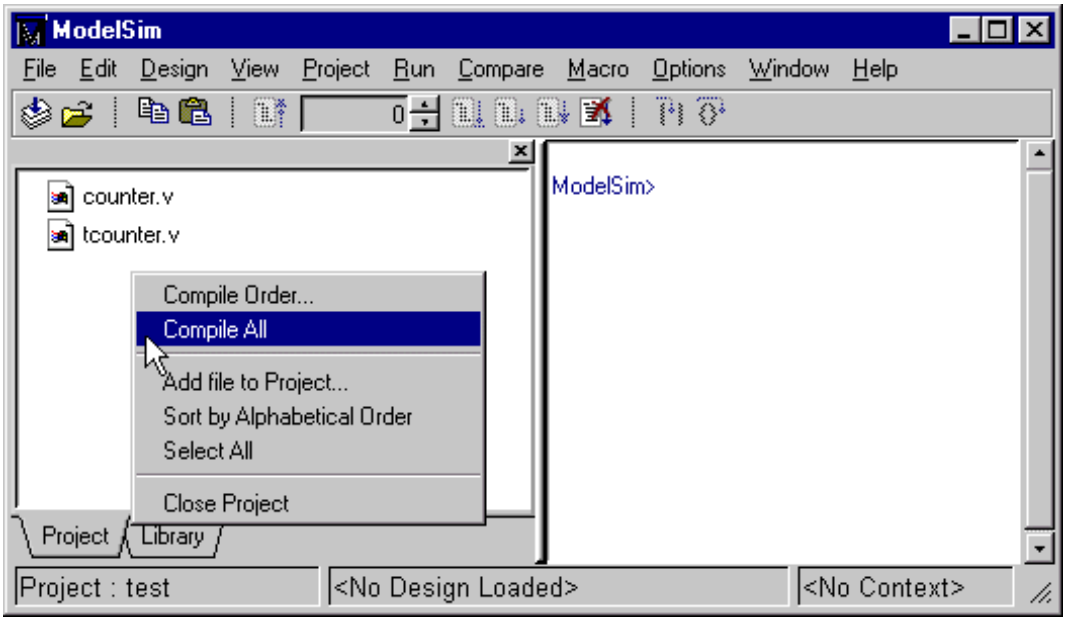


2 Specify one or more files you want to add to the project. (The files used in this example are available in the examples directory that is installed along with Model*Sim*.)

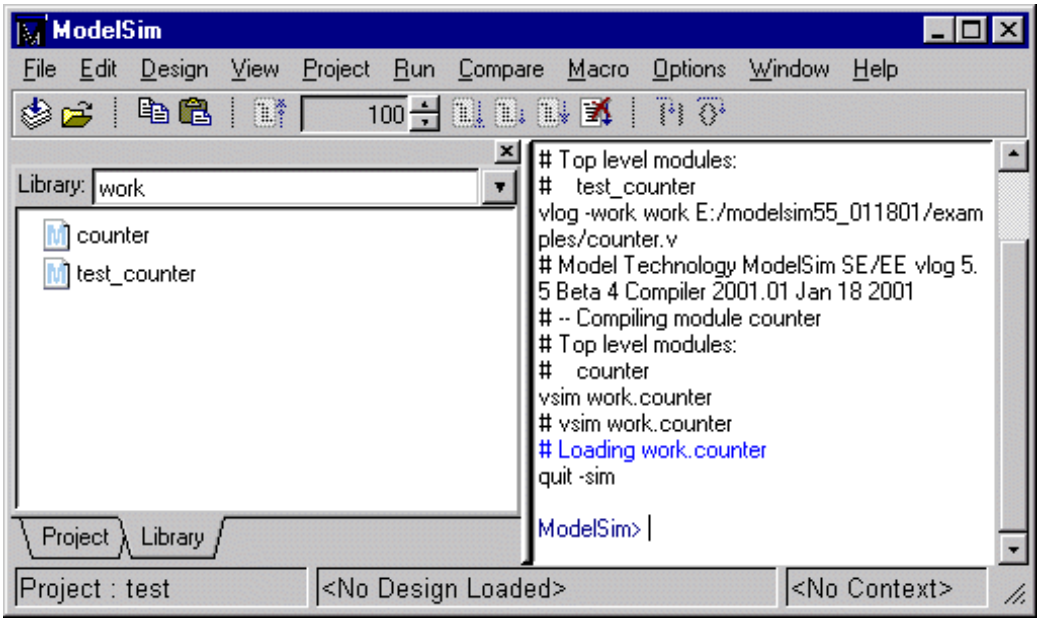
3 For the files you're adding, choose whether to reference them from their current location or copy them into the project directory.

Step 3 — Compile the files

1 To compile the files, right click in the Project page and select **Compile All**. You can also select **Project > Compile All** from the menu bar.

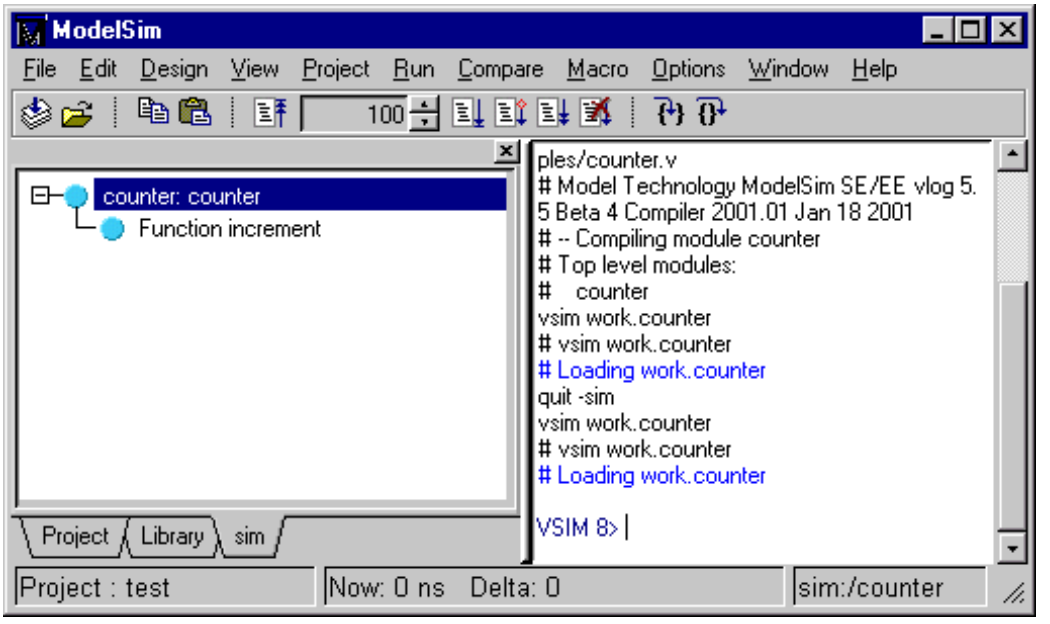


2 Once compilation is finished, click the Library tab and you'll see the two compiled designs.



Step 4 — Simulate a design

1 To simulate one of the designs, either double-click the name or right click the name and select Load. A new page appears showing the structure of the current active simulation.



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 Model*Sim* Tutorial for examples.

Other project operations

In addition to the four actions just discussed, the following are common project operations.

Open an existing project

When you leave a Model*Sim* session, Model*Sim* will remember the last opened project. You can reopen it for your next session by clicking **Open Project** in the Welcome to Model*Sim* dialog. You can also open an existing project by selecting **File > Open > Project** (Main window).

Close a project

Select **File > Close > Project** (Main window). This closes the Project page but leaves the Library and Structure (labeled "Sim" in the graphic above) pages open in the workspace.

Delete a project

Select **File > Delete > Project** (Main window).

Customizing project settings

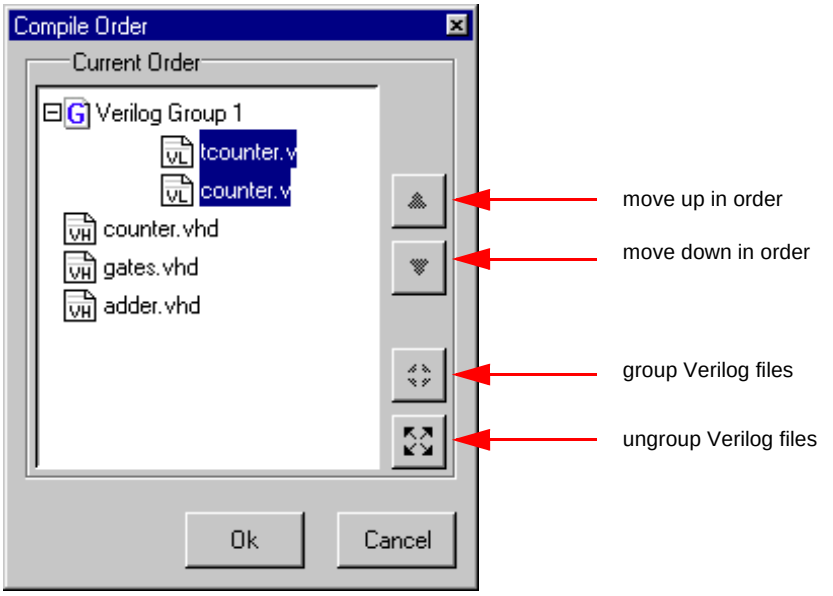
Though the default project settings will work for many designs, it is easy to customize the settings if needed. You can change the compile order and set compiler options.

Changing compile order

When you compile all files in a project, Model*Sim* 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 using the Compile Order dialog.

Note: 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.

To open the Compile Order dialog, right click in an empty area of the Project tab and select **Compile Order**. The dialog shown below opens.



The group and ungroup buttons are used on Verilog files only. They allow you to group two or more Verilog files so they are sent to the compiler at the same time. One case where you might use this is when you have one file with a bunch of define statements and a second file that is a Verilog module. You would want to compile these two files at the same time.

Setting compiler options

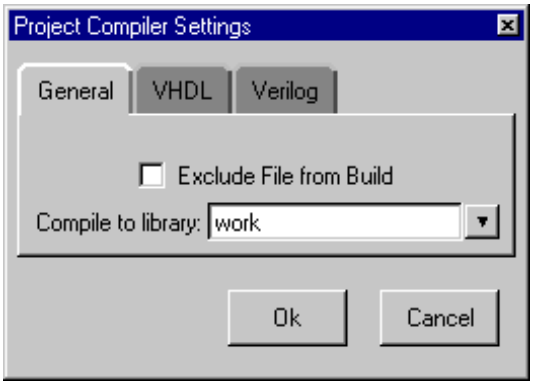
The VHDL and Verilog compilers (vcom and vlog, respectively) have numerous options that affect how a design is compiled and subsequently simulated. Outside of a project you can set the defaults for all future simulations using the **Options > Compile** (Main window) command. Inside of a project you can set these options on individual files or a group of files.

To set the compiler options in a project, select the file(s) in the Project page, right click on the file names, and select **Compile Properties**. The pages that appear in the resulting dialog depend on the type of files you have selected. If you select a VHDL file, you'll see only the General and VHDL pages. If you select a Verilog file, you'll see only the General and Verilog pages. If you select both a VHDL file and a Verilog file, you'll see all three pages (as shown in the dialog below).

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" like this: 

  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, Model*Sim* 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.



- **Exclude File from Build**
  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.

The definitions of the options on the VHDL and Verilog pages can be found in the section "Setting default compile options" (8-252).

Accessing projects from the command line

Generally, projects are used only within the Model*Sim* graphical user interface. However, standalone tools will use the project file if they are invoked in the project's root directory. If invoked outside the project directory, the **MODELSIM** environment variable can be set with the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

You can also use the **project** command (CR-159) from the command line to perform common operations on new projects. The command is to be used outside of a simulation session.

System initialization

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" (B-396) 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; see "How location mapping works" (E-438) for more details
pref.tcl	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics; see "Preference variables located in TCL files" (B-406) 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" (B-406) for specific details on the *modelsim.tcl* file

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" (B-393).

Environment variable	Purpose
MODEL_TECH	set by Model*Sim* 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 path to the *modelsim.ini* file
MGC_WD	identifies the Mentor Graphics working directory (set by Mentor Graphics tools)
MGC_LOCATION_MAP	identifies the path to the location map file; set by Model*Sim* if not defined
MODEL_TECH_TCL	identifies the path to all Tcl libraries installed with Model*Sim*
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the path to the MGC tool suite
TCL_LIBRARY	identifies the path to the Tcl library; set by Model*Sim* to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the path to the Tk library; set by Model*Sim* to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TIX_LIBRARY	identifies the path to the Tix library; set by Model*Sim* to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the path to the [incr]Tcl library; set by Model*Sim* to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the path to the [incr]Tk library; set by Model*Sim* to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the path to the Tcl files that are used by ModelSim; set by Model*Sim* to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with Model*Sim*

Environment variable	Purpose
MODELSIM_TCL	identifies the path to the *modelsim.tcl* file; 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 Model*Sim*'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", Model*Sim* 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" (B-398) for more details.

5 Parses any command line arguments that were included when you started Model*Sim* 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)/tcl8.0*
- set TK_LIBRARY=*$(MODEL_TECH_TCL)/tk8.0*
- set TIX_LIBRARY=*$(MODEL_TECH_TCL)/tix4.1*
- 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)/pref.tcl*.

11 Loads last working directory, project init, project history, and printer defaults from the registry (Windows) or *$(HOME)/.modelsim* (UNIX).

12 Finds the *modelsim.tcl* file by evaluating the following conditions:

- use MODELSIM_TCL 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

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within Model*Sim*, 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.

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 Model*Sim* are compiled into libraries as well.

Design library contents

A *design library* is a directory 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; and Verilog modules and UDPs (user defined primitives). The design units are classified as follows:

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

Design unit information

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

- retargetable, executable code
- debugging information
- dependency information

Design library types

There are two kinds of design libraries: working libraries and resource libraries. A *working library* is the library into which a design unit is placed after compilation. A *resource library* contains design units that can be referenced within the design unit being compiled. Only one library can be the working library; in contrast, any number of libraries (including the working library itself) can be resource libraries during a compilation.

The library named **work** has special attributes within Model*Sim*; 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. In other words the **work** library is the *working* library. In all other aspects it is the same as any other library.

Working with design libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within Model*Sim*, 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.

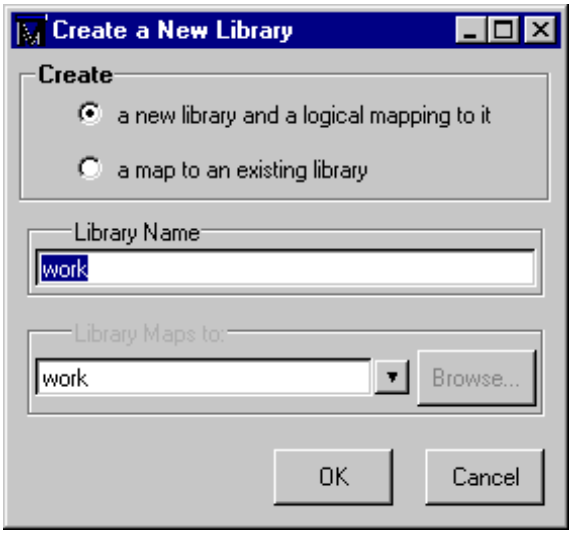
Creating a library

When you create a project (see "Getting started with projects" (2-28)), 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 Model*Sim* graphic interface.

From the Model*Sim* prompt or a UNIX/DOS prompt, use this **vlib** command (CR-249):

```
vlib <directory_pathname>
```

To create a new library with the Model*Sim* graphic interface, select **Design > Create a New Library** (Main window). This brings up a dialog box that allows you to specify the library name and its logical mapping.



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 new library name into this field.

- **Library Maps to**
  Type or **Browse** for a mapping for the specified library. This field can be changed only when the **Create a map to an existing library** option is selected.

When you click **OK**, Model*Sim* 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 Model*Sim* library.

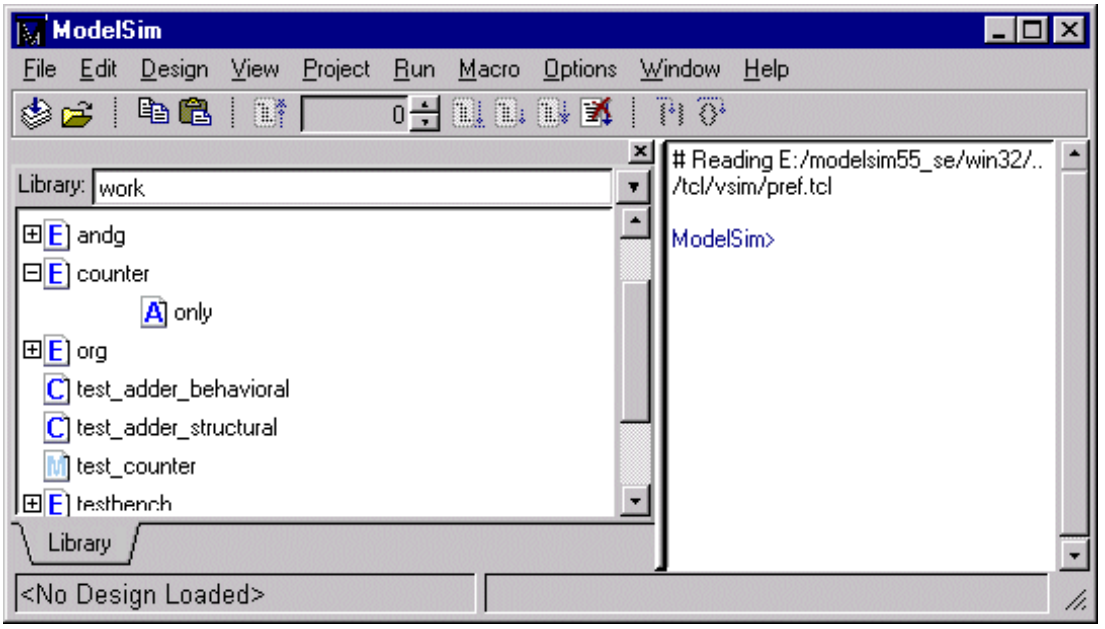
The new map entry is written to the *modelsim.ini* file in the [Library] section. See "[Library] library path variables" (B-396) for more information.

Note: Remember that a design library is a special kind of directory; the only way to create a library is to use the Model*Sim* GUI or the **vlib** command (CR-249). 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 page in the Main window workspace provides access to design units (configurations, modules, packages, entities, and architectures) in a library. Note the icons identify whether a unit is an entity (E), a module (M), and so forth.



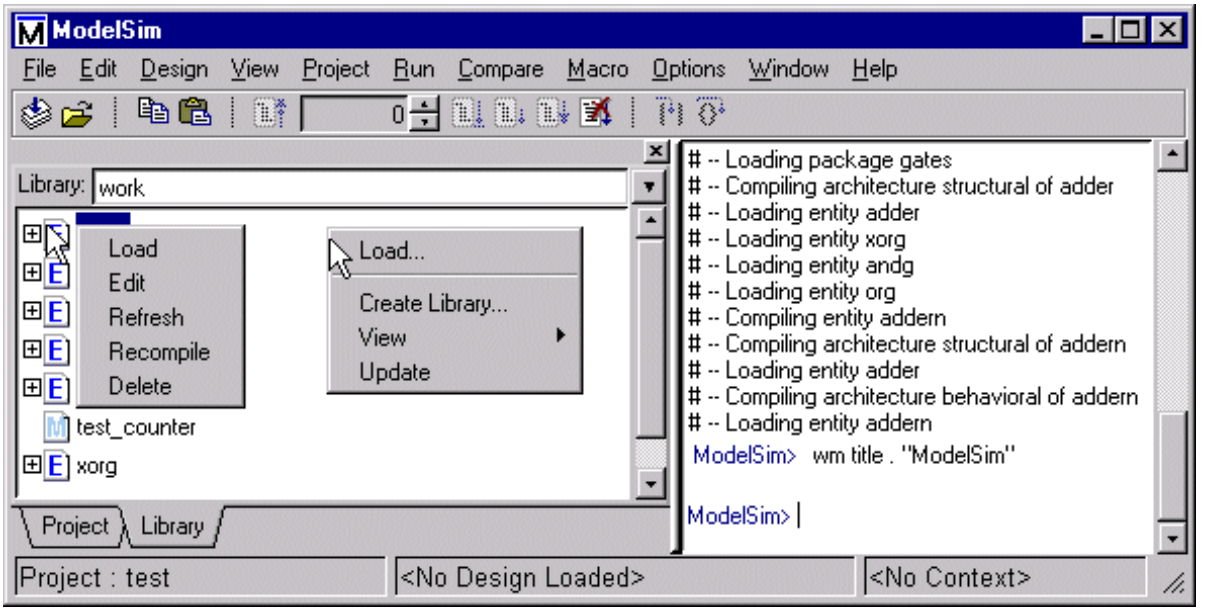
The Library page includes these options:

- **Library**
  Select the library you wish to view from the drop-down list. Related command line command is **vdir** (CR-223).

- **DesignUnit/Description list**

  Select a plus (+) box to view the associated architecture, or select a minus (–) box to hide the architecture.

The Library page also has two context menus that you access with your right mouse button (Windows—2nd button, UNIX—3rd button).One menu is accessed by right-clicking a design unit name; the second is accessed by right-clicking a blank area in the Designs page. The graphic below shows the two menus.



The context menu at the left includes the following commands:

- **Load**
  Simulates the selected design unit and opens a structure page in the workspace. Related command line command is **vsim** (CR-258).
- **Edit**
  Opens the selected design unit in the Source window.
- **Refresh**
  Rebuilds the library image of the selected item(s) without using source code. Related command line command is **vcom** (CR-217) with the -refresh argument.
- **Recompile**
  Recompiles the selected design unit. Related command line command is **vcom** (CR-217).
- **Delete**
  Deletes the selected design unit. Related command line command is **vdel** (CR-222).

  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, the entity and all its associated architectures 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.

The second context menu has the following options:

- **Load**
  Opens the Load Design dialog box. See "Simulating with the graphic interface" (8-256) for details. Related command line command is **vsim** (CR-258).
- **Create Library**
  Opens the Create a New Library dialog box. See "Creating a library" (3-43) earlier in this chapter for details. Related command line command is **vlib** (CR-249).
- **View**
  Provides various options for displaying design units.
- **Update**
  Reloads the library in case any of the design units were modified outside of the current session (e.g., by a script or another user).

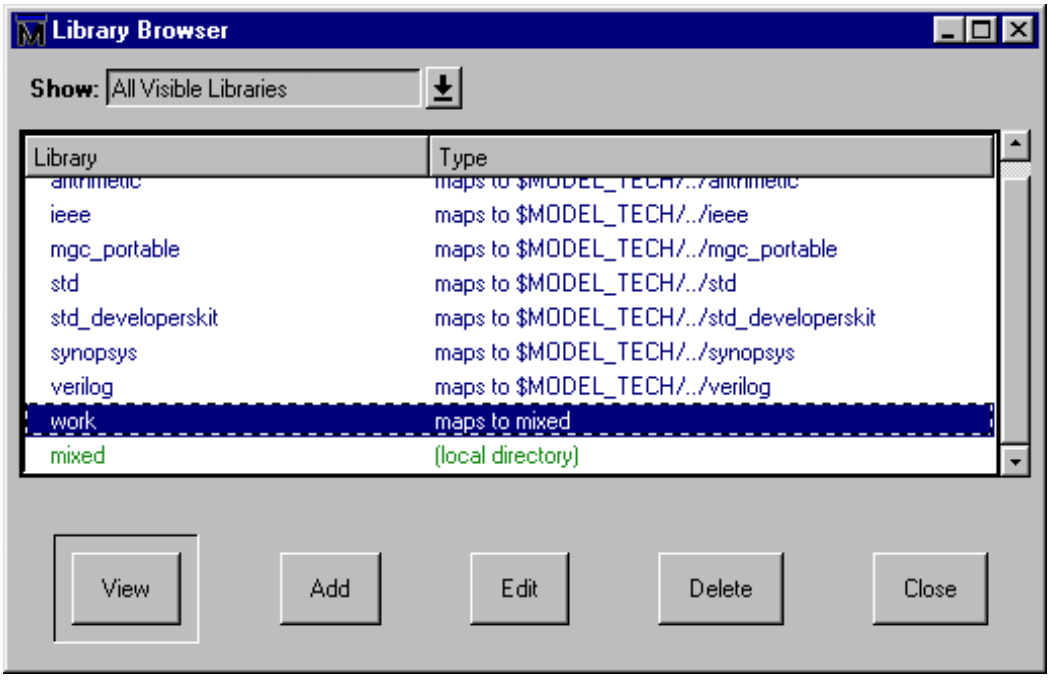
Assigning a logical name to a design library

VHDL uses logical library names that can be mapped to Model*Sim* library directories. By default, Model*Sim* 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 **Design > Browse Libraries** (Main window). This brings up a dialog box that allows you to view, add, edit, and delete mappings, as shown below:



The **Library Browser** dialog box includes these options:

- **Show**
  Choose the mapping and library scope to view from the drop-down list.
- **Library/Type list**

  To view the contents of a library
  Select the library, then click the **View** button. This brings up the Library page (3-44) in the Main window. From there you can also delete design units from the library.

  To create a new library mapping
  Click the **Add** button. This brings up **Create a New Library** (3-43) dialog box that allows you to enter a new logical library name and the pathname to which it is to be mapped.

  It is possible to enter the name of a non-existent directory, but the specified directory must exist as a Model*Sim* library before you can compile design units into it. ModelSim will issue a warning message if you try to map to a non-existent directory.

To edit an existing library mapping
Select the desired mapping entry, then click the **Edit** button. This brings up a dialog box that allows you to modify the logical library name and the pathname to which it is mapped. Selecting **Delete** removes an existing library mapping, but it does not delete the library. The library can be deleted with this **vdel** command (CR-222):

```
vdel -lib <library_name> -all
```

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>
```

This command may be invoked from either a UNIX/DOS prompt or from the command line within Model*Sim*.

When you use **vmap** (CR-257) this way you are modifying the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, edit the *modelsim.ini* file using any 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-257) 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.

See also

See "ModelSim Commands" (CR-9) for more information about the library management commands, "ModelSim Graphic Interface" (8-149) for more information about the graphical user interface, and "Projects and system initialization" (2-25) for more information about the *modelsim.ini* file.

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.

Specifying the resource libraries

VHDL resource libraries

Within a VHDL source file, you can 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-217) 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.

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 1076-1987* and *ANSI/IEEE Std 1076-1993*. See also, "Using the TextIO package" (4-60).

A VHDL **use** clause can be used to select specific 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 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, you can 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 file in which the **use** clause is placed. 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 std_logic_1164 packages (accelerated for Model*Sim*).
- *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, vital_timing, and vital_memory.

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.

VITAL 2000 library

ModelSim versions 5.5 and later include a separate VITAL 2000 library that contains an accelerated vital_memory package.

You'll need to add a **use** clause to your VHDL code to access the package. For example:

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

Also, when you compile, use the **-vital2000** switch to **vcom** (CR-217).

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 Model*Sim* with this command:

```
do rebldlibs.do
```

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

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.

Note: Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using **vcom** (CR-217) with the **-93** option.

Regenerating your design libraries

Depending on your current Model*Sim* 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 page context menu (see "Managing library contents" (3-44)), or by using the **-refresh** argument to **vcom** (CR-217) and **vlog** (CR-250).

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 Model*Sim* (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 (Verilog only) 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.

Verilog resource libraries

Model*Sim* supports and encourages separate compilation of distinct portions of a Verilog design. The **vlog** (CR-250) compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs (and, perhaps, VHDL design units) that are referenced by the simulator as it loads the design. See "Library usage" (5-78).

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

It is possible with Model*Sim* to maintain 32-bit and 64-bit versions of a design in the same library. To do this, you must compile the design with one of the versions (32-bit or 64-bit), and "refresh" the design with the other version. For example:

Using the 32-bit version of Model*Sim*:

```
vcom file1.vhd
vcom file2.vhd
```

Next, using the 64-bit version of Model*Sim*:

```
vcom -refresh
```

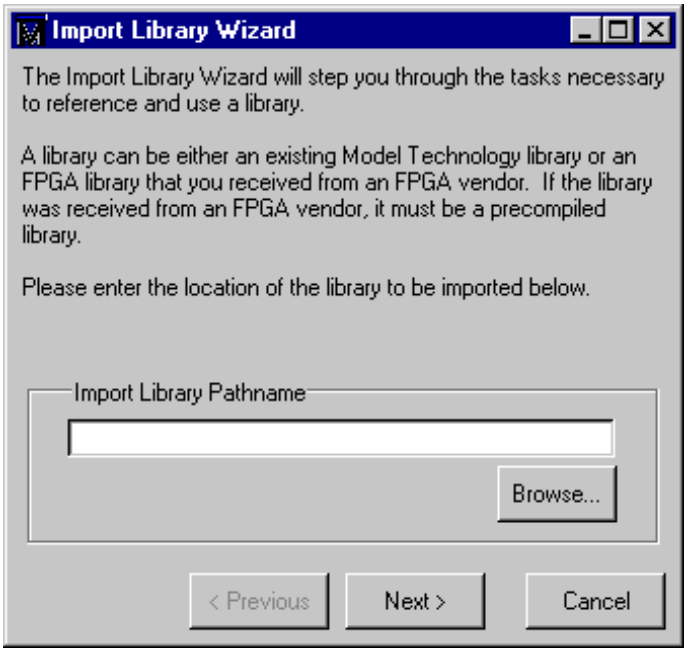
Do not compile the design with one version, and then recompile it with the other. If you do this, Model*Sim* will remove the first module, because it could be "stale."

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 **Design > Import Library** (Main window).



Follow the instructions in the wizard to complete the import.

4 - VHDL Simulation

Chapter contents

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

The TextIO package is defined within the *VHDL Language Reference Manuals, IEEE Std 1076-1987* and *IEEE Std 1076-1993*; it allows human-readable text input from a declared source within a VHDL file during simulation.

Compiling and simulating with the GUI

Many of the examples in this chapter are shown from the command line. For compiling and simulating within a project or the Model*Sim* GUI, see:

- Getting started with projects (2-28)
- Compiling with the graphic interface (8-250)
- Simulating with the graphic interface (8-256)

ModelSim variables

Several variables are available to control simulation, provide simulator state feedback, or modify the appearance of the Model*Sim* GUI. To take effect, some variables, such as environment variables, must be set prior to simulation. See *Appendix B - ModelSim Variables* for a complete listing of Model*Sim* variables.

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-249) 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.

Note: 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-249).

See "Design libraries" (3-41) for additional information on working with libraries.

Invoking the VHDL compiler

Model*Sim* compiles one or more VHDL design units with a single invocation of **vcom** (CR-217), 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 both the 1076 -1987 and 1076 -1993 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The **vcom** (CR-217) command compiles units written with version 1076 -1987 by default; use the -93 option with **vcom** (CR-217) to compile units written with version 1076 -1993. You can also change the default by modifying the *modelsim.ini* file (see "Preference variables located in INI files" (B-396) 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-217) 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.

Simulating VHDL designs

After compiling the design units, you can simulate your designs with **vsim** (CR-258). 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" (2-28)) or the Load Design dialog box (see "Simulating with the graphic interface" (8-256)).

Note: Simulation normally stops if a failure occurs; however, if a bounds check on a signal fails the simulator will continue running.

Invoking the simulator from the Main window

For VHDL, invoke **vsim** (CR-258) 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-258) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

If a design unit name is not specified, **vsim** (CR-258) will present the **Load Design** dialog box from which you can choose a configuration or entity/architecture pair. See "Simulating with the graphic interface" (8-256) for more information.

Selecting the time resolution

The simulation time resolution is 1 ns by default. You can select a specific time resolution with the **vsim** (CR-258) **-t** option or from the **Load Design** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example, to run in picosecond resolution, or 10ps resolution respectively:

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

Note that there is no space between the value and the units (i.e.., 10ps, not 10 ps).

The default time resolution can also be changed by modifying the Resolution (B-400) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-168) with the **simulator state** option.

See "Preference variables located in INI files" (B-396) for more information on modifying the *modelsim.ini* file.

vsim (CR-258) 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** and **-sdfmax** options. 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
```

Timing check disabling

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

For example:

```
vsim +notimingchecks topmod
```

Invoking Code Coverage with vsim

ModelSim's Code Coverage feature gives you graphical and report file feedback on how the source code is being executed. It allows line number execution statistics to be kept by the simulator. It can be used during any design phase and in all levels and types of designs. For complete details, see *Chapter 10 - Code Coverage*.

To acquire code coverage statistics, the **-coverage** switch must be specified during the command-line invocation of the simulator.

```
vsim -coverage ...
```

This will allow you to use the various code coverage commands: **coverage clear** (CR-92), **coverage reload** (CR-93), and **coverage report** (CR-94).

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.

The VHDL'93 syntax for a file declaration is:

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

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 B - ModelSim Variables* for more details.

Using STD_INPUT and STD_OUTPUT within Model*Sim*

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";
```

The standard VHDL'93 TextIO package contains 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 Model*Sim*, reading from the STD_INPUT file allows you to enter text into the current buffer from a prompt in the Main window. The last line written to the STD_OUTPUT file appears at the prompt.

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 */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, Model*Sim* supplies hexadecimal routines in the package io_utils, which is located in the file */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, IEEE Std 1076-1987 and IEEE Std 1076-1993*, 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.

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.

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 Model*Sim* install files as an example. Check for this file:

<*install_dir*>*/modeltech/examples/stimulus.vhd*

Obtaining the 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 08855-1331

Tel: (800)678-4333 ((908)562-5420 from outside the U.S.)
Fax: (908)981-9667
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 v3.0 accelerated packages are pre-compiled into the **ieee** library in the installation directory.

Note: By default, Model*Sim* is optimized for VITAL v3.0. You can, however, revert to VITAL v2.2b by invoking **vsim** (CR-258) with the **-vital2.2b** option, and by mapping library **vital** to *<install_dir>/modeltech/vital2.2b*.

Model*Sim* 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. Model*Sim* is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, Model*Sim* accelerates the VITAL_Timing and VITAL_Primitives packages. The procedures in these packages are optimized and built into the simulator kernel. By default, **vsim** (CR-258) uses the optimized procedures. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL v3.0).

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-217) automatically checks for VITAL 3.0 compliance on all entities with the VITAL_Level0 attribute set, and all architectures with the VITAL_Level0 or VITAL_Level1 attribute set. It also checks for VITAL 2000 compliance on all architectures using the vital2000 library.

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-217) with the option **-novitalcheck**. It is, of course, possible to turn off compliance checking for VITAL 3.0 as well; 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 3.0 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 just 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 it is not read elsewhere.

You can control the visibility of VITAL compliance-check warnings in your **vcom** (CR-217) 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 (B-396) section.

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

Compiling and Simulating with accelerated VITAL packages

vcom (CR-217) 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.

Compiler options for VITAL optimization

Several **vcom** (CR-217) options control and provide feedback on VITAL optimization:

`-O0 | -O4`
Lower 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.

`-vital2000`
Turns on acceleration for the VITAL 2000 vital_memory package.

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

Util package

The util package is included in ModelSim versions 5.5 and later and serves as a container for various VHDL utilities. The package is part of the modelsim_lib library which is located in the modelsim tree and 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() (4-70)

to_time() (4-71)

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_spy()

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

This system task works only in ModelSim versions 5.5 and newer.

Syntax

```
init_signal_spy( spy_object, dest_object, verbose);
```

Returns

Nothing

Arguments

Name	Type	Description
spy_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/wire. 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 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 spy_object's value is mirrored onto the dest_object. Default is 0, no message.

Related functions

None

Limitations

- When mirroring the value of a Verilog register/wire onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Mirroring slices or single bits of a vector is not supported. If you do reference a slice or bit of a vector, the function will assume that you are referencing the entire vector.

Example

```
library modelsim_lib;
use modelsim_lib.util.all;
entity top is
end;

architecture ...
  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"`.

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() (4-68)

to_time() (4-71)

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() (4-68) 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() (4-68)

to_real() (4-70)

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.

5 - Verilog Simulation

Chapter contents

This chapter describes how to compile and simulate Verilog designs with Model*Sim* Verilog. Model*Sim* Verilog implements the Verilog language as defined by the IEEE Std 1364, and it is recommended that you obtain this specification as a reference manual.

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

- Standard Delay Format (SDF) annotator compatible with many ASIC and FPGA vendor's Verilog libraries
- Value Change Dump (VCD) file extensions for ASIC vendor test tools
- Dynamic loading of PLI/VPI applications
- 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 Model*Sim* VHDL
- Extensions to provide compatibility with Verilog-XL

The following IEEE Std 1364 functionality is partially implemented in Model*Sim* Verilog:

- Verilog Procedural Interface (VPI) (see */<install_dir>/modeltech/docs/technotes/Verilog_VPI.note* for details)

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

- Getting started with projects (2-28)
- Compiling with the graphic interface (8-250)
- Simulating with the graphic interface (8-256)

Model*Sim* variables

Several variables are available to control simulation, provide simulator state feedback, or modify the appearance of the Model*Sim* GUI. To take effect, some variables, such as environment variables, must be set prior to simulation. See *Appendix B - ModelSim Variables* for a complete listing of Model*Sim* variables.

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 Model*Sim* 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** option. 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** option. 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 file "transcript" in the current directory.

Incremental compilation

By default, Model*Sim* 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" (5-90)).

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(n1, a, b);
    and2(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** option, 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 options 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 Model*Sim* 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** option to instruct the compiler to place the results in the **asiclib** library rather than the default **work** library.

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 specify an alternate library with the **-lib** option. All other Verilog instantiations are resolved in the following order:

- Search libraries specified with **-Lf** options in the order they appear on the command line.
- Search the library specified in the "Verilog-XL `uselib compiler directive" (5-81).
- Search libraries specified with **-L** options 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.

It is important to recognize that the work library is not necessarily a library named **work** - 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 if sub-module names overlap among the libraries. In this situation you want the modules to search for their sub-modules in the work library first. This is accomplished by specifying **-L work** first in the list of search libraries.

For example, assume you have a top level module "top" that instantiates module "modA" from library "libA" and module "modB" from library "libB". Furthermore, "modA" and "modB" both instantiate modules named "cellA", but the definition of "cellA" compiled into "libA" is different from that compiled into "libB". In this case, it is insufficient to just specify "-L libA - L libB" as the search libraries because instantiations of "cellA" from "modB" resolve to the "libA" version of "cellA". The appropriate search library options are "-L work -L libA -L libB".

Verilog-XL compatible compiler options

See **vlog** (CR-250) for a complete list of compiler options. The options described here are equivalent to Verilog-XL options. Many of these are provided to ease the porting of a design to Model*Sim* Verilog.

`+define+<macro_name>[=<macro_text>]`
This option allows you to define a macro from the command line that is equivalent to the following compiler directive:

```
‘define <macro_name> <macro_text>
```

Multiple **+define** options are allowed on the command line. A command line macro overrides a macro of the same name defined with the ‘define compiler directive.

`+incdir+<directory>`
This option specifies which directories to search for files included with **‘include** compiler directives. By default, the current directory is searched first and then the directories specified by the **+incdir** options in the order they appear on the command line. You may specify multiple **+incdir** options as well as multiple directories separated by "+" in a single **+incdir** option.

`+delay_mode_distributed`
This option disables path delays in favor of distributed delays. See Delay modes (5-97) for details.

`+delay_mode_path`
This option sets distributed delays to zero in favor of path delays. See Delay modes (5-97) for details.

`+delay_mode_unit`
This option sets path delays to zero and non-zero distributed delays to one time unit. See Delay modes (5-97) for details.

`+delay_mode_zero`
This option sets path delays and distributed delays to zero. See Delay modes (5-97) for details.

`-f <filename>`
This option reads more command line arguments from the specified text file. Nesting of **-f** options is allowed.

`+mindelays`
This option selects minimum delays from the "min:typ:max" expressions. If preferred, you can defer delay selection until simulation time by specifying the same option to the simulator.

`+typdelays`
This option selects typical delays from the "min:typ:max" expressions. If preferred, you can defer delay selection until simulation time by specifying the same option to the simulator.

`+maxdelays`
This option selects maximum delays from the "min:typ:max" expressions. If preferred, you can defer delay selection until simulation time by specifying the same option to the simulator.

`+nowarn<mnemonic>`
This option disables the class of warning messages specified by <mnemonic>. This option only disables warning messages accompanied by a mnemonic enclosed in square brackets. For example,

```
# WARNING: test.v(2): [TFMPC] - Too few port connections.
```

This warning message can be disabled with the **+nowarnTFMPC** option.

`-u`
This option treats all identifiers in the source code as all uppercase.

Options supporting source libraries

The following options support source libraries in the same manner as Verilog-XL. Note that these libraries are source libraries and are very different from the libraries that the Model*Sim* compiler uses to store compilation results. You may find it convenient to use these options if you are porting a design to Model*Sim* or if you are familiar with these options 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>`
This option specifies a source library file containing module and UDP definitions. Modules and UDPs within the file are compiled only if they match previously unresolved references. Multiple **-v** options are allowed.

`-y <directory>`
This option specifies a source library directory containing module and UDP definitions. Files within this directory are compiled only if the file names match the names of previously unresolved references. Multiple **-y** options are allowed.

`+libext+<suffix>`
This option works in conjunction with the **-y** option. It specifies file extensions for the files in a source library directory. By default the compiler searches for files without extensions. If you specify the **+libext** option, then the compiler will search for a file with the suffix appended to an unresolved name. You may specify only one **+libext** option, but it may contain multiple suffixes separated by "+". The extensions are tried in the order they appear in the **+libext** option.

`+librescan`
This option changes how unresolved references are handled that are added while compiling a module or UDP from a source library. By default, the compiler attempts to resolve these references as it continues searching the source libraries. If you specify the **+librescan** option, then the new unresolved references are deferred until after the current pass through the source libraries. They are then resolved by searching the source libraries from the beginning in the order they are specified on the command line.

`+nolibcell`
By default, all modules compiled from a source library are treated as though they contain a **‘celldefine** compiler directive. This option disables this default. The **‘celldefine** directive only affects the PLI Access routines **acc_next_cell** and **acc_next_cell_load**.

`-R <simargs>`
This option instructs the compiler to invoke the simulator after compiling the design. The compiler automatically determines which top level modules are to be simulated. The command line arguments following **-R** are passed to the simulator, not the compiler. Place the **-R** option at the end of the command line or terminate the simulator command line arguments with a single "-" character to differentiate them from compiler command line arguments.

The **-R** option is not a Verilog-XL option, but it is used by Model*Sim* Verilog to combine the compile and simulate phases together as you may be used to doing with Verilog-XL. It is not recommended that you regularly use this option because you will incur the unnecessary overhead of compiling your design for each simulation run. Mainly, it is provided to ease the transition to Model*Sim* Verilog.

Verilog-XL ‘uselib compiler directive

The **‘uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler options. 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 vlog switch (described below).

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>
```

In Verilog-XL, the library references are equivalent to command line options 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 options:

```
-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.

For example, the following code fragment shows how two different modules that have the same name can be instantiated within the same design:

```
‘uselib dir=/h/vendorA file=.v
NAND2 u1(n1, n2, n3);
```

```
‘uselib dir=/h/vendorB file=.v
NAND2 u2(n4, n5, n6);
```

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

-compile_uselibs argument

In Model*Sim* versions 5.5 and later, a vlog argument eases the use of **‘uselib** directives. The **-compile_uselibs** 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, Model*Sim* determines into what 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" (B-393))
- A directory named "mti_uselibs" that is created in the current working directory

pre-5.5 release implementation

In Model*Sim* versions prior to 5.5, the library files referenced by the **‘uselib** directive were not automatically compiled by Model*Sim* Verilog. To maintain backwards compatibility, this is still the default behavior when -compile_uselibs (see above) is not used. The following describes the pre-5.5 release implementation.

Because it is an important feature of **‘uselib** to allow a design to reference multiple modules having the same name, independent compilation of the source libraries referenced by the **‘uselib** directives is required. Each source library should be compiled into its own object library. The compilation of the code containing the **‘uselib** directives only records which object libraries to search for each module instantiation when the design is loaded by the simulator.

Because the **‘uselib** directive is intended to reference source libraries, Model*Sim* Verilog must infer the object libraries from the library references. The rule is to assume an object library named **work** in the directory defined in the library reference **dir=<library_directory>** or the directory containing the file in the library reference **file=<library_file>**. The library reference **libext=<file_extension>** is ignored in the pre-5.5 release implementation. For example, the following **‘uselib** directives infer the same object library:

```
‘uselib dir=/h/vendorA

‘uselib file=/h/vendorA/libcells.v
```

In both cases Model*Sim* Verilog assumes that the library source is compiled into the object library **/h/vendorA/work**.

Model*Sim* Verilog also extends the **‘uselib** directive to explicitly specify the object library with the library reference **lib=<library_name>**. For example,

```
‘uselib lib=/h/vendorA/work
```

The library name can be a complete path to a library, or it can be a logical library name defined with the **vmap** command. Since this usage of **‘uselib** is an extension, it may be desirable to qualify it with an **‘ifdef** to make it portable to other Verilog systems. For example,

```
‘ifdef MODEL_TECH
‘uselib lib=vendorA
‘else
‘uselib dir=/h/vendorA libext=.v
‘endif
```

The MODEL_TECH macro is automatically defined by the Model*Sim* compiler.

Simulation

The Model*Sim* 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
```

Note: When working with designs that contain optimized code, this syntax may vary. Please see "Compiling for faster performance" (5-90) for details.

If a top-level module name is not specified, Model*Sim* will present the **Load Design** dialog box from which you can choose one or more top-level modules. See "Simulating with the graphic interface" (8-256) for more information.

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**.

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.

Simulation 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 simulation resolution limit. The resolution limit defaults to the smallest time precision found among all of the ‘timescale compiler directives in the design. The time precision is the second number in the ‘timescale directive. For example, "10 ps" in the following directive:

```
‘timescale 1 ns / 10 ps
```

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. If the design contains no ‘timescale directives, then the resolution limit defaults to the "resolution" value specified in the *modelsim.ini* file (default is 1 ns). In any case, you can override the default resolution limit by specifying the **-t** option on the command line.

For example, to explicitly choose 100 ps resolution:

```
vsim -t 100ps top
```

This forces 100 ps resolution even if the design has finer time precision. As a result, time values with finer precision are rounded to the nearest 100 ps.

Event order issues

The Verilog language is defined such that the simulator is not required to execute simultaneous events in any particular order. Unfortunately, some models are inadvertently written to rely on a particular event order, and these models may behave differently when ported to another Verilog simulator. A model with event order dependencies is ambiguous and should be corrected. For example, the following code is ambiguous:

```
module top;
    reg r;

    initial r = 0;
    initial r = 1;

    initial #10 $display(r);
endmodule
```

The value displayed for "r" depends on the order that the simulator executes the initial constructs that assign to "r". Conceptually, the initial constructs run concurrently and the simulator is allowed to execute them in any order. Model*Sim* Verilog executes the initial constructs in the order they appear in the module, and the value displayed for "r" is "1". Verilog-XL produces the same result, but a simulator that displays "0" is not incorrect because the code is ambiguous.

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

Tracking down event order dependencies is a tedious task, so Model*Sim* Verilog aids you with a couple of compiler options:

`-compat`
This option turns off optimizations that result in different event ordering than Verilog-XL. Model*Sim* Verilog generally duplicates Verilog-XL event ordering, but there are cases where it is inefficient to do so. Using this option does not help you find the event order dependencies, but it allows you to ignore them. Keep in mind that this option does not account for all event order discrepancies, and that using this option may degrade performance.

`-hazards`
This option detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes.

vsim (CR-258) 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. Model*Sim* calls this a READ/WRITE hazard if it executed the read first.

- WRITE/READ:
  Same as a READ/WRITE hazard except that Model*Sim* 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-250) with the **-hazards** option when you compile your source code and you must also invoke **vsim** with the **-hazards** option when you simulate.

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.

Verilog-XL compatible simulator options

See **vsim** (CR-258) for a complete list of simulator options. The options described here are equivalent to Verilog-XL options. Many of these are provided to ease the porting of a design to Model*Sim* Verilog.

`+alt_path_delays`
Specify path delays operate in inertial mode by default. In inertial mode, a pending output transition is cancelled when a new output transition is scheduled. The result is that an output may have no more than one pending transition at a time, and that pulses narrower than the delay are filtered. The delay is selected based on the transition from the cancelled pending value of the net to the new pending value. The **+alt_path_delays** option modifies the inertial mode such that a delay is based on a transition from the current output value rather than the cancelled pending value of the net. This option has no effect in transport mode (see `+pulse_e/<percent>` and +pulse_r/<percent>).

`-l <filename>`
By default, the simulation log is written to the file "transcript". The **-l** option allows you to specify an alternate file.

`+maxdelays`
This option selects the maximum value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

`+mindelays`
This option selects the minimum value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

`+multisource_int_delays`
This option enables multisource interconnect delays with transport delay behavior and pulse handling. Model*Sim* uses a unique delay value for each driver-to-driven module

interconnect path specified in the SDF file. Pulse handling is configured using the **+pulse_int_e** and **+pulse_int_r** switches (described below).

`+no_neg_tchk`
This option disables negative timing check limits by setting them to zero. By default negative timing check limits are enabled. This is just the opposite of Verilog-XL, where negative timing check limits are disabled by default, and they are enabled with the +neg_tchk option.

`+no_notifier`
This option disables the toggling of the notifier register argument of the timing check system tasks. By default, the notifier is toggled when there is a timing check violation, and the notifier usually causes a UDP to propagate an X. Therefore, the **+no_notifier** option suppresses X propagation on timing violations.

`+no_path_edge`
This option causes ModelSim to ignore the input edge specified in a path delay. The result is that all edges on the input are considered when selecting the output delay. Verilog-XL always ignores the input edges on path delays.

`+no_pulse_msg`
This option disables the warning message for specify path pulse errors. A path pulse error occurs when a pulse propagated through a path delay falls between the pulse rejection limit and pulse error limit set with the **+pulse_r** and **+pulse_e** options. A path pulse error results in a warning message, and the pulse is propagated as an X. The **+no_pulse_msg** option disables the warning message, but the X is still propagated.

`+no_tchk_msg`
This option disables error messages issued by timing check system tasks when timing check violations occur. However, notifier registers are still toggled and may result in the propagation of X's for timing check violations.

`+nosdfwarn`
This option disables warning messages during SDF annotation.

`+notimingchecks`
This option completely disables all timing check system tasks.

`+nowarn<mnemonic>`
This option disables the class of warning messages specified by <mnemonic>. This option only disables warning messages accompanied by a mnemonic enclosed in square brackets. For example,

```
# WARNING: test.v(2): [TFMPC] - Too few port connections.
```

This warning message can be disabled with the **+nowarnTFMPC** option.

`+ntc_warn`
This option enables warning messages from the negative timing constraint algorithm. This algorithm attempts to find a set of delays for the timing check delayed net arguments such that all negative limits can be converted to non-negative limits with respect to the delayed nets. If there is no solution for this set of limits, then the algorithm sets one of the negative limits to zero and recalculates the delays. This process is repeated until a solution is found. A warning message is issued for each negative limit set to zero. By default these warnings are disabled.

`+pulse_e/<percent>`
This option controls how pulses are propagated through specify path delays, where <percent> is a number between 0 and 100 that specifies the error limit as a percentage of the path delay. A pulse greater than or equal to the error limit propagates to the output in transport mode (transport mode allows multiple pending transitions on an output). A pulse less than the error limit and greater than or equal to the rejection limit (see +pulse_r/<percent>) propagates to the output as an X. If the rejection limit is not specified, then it defaults to the error limit. For example, consider a path delay of 10 along with a **+pulse_e/80** option. The error limit is 80% of 10 and the rejection limit defaults to 80% of 10. This results in the propagation of pulses greater than or equal to 8, while all other pulses are filtered. Note that you can force specify path delays to operate in transport mode by using the **+pulse_e/0** option.

`+pulse_int_e/<percent>`
This option is analogous to **+pulse_e**, except it applies to interconnect delays only.

`+pulse_int_r/<percent>`
This option is analogous to **+pulse_r**, except it applies to interconnect delays only.

`+pulse_r/<percent>`
This option controls how pulses are propagated through specify path delays, where <percent> is a number between 0 and 100 that specifies the rejection limit as a percentage of the path delay. A pulse less than the rejection limit is suppressed from propagating to the output. If the error limit is not specified (see **+pulse_e** (5-88)), then it defaults to the rejection limit.

`+pulse_e_style_ondetect`
This option selects the "on detect" style of propagating pulse errors (see `+pulse_e/<percent>`). A pulse error propagates to the output as an X, and the "on detect" style is to schedule the X immediately, as soon as it has been detected that a pulse error has occurred. The "on event" style is the default for propagating pulse errors (see `+pulse_e_style_onevent`).

`+pulse_e_style_onevent`
This option selects the "on event" style of propagating pulse errors (see `+pulse_e/<percent>`). A pulse error propagates to the output as an X, and the "on event" style is to schedule the X to occur at the same time and for the same duration that the pulse would have occurred if it had propagated through normally. The "on event" style is the default for propagating pulse errors.

`+sdf_nocheck_celltype`
By default, the SDF annotator checks that the CELLTYPE name in the SDF file matches the module or primitive name for the CELL instance. It is an error if the names do not match. The **+sdf_nocheck_celltype** option disables this error check.

`+sdf_verbose`
This option displays a summary of the design objects annotated for each SDF file.

`+transport_int_delays`
By default, interconnect delays operate in inertial mode (pulses smaller than the delay are filtered). The **+transport_int_delays** option selects transport mode with pulse control for single-source nets (one interconnect path). In transport mode, narrow pulses are propagated through interconnect delays. This option works independent from **+multisource_int_delays**.

`+transport_path_delays`

By default, path delays operate in inertial mode (pulses smaller than the delay are filtered). The **+transport_path_delays** option selects transport mode for path delays. In transport mode, narrow pulses are propagated through path delays. Note that this option affects path delays only, and not primitives. Primitives always operate in inertial delay mode.

`+typdelays`

This option selects the typical value in min:typ:max expressions. This is the default. This option has no effect if the min:typ:max selection was determined at compile time.

Compiling for faster performance

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

Model*Sim*'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" (5-76)). 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 option allows the compiler to propagate parameters and perform global optimizations. A requirement of using the "-fast" option 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.

Note: If you want to optimize a very large netlist, you should only optimize the cell libraries using the -fast option. (The -forcecode option should also be specified.) The netlist itself should be compiled with the default settings. Optimizing in this manner reduces compilation time and compiler memory usage significantly.

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 in-lined modules.
- Compile the device under test and testbench *without* -fast.
- Create separate work directories for the cell library and the rest of the design.

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.

You can use the **write report** command (CR-281) command and the **-debugCellOpt** argument to **vlog** command (CR-250) to obtain information about which cells have and have not been optimized. **write 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 Transcript window output that identifies why certain cells were not optimized.

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=<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 "Load Design" 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, select **Design > Browse Libraries** to see the modules and their associated secondary names. Also, you can execute the **vdir** command (CR-223) 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 option 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 option 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
r	Enable access to registers (including memories, integer, time, and real types).
n	Enable access to nets.
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.
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).
c	Enable access to library cells. By default any Verilog module bracketed with a ‘celldefine / ‘endcelldefine may be optimized, and debug and PLI access may be limited. This option keeps module cell visibility.

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.

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 that it operates on. 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 a module named "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 option, if the source code is unavailable for any of the modules referenced in a design, then you must instruct the compiler to search libraries for the precompiled modules. 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 (5-81)). If your design exclusively uses ‘uselib directives to reference modules in other libraries, then you don’t need to specify library search options to the compiler.

The library search options supported by the compiler are identical to those supported by the simulator (e.g., -L and -Lf; see Library usage (5-78)). The compiler also searches the libraries in the same order as the simulator (-Lf libraries first, followed by ‘uselib libraries, and finally -L libraries). However, unlike the simulator, the compiler does not search the work library by default.

Note: The library search options you specify to the compiler must also be specified to the simulator when you simulate the design.

Cell Libraries

Model Technology is the first Verilog simulation vendor to pass the ASIC Council's Verilog test suite and achieve 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 Model*Sim* 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. Model*Sim* Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

SDF timing annotation

Model*Sim* Verilog supports timing annotation from Standard Delay Format (SDF) files. See *Chapter 12 - 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 options. These options and directives are compatible with Verilog-XL. Compiler delay mode options 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 option or the **‘delay_mode_distributed** compiler directive.

Path delay mode

In path delay mode the distributed delays are set to zero. Select this delay mode with the **+delay_mode_path** compiler option or the **‘delay_mode_path** compiler directive.

Unit delay mode

In unit delay mode the distributed delays are set to one (the unit is the time_unit specified in the **‘timescale** directive), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler option 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 option or the **‘delay_mode_zero** compiler directive.

System Tasks

The IEEE Std 1364 defines many system tasks as part of the Verilog language, and Model*Sim* 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
			$removal
			$recrem

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

Note: $readmemb and $readmemh match the behavior of Verilog-XL rather than IEEE Std 1364. Specifically, it loads data into memory starting with the lowest address. For example, whether you make the declaration `memory[127:0]` or `memory[0:127]`, Model*Sim* will load data starting at address 0 and work upwards to address 127.

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, but they are not described in the IEEE Std 1364.

`$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 does Verilog-XL.

```
$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.

`$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.

The following system tasks are Verilog-XL system tasks that are not implemented in Model*Sim* 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**.

$init_signal_spy

The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/wire (called the spy_object) onto an existing Verilog register or VHDL signal (called the dest_object). This system task allows you to reference VHDL signals at any level of hierarchy from within a Verilog module; or, reference Verilog registers/wires at any level of hierarchy from within a Verilog module when there is an interceding VHDL block.

This system task works only in ModelSim versions 5.5 and newer.

Syntax

```
$init_signal_spy( spy_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
spy_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/wire. 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 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 spy_object's value is mirrored onto the dest_object. Default is 0, no message.

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.
- Mirroring slices or single bits of a vector is not supported. If you do reference a slice or bit of a vector, the function will assume that you are referencing the entire vector.

Example

```
module ...
...
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"`.

Compiler Directives

Model*Sim* Verilog supports all of the compiler directives defined in the IEEE Std 1364 and some additional Verilog-XL compiler directives for compatibility.

Many of the compiler directives (such as **‘define** and **‘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
`define_nettype
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`timescale
`unconnected_drive
`uselib
```

Model*Sim* 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
`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 (5-97) for details.

```
‘delay_mode_path
```

This directive sets distributed delays to zero in favor of path delays. See Delay modes (5-97) for details.

```
‘delay_mode_unit
```

This directive sets path delays to zero and non-zero distributed delays to one time unit. See Delay modes (5-97) for details.

```
‘delay_mode_zero
```

This directive sets path delays and distributed delays to zero. See Delay modes (5-97) for details.

```
‘uselib
```

This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler options. See Verilog-XL ‘uselib compiler directive (5-81) for details.

The following Verilog-XL compiler directives are silently ignored by Model*Sim* Verilog. Many of these directives are irrelevant to Model*Sim* Verilog, but may appear in code being ported from Verilog-XL.

```
‘accelerate
‘autoexpand_vectornets
‘disable_portfaults
‘enable_portfaults
‘endprotect
‘expand_vectornets
‘noaccelerate
‘noexpand_vectornets
‘noremove_gatenames
‘noremove_netnames
‘nosuppress_faults
‘protect
‘remove_gatenames
‘remove_netnames
‘suppress_faults
```

The following Verilog-XL compiler directives produce warning messages in Model*Sim* Verilog. These are not implemented in Model*Sim* Verilog, and any code containing these directives may behave differently in Model*Sim* Verilog than in Verilog-XL.

```
‘default_trireg_strength
‘signed
‘unsigned
```

Using the Verilog PLI/VPI

The Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) both 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 (5-120)). In addition, you may write your own PLI/VPI applications.

Model*Sim* 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. In version 5.5, the VPI is partially implemented as defined in the IEEE Std 1364. The list of currently supported functionality can be found in the following directory:

```
<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 Model*Sim* 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, Model*Sim* 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 Model*Sim* 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. It is recommended 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 applications" (5-111)). The PLI 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 option to the simulator (multiple options 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 )
  {
      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;
      vpi_register_systf( &systf_data );


      callback.reason    = cbEndOfCompile;
      callback.cb_rtn    = MyEndOfCompCB;
      callback.user_data = 0;
      (void) vpi_register_cb( &callback );


      callback.reason    = cbStartOfSimulation;
      callback.cb_rtn    = MyStartOfSimCB;
      callback.user_data = 0;
      (void) vpi_register_cb( &callback );
  }

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 (5-108).

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 applications

Model*Sim* Verilog uses operating system calls to dynamically load PLI and VPI applications when the simulator loads a design. Therefore, the applications must be compiled and linked for dynamic loading on a specific operating system. The PLI/VPI routines are declared in the include files located in the Model*Sim* <*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.

Windows NT/95/98/2000 platforms

Under Windows Model*Sim* loads a 32-bit dynamically linked library for each PLI/VPI application. The following compile and link steps are used to create the necessary.dll file (and other supporting files) using the Microsoft Visual C/C++ compiler.

```
cl -c -I<install_dir>\modeltech\include app.c
link -dll -export:<init_function> app.obj \
<install_dir>\modeltech\win32\mtipli.lib
```

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 Model*Sim* 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.

Linux platform

Under Linux, ModelSim loads shared objects. Use these **gcc** or **cc** compiler commands to create a shared object:

gcc compiler:

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -shared -E -o app.so app.o
```

cc compiler:

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -shared -E -o app.so app.o
```

Solaris platform

Under SUN Solaris, Model*Sim* loads shared objects. Use these **gcc** or **cc** compiler commands to create a shared object:

gcc compiler:

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -G -B symbolic -o app.so app.o
```

cc compiler:

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -G -B symbolic -o app.so app.o
```

Note: 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 in your current directory you must force Solaris to search the directory. There are two ways you can do this:

- Add "**.** / " before *app.so* in the PLI library specification, or
- Load the path as a UNIX shell environment variable: LD_LIBRARY_PATH= <*library path without filename*>

64-bit Solaris platform

On a 64-bit Sun system, use the following **cc** compiler commands to prepare PLI/VPI code for dynamic linking with Model*Sim*:

```
cc -v -xarch=v9 -O -I$MTI_HOME/include -c app.c
ld -G app.o -o app.so
```

HP700 platform

Model*Sim* loads shared libraries on the HP700 workstation. A shared library is created by creating object files that contain position-independent code (use the **+z** or **-fpic** compiler option) and by linking as a shared library (use the **-b** linker option). Use these **gcc** or **cc** compiler commands:

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 -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.

for HP-UX 11.0 users

If you are building the PLI/VPI library under HP-UX 11.0, you should not specify the "-lc" option to the invocation of ld, since this will cause an incorrect version of the standard C library to be loaded.

In other words, build libraries like this:

```
cc -c +z -I<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o
```

If you receive the error "Exec format error" when the simulator is trying to load a PLI/VPI library, then you have most likely built under 11.0 and specified the "-lc" option. Just rebuild without "-lc" (or rebuild on an HP-UX 10.0 machine).

64-bit HP platform

On a 64-bit HP system, use the following **cc** compiler commands to prepare PLI/VPI code for dynamic linking with Model*Sim*:

```
cc -v +DA2.0W -O -I<install_dir>/modeltech/include -c app.c
ld -G app.o -o app.so
```

IBM RS/6000 platform

Model*Sim* loads shared libraries on the IBM RS/6000 workstation. The shared library must import Model*Sim*'s PLI/VPI symbols, and it must export the PLI or VPI application's initialization function or table. Model*Sim*'s export file is located in the Model*Sim* 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 Model*Sim* can find the symbol when it dynamically loads the shared object.

64-bit RS/60000 platform

Only version 4.3 of AIX supports the 64-bit platform. A gcc 64-bit compiler is not available at this time. The cc commands are as follows:

```
cc -c -q64 -I/<install_dir>/modeltech/include app.c
cc -o app.sl app.o -q64 -bE:app.exp \
-bI:/<install_dir>/modeltech/rs64/mti_exports\
-Wl-G -bnoentry
```

Note: When using AIX 4.3 in 32-bit mode, you must add the switch `-d use_inttypes` to the compile command lines. This switch prevents a name conflict that occurs between *inttypes.h* and *mti.h*.

Using 64-bit ModelSim with 32-bit PLI/VPI Applications

If you have 32-bit PLI/VPI applications and wish to use 64-bit Model*Sim*, 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 following 64-bit porting guides for the appropriate platform:

Sun

Solaris 7 64-bit Developer's Guide

http://docs.sun.com:80/ab2/coll.45.10/SOL64TRANS/

HP

HP-UX 64-bit Porting and Transition Guide

http://docs.hp.com:80/dynaweb/hpux11/hpuxen1a/0462/@Generic__BookView

HP-UX 11.x Software Transition Kit

http://software.hp.com/STK/

IBM

AIX 64-bit Migration Guide

http://www.developer.ibm.com/library/aix4.3/Sun

Specifying the PLI/VPI file to load

The PLI 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 option to the simulator (multiple options 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 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 B - 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 hello()
    {
        io_printf("Hi there\n");
    }
    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.

```
hello.c:

    #include "vpi_user.h"
    static hello()
    {
        vpi_printf( "Hello world!\n" );
    }

    void RegisterMyTfs( void )
    {
        s_vpi_systf_data systf_data;
        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;
        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
```

Note: A general VPI example can be found in <install_dir>/modeltech/examples/vpi.

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 Model*Sim* 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-260) and `-keeploadedrestart` (CR-260) vsim arguments 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 Model*Sim* Verilog.

`reason_force`
Not supported in Model*Sim* Verilog.

`reason_release`
Not supported in Model*Sim* Verilog.

`reason_disable`
Not supported in Model*Sim* 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 Model*Sim* 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)
accRegBit (acc_handle_by_name, acc_handle_tfarg, and acc_handle_itfarg)
```

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 Model*Sim* Verilog. Even without the instructions, it is still likely that you can get it to work with Model*Sim* Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with Model*Sim* 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". To prepare the application for Model*Sim* 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 applications" (5-111)). 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
```

That's all there is to it. The PLI application is ready to be run with Model*Sim* Verilog. All that's left is to specify the resulting object file to the simulator for loading using the Veriuser modesim.ini file entry, the **-pli** simulator option, or the PLIOBJS environment variable (see "Registering PLI applications" (5-108)).

Note: On the HP700 platform, the object files must be compiled as position-independent code by using the **+z** compiler option. 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 Model*Sim* 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

Note: 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 Model*Sim* 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.

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 file 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-258) with the **-trace_foreign** option:

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.

6 - Mixed VHDL and Verilog Designs

Chapter contents

Model*Sim* single-kernel simulation (SKS) allows you to simulate designs that are written in VHDL and/or Verilog. This chapter outlines data mapping and the criteria established to instantiate design units between HDLs.

The boundaries between VHDL and Verilog are enforced at the level of a design unit. This means that although a design unit must be either all VHDL or all Verilog, it may instantiate design units from either language. Any instance in the design hierarchy may be a design unit from either HDL without restriction. SKS technology allows the top-level design unit to be either VHDL or Verilog. As you traverse the design hierarchy, instantiations may freely switch back and forth between VHDL and Verilog.

Separate compilers, common libraries

VHDL source code is compiled by **vcom** (CR-217) and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in a library. Likewise, Verilog source code is compiled by **vlog** (CR-250) and the resulting design units (modules and UDPs) are stored in a library.

Libraries can store any combination of VHDL and Verilog design units, provided the design unit names do not overlap (VHDL design unit names are changed to lower case).

See "Design libraries" (3-41) for more information about library management and see the **vcom** (CR-217) and the **vlog** commands.

Mapping data types

Cross-HDL instantiation does not require any extra effort on your part. As ModelSim loads a design it detects cross-HDL instantiations – made possible because a design unit's HDL type can be determined as it is loaded from a library – and the necessary adaptations and data type conversions are performed automatically.

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. ModelSim automatically maps between the HDL data types as shown below.

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
string	string

The type of a Verilog parameter is determined by its initial value.

VHDL and 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 Model*Sim*. (See *\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'
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 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

VHDL instantiation of Verilog design units

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. In addition, 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 architecture name, but it will be ignored because Verilog modules do not have architectures.

Verilog instantiation criteria

A Verilog design unit may be instantiated from VHDL if it meets the following criteria:

- The design unit is a module (UDPs are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- 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. The interface to the module can be extracted from the library in the form of a component declaration by running **vgencomp** (CR-224). Given a library and module name, **vgencomp** (CR-224) 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 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 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-224) 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 and generic names are not unique unless case is preserved. In this event, **vgencomp** (CR-224) behaves as if the module was compiled with the -93 switch for those names only.

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-224) 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.

VCD output

When creating a VCD file for designs that have bi-directional ports, you first have to use the **splitio** command (see "Extracting the proper stimulus for bidirectional ports" (13-344)). Be aware that VCD file output will vary between a design coded in VHDL and the same design coded in Verilog with timing wrapped in VHDL. The difference occurs because **splitio** generates Extended VCD stimulus files, and the Extended VCD format is supported only for pure VHDL designs.

Verilog instantiation of VHDL design units

You can reference a VHDL entity or configuration from Verilog as though the design unit is a module of the same name (in lower case).

VHDL instantiation criteria

A VHDL design unit may be instantiated from Verilog if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration declaration.
- 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.

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

Named port associations

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 are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. The **defparam** statement is not allowed for setting generic values.

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise.

Verilog does not have the concept of architectures or libraries, so the escaped identifier is employed 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

SDF annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See "SDF for Mixed VHDL and Verilog Designs" (12-336) for more information.

7 - Datasets (saved simulations) and virtuals

Chapter contents

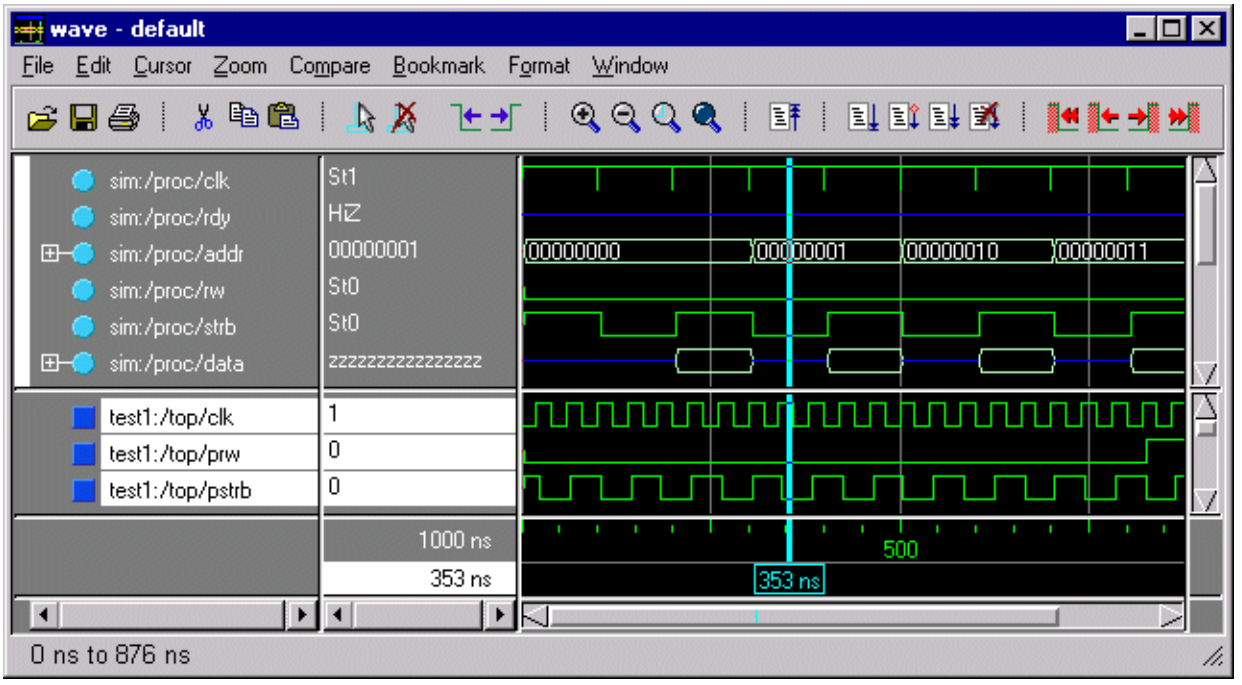
A Model*Sim* simulation can be saved to a logfile (using the `-wlf <filename>` argument to the **vsim** command (CR-258)) for future viewing or comparison to a current simulation. We use the term "dataset" to refer to a logfile that has been reopened in the program.

With Model*Sim* release 5.3 and later, you can open more than one dataset for simultaneous viewing. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different datasets.

Datasets

The term "dataset" refers to a simulation waveform database that was saved and then subsequently reloaded for viewing or comparing. Any number of datasets can be opened in view mode. View mode allows you to view, but not run, a previous simulation.

A prefix identifies each dataset that is opened. The current active simulation is prefixed by "sim," while any datasets loaded for viewing are prefixed by the filename of the logfile. For example, 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 "test1" prefix.



Note: The simulator time resolution (see Resolution (B-400)) must be the same for all datasets you're comparing, including the current simulation.

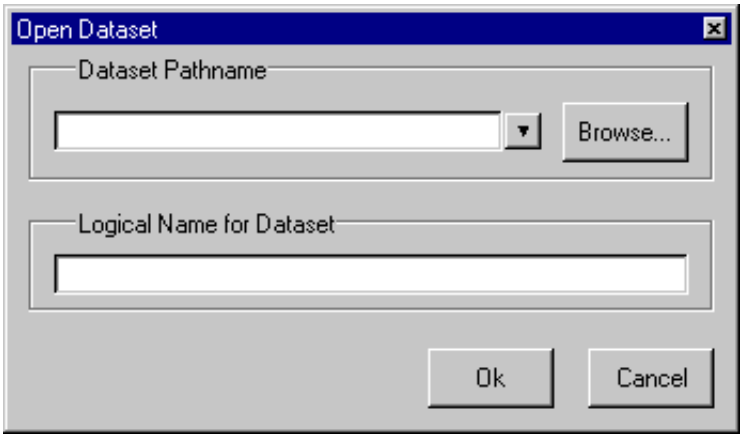
Saving a simulation to a dataset

The results of each simulation run are automatically saved to a dataset 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. Therefore, you should use the `-wlf <filename>` argument to the **vsim** command (CR-258) to specify a different name if you want to save the dataset.

Important: You must end a simulation session with a quit or quit -sim command in order to produce a valid dataset. If you don't end the simulation in this manner, the dataset will not close properly, and ModelSim will issue an error when you try to open the dataset in subsequent sessions.

Opening datasets

To open a dataset, select either **File > Open > Dataset** (Main window) or **File > Open Dataset** (Wave window).



The Open Dataset dialog box includes the following options.

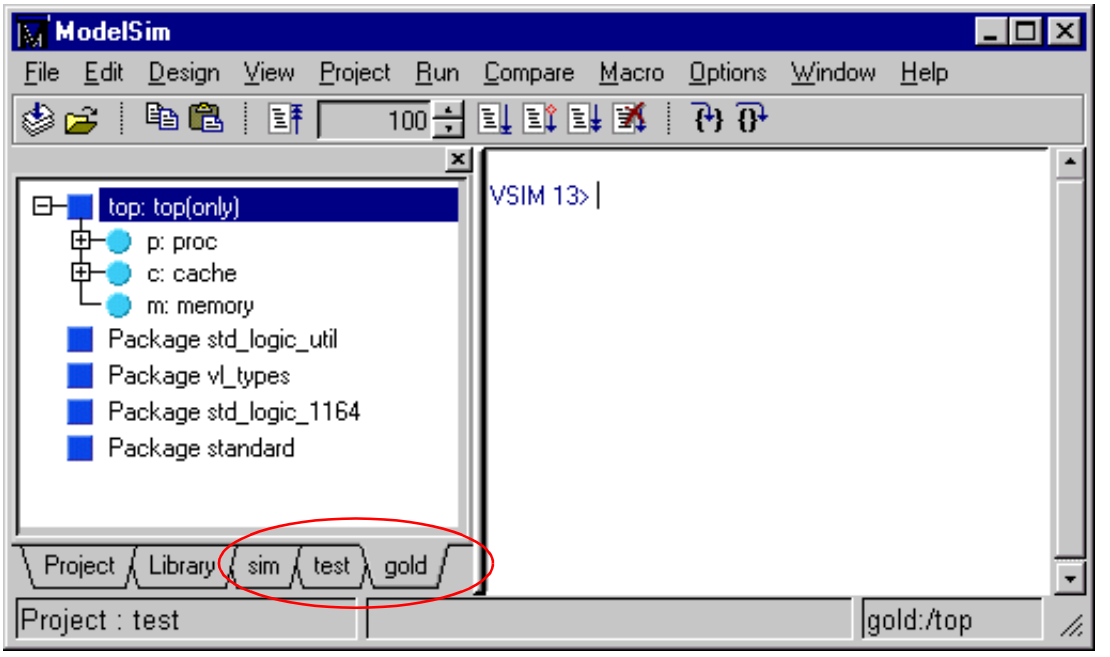
- **Dataset Pathname**
  Identifies the path and filename of the logfile you want to open.
- **Logical Name for Dataset**
  This is the name by which the dataset will be referred. By default this is the filename of the logfile.

Important: You must end a simulation session with a quit or quit -sim command in order to produce a valid dataset. If you don't end the simulation in this manner, the dataset will not close properly, and ModelSim will issue an error when you try to open the dataset in subsequent sessions.

Viewing dataset structure

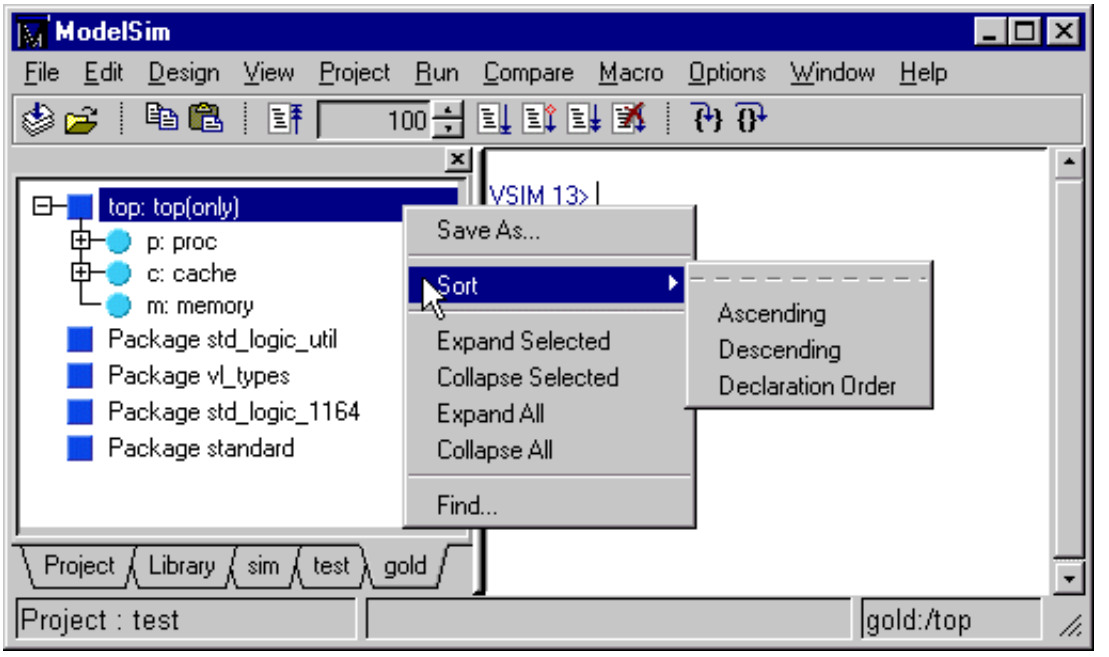
In versions 5.5 and later, each dataset you open creates a Structure page in the Main window workspace. This page contains the same data as the "Structure window" (8-210), but you get one for each dataset.

The graphic below shows three Structure pages: one for the active simulation ("Sim") and one each for two open datasets ("Test" and "Gold").



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 page has a context menu that you access by clicking the right mouse button (Windows—2nd button, UNIX—3rd button) anywhere within the Structure page.

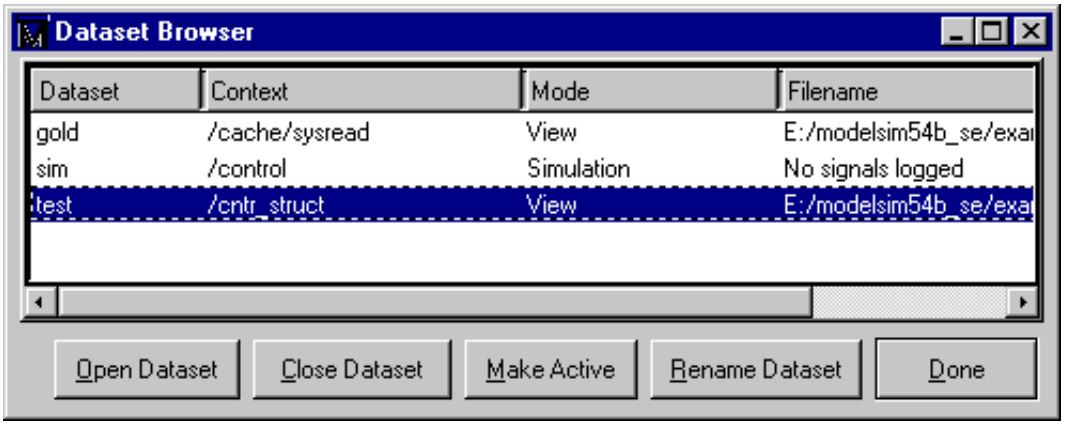


The Structure page context menu includes the following options.

- **Save As**
  Writes the HDL item names in the Structure page to a text file.
- **Sort**
  Sorts the HDL items in the Structure page by alphabetic (ascending or descending) or declaration order.
- **Expand Selected**
  Shows the hierarchy of the selected HDL item.
- **Collapse Selected**
  Hides the hierarchy of the selected HDL item.
- **Expand All**
  Shows the hierarchy of all HDL items in the list.
- **Collapse All**
  Hides the hierarchy of all HDL items in the list.
- **Find**
  Opens the Find dialog. See "Finding items in the Structure window" (8-212) for details.

Managing datasets

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).



The Dataset Browser dialog box includes the following options.

- **Open Dataset**
  Opens the View Dataset dialog box (see "Opening datasets" (7-139)) so you can open additional datasets.
- **Close Dataset**
  Closes the selected dataset. This will also remove the dataset's Structure page 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.
- **Rename Dataset**
  Allows you to assign a new logical name for the selected dataset.

Using datasets with Model*Sim* commands

Multiple datasets can be opened 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. A different dataset name can also be specified 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**

Design regions and signal names can be fully specified over multiple logfiles 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 default dataset. When more than one dataset is open, Model*Sim* will automatically prefix names in the Wave and List window with the dataset name. You can change this default by selecting **Edit > Display Properties** (Wave window) and **Prop > Display Props** (List window).

Model*Sim* 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 page, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the **environment** command (CR-114) to change the active dataset.

Model*Sim* remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command (CR-114), 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 **Options > Edit Preferences** (Main window) command to change the variable value.

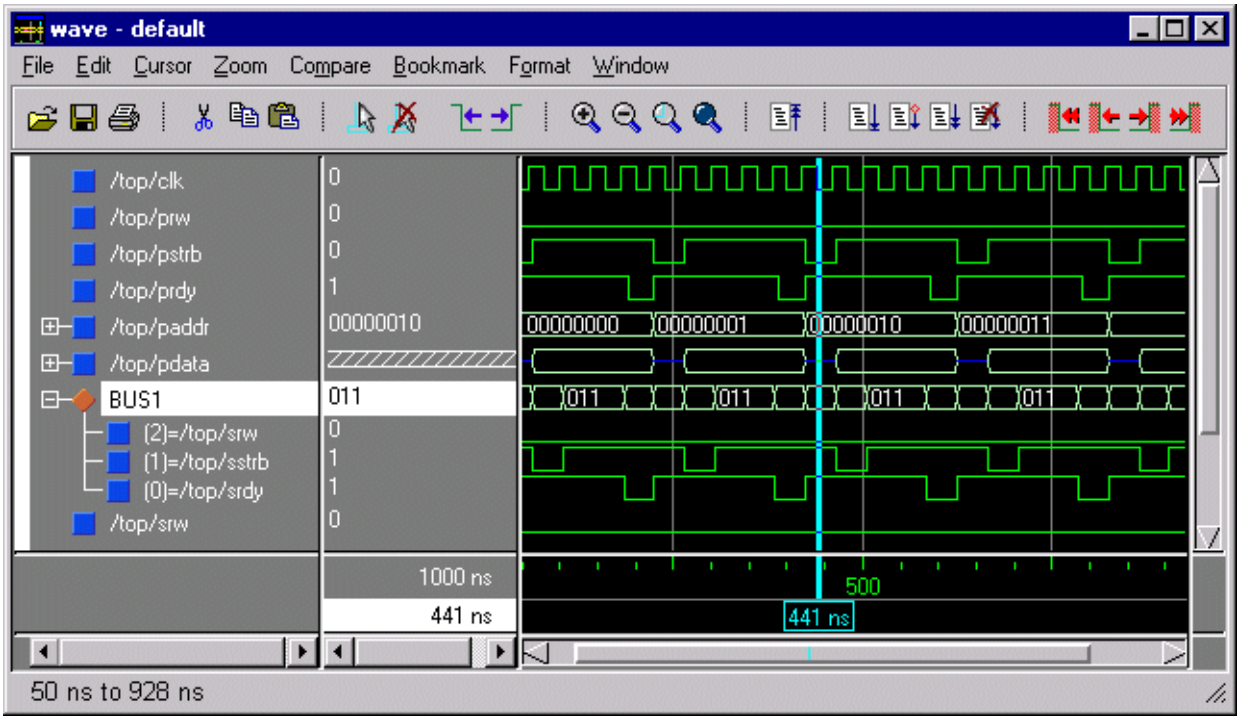
Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-114) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

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 Model*Sim* simulation kernel. Beginning with release 5.3, Model*Sim* supports the following kinds of virtual objects:

- Virtual signals (7-144)
- Virtual functions (7-145)
- Virtual regions (7-146)
- Virtual types (7-146)

Virtual objects are indicated by an orange diamond as illustrated by *BUS1* below:



Virtual signals

Virtual signals are aliases for combinations or subelements of signals written to the logfile 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. Virtual signals can be created via a menu in the Wave and List windows (**Edit > Combine**), or with the **virtual signal** command (CR-245). Virtual signals can also 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-236) 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 logfile, 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 subelements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (CR-243). By default, when quitting, Model*Sim* 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 Model*Sim* 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-115), but cannot be set by the **force** command (CR-121).

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-233).

Virtual functions are also implicitly created by Model*Sim* when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Signals, Wave or List windows. 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-242).

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-248).

Dataset, logfile, and virtual commands

The table below provides a brief description of the actions associated with datasets, logfiles, and virtual commands. For complete details about syntax, arguments, and usage, refer to the *ModelSim Command Reference*.

Command name	Action
dataset close (CR-95)	closes the specified dataset
dataset list (CR-96)	lists all open datasets
dataset open (CR-97)	opens a dataset
dataset rename (CR-98)	assigns a new logical name to the specified dataset
log (CR-131)	creates a logfile for the current simulation
nolog (CR-139)	suspends writing of data to the logfile for the specified signals
searchlog (CR-180)	searches one or more of the currently open logfiles for a specified condition
virtual function (CR-233)	creates a new signal that consists of logical operations on existing signals and simulation time
virtual region (CR-242)	creates a new user-defined design hierarchy region
virtual signal (CR-245)	creates a new signal that consists of concatenations of signals and subelements
virtual type (CR-248)	creates a new enumerated type
vsim (CR-258) -wlf <filename>	creates a logfile for the simulation which can be reopened as a dataset

8 - Model*Sim* Graphic Interface

Chapter contents

The example graphics in this chapter illustrate Model*Sim*'s graphic interface within a Windows environment; however, Model*Sim*'s interface is designed to provide consistency across all supported platforms. Your operating system provides the basic window-management frames, while Model*Sim* controls all internal window features such as menus, buttons, and scroll bars.

Because Model*Sim*'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 Model*Sim* simulation and debugging environment consists of nine window types. Multiple windows of each type can be used during simulation (with the exception of the Main window). To make an additional window select **View > New** (Main window). A brief description of each window follows:

- Main window (8-157)
  The initial window that appears upon startup. All subsequent Model*Sim* windows are opened from the Main window. This window contains the session transcript.
- Dataflow window (8-171)
  Lets you trace signals and nets through your design by showing related processes.
- List window (8-175)
  Shows the simulation values of selected VHDL signals and variables and Verilog nets and register variables in tabular format.
- Process window (8-190)
  Displays a list of processes in the region currently selected in the Structure window.
- Signals window (8-193)
  Shows the names and current values of VHDL signals, and Verilog nets and register variables in the region currently selected in the Structure window.
- Source window (8-201)
  Displays the HDL source code for the design. (Your source code can remain hidden if you wish, see "Source code security and -nodebug" (E-433).)
- Structure window (8-210)
  Displays the hierarchy of structural elements such as VHDL component instances, packages, blocks, generate statements, and Verilog model instances, named blocks, tasks and functions. In versions 5.5 and later, this same information is displayed in the Main window workspace.
- Variables window (8-213)
  Displays VHDL constants, generics, variables, and Verilog register variables in the current process and their current values.
- Wave window (8-216)
  Displays waveforms, and current values for the VHDL signals and variables and Verilog nets and register variables 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 (8-152)	Main, Source, and Wave windows
Drag and Drop (8-152)	Dataflow, List, Signals, Source, Structure, Variables, and Wave windows
Command history (8-152)	Main window command line
Automatic window updating (8-153)	Dataflow, Process, Signals, and Structure windows
Finding names, searching for values, and locating cursors (8-153)	various windows
Sorting HDL items (8-154)	Process, Signals, Source, Structure, Variables and Wave windows
Multiple window copies (8-154)	all windows except the Main window
Menu tear off (8-154)	all windows
Customizing menus and buttons (8-154)	all windows
Combining signals into a user-defined bus (8-154)	List and Wave windows
Tree window hierarchical view (8-155)	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)

- The Main window includes context menus that are accessed by clicking the right mouse button.
- 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

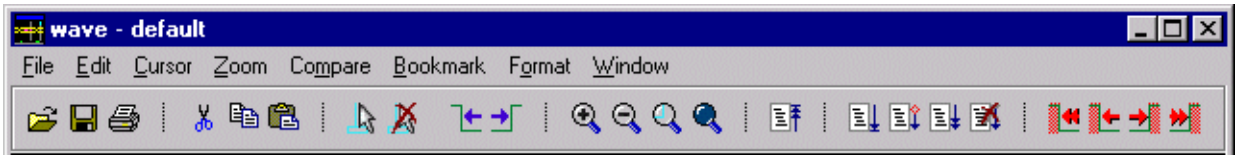
  –contents of the clipboard.

Transcript window context menu

Note: 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 window will ONLY paste from the clipboard.
- All menus highlight their accelerator keys.

Quick access toolbars



Buttons on the Main, Source, and Wave windows provide access to commonly used commands and functions. See, "The Main window toolbar" (8-166), "The Source window toolbar" (8-204), and "The Wave window toolbar" (8-224).

Drag and Drop

Drag and drop of HDL 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, Signals, Source, Structure, Variables, and Wave windows
- **Drop items into these windows:**
  Dataflow, List, and Wave windows

Note: Drag and drop works to rearrange items *within* the List and Wave windows as well.

Command history

Avoid entering long commands twice; use the down and up keyboard arrows to move through the command history for the current simulation.

Automatic window updating

Selecting an item in the following windows automatically updates other related Model*Sim* windows as indicated below:

Select an item in this window	To update these windows
Dataflow window (8-171) (with a process selected in the center of the window)	Process window (8-190)
	Signals window (8-193)
	Source window (8-201)
	Structure window (8-210)
	Variables window (8-213)
Process window (8-190)	Dataflow window (8-171)
	Signals window (8-193)
	Structure window (8-210)
	Variables window (8-213)
Signals window (8-193)	Dataflow window (8-171)
Structure window (8-210)	Process window (8-190)
	Signals window (8-193)
	Source window (8-201)

Finding names, searching for values, and locating cursors

- **Find** HDL item names with the **Edit** > **Find** menu selection in these windows: List, Process, Signals, Source, Structure, Variables, and Wave windows.
- **Search** for HDL item values with the **Edit** > **Search** menu selection in these windows: List, and Wave windows.

You can also:

- **Locate** time markers in the List window with the **Markers** > **Goto** menu selection.
- **Locate** time cursors in the Wave window with the **Cursor** > **Goto** menu selection.

In addition to the menu selections above, the virtual event **<<Find>>** is defined for all windows. The default binding is to **<Key-F19>** in most windows (the Find key on a Sun keyboard). You can bind **<<Find>>** to other events with the Tcl/Tk command **event add**. For example,

```
event add <<Find>> <control-Key-F>
```

Sorting HDL items

Use the **Edit > Sort** menu selection in the windows below to sort HDL items in ascending, descending or declaration order.

Process, Signals, Structure, Variables and Wave windows

Names such as net_1, net_10, and net_2 will sort numerically in the Signals and Wave windows.

Multiple window copies

Use the **View > New** menu selection from the Main window (8-157) to create multiple copies of the same window type. The new window will become the default window for that type.

Context menus

Context menus refer to menus that "pop-up" in the middle of the interface by clicking the right mouse button (Windows—2nd button, UNIX—3rd 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.

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

- "Customizing the interface" (8-279),
- "Customizing menus and buttons" (8-154), and
- "The Button Adder" (8-269) for more information.

Combining signals into a user-defined bus

You can collect selected items in the List window (8-175) and Wave window (8-216) displays and combine them into a bus named by you. In the List window, the **Edit > Combine** menu selection allows you to move the selected items to the new bus as long as they are all scalars or arrays of the same base type (records are not yet supported).

In the Wave window (8-216), the **Edit > Combine** menu selection requires all selected items to be either all scalars or all arrays of the same size. The benefit of this added restriction is that the bus can be expanded to show each element as a separate waveform. Using the **flatten** option allows scalars and various array sizes to be mixed, but foregoes display of child waveforms.

The **keep** option in both windows copies the signals rather than moving them.

Tree window hierarchical view

ModelSim provides a hierarchical, or "tree view" of some aspects of your design in the Main window Structure pages and the Structure, Signals, Variables, and Wave windows.

HDL items you can view

Depending on which window you are viewing, one entry is created for each of the following VHDL and Verilog HDL items within the design:

VHDL items

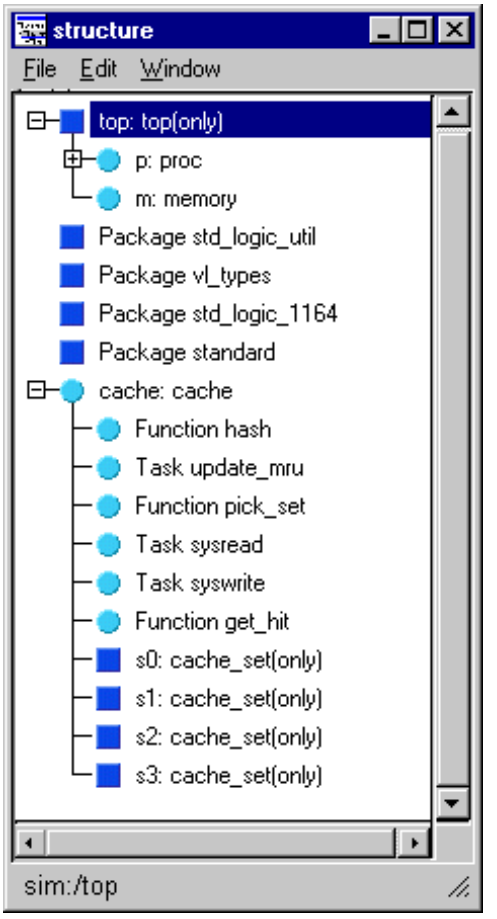
(indicated by a dark blue square icon) signals, variables, component instantiations, generate statements, block statements, and packages

Verilog items

(indicated by a lighter blue circle icon) parameters, registers, nets, module instantiations, named forks, named begins, tasks, and functions

Virtual items

(indicated by an orange diamond icon) virtual signals, buses, and functions, see "Virtual Objects (User-defined buses, and more)" (7-144) for more information



Viewing the hierarchy

Whenever you see a tree view, as in the Structure window displayed here, 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

Finding items within tree windows

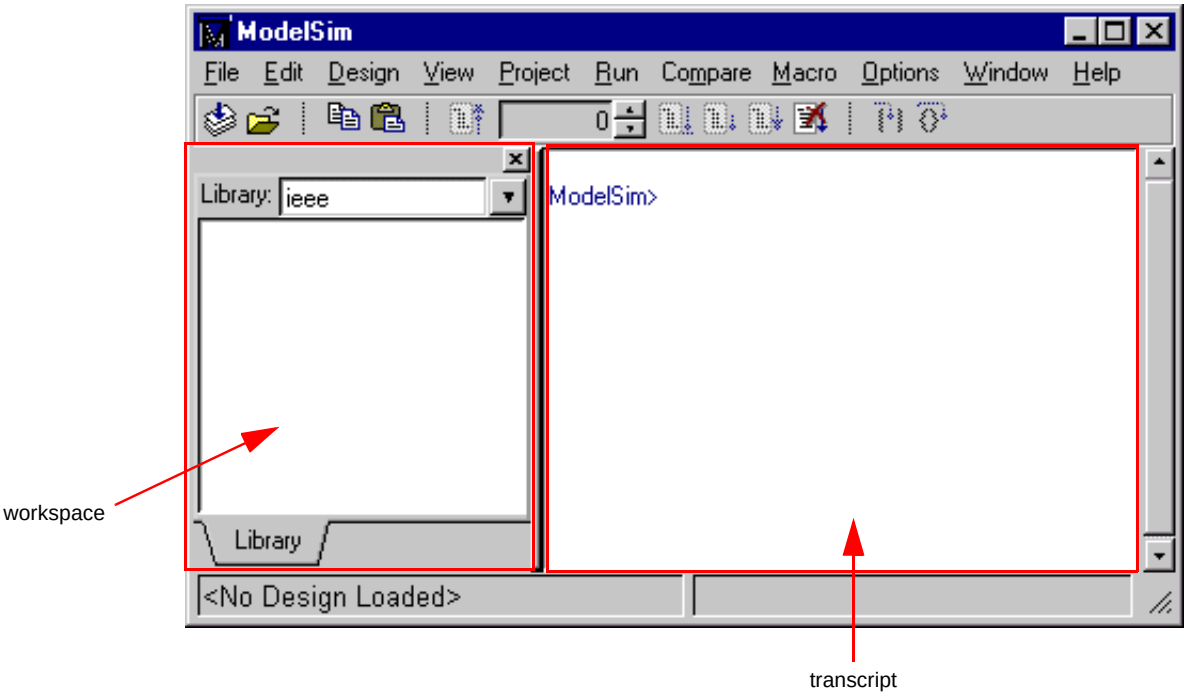
You can open the Find dialog box within all windows (except the Dataflow windows) by selecting **Edit > Find** or by using **<control-s>** (Unix) or **<control-f>** (Windows).

Options within the Find dialog box allow you to search unique text-string fields within the specific window. See also,

- "Finding items by name in the List window" (8-185),
- "Finding HDL items in the Signals window" (8-198), and
- "Finding items by name or value in the Wave window" (8-237).

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; Model*Sim* handles all internal-window features including menus, buttons, and scroll bars.

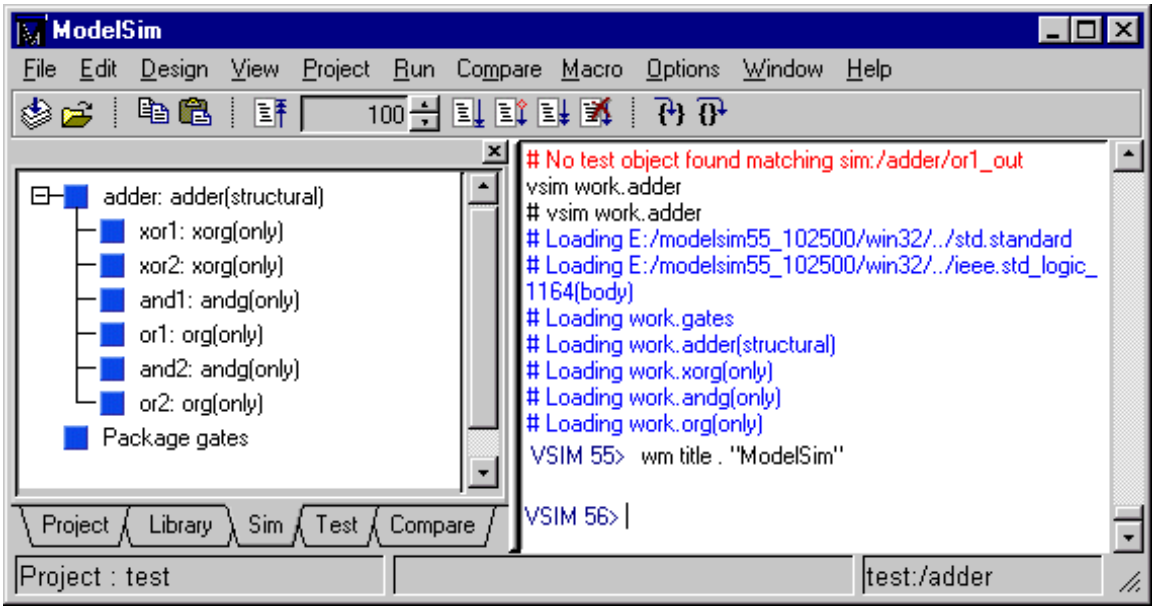


The menu bar at the top of the window provides access to a wide variety of simulation commands and Model*Sim* preferences. The toolbar provides buttons for quick access to the many common commands. The status bar at the bottom of the window gives you information about the data in the active Model*Sim* window. The menu bar, toolbar, and status bar are described in detail below.

Workspace

The workspace is available in software versions 5.5 and later. It provides convenient access to projects, compiled design units, and simulation/dataset structures. It can be hidden or displayed by selecting the **View > Hide/Show Workspace** command.

The workspace can display three types of pages, as shown in the graphic below.



- **Project page**
  Shows all files that are included in the open project. See *Chapter 2 - Projects and system initialization* for details.
- **Library page**
  Shows compiled design units in the specified library. See "Managing library contents" (3-44) for details.
- **Structure pages**
  Shows a hierarchical view of the active simulation and any open datasets. This is the same data that is displayed in the "Structure window" (8-210). There is one page for the current simulation and one page for each open dataset. See "Viewing dataset structure" (7-140) for details.
- **Compare page**
  Shows comparison objects that were created by doing a waveform comparison. See *Chapter 11 - Waveform Comparison* 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 Model*Sim*. 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 in the Transcript and Source windows" (8-168) 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 *modelsim.tcl*, or TranscriptFile in *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 > Save Transcript As**, or **File > 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-104) for more information.

The Main window menu bar

The menu bar at the top of the Main window lets you access many Model*Sim* commands and features. The menus are listed below with brief descriptions of each command's use.

File menu

New	provides three options: Folder – create a new folder in the current directory Source – create a VHDL, Verilog, or Other source file Project – create a new project
Open	provides three options: File – open the selected hdl file Project – open the selected .mpf project file Dataset – open the specified logfile and assign it the specified dataset name
Close	provides three options: Project – close the currently open project file Dataset – close the specified dataset
Delete	provides one option: Project – delete the selected .mpf project file
Change Directory	change to a different working directory
Save Transcript	save the current contents of the transcript window to the file indicated with a "Save Transcript As" selection (this selection is not initially available because the transcript is written to the *transcript* file by default), see "Saving the Main window transcript file" (8-159)
Save Transcript As...	save the current contents of the transcript window to a file
Clear Transcript	clear the Main window transcript display
Options (all options are set for the current session only)	Transcript File: set a transcript file to save for this session only Command History: 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 status bar Model*Sim* Prompt: change the title of the Model*Sim* prompt VSIM Prompt: change the title of the VSIM prompt Paused Prompt: change the title of the Paused prompt
<path list>	a list of the most recent working directory changes
Quit	quit Model*Sim*

Edit menu

Copy	copy the selected text
Paste	paste the previously cut or copied text to the left of the currently selected text
Select All	select all text in the Main window transcript
Unselect All	deselect all text in the Main window transcript
Find	search the transcript forward or backward for the specified text string
Breakpoints	open the Breakpoints dialog box; see "Setting file-line breakpoints" (8-205) for details

Design menu

Browse Libraries	browse all libraries within the scope of the design; see also "Managing library contents" (3-44)
Create a New Library	create a new library or map a library to a new name; see "Creating a library" (3-43)
Import Library	import FPGA libraries; see "Importing FPGA libraries" (3-53) for details
Compile	compile HDL source files into the current project's work library
Load Design	initiate simulation by specifying the top level design unit in the Design tab; specify HDL specific simulator settings with the VHDL and Verilog tabs; specify the library to search for design units instantiated from Verilog with the Libraries tab; specify settings relating to the annotation of design timing with the SDF tab
End Simulation	end the simulation (returns to the Model*Sim* command line)

View menu

All	open all ModelSim windows
Hide/Show Workspace	hide or show the workspace
Layout Style[a]	provides five options: Default - restore the window layout to that used for versions 5.5 and later Classic - restore the window layout to that used in versions prior to 5.5 Cascade - Cascade all open windows Horizontal - Tile all open windows horizontally Vertical - Tile all open windows vertically
Source	open and/or view the Source window (8-201)
Structure	open and/or view the Structure window (8-210)
Variables	open and/or view the Variables window (8-213)
Signals	open and/or view the Signals window (8-193)
List	open and/or view the List window (8-175)
Process	open and/or view the Process window (8-190)
Wave	open and/or view the Wave window (8-216)
Dataflow	open and/or view the Dataflow window (8-171)
Datasets	open the Dataset Browser for selecting the current Dataset
New	create a new window of the specified type
Other	if the Performance Analyzer and/or Code Coverage is turned on, this selection will allow viewing of: Hierarchical Profile, Ranked Profile, and Source Coverage

a.You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Options > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Project menu

Compile Order	set the compile order of the files in the open Project; see "Changing compile order" (2-34) for details
Compile All	compile all files in the open Project; see "Step 3 — Compile the files" (2-32) for details
Add File to Project	add file(s) to the open Project; see "Step 2 — Add files to the project" (2-31) for details

Run menu

Run <default>	run simulation for one default run length; change the run length with **Options > Simulation**, or use the Run Length text box on the toolbar
Run -All	run simulation until you stop it; see also the **run** command (CR-176)
Continue	continue the simulation; see also the **run** command (CR-176) and the **-continue** option
Run -Next	run to the next event time
Step	single-step the simulator; see also the **step** command (CR-187)
Step -Over	execute without single-stepping through a subprogram call
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, and virtual definitions; see also the **restart** command (CR-170)

Compare menu

Start Comparison	start a new comparison
Comparison Wizard	receive step-by-step assistance while creating a waveform comparison
Run Comparison	compute differences from time zero until the end of the simulation
End Comparison	stop difference computation and close the currently open comparison
Add	provides three options: Compare by Signal - specify signals for comparison Compare by Region - designate a reference region for a comparison Clocks - define clocks to be used in a comparison
Options	set options for waveform comparisons
Differences	provides four options: Clear - clear all differences from the Wave window Show - display differences in a text format in the Main window Transcript Save - save computation differences to a file that can be reloaded later Write Report - save computation differences to a text file

Rules	provides two options: Show - display the rules used to set up the waveform comparison Save - save rules for waveform comparison to a file
Reload	load saved differences and rules files

Macro menu

Execute Macro	browse for and execute a DO file (macro)
Execute Old PE Macro	call and execute an old PE 4.7 macro without changing the macro to SE 5.5; backslashes can be selected as pathname delimiters
Convert Old PE Macro	convert old PE 4.7 macro to SE 5.5 macro without changing the file; backslashes can be selected as pathname delimiters
Macro Helper	**UNIX only** - invoke the Macro Helper tool; see also "The Macro Helper" (8-270)
Tcl Debugger	invoke the Tcl debugger, TDebug; see also "The Tcl Debugger" (8-271)
TclPro Debugger	invoke the TclPro Debugger by Scriptics® if installed. TclPro Debugger can be acquired from Scriptics at www.scriptics.com.

Options menu

Compile	set both VHDL and Verilog compile options; see "Setting default compile options" (8-252)
Simulation	set various simulation options; see "Setting default simulation options" (8-265)
Edit Preferences	set various preference variables; see http://www.model.com/resources/pref_variables/frameset.htm
Save Preferences	save current Model*Sim* settings to a Tcl preference file; see http://www.model.com/resources/pref_variables/frameset.htm

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window

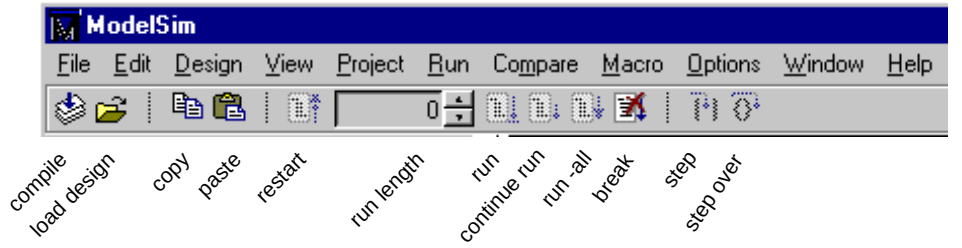
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

Help menu

About Model*Sim*	display Model*Sim* application information (e.g., software version)
Release Notes	view current release notes with the Model*Sim* **notepad** (CR-141)
Enable Welcome	enable the Welcome screen for starting a new project or opening an existing project when Model*Sim* is initiated
Welcome Menu	open the Welcome screen
Information about Help	view the readme file pertaining to Model*Sim*'s online documentation
SE Documentation	open and read Model*Sim* documentation in PDF or HTML format; PDF files can be read with a free Adode Acrobat reader available on the Model*Sim* installation CD or from www.adobe.com
Tcl Help	open the Tcl command reference (man pages) in Windows help format
Tcl Syntax	open Tcl syntax details in HTML format
Tcl Man Pages	open the Tcl /Tk 8.0 manual in HTML format
Technotes	select a technical note to view from the drop-down list

The Main window toolbar

Buttons on the Main window toolbar give you quick access to these Model*Sim* commands and functions.



Main window toolbar buttons

Button	Menu equivalent	Command equivalents
Compile open the Compile HDL Source Files dialog box to select files for compilation	Design > Compile, also Options > Compile (opens the Compile Options dialog box)	**vcom** <arguments>, or **vlog** <arguments> see: **vcom** (CR-217) or **vlog** (CR-250)
Load Design open the Load Design dialog box to initiate simulation	Design > Load Design	**vsim** <arguments> see: **vsim** (CR-258)
Copy copy the selected text within the Main window transcript	Edit > Copy	see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168)
Paste paste the copied text to the cursor location	Edit > Paste	see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168)
Restart reload the design elements and resets the simulation time to zero, with the option of using current formatting, breakpoints, and logfile	Run > Restart	**restart** <arguments> see: **restart** (CR-170)
Run Length specify the run length for the current simulation	none	**run** <specific run length> see: **run** (CR-176)

Main window toolbar buttons			
Button		**Menu equivalent**	**Command equivalents**
	**Run** run the current simulation for the specified run length	Run > Run <default_run_length>	**run** (no arguments) see: **run** (CR-176)
	**Continue Run** continue the current simulation run until the end of specified run length or until it hits a breakpoint or specified break event	Run > Continue	**run -continue** see: **run** (CR-176)
	**Run -All** run the current simulation forever, or until it hits a breakpoint or specified break event	Run > Run -All	**run -all** see: **run** (CR-176), see "Assertion settings page" (8-266)
	**Break** stop the current simulation run	none	none
	**Step** step the current simulation to the next HDL statement	Run > Step	**step** see: **step** (CR-187)
	**Step Over** HDL statements are executed but treated as simple statements instead of entered and traced line by line	Run > Step -Over	**step -over** see: **step** (CR-187)

The Main window status bar

Fields at the bottom of the Main window provide the following information about the current simulation:

Field	Description
Now	the current simulation time, using the default resolution units (see "Simulating with the graphic interface" (8-256)), or a larger time unit if one can be used without a fractional remainder
Delta	the current simulation iteration number
<dataset name>	name of the current dataset (item selected in the Structure window (8-210))

Mouse and keyboard shortcuts in the Transcript and Source windows

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 Model*Sim* 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 >	none	paste clipboard
< middle-button - press > + drag	none	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
< 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 >		move cursor left
< control - d >		delete character to the right
< control - e >, < end >	< end >	move cursor to the end of the line
< control - f >		move cursor right one character
< control - k >		delete to the end of line
< control - n >		move cursor one line down (Source window only under Windows)
< control - o >	none	insert a newline character in front of the cursor
< control - p >		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 to the right of the cursor
< control - u >		delete line

Keystrokes - UNIX	Keystrokes - Windows	Result
< control - v >	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	move cursor up one screen
< Meta - w>	< control - c >	copy selection
< F8 >		search for the most recent command that matches the characters typed (Main window only)

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.

Dataflow window

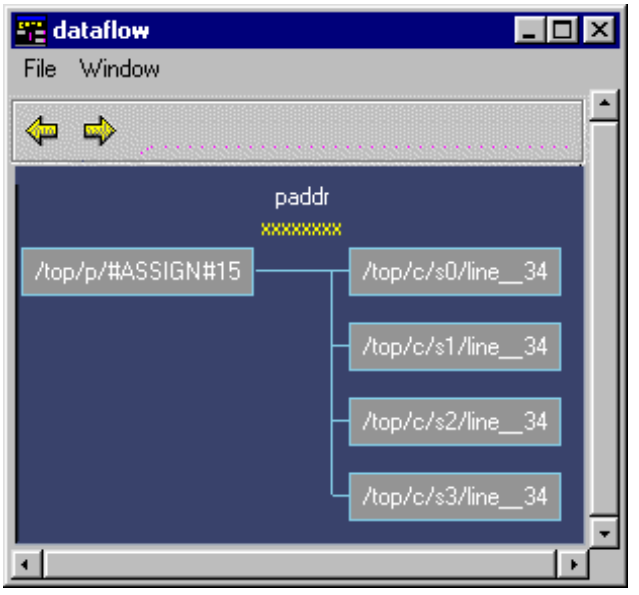
The Dataflow window allows you to trace VHDL signals or Verilog nets and registers through your design. Double-click an item with the left mouse button to move it to the center of the Dataflow display.

VHDL signals or processes in the Dataflow window:

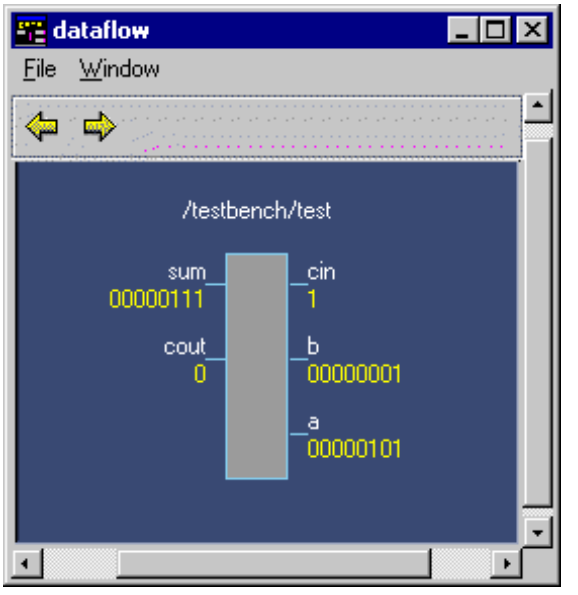
- A signal is displayed in the center of the window with all the processes that drive the signal on the left, and all the processes that read the signal on the right.
- A process is displayed with all the signals read by the process shown as inputs on the left of the window, and all the signals driven by the process on the right.

Verilog nets/registers or processes in the Dataflow window:

- A net or register is displayed in the center of the window with all the processes that drive the net or register on the left, and all the processes triggered by the net or register on the right.
- A process is displayed with all the nets or registers that trigger the process shown as inputs on the left of the window, and all the nets or registers driven by the process on the right.



signal, net, register



process

Link to active cursor in Wave window

In versions 5.5 and later, the value of a signal, net, or register in the Dataflow window is linked to the active cursor in the Wave window. As you move the active cursor in the Wave window, the value of the signal, net, or register in the Dataflow window will update.

Dataflow window menu bar

The following menu commands and button options are available from the Dataflow window menu bar.

File menu

Save Postscript	save the current dataflow view as a Postscript file; see "Saving the Dataflow window as a Postscript file" (8-174)
Selection	Selection > Follow Selection updates the Dataflow window when the Process window (8-190) or Signals window (8-193) changes; Selection > Fix Selection freezes the view selected from within the Dataflow window
Close	close this copy of the Dataflow window; you can create a new window with View > New from the "The Main window menu bar" (8-160)

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

Tracing HDL items with the Dataflow window

The Dataflow window is linked with the Signals window (8-193) and the Process window (8-190). To examine a particular process in the Dataflow window, click on the process name in the Process window. To examine a particular HDL item in the Dataflow window, click on the item name in the Signals window.

With a signal in the center of the Dataflow window, you can:

- click once on a process name in the Dataflow window to make the Source, Process, Signals, and Variable windows update to show that process,
- click twice on a process name in the Dataflow window to move the process to the center of the Dataflow window

With a process in the center of the Dataflow window, you can:

- click twice on an item name to move that item to the center of the Dataflow window.

The backward and forward buttons on the toolbar are analogous to Back and Forward buttons in a web browser. They move backward or forward through previous views of the dataflow.

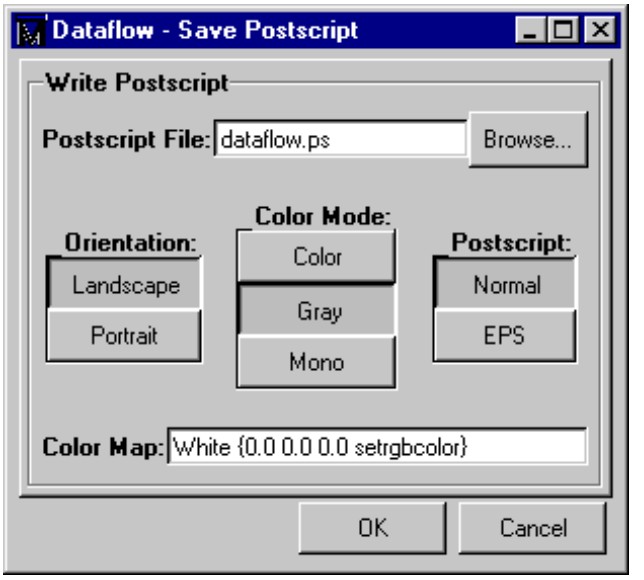
	move backward through dataflow views
	move forward through dataflow views

The Dataflow window will display the current process when you single-step or when Model*Sim* hits a breakpoint.

Saving the Dataflow window as a Postscript file

Select **File > Save Postscript** (Dataflow window) to save the current Dataflow view as a Postscript file. Configure the Postscript output with the following dialog box, or use the **Options > Edit Preferences** (Main window) command.

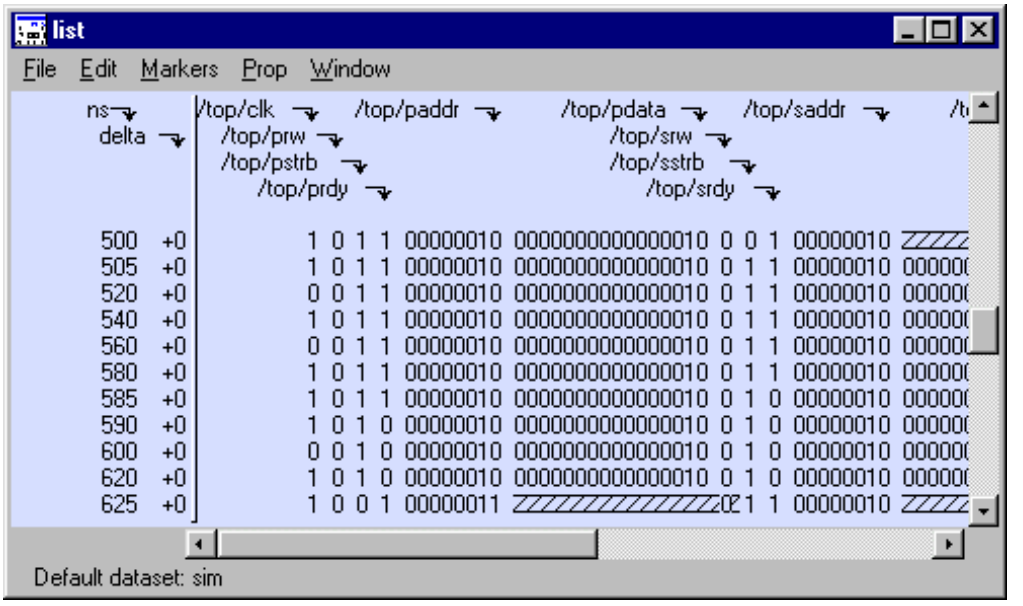
The dialog box has the following options:



- **Postscript File**
  specify the name of the file to save, default is *dataflow.ps*
- **Orientation**
  specify **Landscape** (horizontal) or **Portrait** (vertical) orientation
- **Color Mode**
  specify **Color** (256 colors), **Gray** (gray-scale) or **Mono** (monochrome) color mode
- **Postscript**
  specify Normal Postscript or EPS (Encapsulated Postscript) file type
- **Color Map**
  specify the color mapping from current Dataflow window colors to Postscript colors

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.



HDL items you can view

One entry is created for each of the following VHDL and Verilog HDL items within the design:

- *VHDL items*
  signals and process variables
- *Verilog items*
  nets and register variables
- Comparison items
  comparison regions and comparison signals; see *Chapter 11 - Waveform Comparison* for more information
- Virtual items
  Virtual signals and functions

Note: Constants, generics, and parameters are not viewable in the List or Wave windows.

The List window menu bar

The following menu commands are available from the List window menu bar.

File menu

Write List (format)	save the listing as a text file in one of three formats: tabular, events, or TSSI
Load Format	run a List window format DO file previously saved with Save Format
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
Close	close this copy of the List window; you can create a new window with View > New from the "The Main window menu bar" (8-160)

Edit menu

Cut	cut the selected item field from the listing; see "Editing and formatting HDL items in the List window" (8-181)
Copy	copy the selected item field
Paste	paste the previously cut or copied item to the left of the currently selected item
Delete	delete the selected item field
Combine	combine the selected fields into a user-defined bus; keep copies of the original items rather than moving them; see "Combining signals into a user-defined bus" (8-154)
Select All	select all signals in the List window
Unselect All	deselect all signals in the List window
Find	find the specified item label within the List window
Search	search the List window for a specified value, or the next transition for the selected signal

Markers menu

Add Marker	add a time marker at the currently selected line
Delete Marker	delete the selected marker from the listing
Goto	choose the time marker to go to from a list of current markers

Prop menu

Display Props	set display properties for all items in the window: delta settings, trigger on selection, strobe period, label size, and dataset prefix
Signal Props	set label, radix, trigger on/off, and field width for the selected item

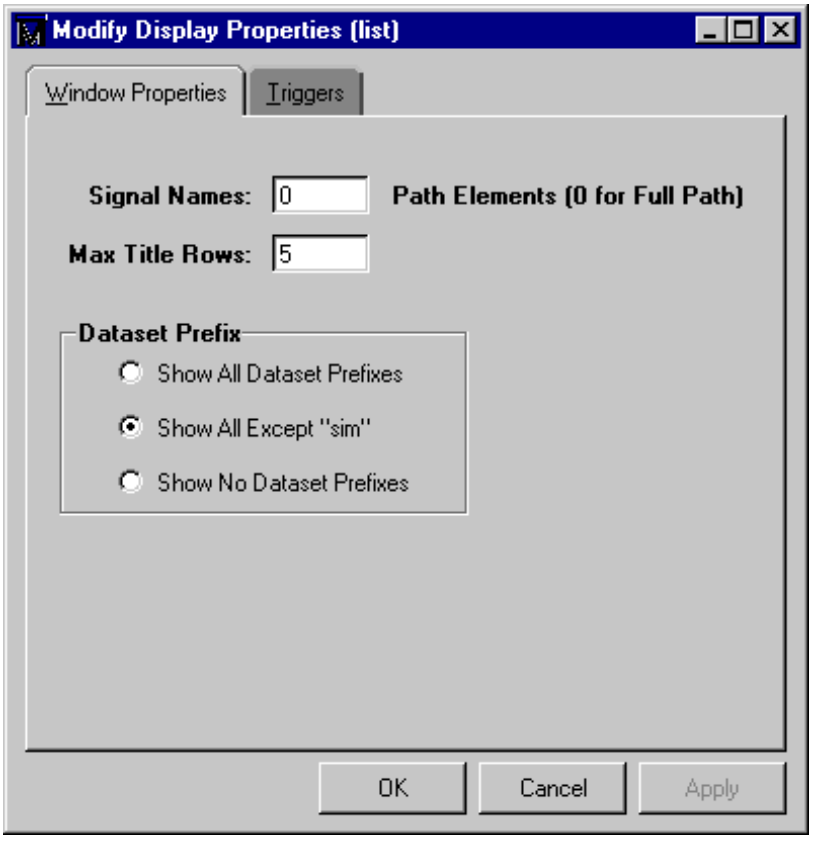
Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

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 **Prop > Display Props** (List window). The resulting Modify Display Properties dialog box contains options for Trigger Settings and Window Properties.

Window Properties page



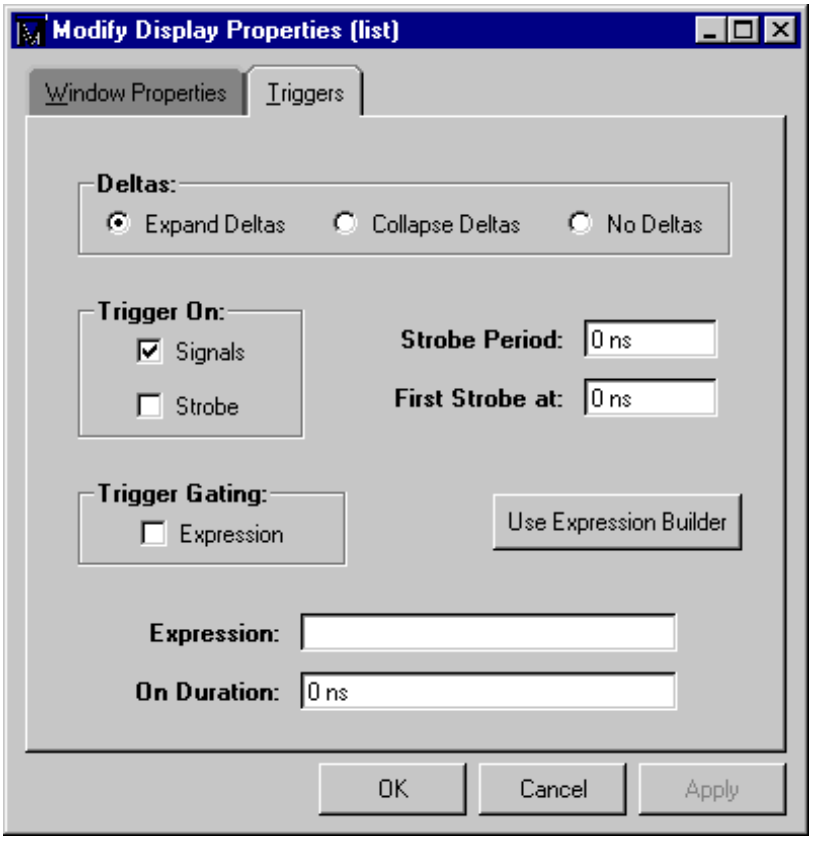
The **Window Properties** page includes these options:

- **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: Show All Dataset Prefixes**
  Displays the dataset prefix associated with each signal pathname. Useful for displaying signals from multiple datasets.
- **Dataset Prefix: Show All Except "sim"**
  Displays all dataset prefixes except the one associated with the current simulation – "sim." Useful for displaying signals from multiple datasets.

- **Dataset Prefix: Show No Dataset Prefixes**
  Turns off display of dataset prefixes.

Trigger settings page

The Triggers page controls the triggering for the display of new lines in the List window. You can specify whether an HDL item trigger or a strobe trigger is used to determine when the List window displays a new line. If you choose **Trigger on: Signals**, then you can choose between collapsed or expanded delta displays. You can also choose a combination of signal or strobe triggers. To use gating, Signals or Strobe or both must be selected.



The Triggers page includes the following options:

- **Deltas:Expand Deltas**
  When selected with the **Trigger on: Signals** check box, displays a new line for each time step on which items change, including deltas within a single unit of time resolution.
- **Deltas:Collapse Deltas**
  Displays only the final value for each time unit.
- **Deltas:No Deltas**
  Hides simulation cycle (delta) column.
- **Trigger On: Signals**
  Triggers on signal changes. Defaults to all signals. Individual signals can be excluded from triggering by using the **Prop > Signals Props** dialog box or by originally adding them with the **-notrigger** option to the **add list** command (CR-28).

- **Trigger On: Strobe**
  Triggers on the **Strobe Period** you specify; specify the first strobe with **First Strobe at:**.
- **Trigger Gating: Expression**
  Enables triggers to be gated on and off by an overriding 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" (8-275)
- **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 config -usegating 1
configure list config -gateduration 100
configure list config -gateexpr {<expression>}
```

Adding HDL 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" (8-178)). 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, Dataflow, 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-28) 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 HDL items to your List window.
- Edit and format the items to create the view you want (see "Editing and formatting HDL items in the List window" (8-181)).
- Save the format to a file by selecting **File > Save Format** (List window).

To use the format (do) file, start with a blank List window, and run the DO file in one of two ways:

- Invoke the **do** (CR-104) command from the command line:
  `do <my_list_format>`
- Select **File > Load Format** from the List window menu bar.

Select **Edit > Select All** and **Edit > Delete** to remove the items from the current List window or create a new, blank List window by selecting **View > New > List** (Main window). You may find it useful to have two differently formatted windows open at the same time, see "Examining simulation results with the List window" (8-184).

Note: 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, Model*Sim* will advise you of the HDL items it expects to find.

Editing and formatting HDL items in the List window

Once you have the HDL 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 HDL items to the List window" (8-180))

To edit an item:

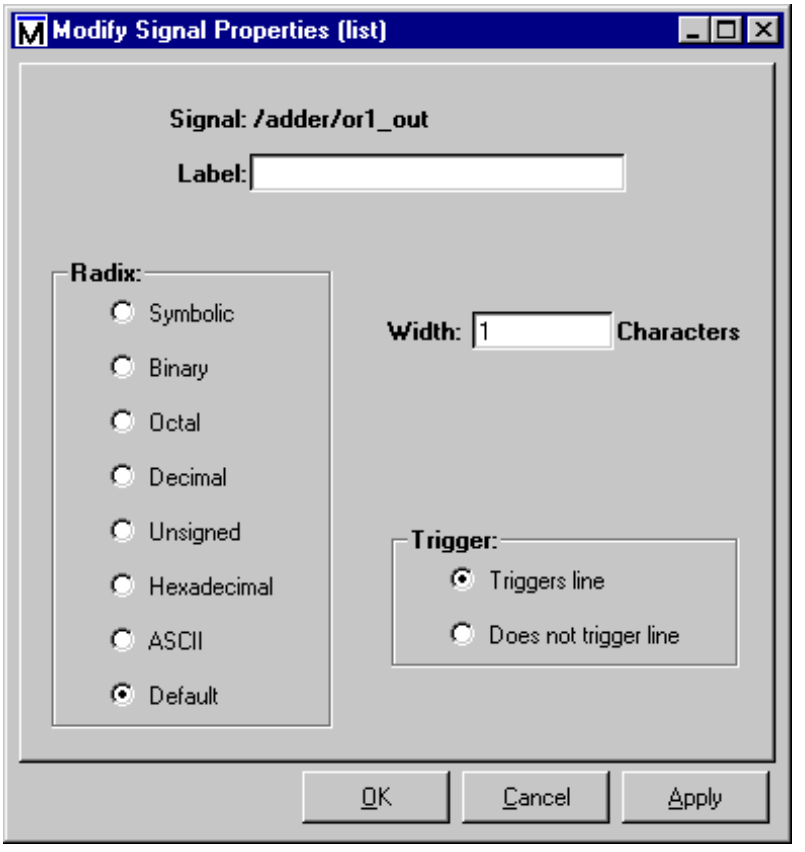
Select the item's label at the top of the List window or one of its values from the listing. Move, copy or remove the item by selecting commands from the List window Edit menu (8-176) menu.

You can also click+drag to move items within the window:

- to select several contiguous items:
  click+drag to select additional items to the right or the left of the original selection
- to select several items randomly:
  Control+click to add or subtract from the selected group
- to move the selected items:
  re-click on one of the selected items, hold and drag it to the new location

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 **Prop > Signal Props** (List window). The resulting Modify Signal Properties dialog box allows you to set the item's label, label width, triggering, and radix.



The **Modify Signal Properties** dialog box includes these options:

- **Signal**
  Shows the full pathname of the selected signal.
- **Label**
  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 **Options > Simulation** (Main window) or the **radix** command (CR-166). You can change the default radix permanently by editing the DefaultRadix (B-399) 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.

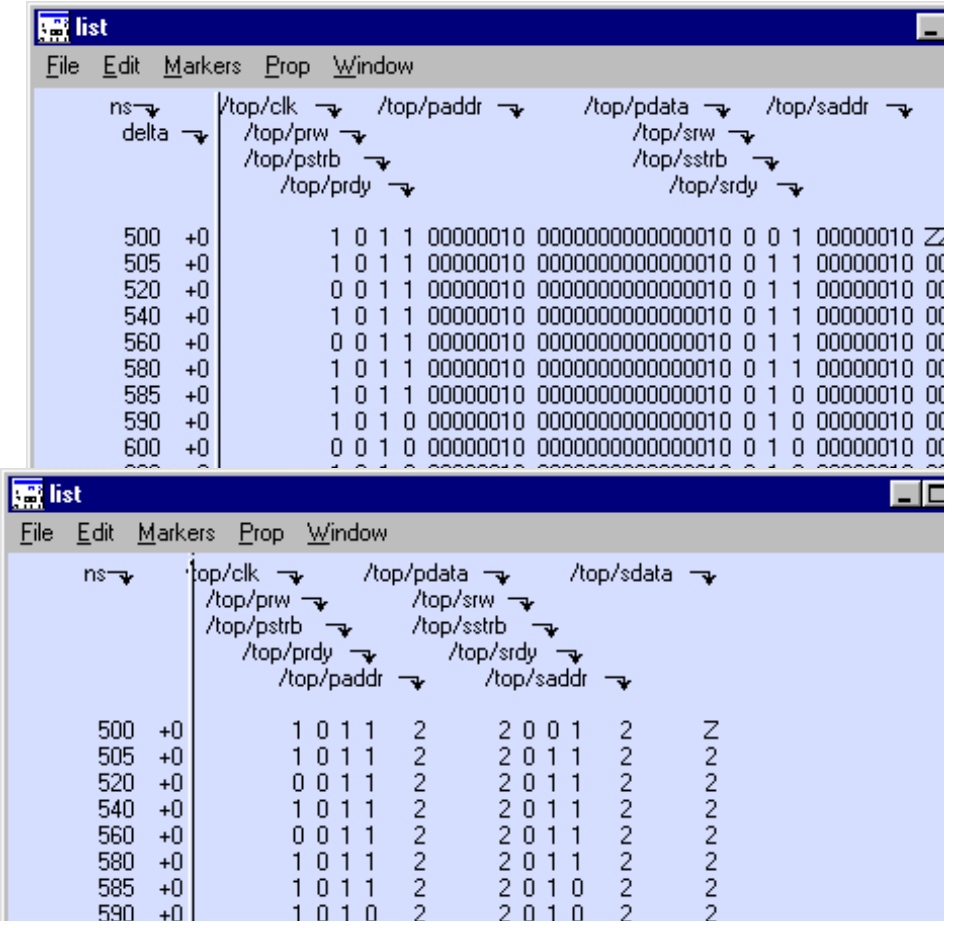
- **Width**
  Allows you to specify the desired width of the column used to list the item value. The default is an approximation of the width of the current value.
- **Trigger: Triggers line**
  Specifies that a change in the value of the selected item causes a new line to be displayed in the List window.
- **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" (8-178).

Examining simulation results with the List window

Because you can use the Main window View menu (8-162) to create a second List window, you can reformat another List window after the simulation run if you decide a different format would reveal the information you're after. Compare the two illustrations.

The divider bar separates time and delta from values; signal values are listed in symbolic format; and an item change triggers a new line.



Signal values are listed in decimal format;

In the first List window, the HDL items are formatted as symbolic and use an item change to trigger a line; the field width was changed to accommodate the default label width. The window divider maintains the time and delta in the left pane; signals in the right pane can be viewed by scrolling. For the second listing, the item radix for **paddr**, **pdata**, **saddr**, and **sdata** is now decimal.

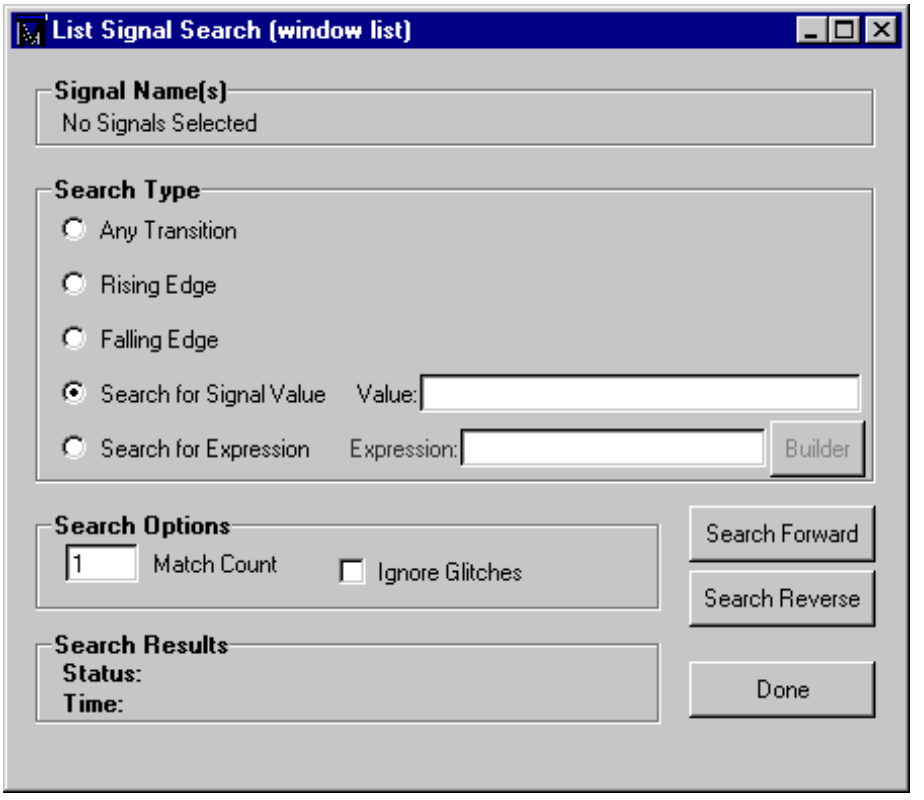
Finding items by name in the List window

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" (8-178)). Checking **Auto Wrap** makes the search continue at the beginning of the window. Note that you can change an item's label.

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-291).

Note: If your signal values are displayed in binary radix, see "Searching for binary signal values in the GUI" (CR-300) 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" (8-275) if desired.

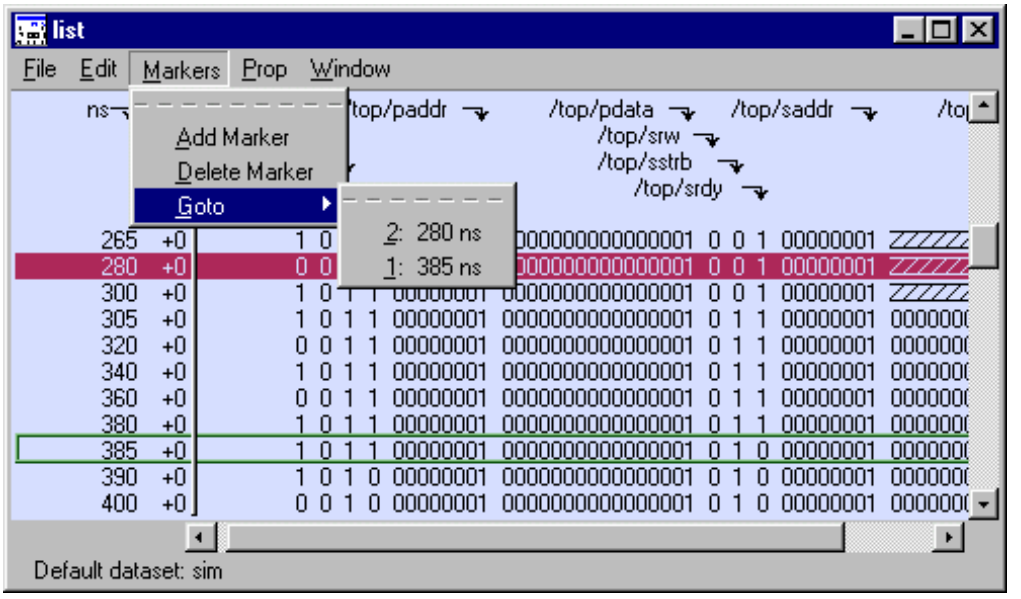
  The expression can involve more than one signal but is limited to signals logged in the List window. Expressions can include constants, variables, and DO files. If no expression is specified, the search will give an error. See "Expression syntax" (CR-302) for more information.
- **Search Options: Match Count**
  Indicates the number of transitions or matches to search.You can search for the n-th transition or the n-th 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 **Markers** > **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 **Markers** > **Delete Marker**.

Finding a marker



Choose a specific marked line to view by selecting **Markers** > **Goto**. The marker name (on the **Goto** list) corresponds to the simulation time of the selected 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
<arrow up>	scroll listing up (selects and highlights the line above the currently selected line)
<arrow down>	scroll listing down (selects and highlights the line below the currently selected line)
<arrow left>	scroll listing left
<arrow right>	scroll listing right
<page up>	scroll listing up by page
<page down>	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)
<control-f> Windows <control-s> UNIX	opens the find dialog box; finds the specified item label within the list display

Saving List window data to a file

Select **File > Write List (format)** (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
```

- **event**
  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-283)

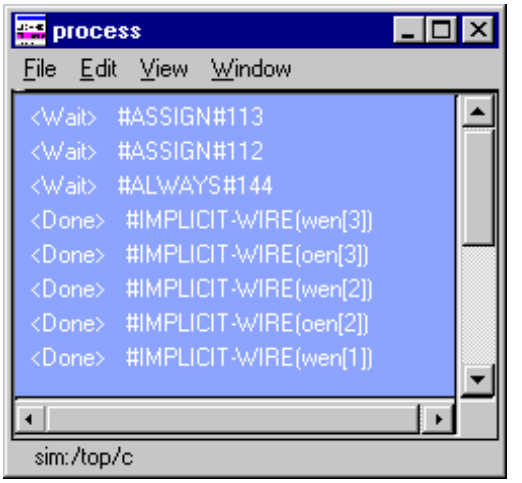
```
0 00000000000000010?????????
2 00000000000000010???????1?
3 00000000000000010??????010
4 00000000000000010000000010
100 00000001000000010000000010
```

You can also save List window output using the **write list** command (CR-279).

Process window

The Process window displays a list of processes. If **View > Active** is selected then all 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.

Each HDL item in the scrollbox is preceded by one of the following indicators:



- **<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.
- **<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.

If you select a "Ready" process, it will be executed next by the simulator.

When you click on a process in the Process window, the following windows are updated:

Window updated	Result
Structure window (8-210)	shows the region in which the process is located
Variables window (8-213)	shows the VHDL variables and Verilog register variables in the process
Source window (8-201)	shows the associated source code
Dataflow window (8-171)	shows the process, the signals, nets, and registers the process reads, and the signals, nets, and registers driven by the process
Source window (8-201)	shows the signals, nets, and registers declared in the region in which the process is located

The Process window menu bar

The following menu commands are available from the Process window menu bar.

File menu

Save As	save the process tree to a text file viewable with the Model*Sim* **notepad** (CR-141)
Environment	**Follow Context Selection**: update the window based on the selection in the Structure window (8-210); **Fix to Current Context**: maintain the current view, do not update
Close	close this copy of the Process window; you can create a new window with View > New from the "The Main window menu bar" (8-160)

Edit menu

Copy	copy the selected process' full name
Sort	sort the process list in either ascending, descending, or declaration order
Select All	select all processes in the Process window
Unselect All	deselect all processes in the Process window
Find	find the specified text string within the process list; choose the Status (ready, wait or done), the Process label, or the path to search, and the search direction: down or up

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

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

Signals window

The Signals window is divided into two window panes. The left pane shows the names of HDL items in the current region (which is selected in the Structure window). The right pane shows the values of the associated HDL items at the end of the current run. The data in this pane is similar to that shown in the Wave window (8-216), except that the values do not change dynamically with movement of the selected Wave window cursor.

You can double-click a signal and it will highlight that signal in the Source window (opening a Source window if one is not open already).

Horizontal scroll bars for each window pane allow scrolling to the right or left in each pane individually. The vertical scroll bar will scroll both panes together.

The HDL items can be sorted in ascending, descending, or declaration order.

HDL items you can view

One entry is created for each of the following VHDL and Verilog items within the design:

VHDL items

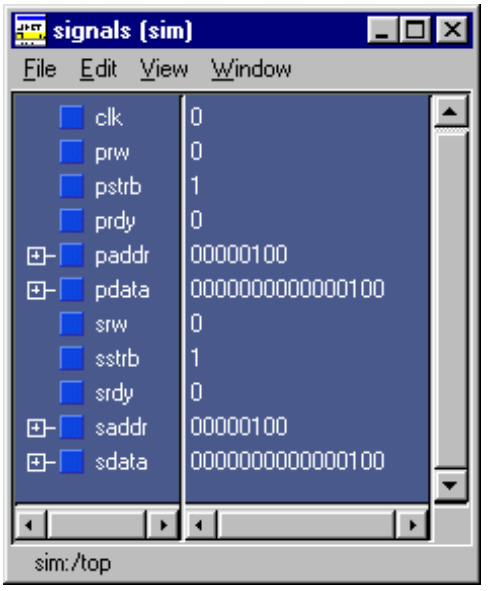
signals

Verilog items

nets, register variables, named events, and module parameters

Virtual items

(indicated by an orange diamond icon) virtual signals and virtual functions; see "Virtual signals" (7-144) for more information



The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion.

Hierarchy also applies to Verilog nets and vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.)

Hierarchy is indicated in typical Model*Sim* fashion with plus (expandable), minus (expanded), and blank (single level) boxes.

See "Tree window hierarchical view" (8-155) for more information.

The Signals window menu bar

The following menu commands are available from the Signals window menu bar.

File menu

Save As	save the signals tree to a text file viewable with the Model*Sim* **notepad** (CR-141)
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 View > New from the "The Main window menu bar" (8-160)

Edit menu

Copy	copy the current selection in the Signals window
Sort	sort the signals tree in either ascending, descending, or declaration order
Select All	select all items in the Signals window
Unselect All	unselect all items in the Signals window
Expand Selected	expand the hierarchy of the selected items
Collapse Selected	collapse the hierarchy of the selected items
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Force	apply stimulus to the specified Signal Name; specify Value, Kind (Freeze/Drive/Deposit), Delay, and Cancel; see also the **force** command (CR-121)
Noforce	remove the effect of any active **force** command (CR-121) on the selected HDL item; see also the **noforce** command (CR-138)
Clock	define clock signals by Signal Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling, see"Defining clock signals" (8-200)
Justify Values	justify values to the left or right margins of the window pane
Find	find the specified text string within the Signals window; choose the Name or Value field to search and the search direction: down or up; see also the **search** command (CR-178)

View menu

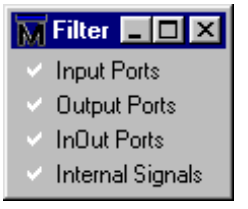
Wave/List/Log	place the Selected Signals, Signals in Region, or Signals in Design in the Wave window (8-216), List window (8-175), or logfile
Filter	choose the port and signal types to view (Input Ports, Output Ports, InOut Ports and Internal Signals) in the Signals window

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

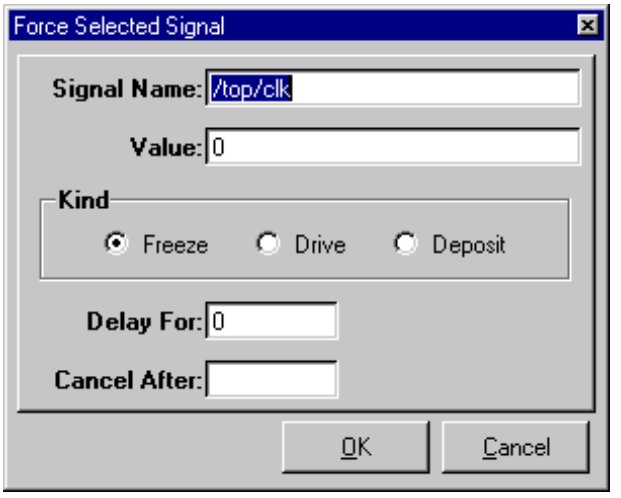
Selecting HDL item types to view

The **View > Filter** menu selection allows you to specify which HDL items are shown in the Signals window. Multiple options can be selected.



Forcing signal and net values

The **Edit > Force** command 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-121).



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-138).

  Freeze is the default for Verilog nets and unresolved VHDL signals and **Drive** is the default for resolved signals.

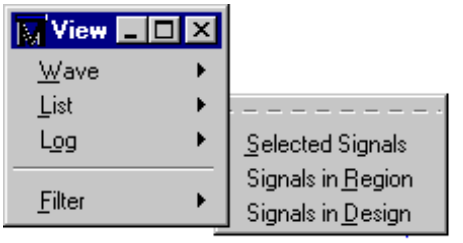
  If you prefer **Freeze** as the default for resolved and unresolved signals, you can change the default force kind in the *modelsim.ini* file; see "Projects and system initialization" (2-25).
- **Kind: Drive**
  Attaches a driver to the signal and drives the specified value until the signal or net is forced again or until it is unforced with a **noforce** command (CR-138). This value 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-138).
- **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-121) after the specified period of simulation time.
- **OK**
  When you click the OK button, a **force** command (CR-121) 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 HDL items to the Wave and List windows or a logfile

Before adding items to the List or Wave window you may want to set the window display properties (see "Setting List window display properties" (8-178)). Once display properties have been set, you can add items to the windows or logfile in several ways.

Adding items with the Signals window View menu



Use the **View** menu with either the **Wave**, **List**, or **Log** selection to add HDL items to the Wave window (8-216), List window (8-175), or a logfile, respectively.

The logfile 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 logfile they cannot be removed. If you begin a simulation by invoking **vsim** (CR-258) with the -view <logfile_name> option, Model*Sim* reads the logfile to drive the Wave and List windows.

Choose one of the following options (Model*Sim* opens the target window for you):

- **Selected Signal**
  Lists only the item(s) selected in the Signals window.
- **Signals in Region**
  Lists all items in the region that is selected in the Structure window.
- **Signals in Design**
  Lists 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 logfile is to enter the one of the following commands at the VSIM prompt (choose either the **add list** (CR-28), **add wave** (CR-37), or **log** (CR-131) 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 *
```

Or add all the items in the design with:

```
add list | add wave | log -r /*
```

If the target window (Wave or List) is closed, Model*Sim* opens it when you when you invoke the command.

Finding HDL 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**.

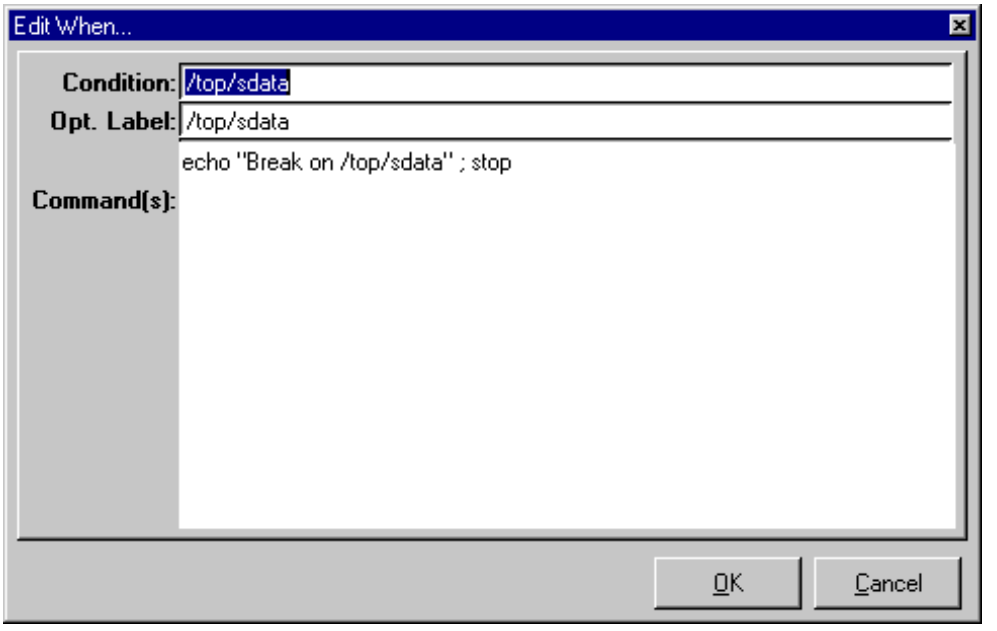
You can also do a quick find from the keyboard. When the Signals window is active, each time you type a letter the signal selector (highlight) will move to the next signal whose name begins with that letter.

Setting signal breakpoints

You can set signal breakpoints (a.k.a., when breakpoints; see the **when** command (CR-273) for more details) via a context menu in the Signal window. When statements instruct Model*Sim* 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 "Time-based breakpoints" (CR-275)) . When a breakpoint is hit, a message appears in the transcript window about which signal caused the breakpoint.

To access the breakpoint commands, select a signal and then click your right mouse button (2nd button in Windows; 3rd button in UNIX). To set a breakpoint on a selected signal, select **Add Breakpoint** from the context menu. To remove a breakpoint from a selected signal, select **Remove Signal Breakpoint**. To remove all breakpoints in the current region, select **Remove All Signal Breakpoints**. To see a list of currently set breakpoints, select **Show Breakpoints**.

The **Edit Breakpoint** command opens the **Edit When** dialog box.



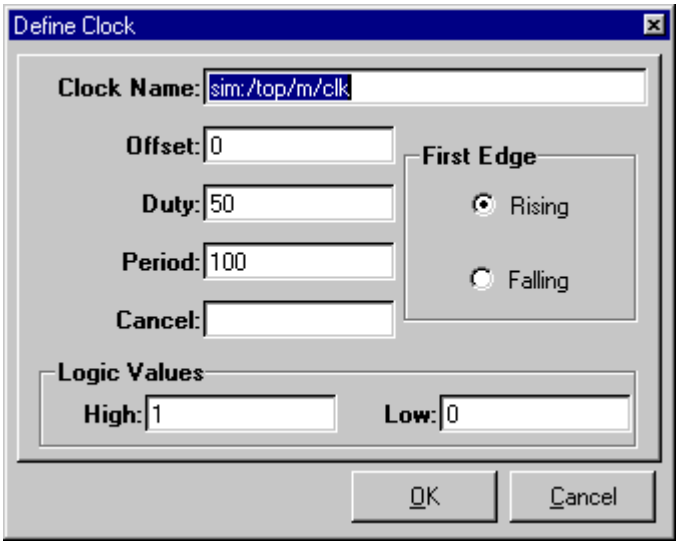
The Edit When dialog includes the following options:

- **Condition**
  The condition(s) to be met for the specified command(s) to be executed. Required. See the **when** command (CR-273) for more information on creating the condition statement.
- **Opt. Label**
  An optional text label for the when statement.
- **Command(s)**
  The command(s) to be executed when the specified condition is met. Any Model*Sim* or Tcl command or series of commands are valid with one exception—the **run** command (CR-176) cannot be used.

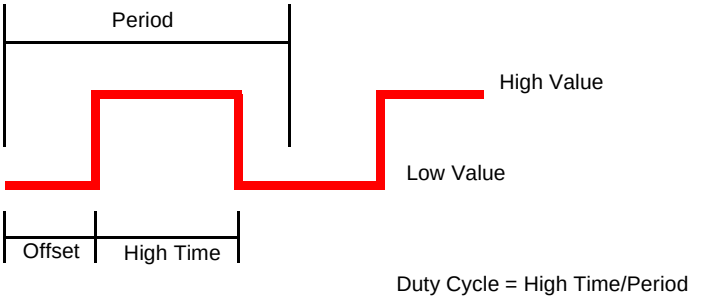
The **Edit All Breakpoints** command opens the Breakpoints dialog box. See "Setting file-line breakpoints" (8-205) for details.

Defining clock signals

Select **Edit > Clock** to define clock signals by Name, Period, Duty Cycle, Offset, and whether the first rising edge is rising or falling. You can also specify a simulation period after which the clock definition should be cancelled.



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.

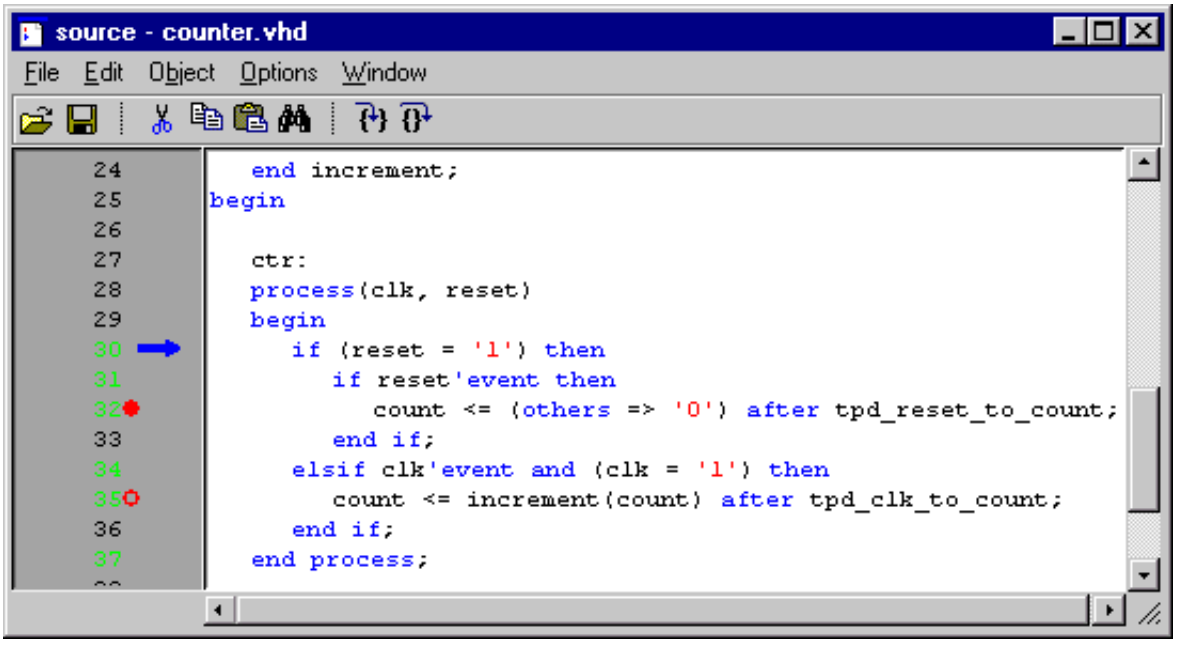
Source window

The Source window allows you to view and edit your HDL 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 the Structure window (8-210) or use the **File > Open** command (Source window) to add a file to the window. (Your source code can remain hidden if you wish; see "Source code security and -nodebug" (E-433)).

The window is divided into two panes—the left-hand pane contains line numbers, and the right-hand pane contains the source file. The pathname of the source file is indicated in the header of the Source window.

As shown in the picture below, you may also see the following in the left-hand pane:

- Green line numbers— denote executable lines
- Blue arrow—denotes a process that you have selected in the Process window (8-190)
- Red circles—denote file-line breakpoints; hollow circles denote breakpoints that are currently disabled



The Source window menu bar

The following menu commands are available from the Source window menu bar.

File menu

New	edit a new (VHDL, Verilog or Other) source file
Open	select a source file to open
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 (the SourceDir variable in modelsim.tcl) to search for source files
Properties	list a variety of information about the source file; for example, file type, file size, file modification date
Save	save the current source file
Save As	save the current source file with a different name
Compile	compile HDL source files
Close	close this copy of the Source window; you can create a new window with View > New from the "The Main window menu bar" (8-160)

Edit menu

To edit a source file, make sure the **Read Only** option in the Source Options dialog box is *not* selected (use the **Edit > read only** (Source menu) selection).

<editing option>	basic editing options include: Undo, Cut, Copy, Paste, Select All, and Unselect All; see "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168)
Find	find the specified text string or regular expression within the source file; there is an option to match case or search backwards
Find Next	find the next occurrence of a string specified with the Find command
Replace	find the specified text string or regular expression and replace it with the specified text string or regular expression
Previous Coverage Miss	when simulating with Code Coverage (10-291), finds the previous line of code that was not used in the simulation
Next Coverage Miss	when simulating with Code Coverage (10-291), finds the next line of code that was not used in the simulation
Breakpoints	add, edit, or delete file-line and signal breakpoints; see "Setting file-line breakpoints" (8-205)

read only	toggle the read-only status of the current source file

Object menu

Describe	display information about the selected HDL item; same as the **describe** command (CR-100); the item name is shown in the title bar
Examine	display the current value of the selected HDL item; same as the **examine** (CR-115) command; the item name is shown in the title bar

Options 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
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 (8-210), the other frozen)
Freeze View	disable updating the source view from the Process window (8-190)

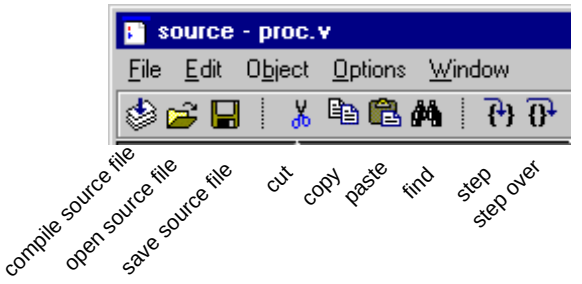
Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window

<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

The Source window toolbar

Buttons on the Source window toolbar give you quick access to these Model*Sim* commands and functions.



Source window toolbar buttons

Button	Menu equivalent	Other equivalents
Compile Source File open the Compile HDL Source File dialog	File > Compile	use **vcom** or **vlog** command at the VSIM prompt see: **vcom** (CR-217) or **vlog** (CR-250) command
Open Source File open the Open File dialog box (you can open any text file for editing in the Source window)	File > Open	select an HDL item in the Structure window, the associated source file is loaded into the Source window
Save Source File save the file in the Source window	File > Save	none
Cut cut the selected text within the Source window	Edit > Cut	see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168)
Copy copy the selected text within the Source window	Edit > Copy	see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168)

Source window toolbar buttons			
Button		**Menu equivalent**	**Other equivalents**
	**Paste** paste the copied text to the cursor location	Edit > Paste	see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168)
	**Find** find the specified text string within the source file; match case option	Edit > Find	<control -f> (Windows) <control -s> (UNIX)
	**Step** steps the current simulation to the next HDL statement	Main window: Run > Step	use **step** command at the VSIM prompt see: **step** (CR-187) command
	**Step Over** HDL statements are executed but treated as simple statements instead of entered and traced line by line	Main window: Run > Step -Over	use the **step -over** command at the VSIM prompt see: **step** (CR-187) command

Setting file-line breakpoints

You can set breakpoints three different ways:

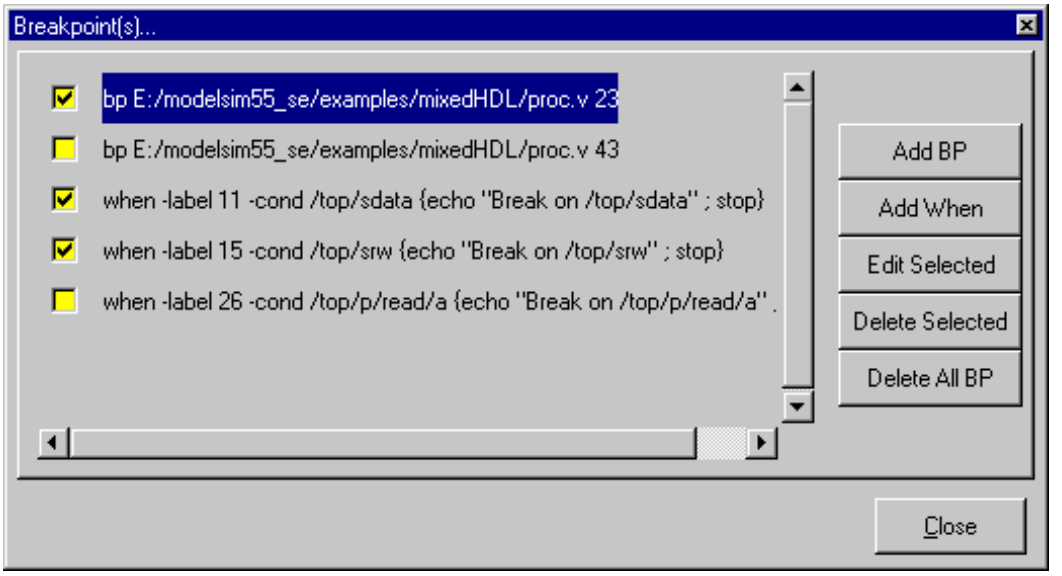
- Using the command line; see the **bp** (CR-48) (breakpoint) command for details
- Using your mouse in the Source window
- Using the **Edit > Breakpoints** menu selection

Setting breakpoints with your mouse

To set a breakpoint with your mouse, click on a green line number at the left side of the window (breakpoints can be set only on executable lines). The breakpoints are toggles – click once to create the colored dot; click again to disable or enable the breakpoint. To delete the breakpoint completely, click the colored dot with your right mouse button, and select **Remove Breakpoint**.

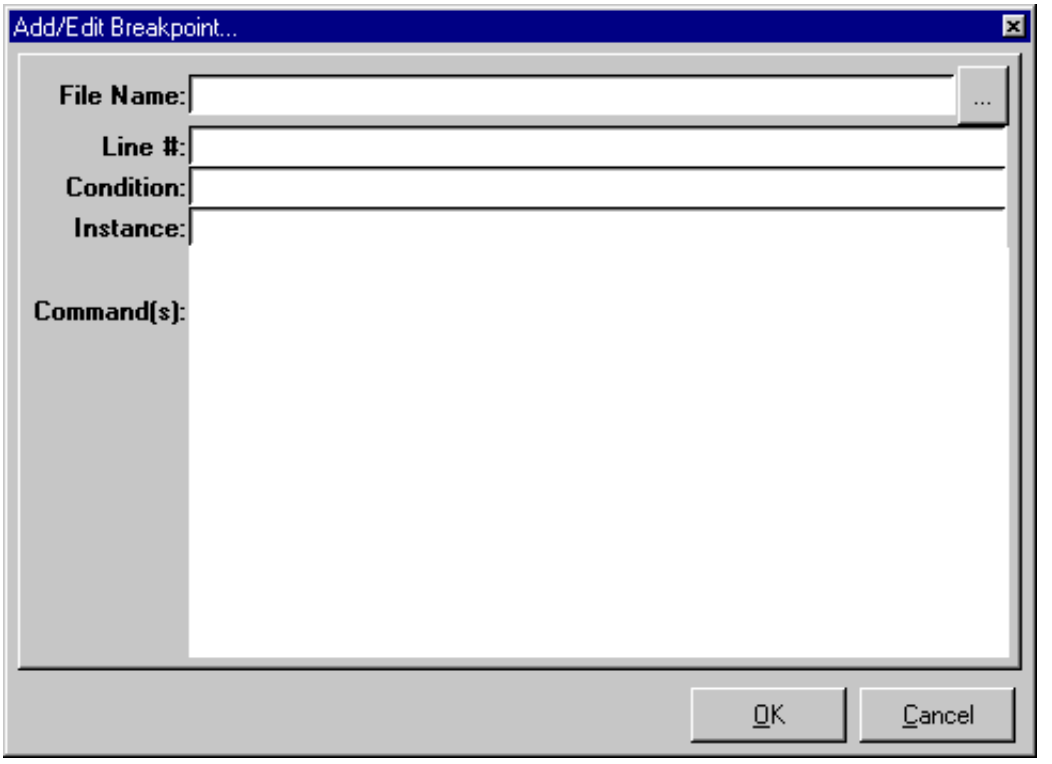
Setting breakpoints with the Edit > Breakpoints command

Selecting **Edit** > **Breakpoints** (Source window) opens the dialog box shown below.



The Breakpoints dialog box allows you to create and manage both file-line and signal breakpoints (a.k.a., when breakpoints). For details on signal breakpoints, see "Setting signal breakpoints" (8-198) and the **when** command (CR-273).

You can enable and disable existing breakpoints by checking or unchecking the box next to the breakpoint's name. To add a new file-line breakpoint, select **Add BP** (or **Edit Selected** for an existing file-line breakpoint).



The **Add/Edit Breakpoint** dialog box includes the following options:

- **File Name**
  The file name in which you want to set the breakpoint. Required. The button next to this field allows you to browse to select a file.
- **Line #**
  The line number on which you want to set the breakpoint. Required.
- **Condition**
  The condition(s) that determine whether the breakpoint is hit. See the **bp** command (CR-48) for more information on creating the condition statement.
- **Instance**
  Specify a region in which the breakpoint should be set. If left blank, the breakpoint affects every instance in the design.
- **Command(s)**
  One or more commands that you want executed at the breakpoint.

Editing the source file in the Source window

Several toolbar buttons (shown above), mouse actions, and special keystrokes can be used to edit the source file in the Source window. See "Mouse and keyboard shortcuts in the Transcript and Source windows" (8-168) for a list of mouse and keyboard editing options.

Checking HDL item values and descriptions

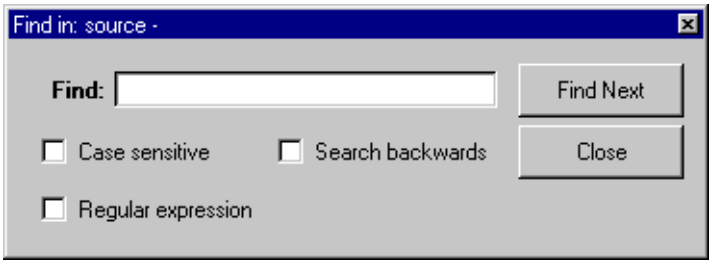
There are two quick methods to determine the value and description of an HDL item displayed in the Source window:

- select an item, then chose **Object > Examine** or **Object > Description** from the Source window menu
- select an item with the right mouse button to see an examine pop-up (select "now" to examine the current simulation time in VHDL code)

You can also invoke the **examine** (CR-115) and/or **describe** (CR-100) 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 tab stops in the Source window by selecting the Main window **Options > Edit Preferences** command. Follow these steps:

1. Select the **By Names** tab.
2. Select Source in the first column, and then select the "tabs" item in the second column.
3. Press the **Change Value** button.
4. In the dialog that appears, enter a single number "n", which sets a tab stop every n characters (where a character width is the width of the "8" character).

   or

   Enter a list of screen distances for the tab stops. For instance,
   21 49 77 105 133 161 189 217 245 273 301 329 357 385 413 441 469

   The number 21 or 21p means 21 pixels; the number 3c means three centimeters; the number 1i means one inch.

Important: Do not use quotes or braces in the list (i.e., "21 49" or {21 49}). This will cause the GUI to hang.

You can also set tab stops using the PrefSource(tabs) Tcl preference variable.

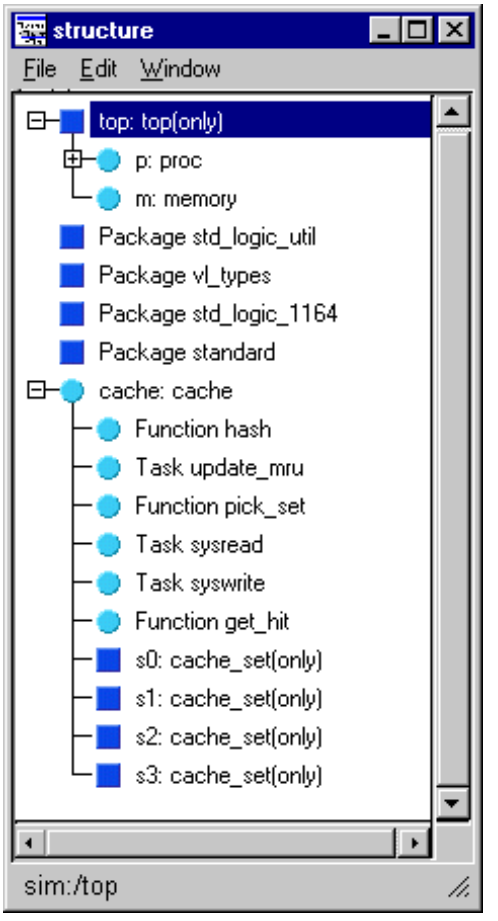
Structure window

Note: In ModelSim versions 5.5 and later the information contained in the Structure window is shown in the structure pages of the Main window Workspace (8-158). The Structure window will not display by default. You can display the Structure window at any time by selecting **View > Structure** (Main window). The discussion below applies to both the Structure window and the structure pages in the Workspace.

The Structure window provides a hierarchical view of the structure of your design. An entry is created by each HDL item within the design. (Your design structure can remain hidden if you wish, see "Source code security and -nodebug" (E-433).)

HDL items you can view

The following HDL items for VHDL and Verilog are represented by hierarchy within Structure window.



VHDL items

(indicated by a dark blue square icon)
component instantiation, generate statements, block statements, and packages

Verilog items

(indicated by a lighter blue circle icon)
module instantiations, named forks, named begins, tasks, and functions

Virtual items

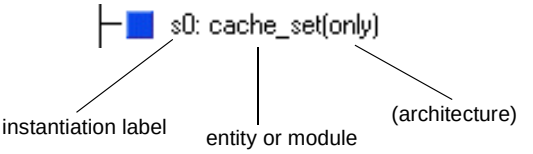
(indicated by an orange diamond icon)
virtual regions; see "Virtual Objects (User-defined buses, and more)" (7-144) for more information.

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.

The first line of the Structure window indicates the top-level design unit being simulated. By default, this is the only level of the hierarchy that is expanded upon opening the Structure window.

Instance name components in the Structure window

An instance name displayed in the Structure window consists of the following parts:



- **instantiation label**
  Indicates the label assigned to the component or module instance in the instantiation statement.
- **entity or module**
  Indicates the name of the entity or module that has been instantiated.
- **architecture**
  Indicates the name of the architecture associated with the entity (not present for Verilog).

When you select a region in the Structure window, it becomes the *current region* and is highlighted; the Source window (8-201) and Signals window (8-193) 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 (8-190) is updated if **In Region** is selected in that window; the Process window will in turn update the Variables window (8-213).

The Structure window menu bar

The following menu commands are available from the Structure window menu bar.

File menu

Save As	save the structure tree to a text file viewable with the ModelSim **notepad** (CR-141)
Environment	allow the window contents to change when the active dataset is changed; or, fix to a specific dataset
Close	close this copy of the Structure window; you can create a new window with View > New from the "The Main window menu bar" (8-160)

Edit menu

Copy	copy the current selection in the Structure window
Sort	sort the structure tree in either ascending, descending, or declaration order
Expand Selected	expand the hierarchy of the selected item
Collapse Selected	collapse the hierarchy of the selected item

Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Find	find the specified text string within the structure tree; see "Finding items in the Structure window" (8-212)

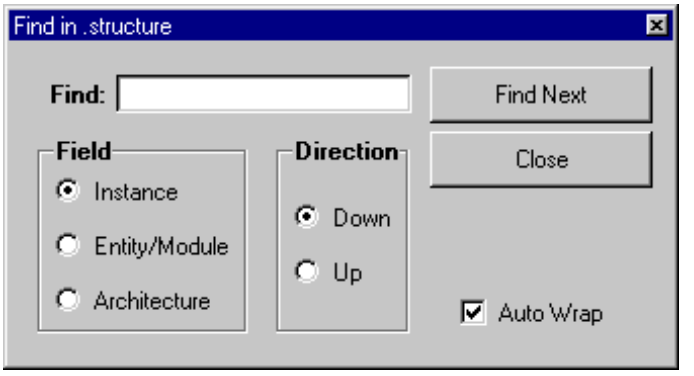
Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

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 **Auto Wrap** to have the search continue at the top of the window.



Variables window

The Variables window is divided into two window panes. The left pane lists the names of HDL 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, see "Source code security and -nodebug" (E-433).)

HDL items you can view

The following HDL items for VHDL and Verilog are viewable within the Variables window.

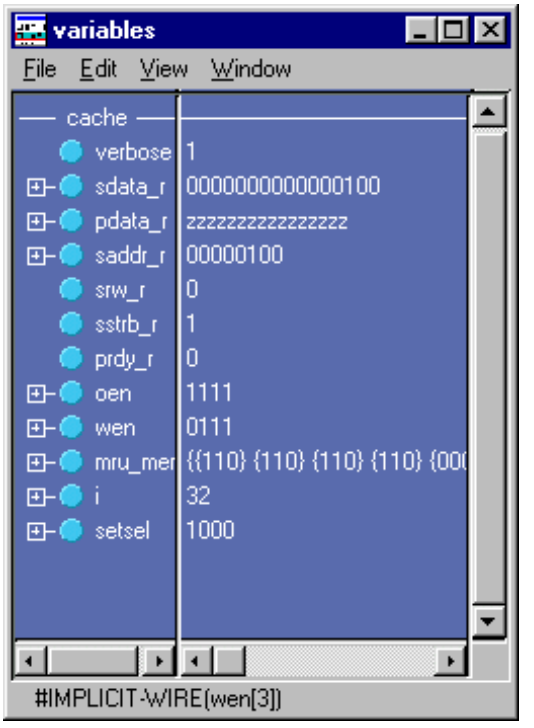
VHDL items

constants, generics, and variables

Verilog items

register variables

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion. Hierarchy also applies to Verilog vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.) Hierarchy is indicated in typical Model*Sim* fashion with plus (expandable) and minus (expanded). See "Tree window hierarchical view" (8-155) for more information.



To change the value of a VHDL variable, constant, or generic or a Verilog register 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. Note that "Variable Name" is a term that is used loosely in this case to signify VHDL constants and generics as well as VHDL and Verilog register variables. 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

The following menu commands are available from the Variables window menu bar.

File menu

Save As	save the variables tree to a text file viewable with the Model*Sim* **notepad** (CR-141)
Environment	**Follow Process Selection**: update the window based on the selection in the Process window (8-190); **Fix to Current Process**: maintain the current view, do not update
Close	close this copy of the Variables window; you can create a new window with View > New from the "The Main window menu bar" (8-160)

Edit menu

Copy	copy the selected items in the Variables window
Sort	sort the variables tree in either ascending, descending, or declaration order
Select All	select all items in the Variables window
Unselect All	deselect all items in the Variables window
Expand Selected	expand the hierarchy of the selected item
Collapse Selected	collapse the hierarchy of the selected item
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Change	change the value of the selected HDL item
Justify Values	justify values to the left or right margins of the window pane
Find	find the specified text string within the variables tree; choose the Name or Value field to search and the search direction: Down or Up

View menu

Wave/List/Log	place the Selected Variables or Variables in Region in the Wave window (8-216), List window (8-175), or logfile

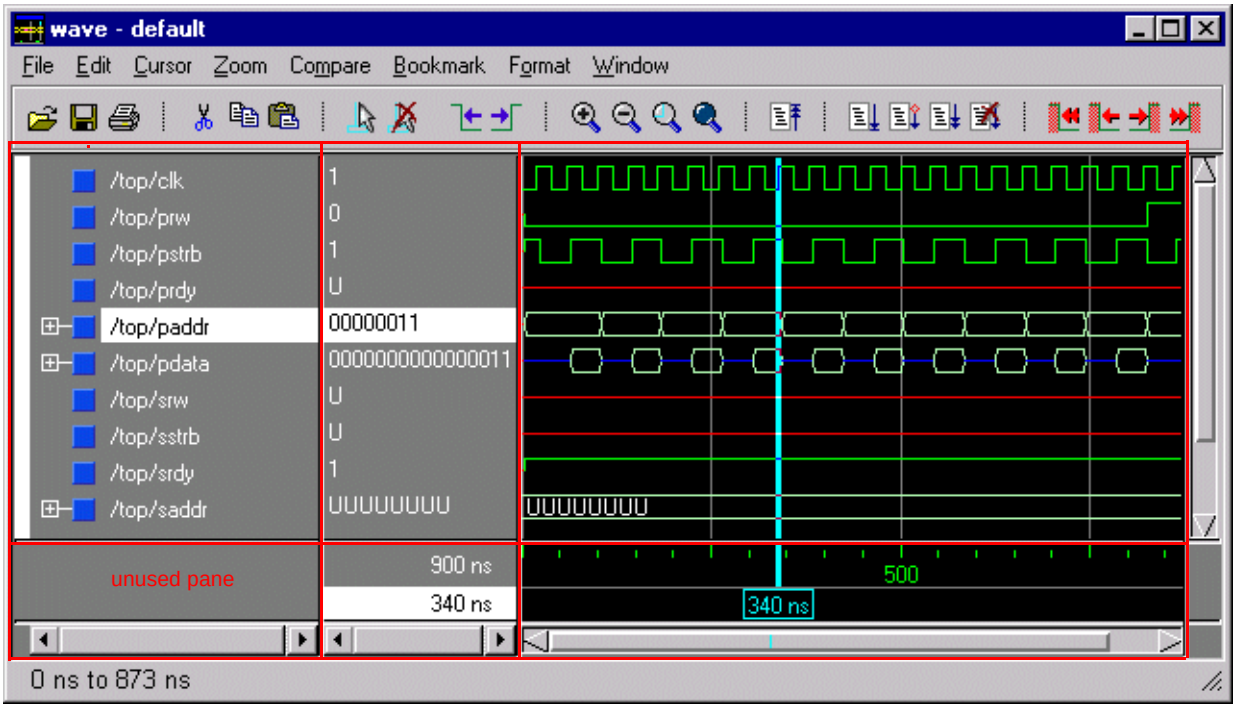
Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

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 HDL waveforms and their values.

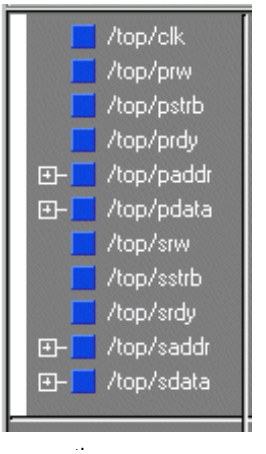
The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.



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 (8-228)).

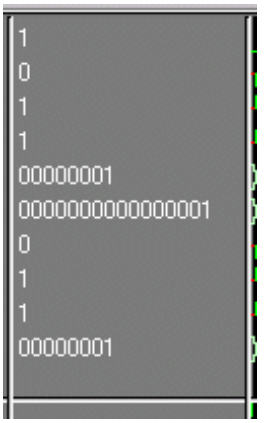


Values pane

A values 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 **Options** > **Simulation** (Main window) (see "Setting default simulation options" (8-265)).

The data in this pane is similar to that shown in the Signals window (8-193), except that the values change dynamically whenever a cursor in the waveform pane (below) is moved.

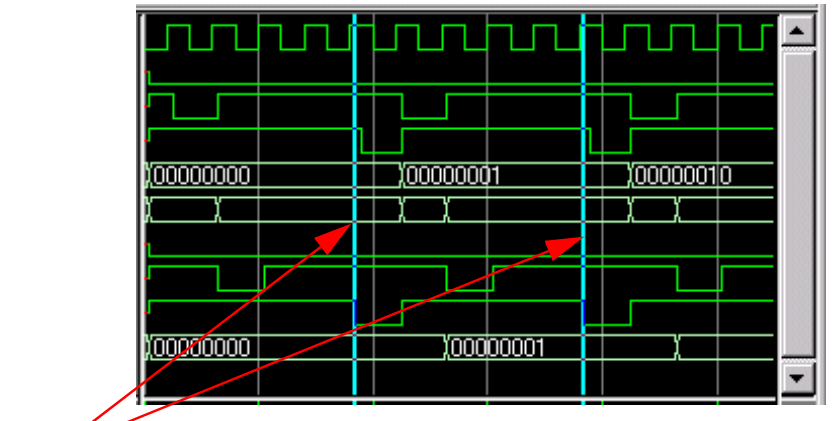


values pane

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.

The window pane below the pathnames window pane and to the left of the cursor panes is unused at this time.

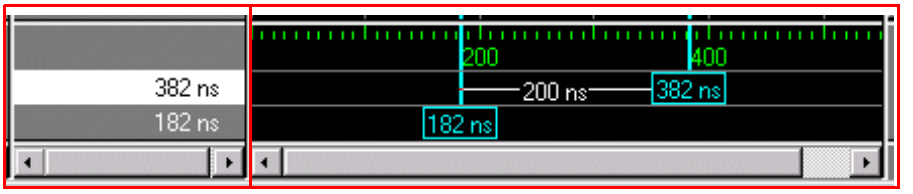


waveform pane

cursors

Cursor panes

There are two cursor panes, as shown below. The left pane shows the time value for each cursor. The selected cursor's value is highlighted. The right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors can be displayed.



two cursor panes

HDL items you can view

VHDL items

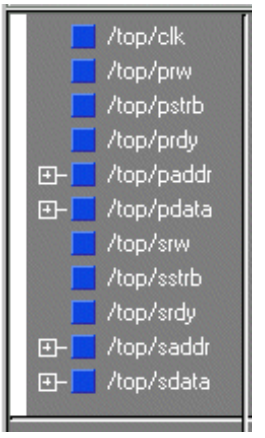
(indicated by a dark blue square) signals and process variables

Verilog items

(indicated by a light blue circle) nets, register variables, and named events

Virtual items

(indicated by an orange diamond) virtual signals, buses, and functions, see; "Virtual Objects (User-defined buses, and more)" (7-144) for more information



Comparison items

(indicated by a yellow triangle) comparison region and comparison signals; see *Chapter 11 - Waveform Comparison* for more information

Note: Constants, generics, and parameters are not viewable in the List or 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 a readout 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 (8-223). You can reuse any formatting changes you make by saving a Wave window format file, see "Adding items with a Wave window format file" (8-219).

Adding HDL 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" (8-235)). You can add items to the Wave window in several ways.

Adding items from the Signals 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 Main window command line

To add specific HDL items to the window, enter (separate the item names with a space):

```
add wave <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add wave *
```

Or add all the items in the design with:

```
add wave -r /*
```

Adding items with a Wave window format file

To use a Wave window format file you must first save a format file for the design you are simulating. Follow these steps:

1. Add the items you want in the Wave window with any method shown above.
2. Edit and format the items, see "Editing and formatting HDL items in the Wave window" (8-230) 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-104) from the command line:

     ```
     do <my_wave_format>
     ```

   - Select **File > Load Format** (Wave window).

   Use **Edit > Select All** and **Edit > Delete** to remove the items from the current Wave window, use the **delete** command (CR-99) with the **wave** option, or create a new, blank Wave window with **View > New > Wave** (Main 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

The following menu commands and button options are available from the Wave window menu bar. If you see a dotted line at the top of a drop-down menu, you can select it to create a separate menu 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 Dataset	open a dataset
New Divider	insert a divider at the current location
New Group	setup a new group element – a container for other items that can be moved, cut and pasted like other objects (NOT CURRENTLY IMPLEMENTED)
Save Format	save the current Wave window display and signal preferences to a DO (macro) file; running the DO file will reformat the Wave window to match the display as it appeared when the DO file was created
Load Format	run a Wave window format (DO) file previously saved with Save Format
Page Setup	setup page for printing; options include: paper size, margins, label width, cursors, color, scaling and orientation
Print (Windows only)	send the contents of the Wave window to a selected printer; options include: All signals – print all signals Current View – print signals in current view for the time displayed Selected – print all or current view signals for user-designated time
Print Postscript	save or print the waveform display as a Postscript file; options include: All Signals – print all signals Current View – print signals in current view for the time displayed Selected – print all or current view signals for user-designated time
New Window Pane	split the pathname, values and waveform window panes to provide room for a new waveset
Remove Window Pane	remove window split and active waveset

Refresh Display	clear the Wave window, empty the file cache, and rebuild the window from scratch
Close	close this copy of the Wave window; you can create a new window with View > New from "The Main window menu bar" (8-160)

Edit menu

Cut	cut the selected item and waveform from the Wave window; see "Editing and formatting HDL items in the Wave window" (8-230)
Copy	copy the selected item and waveform
Paste	paste the previously cut or copied item above the currently selected item
Delete	delete the selected item and its waveform
Select All Unselect All	select, or unselect, all item names in the name pane
Combine	combine the selected fields into a user-defined bus
Signal Breakpoints	add, edit, and delete signal breakpoints; see "Setting signal breakpoints" (8-198)
Sort	sort the top-level items in the name pane; sort with full path name or viewed name; use ascending or descending order
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" (8-237)
Justify Values	justify values to the left or right margins of the window pane
Display Properties	set display properties for signal path length, cursor snap distance, row margin, and dataset prefixes
Signal Properties	set label, height, color, radix, and format for the selected item (use the Format menu selections below to quickly change individual properties); also set properties related to waveform comparisons

Cursor menu

Add Cursor	add a cursor to the center of the waveform window
Delete Cursor	delete the selected cursor from the window
Goto	choose a cursor to go to from a list of current cursors

Zoom menu

Zoom <selection>	selection: Full, In, Out, Last, Area with mouse button 1, or Range to change the waveform display range

Compare menu

Start Comparison	start a new comparison
Comparison Wizard	receive step-by-step assistance while creating a waveform comparison
Run Comparison	compute differences from time zero until the end of the simulation
End Comparison	stop difference computation and close the currently open comparison
Add	provides three options: Compare by Signal - specify signals for comparison Compare by Region - designate a reference region for a comparison Clocks - define clocks to be used in a comparison
Options	set options for waveform comparisons
Differences	provides four options: Clear - clear all differences from the Wave window Show - display differences in a text format in the Main window Transcript Save - save computation differences to a file that can be reloaded later Write Report - save computation differences to a text file
Rules	provides two options: Show - display the rules used to set up the waveform comparison Save - save rules for waveform comparison to a file
Reload	load saved differences and rules files

Bookmark menu

Add Bookmark	add a new bookmark that saves a specific zoom and scroll range
Edit Bookmarks	edit an existing bookmark
<bookmark_name>	list of currently defined bookmarks

Format menu

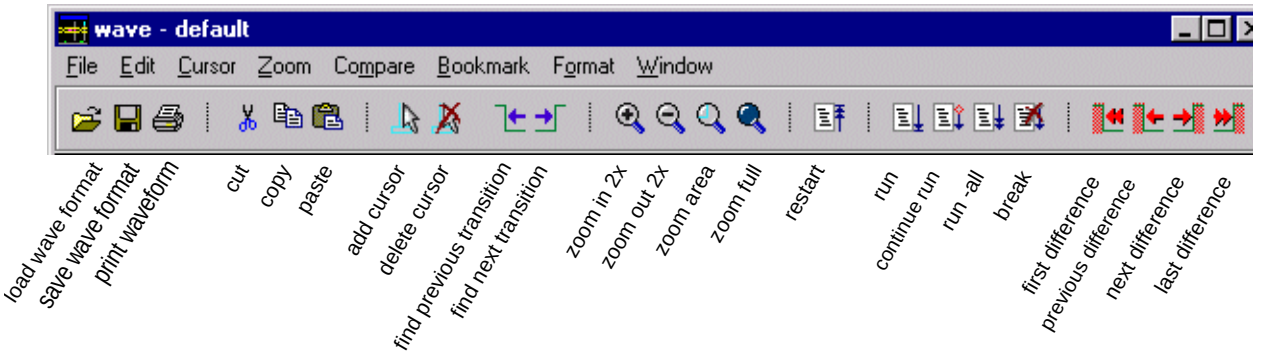
Radix	set the selected item's radix
Format	set the waveform format for the selected item – Literal, Logic, Event, Analog
Color	set the color for the selected item from a color palette
Height	set the waveform height in pixels for the selected item

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (8-269) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (8-162) in the Main window, or use the **view** command (CR-226)

The Wave window toolbar

The Wave window toolbar gives you quick access to these Model*Sim* commands and functions.



Wave window toolbar buttons			
Button		**Menu equivalent**	**Other options**
	**Load Wave Format** run a Wave window format (DO) file previously saved with Save Format	File > Load Format	**do** wave.do see **do** command (CR-104)
	**Save Wave Format** saves the current Wave window display and signal preferences to a do (macro) file	File > Save Format	none
	**Print Waveform** prints a user-selected range of the current Wave window display to a printer or a file	File > Print File > PrintPostscript	none
	**Cut** cut the selected signal from the Wave window	Edit > Cut	right mouse in pathname pane > Cut
	**Copy** copy the selected signal in the signal-name pane	Edit > Copy	right mouse in pathname pane > Copy

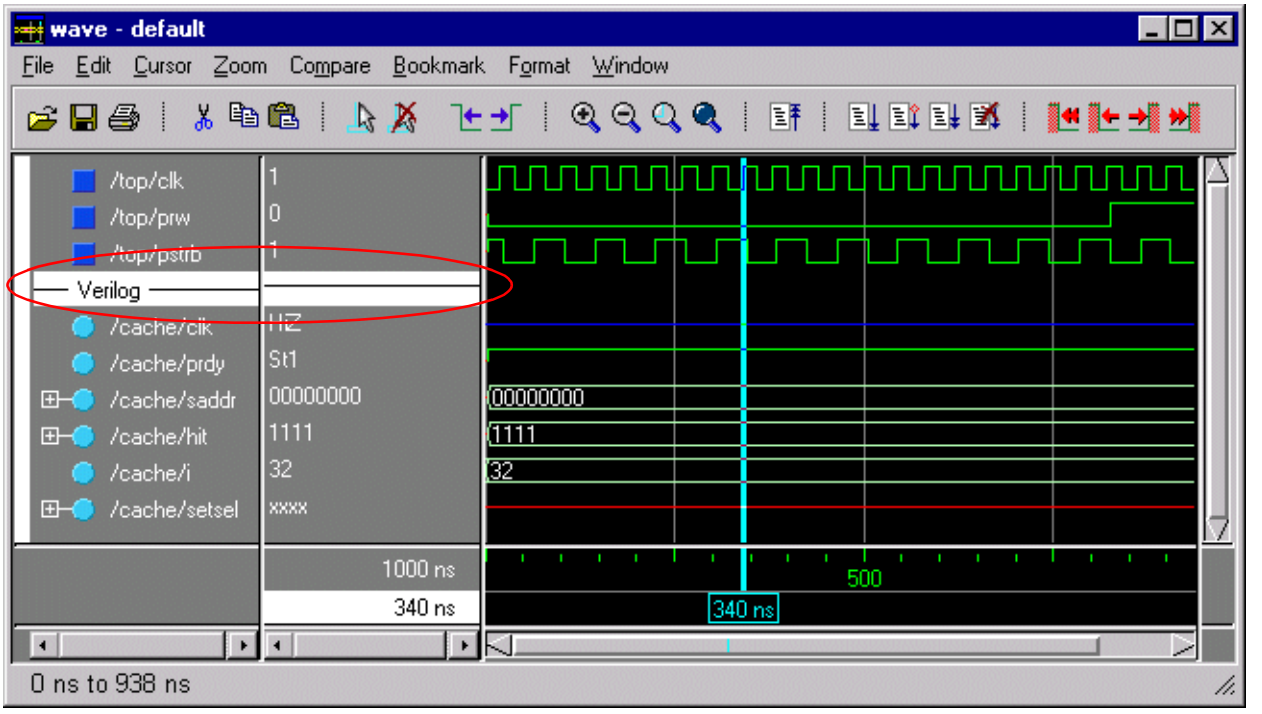
Wave window toolbar buttons		
Button	**Menu equivalent**	**Other options**
 **Paste** paste the copied signal above another selected signal	Edit > Paste	right mouse in pathname pane > Paste
 **Add Cursor** add a cursor to the center of the waveform pane	Cursor > Add Cursor	none
 **Delete Cursor** delete the selected cursor from the window	Cursor > Delete Cursor	none
 **Find Previous Transition** locate the previous signal value change for the selected signal	Edit > Search (Search Reverse)	keyboard: Shift + Tab **left** <arguments> see **left** command (CR-129)
 **Find Next Transition** locate the next signal value change for the selected signal	Edit > Search (Search Forward)	keyboard: Tab **right** <arguments> see **right** command (CR-174)
 **Zoom in 2x** zoom in by a factor of two from the current view	Zoom > Zoom In	keyboard: i I or + right mouse in wave pane > Zoom In
 **Zoom out 2x** zoom out by a factor of two from current view	Zoom > Zoom Out	keyboard: o O or - right mouse in wave pane > Zoom Out
 **Zoom area with mouse button 1** use the cursor to outline a zoom area	Zoom > Zoom Range	keyboard: r or R right mouse in wave pane > Zoom Area
 **Zoom Full** zoom out to view the full range of the simulation from time 0 to the current time	Zoom > Zoom Full	keyboard: f or F right mouse in wave pane > Zoom Full

Wave window toolbar buttons			
Button		**Menu equivalent**	**Other options**
	**Restart** reloads the design elements and resets the simulation time to zero, with the option of keeping the current formatting, breakpoints, and logfile	Main menu: Run > Restart	**restart** <arguments> see: **restart** (CR-170)
	**Run** run the current simulation for the default time length	Main menu: Run > Run <default_length>	use the **run** command at the VSIM prompt see: **run** (CR-176)
	**Continue Run** continue the current simulation run	Main menu: Run > Continue	use the **run -continue** command at the VSIM prompt see: **run** (CR-176)
	**Run -All** run the current simulation forever, or until it hits a breakpoint or specified break event	Main menu: Run > Run -All	use **run -all** command at the VSIM prompt see: **run** (CR-176), also see "Assertion settings page" (8-266)
	**Break** stop the current simulation run	none	none
	**Find First Difference** find the first difference in a waveform comparison	none	none
	**Find Previous Difference** find the previous difference in a waveform comparison	none	none
	**Find Next Difference** find the next difference in a waveform comparison	none	none
	**Find Last Difference** find the last difference in a waveform comparison	none	none

Using Dividers

Dividing lines can be placed in the pathname and values window panes by selecting **File > New Divider** (Wave window). Dividers serve as a visual aid to signal debugging, allowing you to separate signals and waveforms for easier viewing.

Dividing lines can be assigned any name, or no name at all. The default name is "New Divider." In the illustration below, VHDL signals have been separated from Verilog signals with a Divider called "Verilog." Notice that the waveforms in the waveform window pane have been separated by the divider as well.



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

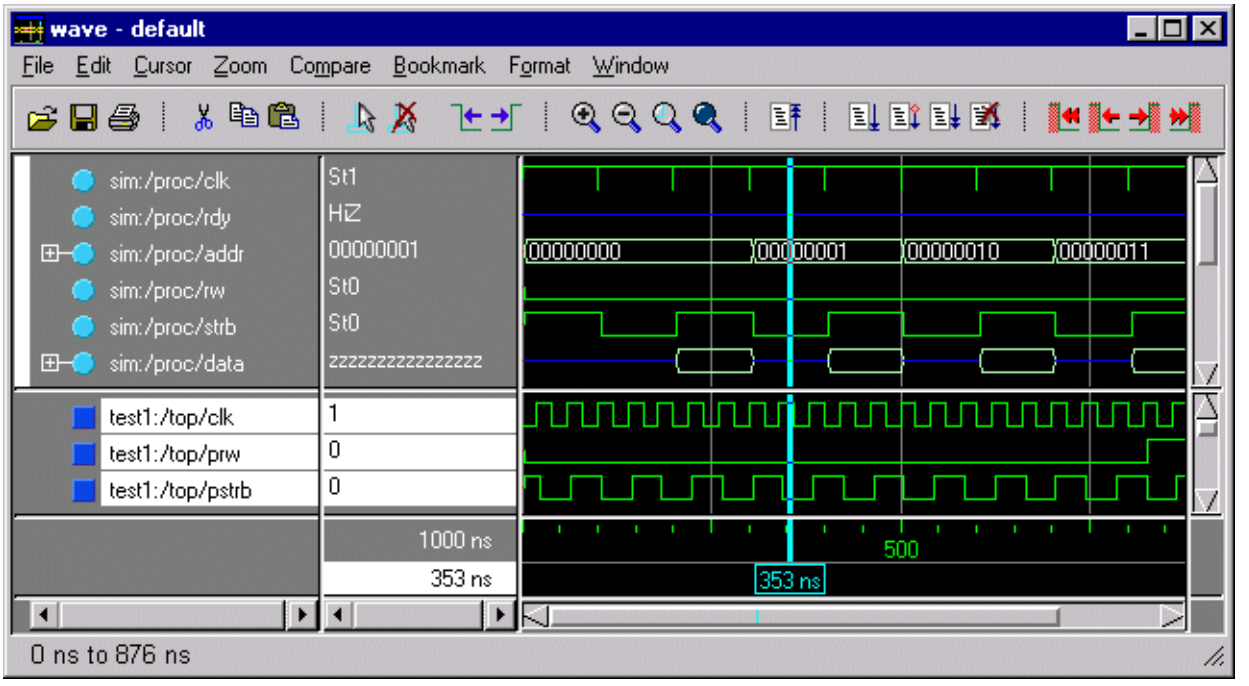
To delete a divider — Select the divider and either press the <Delete> key on your keyboard or select Delete from the pop-up menu

Splitting Wave window panes

The pathnames, values and waveforms window panes of the Wave window display can be split to accommodate signals from one or more datasets. Selecting **File > New Window Pane** (Wave window) creates a space below the selected waveset 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.)

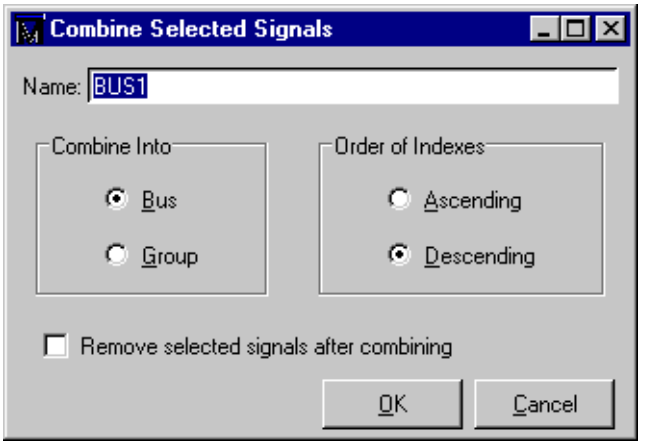
In the illustration below, the Wave window is split, showing the current active simulation with the prefix "sim," and a second view-mode dataset, with the prefix "test1."

For more information on viewing multiple simulations, see *Chapter 7 - Datasets (saved simulations) and virtuals*.



Combining items in the Wave window

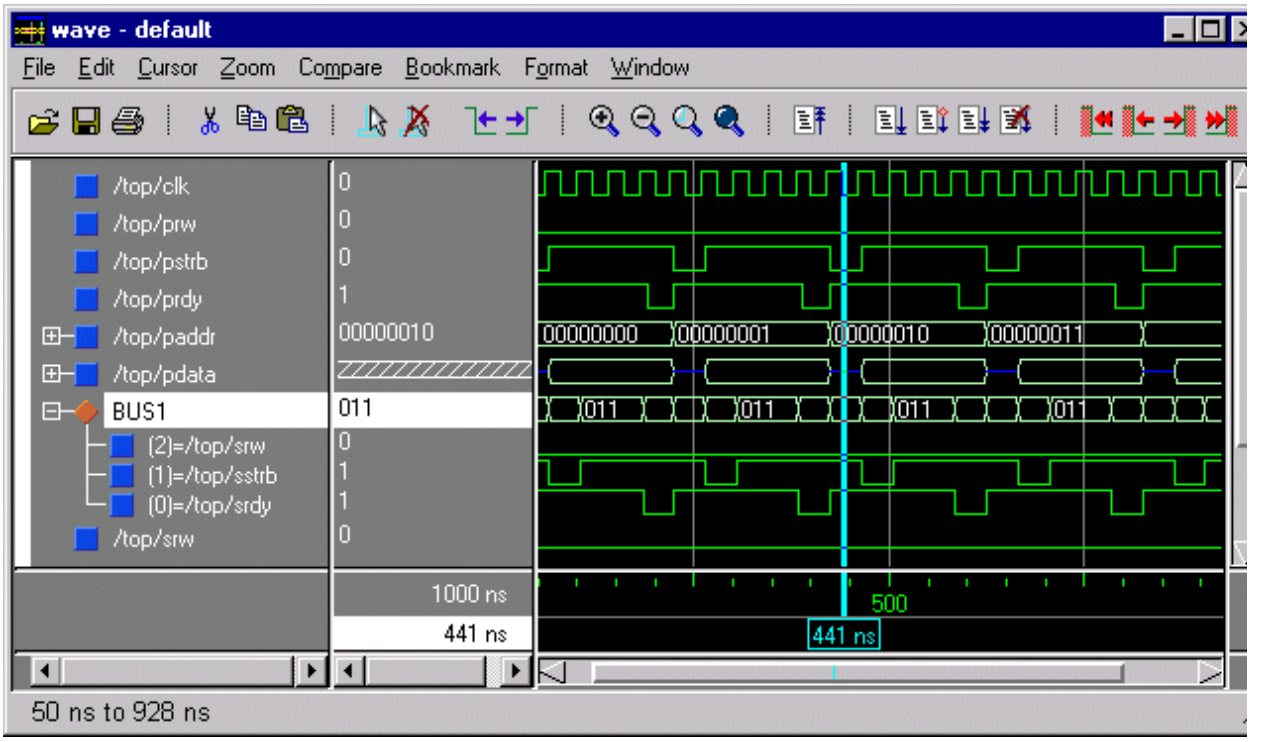
You can combine signals in the Wave window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. To create a bus, select one or more signals in the Wave window and then choose **Edit > Combine**. .



The **Combine Selected Signals** dialog box includes these options:

- **Combine Into**
  Only the Bus option is valid at this time. Groups are not currently implemented.
- **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 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

In the illustration below, three signals have been combined to form a new bus called BUS1. Note that the component signals are listed in the order in which they were selected in the Wave window. Also note that the bus' value 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)" (7-144) for information about other virtual items viewable in the Wave window.

Editing and formatting HDL items in the Wave window

Once you have the HDL 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" (8-235).)

To edit an item:

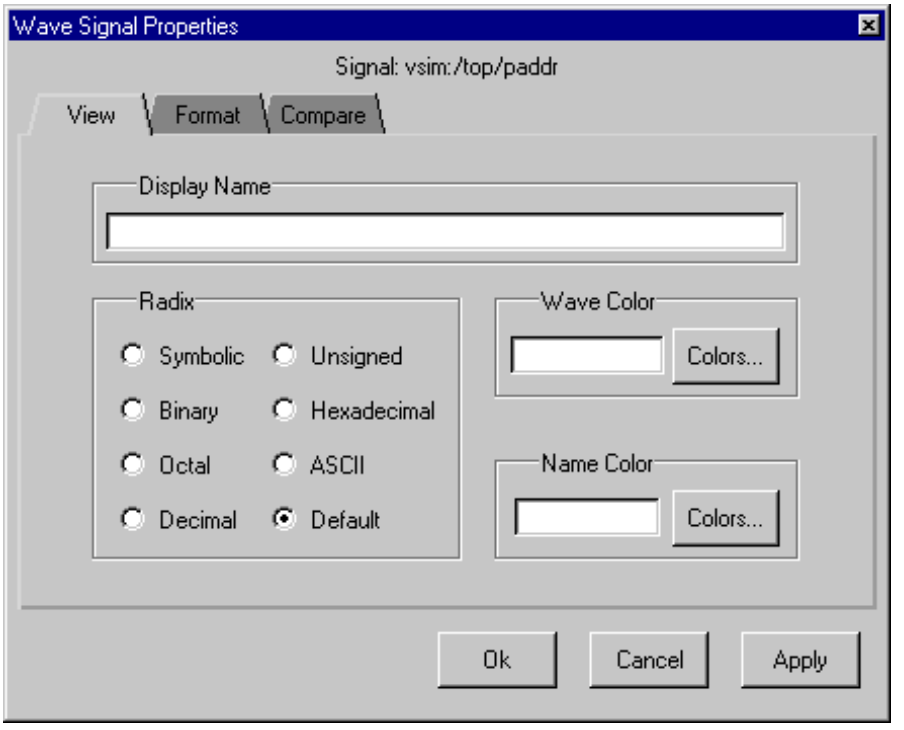
Select the item's label in the pathname pane or its waveform in the waveform pane. Move, copy, or remove the item by selecting commands from the Wave window **Edit menu** (8-221).

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

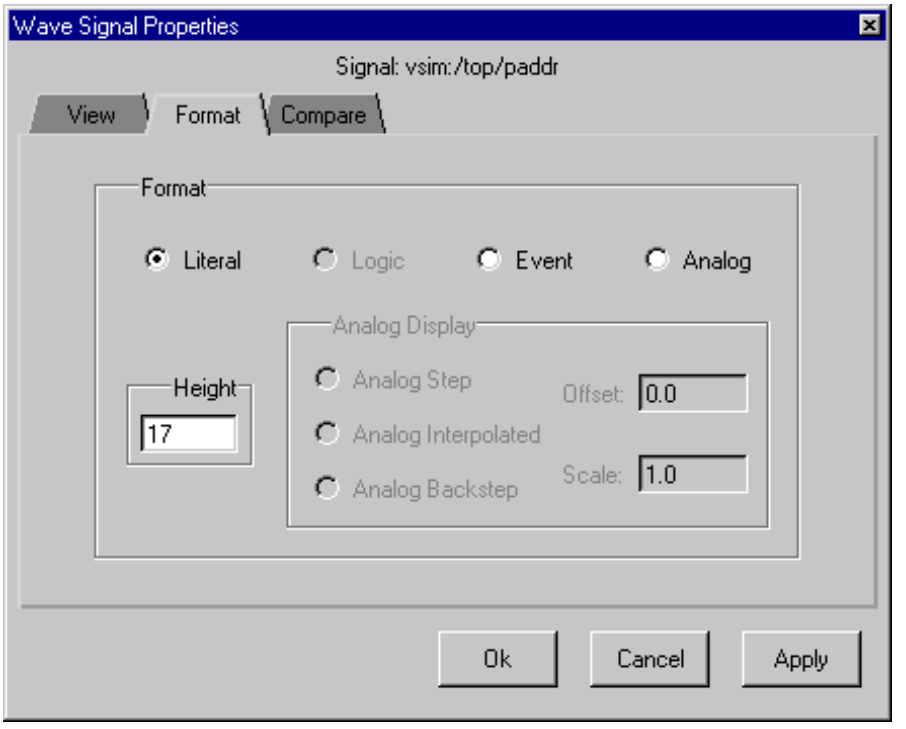
To format an item:

Select the item's label in the pathname pane or its waveform in the waveform pane, then select **Edit > Signal Properties** (Wave window). The resulting Wave Signal Properties dialog box has three tabs: View, Format, and Compare.



The **View** tab includes these options:

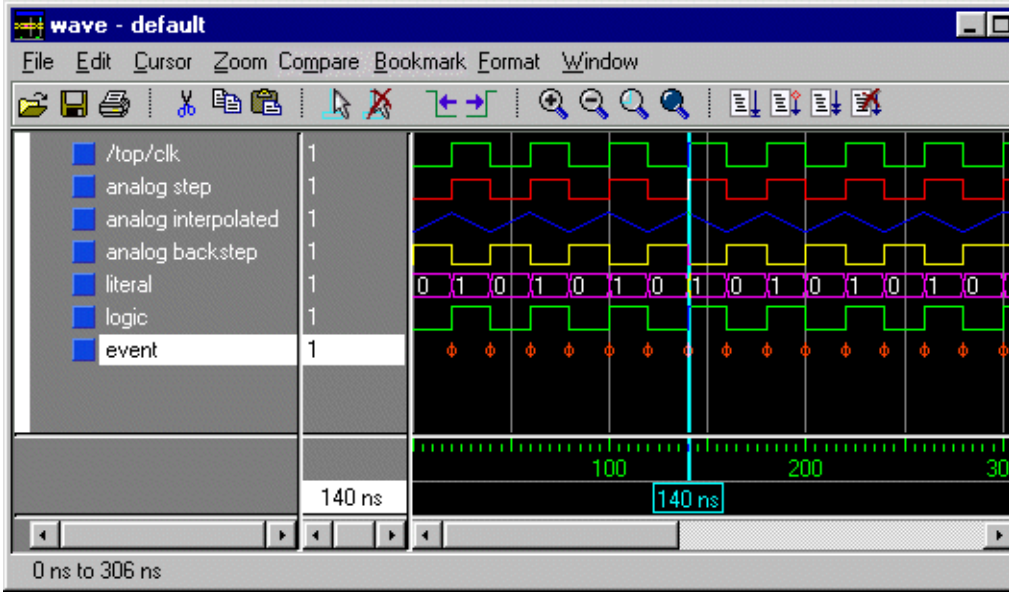
- **Display Name**
  Specifies a new name (in the pathname pane) for the selected signal.
- **Radix**
  Specifies the Radix of the selected signal(s). Setting this to default causes the signal's radix to change whenever the default is modified using the **radix** command (CR-166). Item values are not translated if you select Symbolic.
- **Wave Color**
  Specifies the waveform color. Select a new color from the color palette, or enter an X-Windows color name.
- **Name Color**
  Specifies the signal name's color. Select a new color from the color palette, or enter an X-Windows color name.



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]**
  All signals in the following illustration are the same */top/clk* signal. Starting with "analog step", the */top/clk* signal has been relabeled to illustrate each different wave format.



Analog Step
Displays a 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.

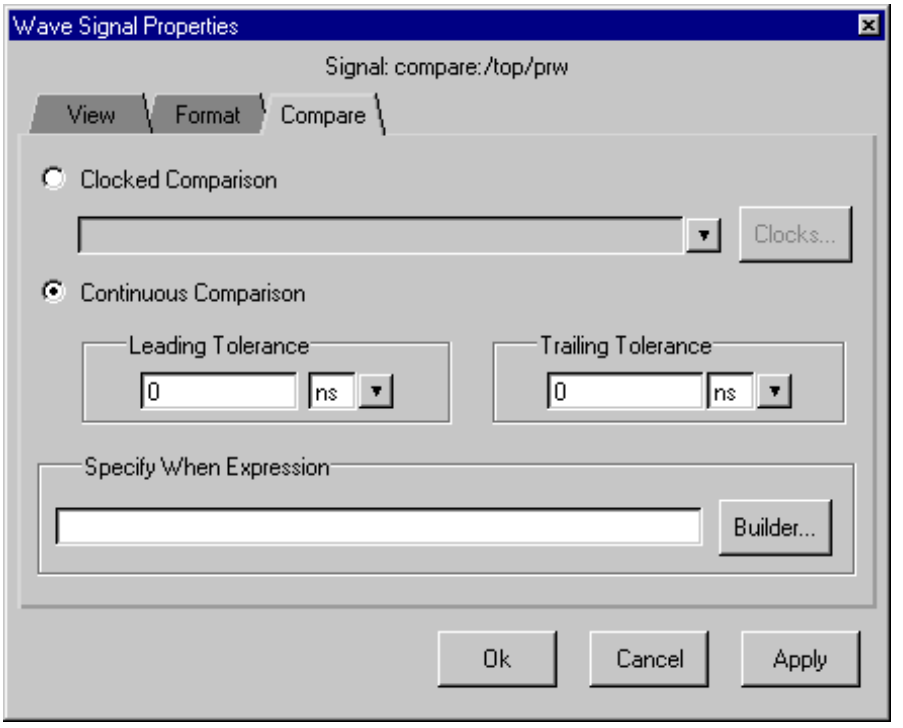
Offset and Scale
Allows you to adjust the scale of the item as it is seen on the display. Offset is the number of pixels offset from zero. The scale factor reduces (if less than 1) or increases (if greater than 1) the number of pixels displayed.

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 time

Verilog types:
All vectors
Scalar real
Scalar integers

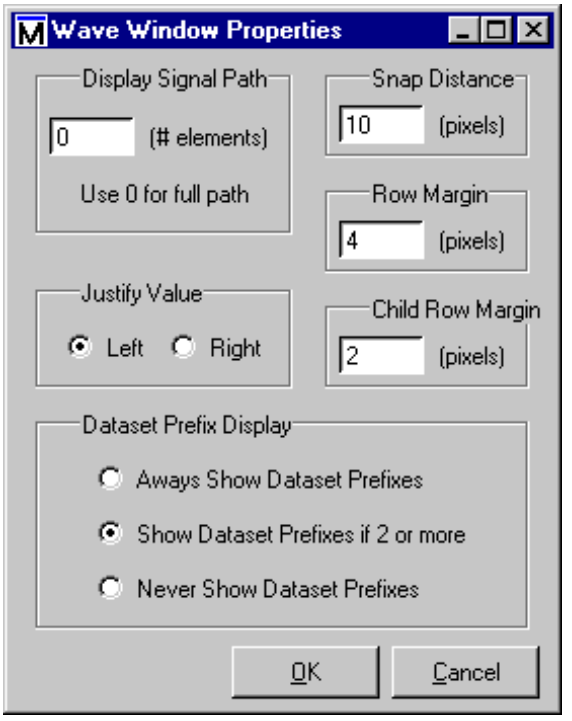
- **Height**
  Allows you to specify the height (in pixels) of the waveform.



The **Compare** tab includes the same options as those in the Add Signal Options dialog box (see Adding Signals, Regions and/or Clocks (11-307)).

Setting Wave window display properties

You can define the display properties of the pathname and values window panes by selecting **Edit > Display Properties** (Wave window).



The **Wave Window Properties** dialog box 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.

- **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 No Dataset Prefixes
  No dataset prefixes will be displayed. This selection is useful if you are running only a single simulation.

Sorting a group of HDL items

Select **Edit > Sort** to sort the items in the pathname and values panes.

Setting signal breakpoints

You can set signal breakpoints (a.k.a., when breakpoints; see the **when** command (CR-273) or "Setting signal breakpoints" (8-198) for more details) using a pop-up menu. Start by selecting a signal and then clicking your second (Windows) or third (UNIX) mouse button. Select Signal Breakpoints from the pop-up menu and you'll see six items:

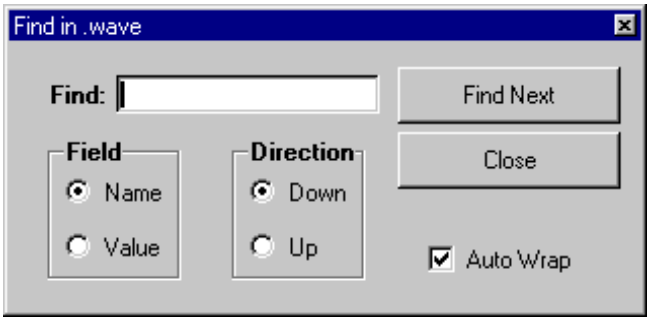
- **Add**
  Creates a signal breakpoint on the selected signal
- **Edit Breakpoints**
  Opens the Edit When dialog. See "Setting signal breakpoints" (8-198) for more information.
- **Edit All Breakpoints**
  Opens the Breakpoints dialog. See "Setting file-line breakpoints" (8-205) for more information.
- **Remove Signal**
  Removes the signal breakpoint from the selected signal
- **Remove All Signals**
  Removes all signal breakpoints
- **Show All**
  Shows a list of all signal breakpoints

When a breakpoint is hit, a message appears in the transcript window about which signal caused the breakpoint. Breakpoints created by the **when** command (CR-273) are not affected by the **Remove All Signals** menu pick, nor are they reported via **Show All**.

Finding items by name or value in the Wave window

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.

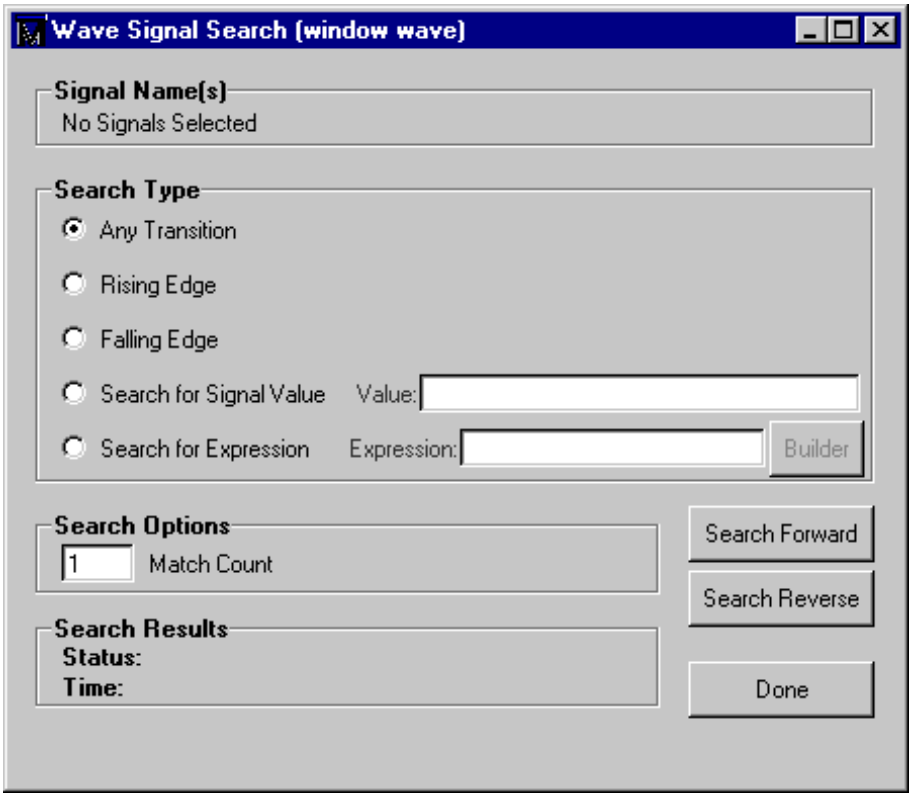
Choose either the Name or Value field to search and enter the value to search for in the Find field. **Find** the item by searching **Down** or **Up** through the Wave window display. **Auto Wrap** continues the search at the top of the window.



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-291).

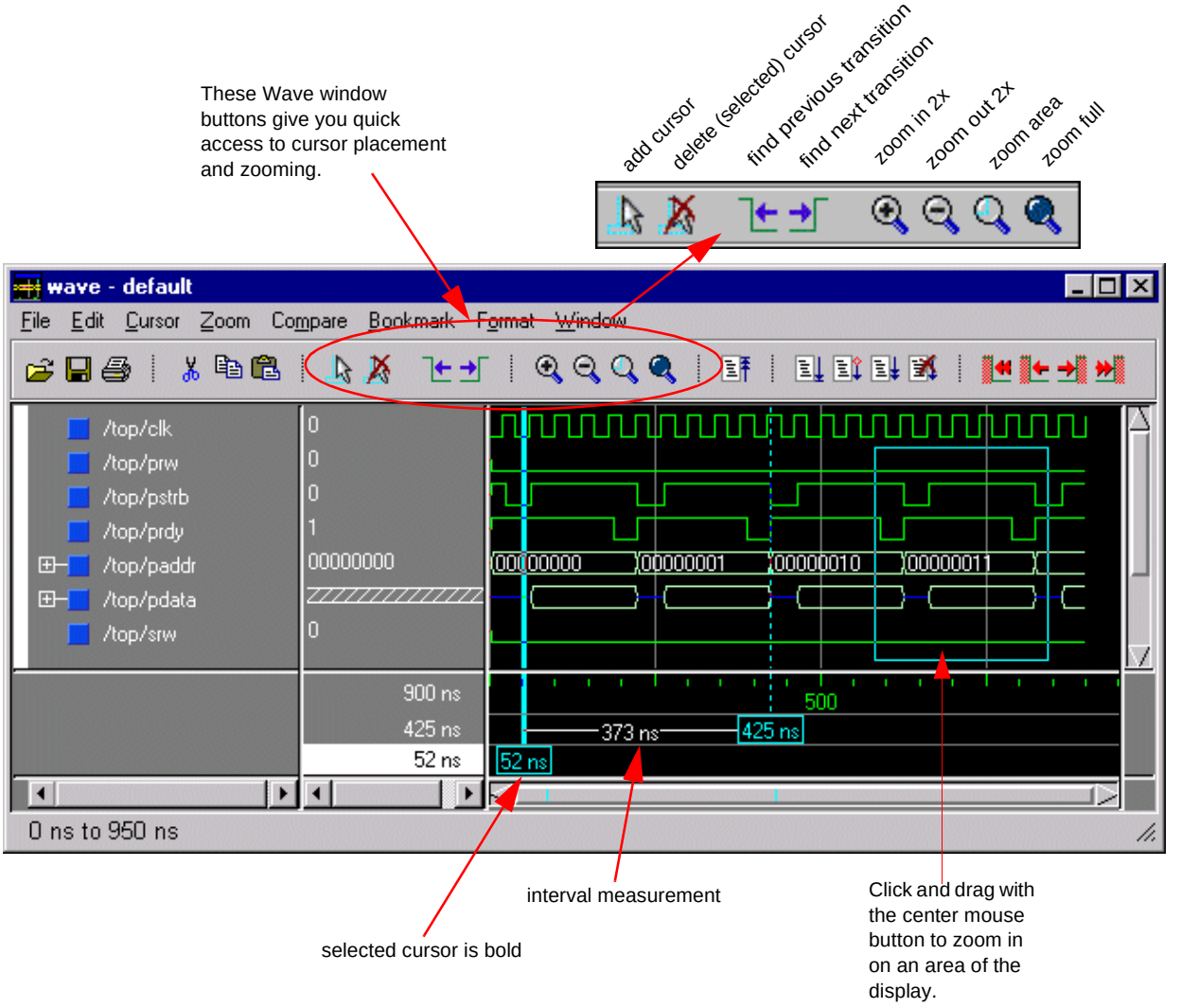
Note: If your signal values are displayed in binary radix, see "Searching for binary signal values in the GUI" (CR-300) for details on how signal values are mapped between a binary radix and std_logic.

- **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" (8-275) 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-302) for more information.
- **Search Options: Match Count**
  You can search for the n-th transition or the n-th 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 with the **Cursor > Add Cursor** menu selection (or the Add Cursor button shown below). The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin dashed lines. Remove cursors by selecting them and selecting **Cursor > Delete Cursor** (or the Delete Cursor button shown below).

Add Cursor add a cursor to the center of the waveform window	**Delete Cursor** delete the selected cursor from the window

Finding a cursor

The cursor value (on the **Goto** list) corresponds to the simulation time of that cursor. Choose a specific cursor view by selecting **Cursor > Goto**.

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. Model*Sim* 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.

The cursors are designed to snap to the closest wave edge to the left on the waveform that the mouse pointer is positioned over. You can control the snap distance via the **Edit > Display Properties** menu selection.

You can position a cursor without snapping by dragging in the area 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

Zooming - changing the waveform display range

Zooming lets you change the simulation range in the waveform pane. You can zoom with either the context menu, toolbar buttons, mouse, keyboard, or commands.

Using the Zoom menu

You can use the Wave window menu bar, or call up the context menu by clicking the right mouse button in the waveform pane.

The Zoom menu options include:

- **Zoom Area with Mouse Button 1**
  Use mouse button 1 to create a zoom area. Position the mouse cursor to the left side of the desired zoom interval, press mouse button 1 and drag to the right. Release when the box has expanded to the right side of the desired zoom interval.
- **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 Full**
  Redraws the display to show the entire simulation from time 0 to the current simulation time.
- **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 area** use the cursor to outline a zoom area
 **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

Zooming with the mouse

To zoom with the mouse, position the mouse cursor to the left side of the desired zoom interval, press the middle mouse button (three-button mouse), or <Ctrl>+left mouse button (two-button mouse), and while continuing to press, drag to the right and then release at the right side of the desired zoom interval.

Zooming keyboard shortcuts

See "Wave window mouse and keyboard shortcuts" (8-244) for a complete list of Wave window keyboard shortcuts.

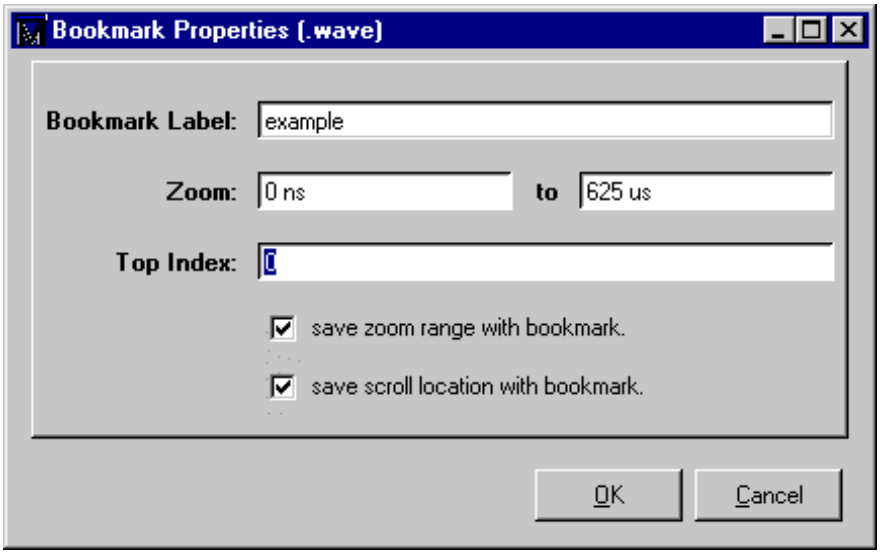
Saving zoom range and scroll position with bookmarks

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" (8-219)) 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 **bookmark add wave** command (CR-44) for details.

To add a bookmark, select **Bookmark > Add Bookmark** (Wave window).

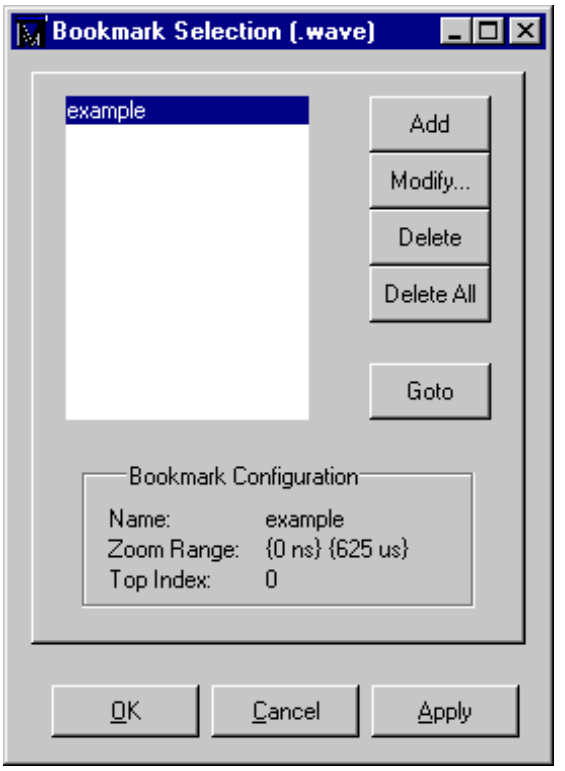


The Bookmark Properties dialog includes the following options.

- **Bookmark Label**
  A text label to assign to the bookmark. The label will identify the bookmark on the Bookmark menu.
- **Zoom**
  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 Bookmark menu, and the Wave window will be zoomed and scrolled accordingly.

To edit or delete a bookmark, select **Bookmark > Edit Bookmarks** (Wave window).



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
- **Delete All** (bookmark delete wave)
  Delete all bookmarks
- **Goto** (bookmark goto wave)
  Zoom and scroll the Wave window using the selected bookmark

Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

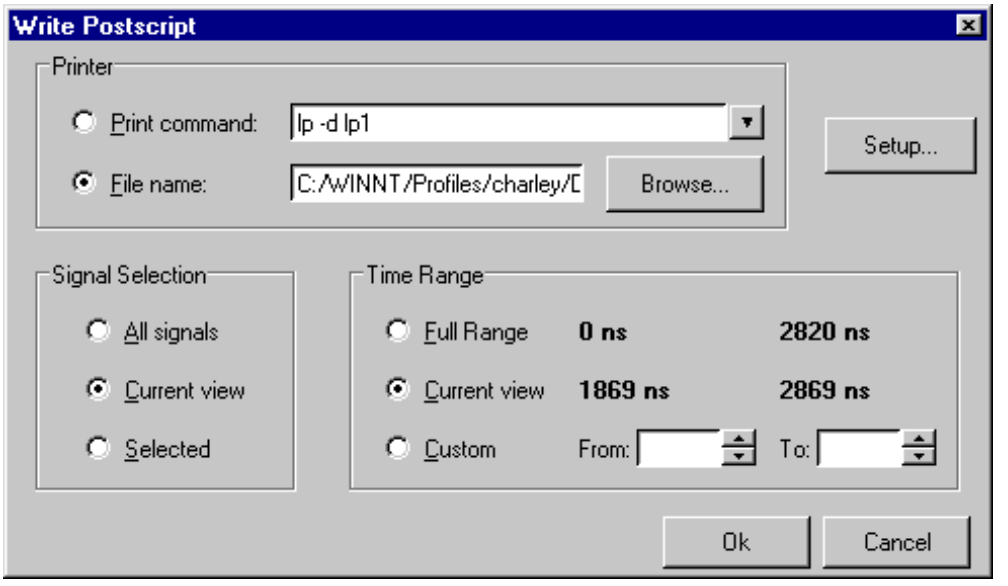
Mouse action	Result
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom(vertical scroll) or far left or right (horizontal scroll)
< middle mouse-button - click in scroll bar trough> (UNIX) only	scrolls window to position of click

Keystroke	Action
i I or +	zoom in
o O or -	zoom out
f or F	zoom full; mouse pointer must be over the the cursor or waveform panes
l or L	zoom last
r or R	zoom range
<arrow up>	scroll waveform display up by selecting the item above the currently selected item
<arrow down>	scroll waveform display down by selecting the item below the currently selected item
<arrow left>	scroll waveform display left
<arrow right>	scroll waveform display right
<page up>	scroll waveform display up by a page
<page down>	scroll waveform display 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 pane for text strings

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 **write wave** command (CR-285)). Printing and writing preferences are controlled by the dialog box shown below.



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

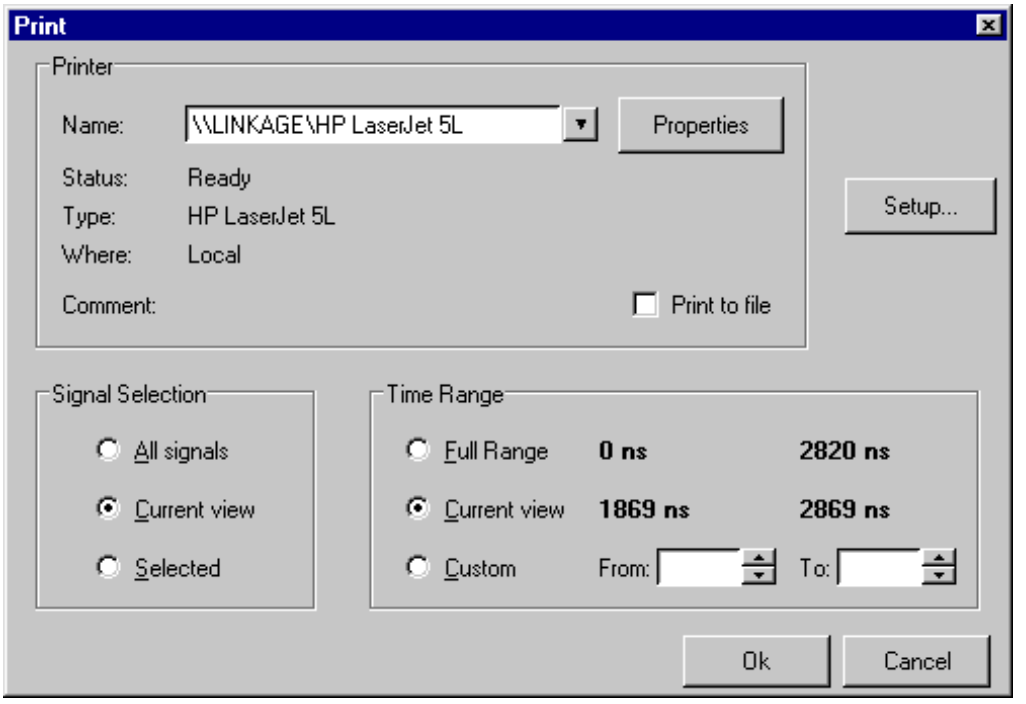
- **Full Range**
  Print all specified signals in the full simulation range.
- **Current view**
  Print the specified signals for the viewable time range.
- **Custom**
  Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "Printer Page Setup" (8-248)

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.



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

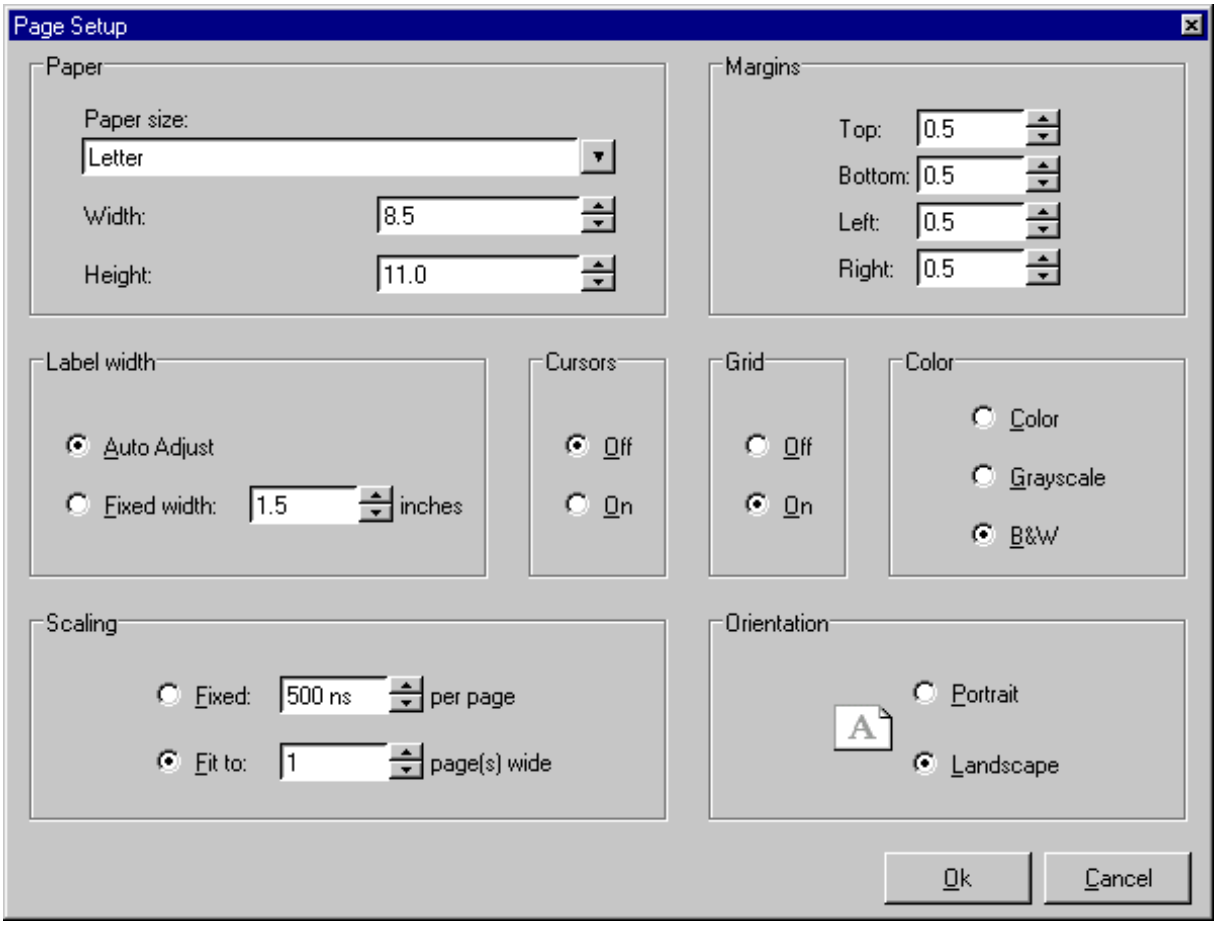
- **Full Range**
  Print all specified signals in the full simulation range.
- **Current view**
  Print the specified signals for the viewable time range.
- **Custom**
  Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "Printer Page Setup" (8-248)

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**).

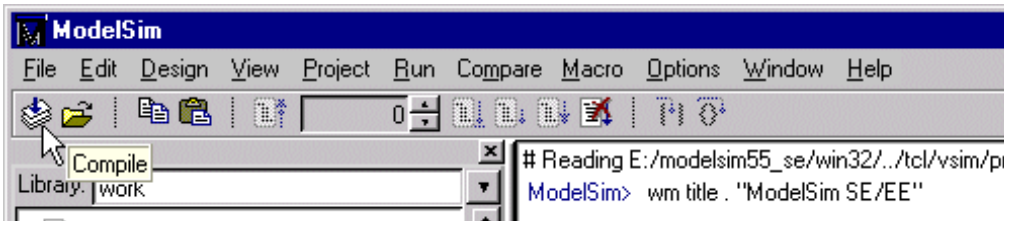


- **Paper Size**
  Select your output page size from a number of options; also choose the paper width and height.
- **Margins**
  Specify the page margins; changing the **Margin** will change the **Scale** and **Page** specifications.
- **Label width**
  Specify Auto Adjust to accommodate any length label, or set a fixed label width.
- **Cursors**
  Turn printing of cursors on or off.
- **Grid**
  Turn printing of grid lines on or off.

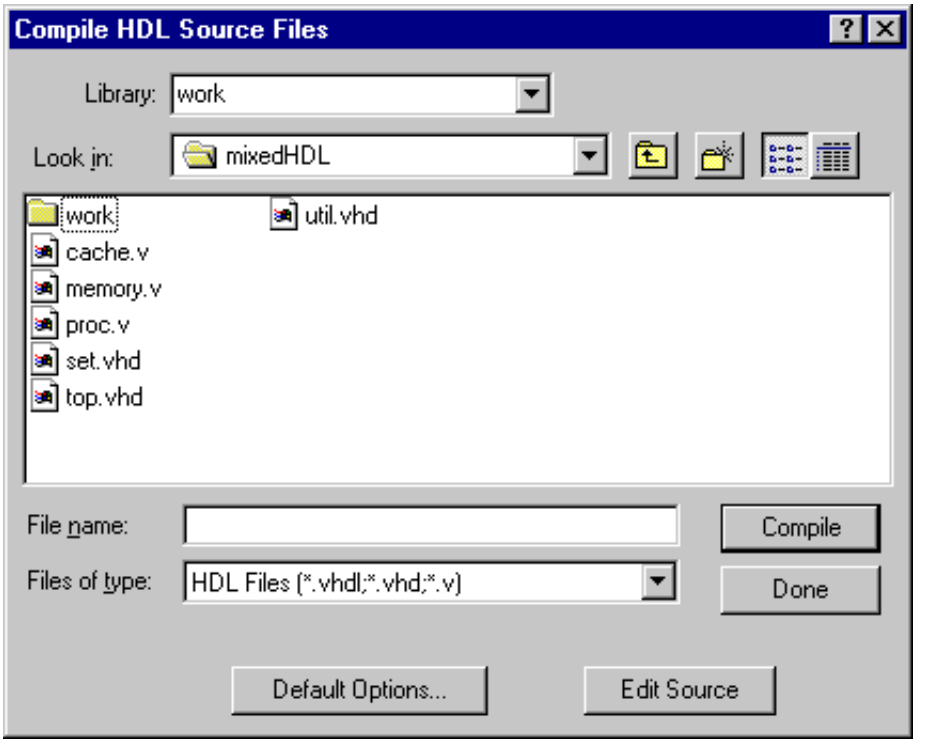
- **Color**
  Select full color printing, grayscale or black and white.
- **Scaling**
  Specify a **Fixed** output time width in nanoseconds per page – the number of pages output is automatically computed; or, select **Fit to** to define the number of pages to be output based on the paper size and time 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 HDL Source Files** dialog box to compile VHDL or Verilog designs. For information on compiling in a project, see "Getting started with projects" (2-28). To open the Compile HDL Source Files dialog, select the **Compile** button (Main window) or **Design > Compile**.



The Compile HDL Source Files dialog box opens as shown below.



From the Compile HDL Source Files dialog box you can:

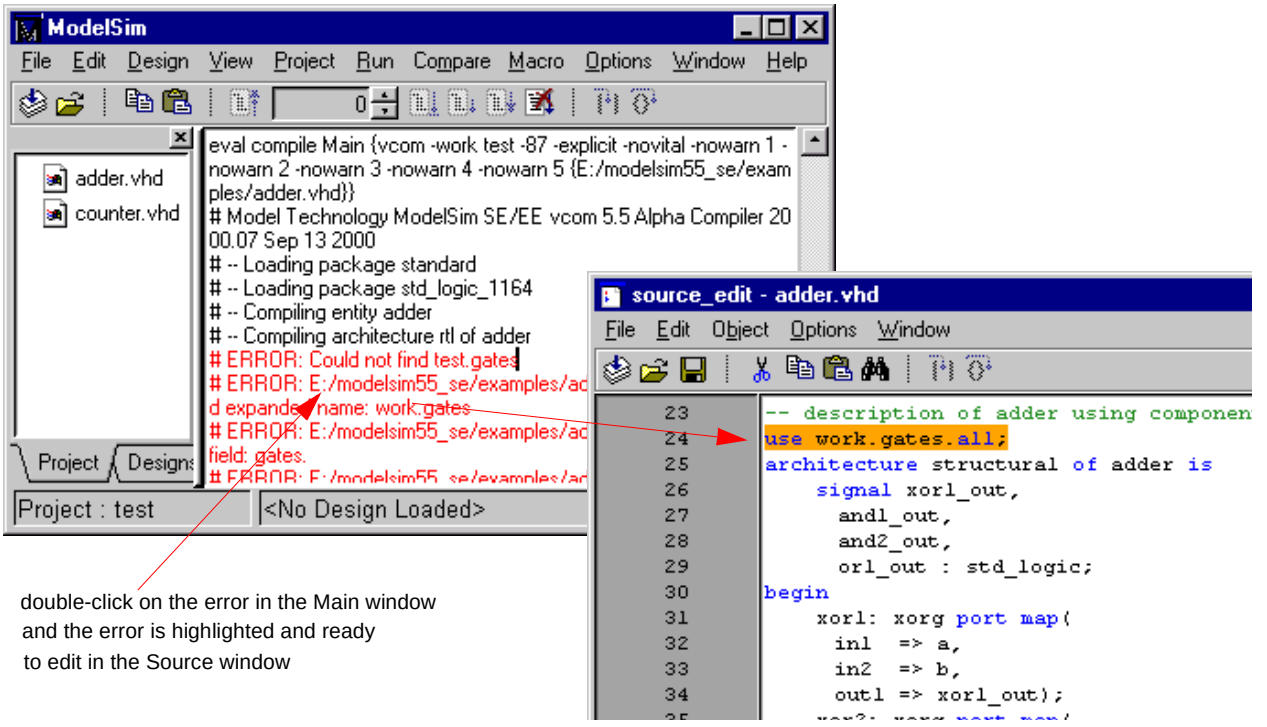
- select source files to compile in any language combination
- specify the target library for the compiled design units
- select among the compiler options for either VHDL or Verilog

Select the **Default Options** button to change the compiler options, see "Setting default compile options" (8-252) for details. The same Compiler Options dialog box can also be accessed by selecting **Options > Compile (**Main window) or by selecting Compile Properties from the context menu in Project tab.

Select the **Edit Source** button to view or edit a source file via the Compile dialog box. See "Source window" (8-201) for additional source file editing information.

Locating source errors during compilation

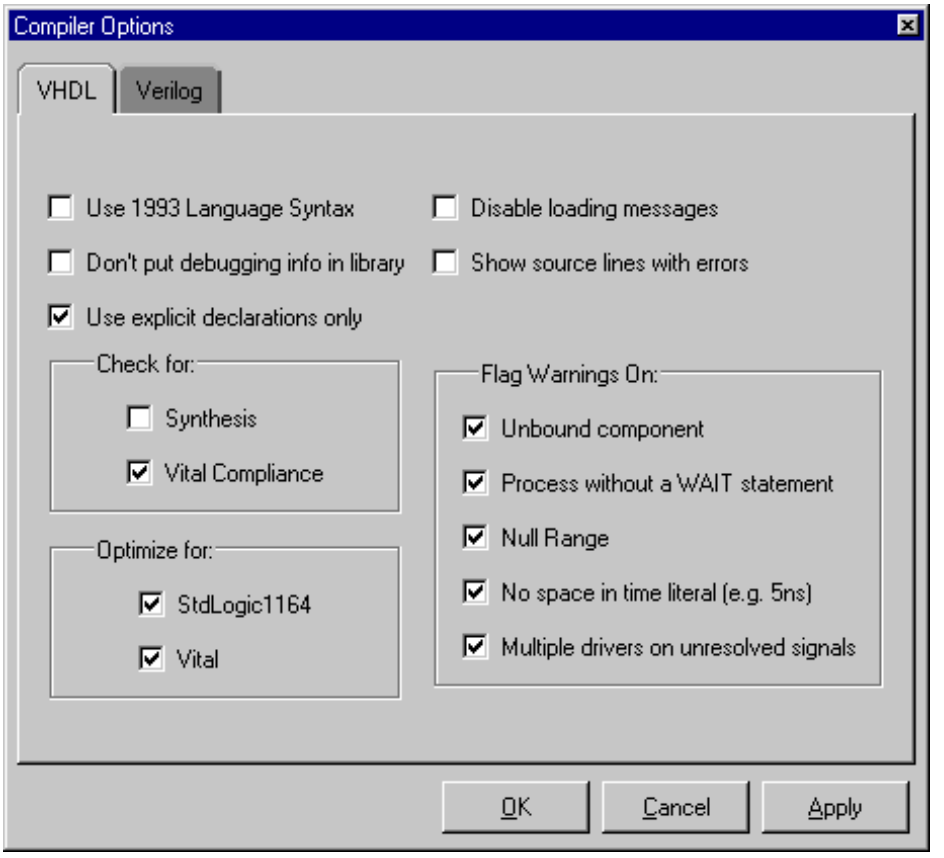
If a compiler error occurs during compilation, a red error message is printed in the Main transcript. Double-click on the error message to open the source file in an editable Source window with the error highlighted.



Setting default compile options

Select **Options** > **Compile** (Main window) to bring up the **Compiler Options** dialog box shown below. **OK** accepts the changes made and closes the dialog box. **Apply** makes the changes with the dialog box open so you can test your settings. **Cancel** closes the dialog box and makes no changes. The options found on each page of the dialog box are detailed below. Changes made in the **Compiler Options** dialog box become the default for all future simulations.

VHDL compiler options page



- **Use 1993 language syntax**
  Specifies the use of VHDL93 during compilation. The 1987 standard is the default. Same as the **-93** switch for the **vcom** command (CR-217). Edit the VHDL93 (B-405) variable in the *modelsim.ini* file to set a permanent default.
- **Don't put debugging info in library**
  Models compiled with this option do not use any of the Model*Sim* 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're done debugging. Same as the **-nodebug** switch for the **vcom** command (CR-217). See "Source code security and -nodebug" (E-433) for more details. Edit the NoDebug (B-397) 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** switch for the **vcom** command (CR-217). Edit the Explicit (B-397) 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** switch for the **vcom** command (CR-217). Edit the Quiet (B-397) 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** switch for the **vcom** command (CR-217). Edit the Show_source (B-397) variable in the *modelsim.ini* file to set a permanent default.

Flag 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 (B-397) 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 (B-397) 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 (B-397) 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 (B-397) 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 (B-397) variable in the *modelsim.ini* file to set a permanent default.

Check for:

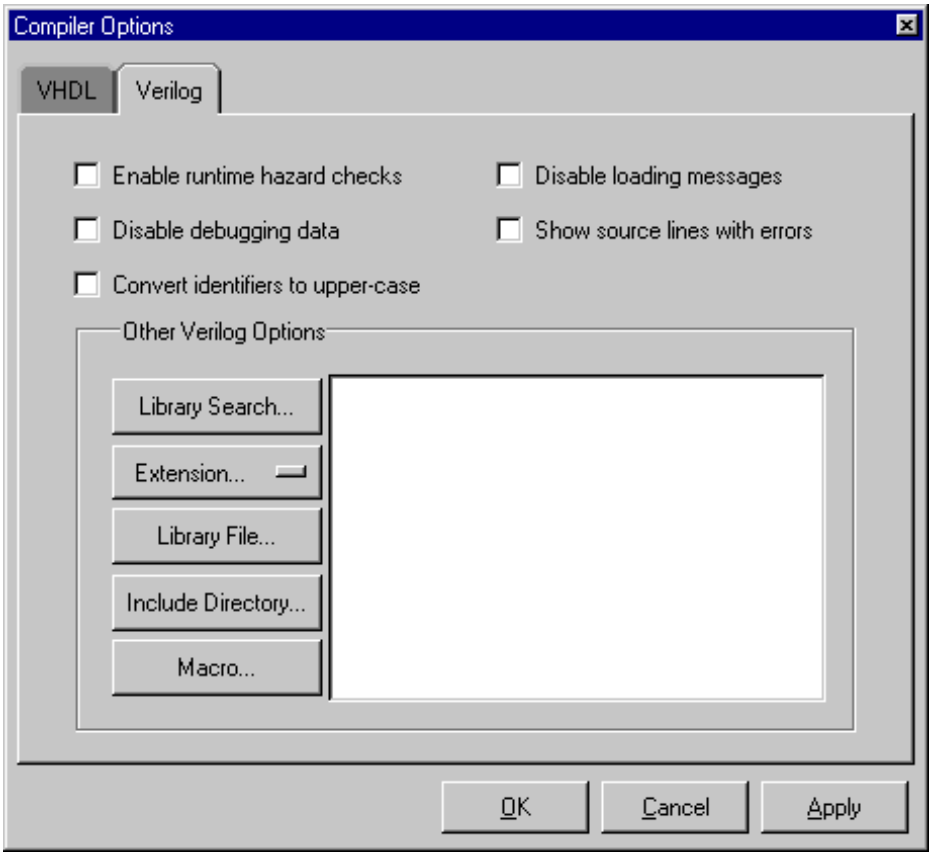
- **Synthesis**
  Turns on limited synthesis-rule compliance checking. Edit the CheckSynthesis (B-396) variable in the *modelsim.ini* file to set a permanent default.

- **Vital Compliance**
  Toggle Vital compliance checking. Edit the NoVitalCheck (B-397) 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 (B-397) variable in the *modelsim.ini* file to set a permanent default.
- **Vital**
  Toggle acceleration of the Vital packages. Edit the NoVital (B-397) variable in the *modelsim.ini* file to set a permanent default.

Verilog compiler options page



- **Enable run-time hazard checks**
  Enables the run-time hazard checking code. Same as the **-hazards** switch for the **vlog** command (CR-250). Edit the Hazard (B-398) variable in the *modelsim.ini* file to set a permanent default.

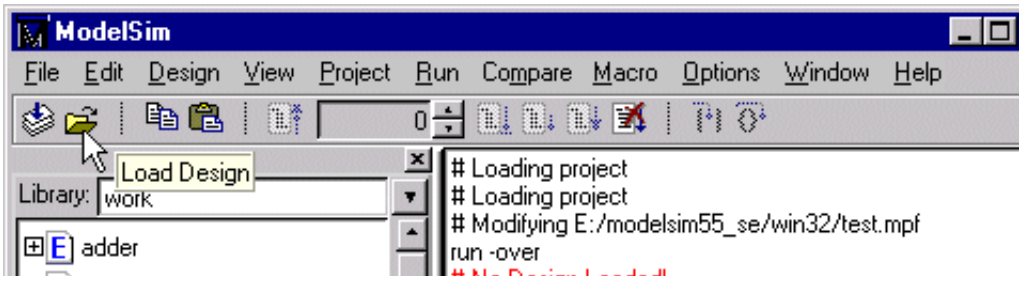
- **Disable debugging data**
  Models compiled with this option do not use any of the Model*Sim* debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you're done debugging. Same as the **-nodebug** switch for the **vlog** command (CR-250). See "Source code security and -nodebug" (E-433) for more details. Edit the NoDebug (B-397) variable in the *modelsim.ini* file to set a permanent default.
- **Convert Verilog identifiers to upper-case**
  Converts regular Verilog identifiers to uppercase. Allows case insensitivity for module names. Same as the **-u** switch for the **vlog** command (CR-250). Edit the UpCase (B-398) 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** switch for the **vlog** command (CR-250). Edit the Quiet (B-397) 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** switch for the **vlog** command (CR-250). Edit the Show_source (B-397) variable in the *modelsim.ini* file to set a permanent default.

Other Verilog Options:

- **Library Search**
  Specifies the Verilog source library directory to search for undefined modules. Same as the **-y <library_directory>** switch for the **vlog** command (CR-250).
- **Extension**
  Specifies the suffix of files in the library directory. Multiple suffixes can be used. Same as the **+libext+<suffix>** switch for the **vlog** command (CR-250).
- **Library File**
  Specifies the Verilog source library file to search for undefined modules. Same as the **-v <library_file>** switch for the **vlog** command (CR-250).
- **Include Directory**
  Specifies a directory for files included with the **'include filename** compiler directive. Same as the **+incdir+<directory>** switch for the **vlog** command (CR-250).
- **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>]** switch for the **vlog** command (CR-250).

Simulating with the graphic interface

You can use a project or the **Load Design** dialog box to simulate a compiled design. For information on simulating in a project, see "Getting started with projects" (2-28). To open the Load Design dialog, select the **Load Design** button (Main window) or **Design > Load Design**.



Five pages - **Design**, **VHDL**, **Verilog**, **Libraries**, and **SDF** - allow you to select various simulation options.

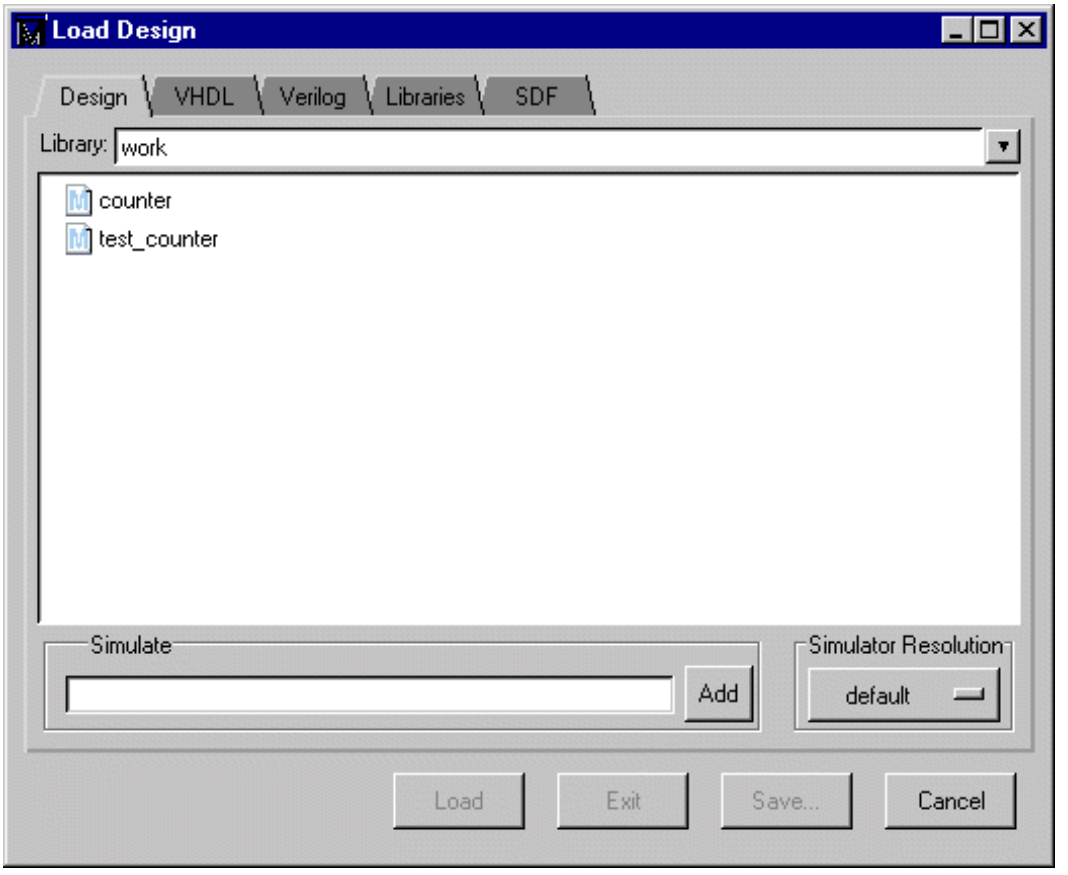
You can switch between pages to modify settings, then begin simulation by selecting the **Load** button. If you select **Cancel**, all selections remain unchanged and you are returned to the Main window; the **Exit** button (only active before simulation) closes Model*Sim*. The **Save Settings** button allows you to save the preferences on all pages to a DO (macro) file.

Compile before you simulate

To begin simulation you must have compiled design units located in a design library, see "Creating a design library" (4-57).

Note: Many of the dialog box options discussed in this section include parenthetical elements that correspond to **vsim** (CR-258) command options. For example, **Simulator Resolution** (-time [<multiplier>]<time_unit>).

Design selection page



Note: The Exit button closes the Load Design dialog box and quits Model*Sim*.

The **Design** page includes these options:

- **Library**
  Specifies a library to view. Make certain your selection is a valid Model*Sim* library — the library must be created by ModelSim and it's directory must include a *_info* file.
- **Design Unit**
  This hierarchical list allows you to select one top-level entity or configuration to be simulated. All entities, configurations, and modules that exist in the specified library are displayed in the list box. Architectures can be viewed by selecting the "+" box before any name.

- **Simulate** (<configuration> | <module> | <entity> [(<architecture>)])
  Specifies the design unit(s) to simulate. You can simulate several Verilog top-level modules or a VHDL top-level design unit in one of three ways:

  1. 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>
     ```

  2. Click on a name in the **Design Unit** list below and click the **Add** button.
  3. Leave this field blank and click on a name in the **Design Unit** list (single unit only).

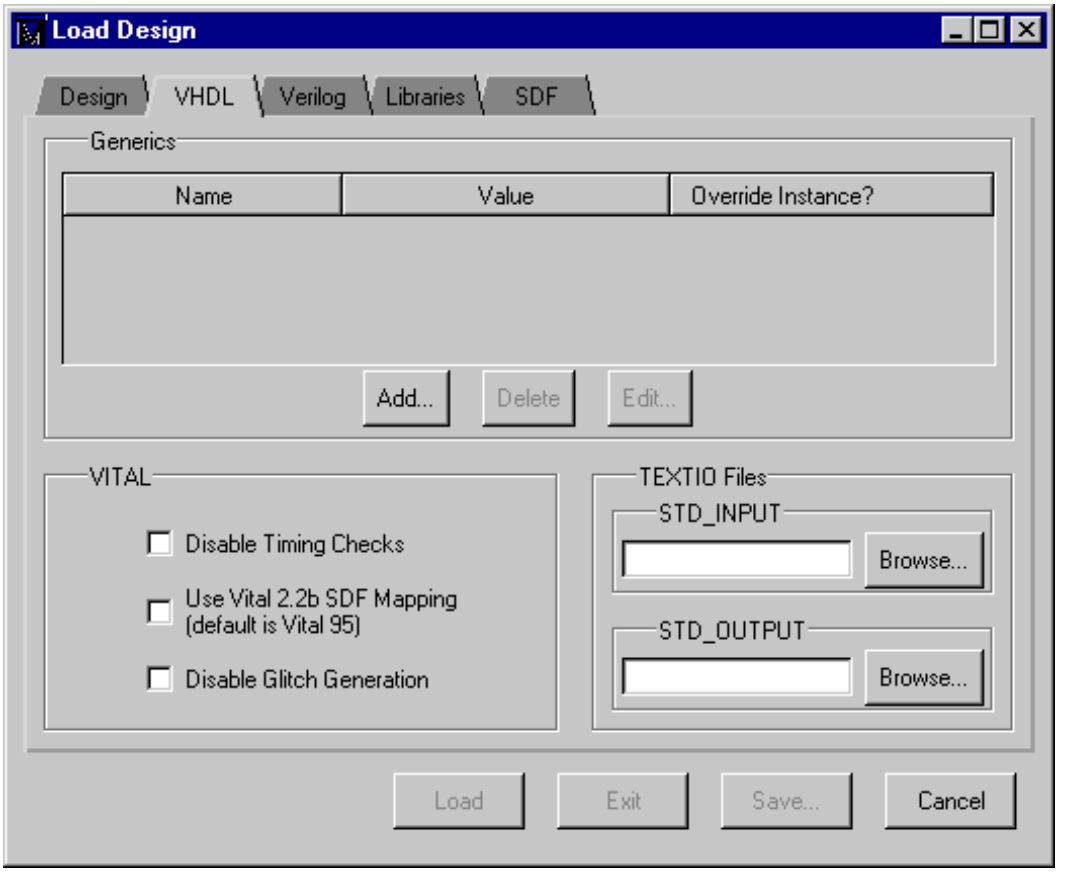
- **Simulator Resolution**
  (-time [<multiplier>]<time_unit>)
  The drop-down menu sets the simulator time units (original default is ns).

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, "Selecting the time resolution" (4-58).

VHDL settings page

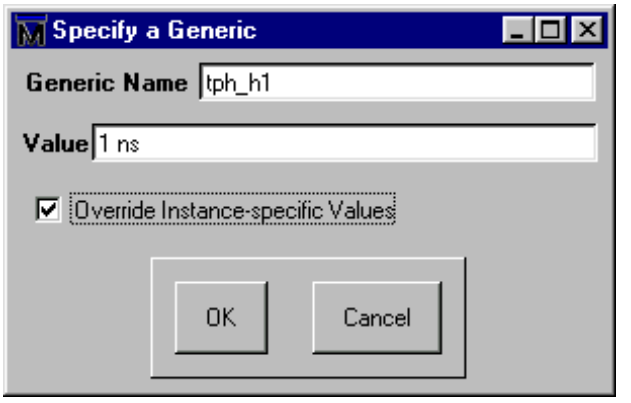


The **VHDL** page includes these options:

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).
- **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 parameter. 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 Instance** column of the **Generics** list.

  The **OK** button adds the generic to the **Generics** listing; **Cancel** dismisses the dialog box without changes.

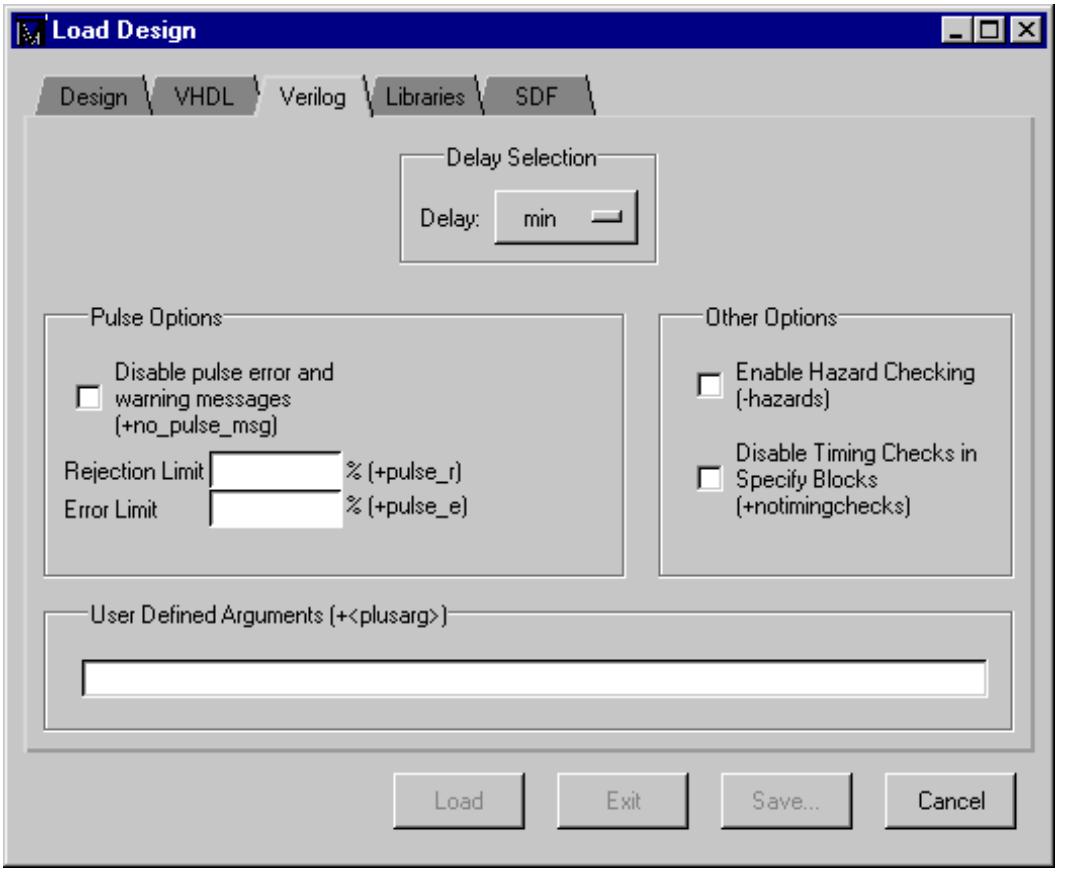
VITAL

- **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 settings page



The **Verilog** page includes these options:

- **Delay Selection** (+mindelays | +typdelays | +maxdelays)
  Use the drop-down menu to select timing for min:typ:max expressions.
  Also see: "Timing check disabling" (4-58).

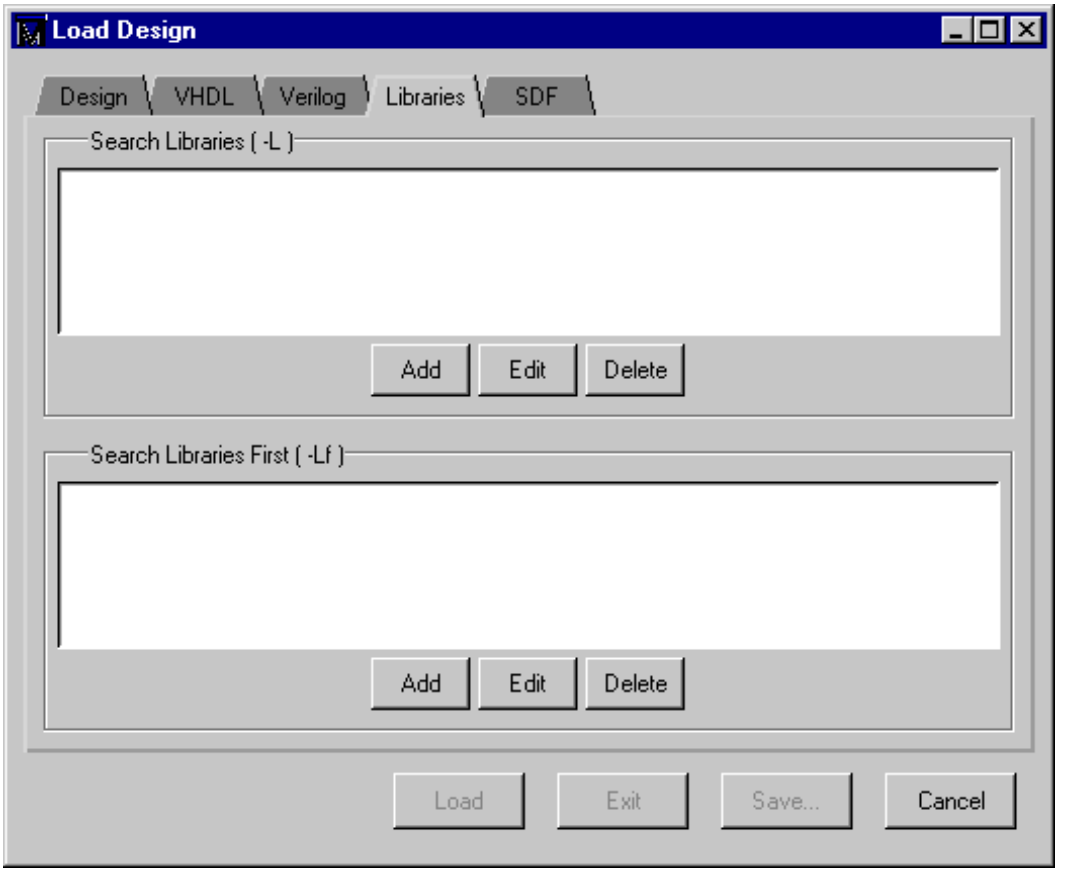
Pulse Options

- **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.
- **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 Model*Sim* may incorrectly parse them.

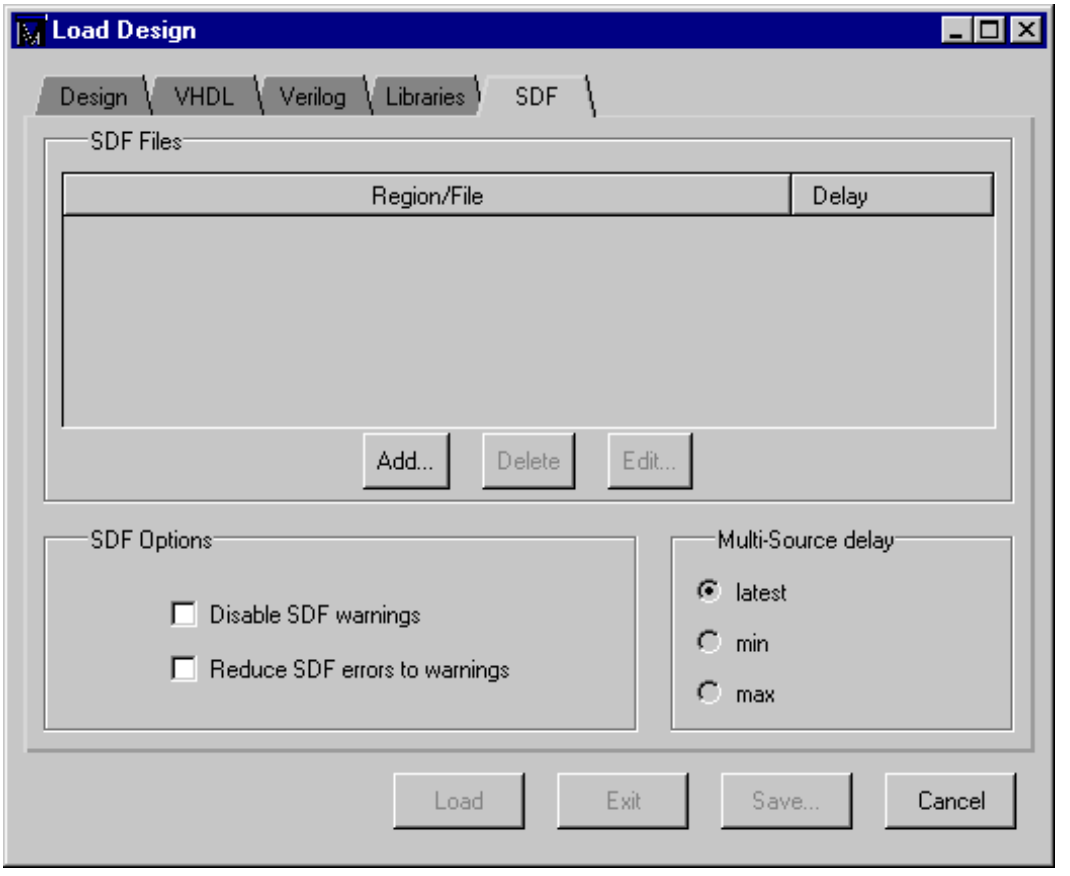
Libraries settings page



The **Libraries** page includes these options:

- **Search Libraries** (-L)
  Specifies the library to search for design units instantiated from Verilog.
- **Search Libraries First** (-Lf)
  Same as Search Libraries but these libraries are searched before ‘uselib.

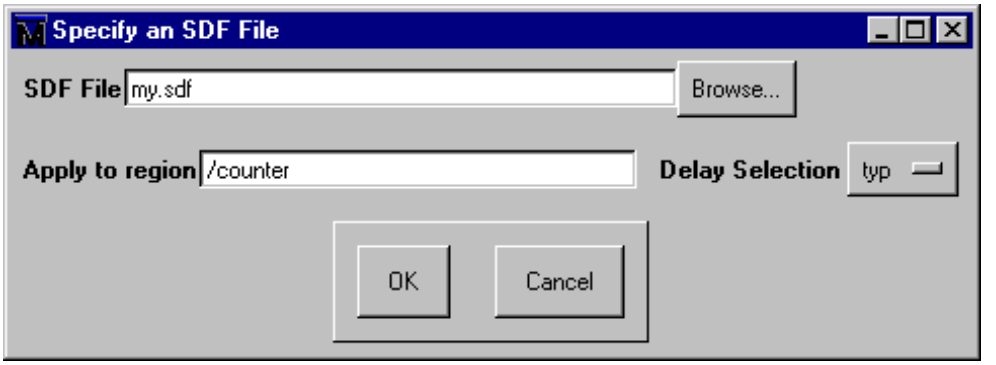
SDF settings page



The **SDF** (Standard Delay Format) page includes these options:

SDF Files

The **Add** button opens a dialog box that allows you to specify the SDF files to load for the current simulation; files are then added to the **Region/File** list. You may also select a file on the listing to **Delete** or **Edit** (opens the dialog box below).



From the **Specify an SDF File** 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 Selection** (-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" (12-326).

  The **OK** button places the specified SDF file and delay on the **Region/File** list; **Cancel** dismisses the dialog box without changes.

SDF options

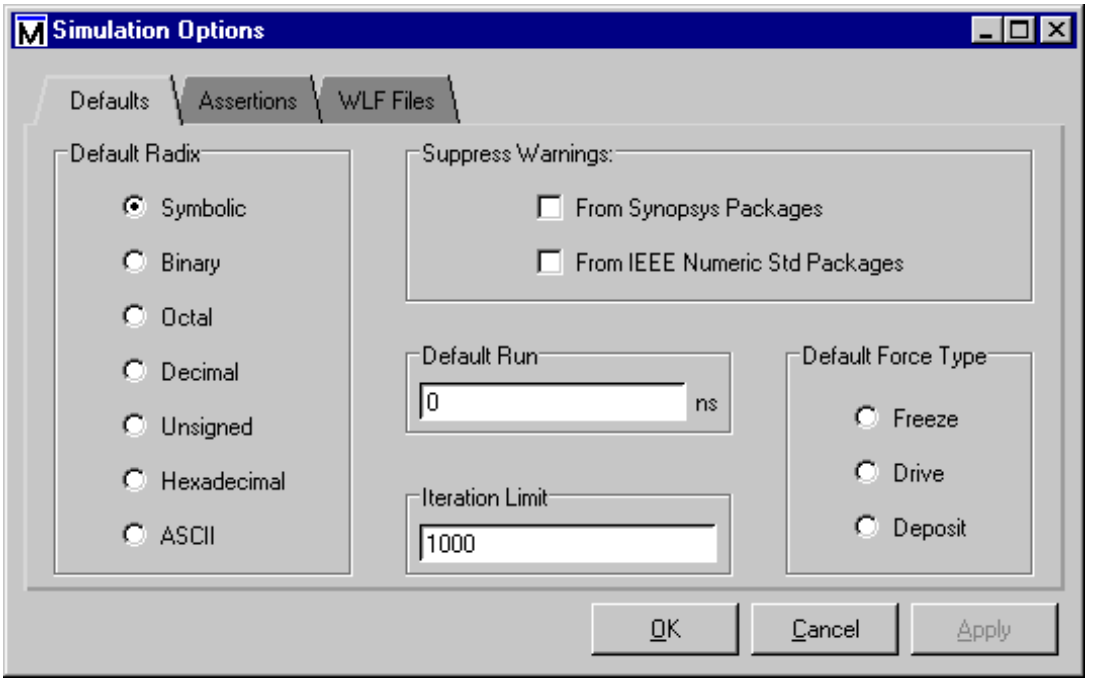
- **Disable SDF warnings** (-sdfnowarn)
  Select to disable warnings from the SDF reader.
- **Reduce SDF errors to warnings** (-sdfnoerror)
  Change SDF errors to warnings so the simulation can continue.
- **Multi-Source Delay** (-multisource_delay <sdf_option>)
  Select **max**, **min** or **latest** 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.

Setting default simulation options

Select **Options > Simulation...** (Main window) to bring up the **Simulation Options** dialog box shown below. Options you can set for the current simulation include: default radix, default force type, default run length, iteration limit, warning suppression, break on assertion specifications, and WLF file configuration. **OK** accepts the changes made and closes the dialog box. **Apply** makes the changes with the dialog box open so you can test your settings. **Cancel** closes the dialog box and makes no changes. The options found on each page are detailed below.

Note: 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. You can use Notepad (see **notepad** command (CR-141)) to edit the variables in *modelsim.ini* if you wish. See also, "Projects and system initialization" (2-25) for more information.

Default settings page



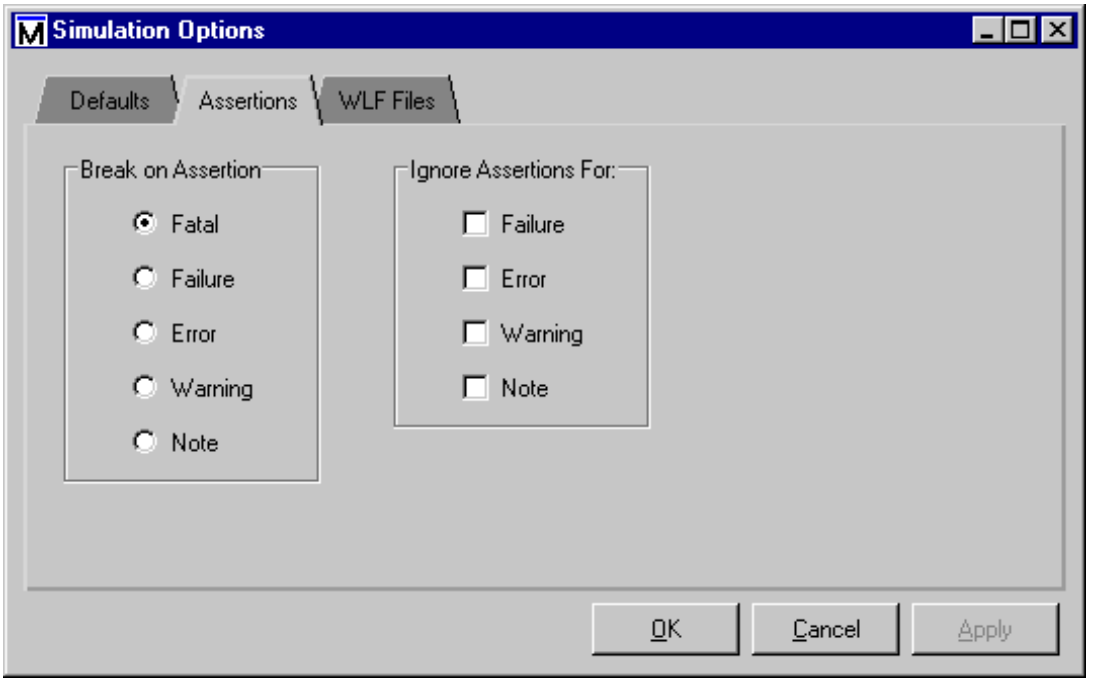
The **Defaults** page includes these options:

- **Default Radix**
  Sets the default radix for the current simulation run. You can also use the **radix** (CR-166) command to set the same temporary default. A permanent default can be set by editing the DefaultRadix (B-399) variable in the *modelsim.ini* file. The chosen radix is used for all commands (**force** (CR-121), **examine** (CR-115), **change** (CR-52) 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 (B-400) 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 (B-400) 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 (B-400) 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 (B-399) 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 (B-399) variable in the *modelsim.ini* file to set a permanent default.

Assertion settings page



The **Assertions** page includes these options:

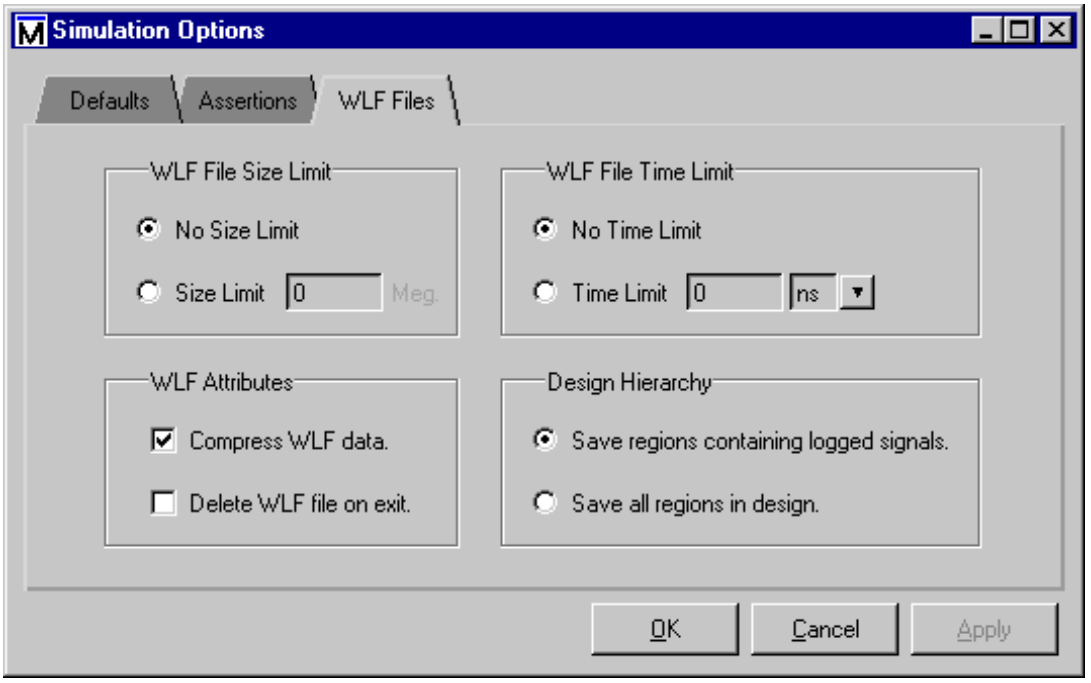
- **Break on Assertion**
  Selects the assertion severity that will stop simulation. Edit the BreakOnAssertion (B-398) variable in the *modelsim.ini* file to set a permanent default.

- **Ignore Assertions For**
  Selects the assertion type to ignore for the current simulation. Multiple selections are possible. Edit the IgnoreFailure, IgnoreError, IgnoreWarning, and IgnoreNote (B-399) 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 settings page



The **WLF Files** page 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 (B-401) 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 (B-401) variable in the *modelsim.ini* file to set a permanent default.

- **Compress WLF data**
  Compresses WLF files to reduce their size. You would typically only disable compression for troubleshooting purposes. Edit the WLFCompress (B-401) variable in the *modelsim.ini* file to set a permanent default.
- **Delete WLF file on exit**
  Specifies whether the WLF file should be deleted when the simulation ends. Edit the WLFDeleteOnQuit (B-401) variable in the *modelsim.ini* file to set a permanent default.
- **Design Hierarchy**
  Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. Edit the WLFSaveAllRegions (B-401) variable in the *modelsim.ini* file to set a permanent default.

Model*Sim* tools

Several tools are available from Model*Sim* menus. The menu selections to locate the tools are below the tool names. Follow the links for more information on each tool.

- "The Button Adder" (8-269)
  Window > Customize (any window)
  Allows you to add a temporary function button or toolbar to any window.
- "The Macro Helper" (8-270)
  Macro > Macro Helper (Main window)
  Creates macros by recording mouse movements and key strokes. UNIX only (excluding Linux).
- "The Tcl Debugger" (8-271)
  Macro > Tcl Debugger (Main window)
  Helps you debug your Tcl procedures.
- "The GUI Expression Builder" (8-275)
  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-297).

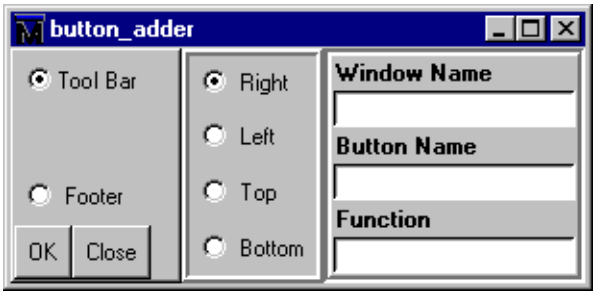
The Button Adder

The Model*Sim* Button Adder creates a single button, or a combined button and toolbar in any currently opened Model*Sim* window. The button exists until you close the window. (See "Buttons the easy way" (8-279).)

Note: When a button is created with the Button Adder, the commands that created the button are echoed in the transcript. The transcript can then be saved and used as a DO file, allowing you to reuse the commands to recreate the button from a startup DO file. See "Using a startup file" (B-404) for additional information.

Invoke the Button Adder from any Model*Sim* window menu: **Window > Customize**.

You have the following options for adding a button:



- **Window Name** is the name of the window to which you wish to add the button.
- **Button Name** is the button's label.
- **Function** can be any command or macro you might execute from the Model*Sim* 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:

- **Toolbar** places the button on a new toolbar.
- **Footer** adds the button to the Main window's status bar.

Justify the button within the menu bar/toolbar with these selections:

- **Right** places the button on the right side of the menu/toolbar.
- **Left** adds the button on the left side of the menu/toolbar.
- **Top** places the button at the top/center of the menu bar or toolbar.
- **Bottom** places the button at the bottom/center of the menu bar or toolbar.

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-148) command. Actions recorded by the Macro Helper can only take place within the Model*Sim* GUI (window sizing and repositioning are not recorded because they are handled by your operating system's window manager). In addition, the **run** (CR-176) commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.

Select **Macro > 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-141).

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-135) 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 **Macro > Tcl Debugger** (Main window) to run the debugger. Make sure you use the Model*Sim* 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.

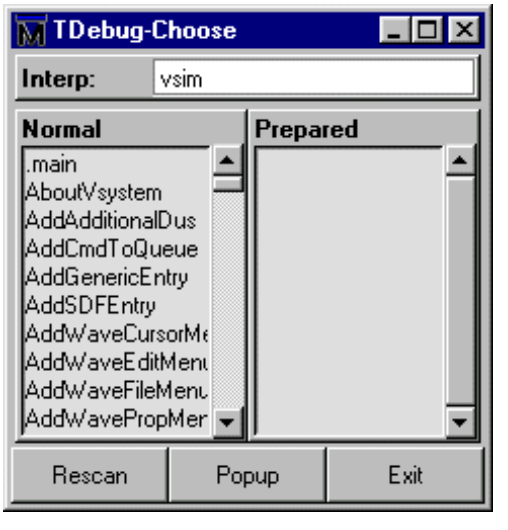
How it works

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 what 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 **Macro > 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.

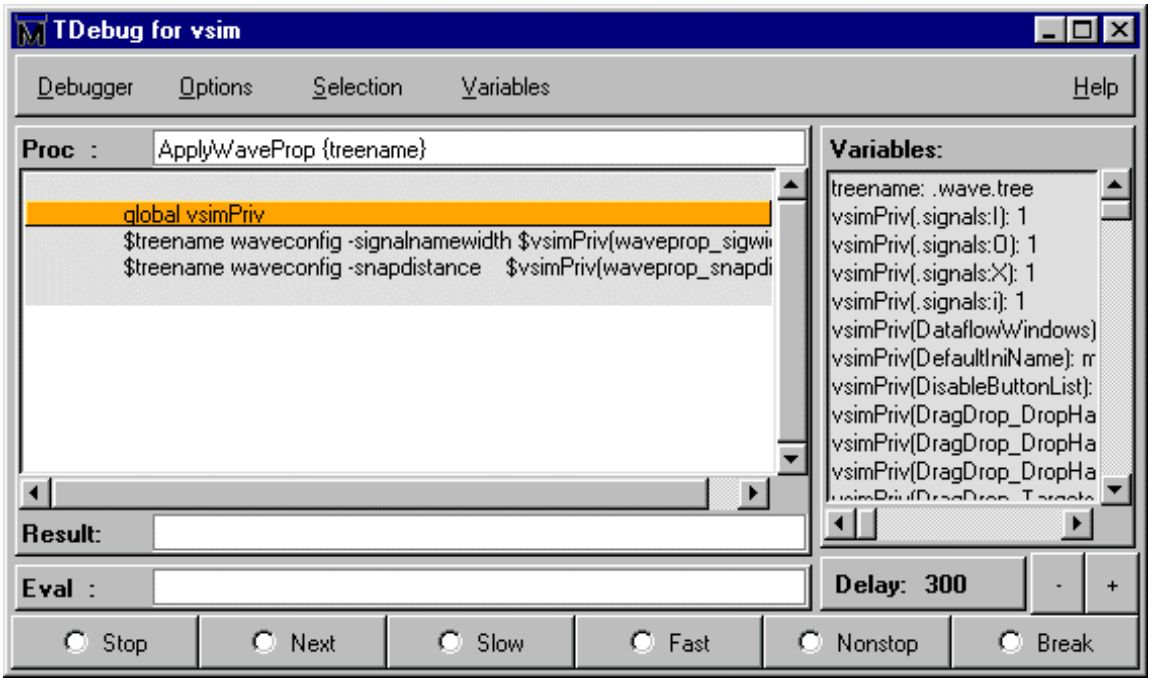
Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.

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 Model*Sim*, 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.



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.

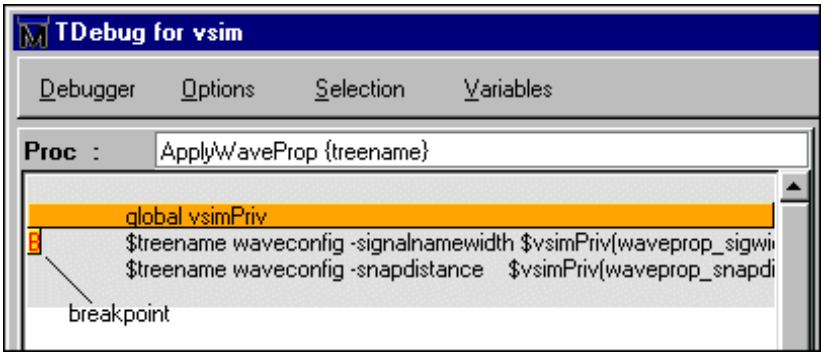
There are seven possible debugger states, one for each button and an ‘idle’ or ‘waiting’ state when no button is active. The button-activated states are:

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**.

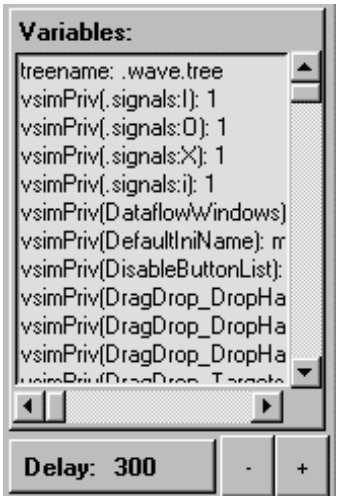
Breakpoints

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. There’s no support for conditional or counted breakpoints.



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).

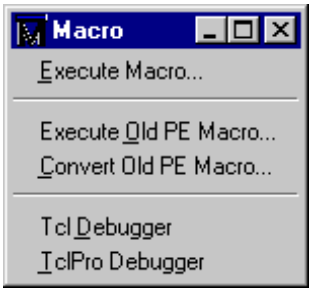


Configuration

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 Macro menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger can be acquired from Scriptics at their web site: www.scriptics.com. Once acquired, do the following steps to use the TclPro Debugger:



1. Launch TclPro Debugger
2. Launch Model*Sim*
3. Select **Macro > TclPro Debugger** (Main window)

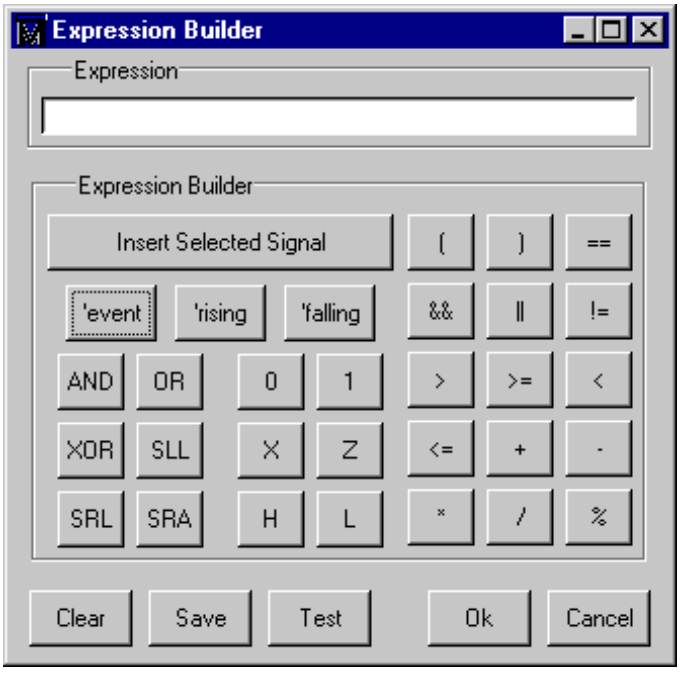
   This will connect Model*Sim* to the Scriptics TclPro Debugger.

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-297).

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-302).

To search for when a signal reaches a particular value

Select the signal in the Wave window and click **Insert Reference 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 Reference 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-302)), or you can type them in.

To save the expression as a Tcl variable

The Save button allows you to save the expression to a Tcl variable.

See"Setting up a List trigger with Expression Builder" (E-444) for an additional Expression builder example.

Graphic interface commands

The following commands provide control and feedback during simulation as well as the ability to edit, and add menus and buttons to the interface. Only brief descriptions are provided here; for more information and command syntax see the *ModelSim Command Reference*.

Window control and feedback commands	Description
batch_mode (CR-42)	returns a 1 if Model*Sim* is operating in batch mode, otherwise returns a 0; it is typically used as a condition in an if statement
configure (CR-88)	invokes the List or Wave widget configure command for the current default List or Wave window
down (CR-105)	moves the active marker in the List window down to the next transition on the selected signal that matches the specifications
getactivecursortime (CR-124)	gets the time of the active cursor in the Wave window
getactivemarkertime (CR-125)	gets the time of the active marker in the List window
left (CR-129)	searches left through the specified Wave window for signal transitions or values
notepad (CR-141)	a simple text editor; used to view and edit ASCII files or create new files
play (CR-148)	**UNIX only (excluding Linux)** - replays a sequence of keyboard and mouse actions that were previously saved to a file with the **record** command (CR-167)
property list (CR-161)	changes properties of an HDL item in the List window display
property wave (CR-162)	changes properties of an HDL item in the waveform or signal name display in the Wave window
record (CR-167)	**UNIX only (excluding Linux)** - starts recording a replayable trace of all keyboard and mouse actions
right (CR-174)	searches right through the specified Wave window for signal transitions or values
search (CR-178)	searches the specified window for one or more items matching the specified pattern(s)
seetime (CR-182)	scrolls the List or Wave window to make the specified time visible
transcribe (CR-193)	displays a command in the Main window, then executes the command
up (CR-196)	moves the active marker in the List window up to the next transition on the selected signal that matches the specifications
write preferences (CR-280)	saves the current GUI preference settings to a Tcl preference file

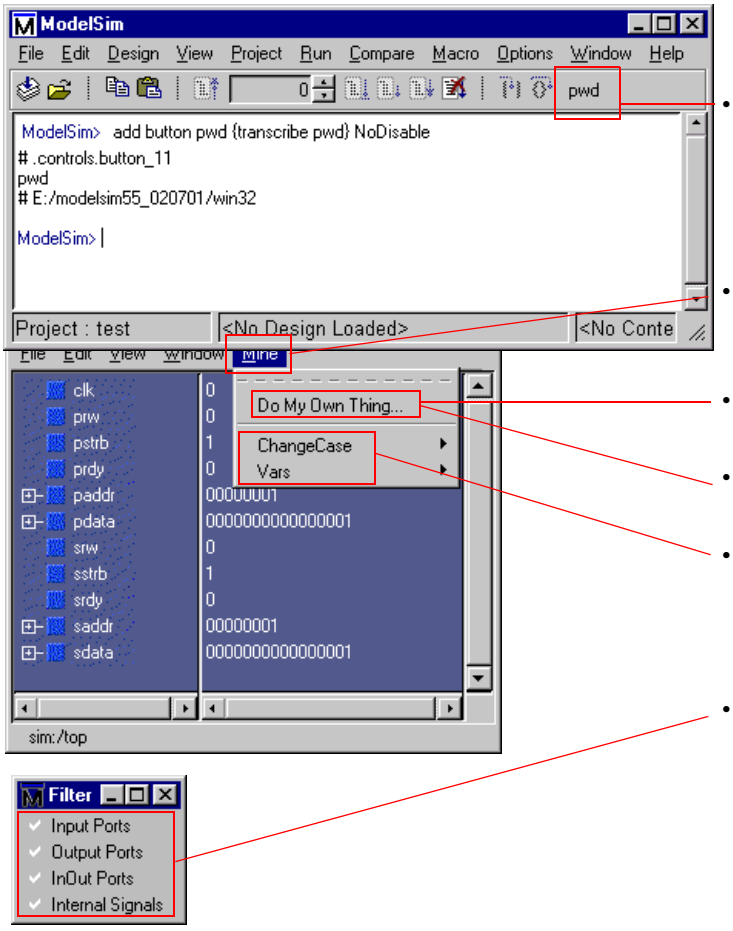
Window menu and button commands	Description
add button (CR-26)	adds a user-defined button to the Main window button bar
add_menu (CR-31)	adds a menu to the menu bar of the specified window
add_menucb (CR-33)	creates a checkbox within the specified menu of the specified window
add_menuitem (CR-34)	creates a menu item within the specified menu of the specified window
add_separator (CR-35)	adds a separator as the next item in the specified menu path in the specified window
add_submenu (CR-36)	creates a cascading submenu within the specified menu_path of the specified window
change_menu_cmd (CR-53)	changes the command to be executed for a specified menu item label, in the specified menu, in the specified window
disable_menu (CR-102)	disables the specified menu within the specified window; useful if you want to restrict access to a group of Model*Sim* features
disable_menuitem (CR-103)	disables a specified menu item within the specified menu_path of the specified window; useful if you want to restrict access to a specific Model*Sim* feature
enable_menu (CR-112)	enables a previously-disabled menu
enable_menuitem (CR-113)	enables a previously-disabled menu item

Customizing the interface

Try customizing Model*Sim*'s interface yourself; use the command examples for **add button** (CR-26) and **add_menu** (CR-31) to add a button to the Main window, and a new menu to the Signals window (8-193). Results of the button and menu commands are shown below

Buttons the easy way

"The Button Adder" (8-269) tool makes adding buttons easy. Select **Window > Customize** in any window to access the Button Adder. Buttons you create are not permanent; they exist only during the current session. To reuse a button, save the Main transcript (**File > Save Transcript As**) after the button is created. Edit the file to contain only button-creation commands, then pass the filename as an argument to the **do** command (CR-104) to recreate the button.



- The **pwd** button was added to the Main window with the **add button** command (CR-26). Buttons can be added to the status bar as well.
- The Mine menu was added to the Signals window with the **add_menu** command (CR-31).
- The Do My Own Thing menu item was added with the **add_menuitem** command (CR-34)
- The menu separator was added with the **add_separator** command (CR-35).
- The ChangeCase and Vars submenus were added with the **add_submenu** command (CR-36).
- You can also add a menu checkbox (like those in this menu tearoff) with the **add_menucb** command (CR-33).

9 - 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: The Performance Analyzer *does not* work on Windows 95.

Introducing Performance Analysis

The Performance Analyzer provides an interactive graphical representation of where Model*Sim* 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

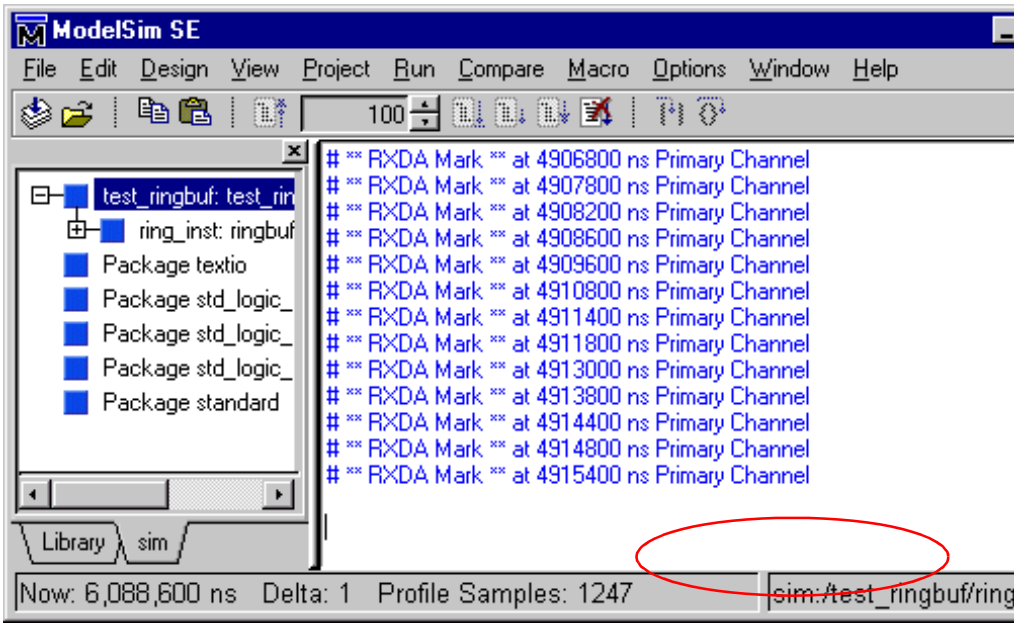
- A non-accelerated VITAL library cell is impacting simulation run time
- A process is consuming more time than necessary because of non-required items in its sensitivity list
- A testbench process is active even though it is not needed
- A random number process is consuming simulation resources when in a test bench 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 feature in Model*Sim* is a statistical sampling profiler. It periodically "wakes up" and samples the current simulation at a user-determined rate, and records what is executing in the simulation during the sample period. 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.

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.



Getting Started

Performance analysis occurs during the Model*Sim* **run** command and is displayed graphically as a profile of simulator performance. To enable the Performance Analyzer, use the **profile on** command at the VSIM prompt. 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.

The Performance Analyzer is turned off by issuing the **profile off** command at the VSIM prompt. Any Model*Sim* **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, issue the **profile clear** command at the VSIM prompt.

Interpreting the data

The Performance Analyzer is most helpful in those situations where a high percentage of simulation time is being spent in a particular module. For example, say the Performance Analyzer shows that 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 several times faster.

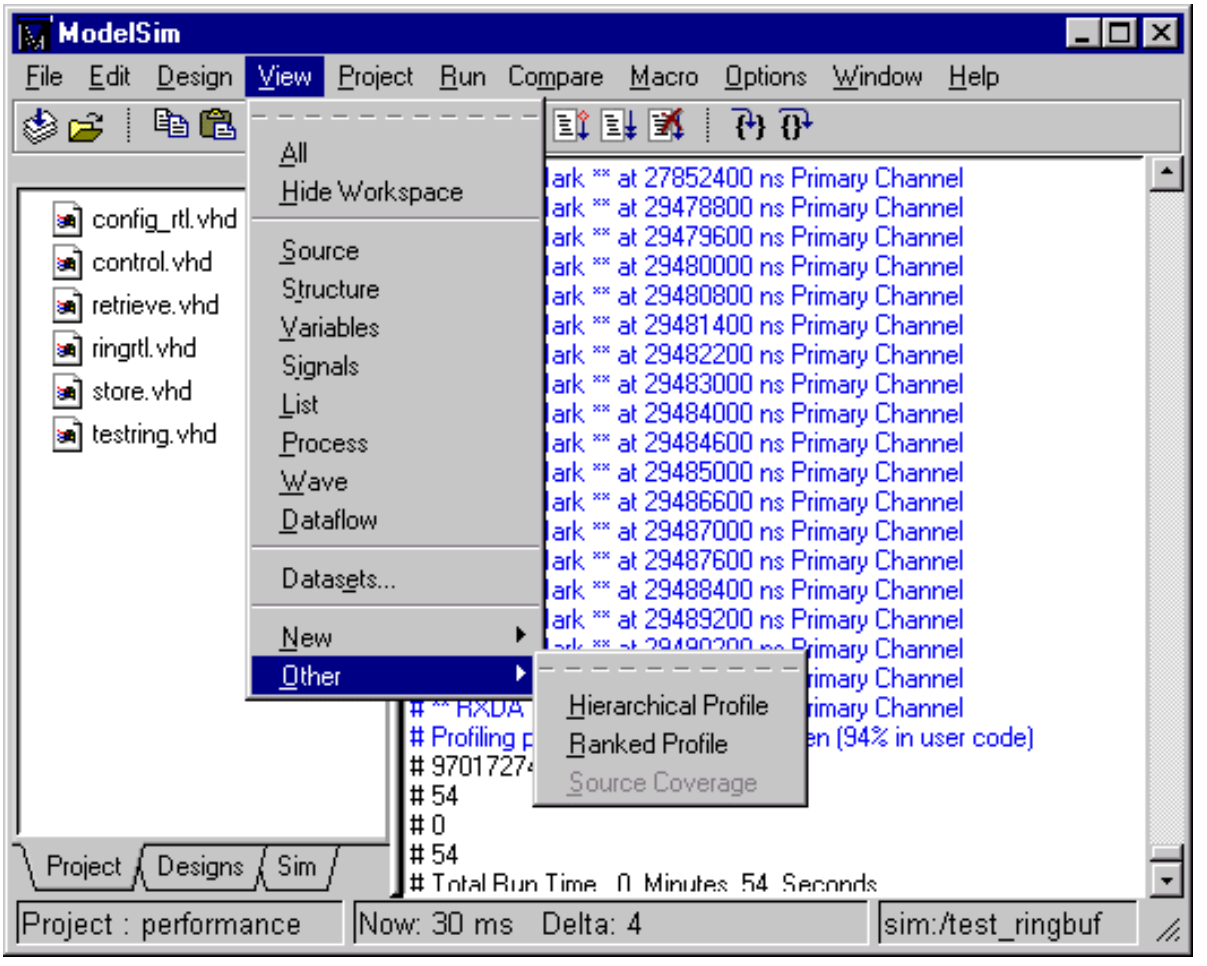
More commonly, the Performance Analyzer will tell you that 30% of simulation time was spent in model X, 25% in model Y, and 20% in model Z. In such situations, careful

examination and improvement of each model may result in a significant overall speed improvement.

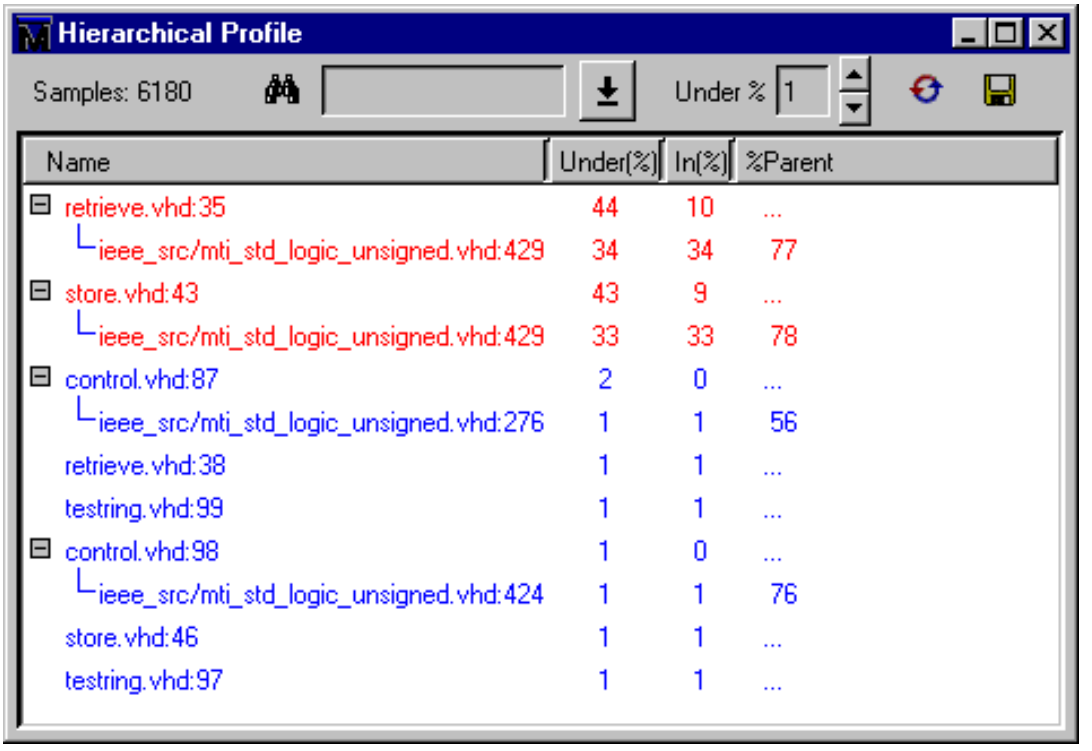
There are times, however, when the Performance Analyzer tells you nothing better than that the simulation has executed in several hundred different models and has spent less than 1% of its time in any one of them. In such situations, the Performance Analyzer provides little helpful information and simulation improvement must come from a higher level examination of how the design can be changed or optimized.

Viewing Performance Analyzer Results

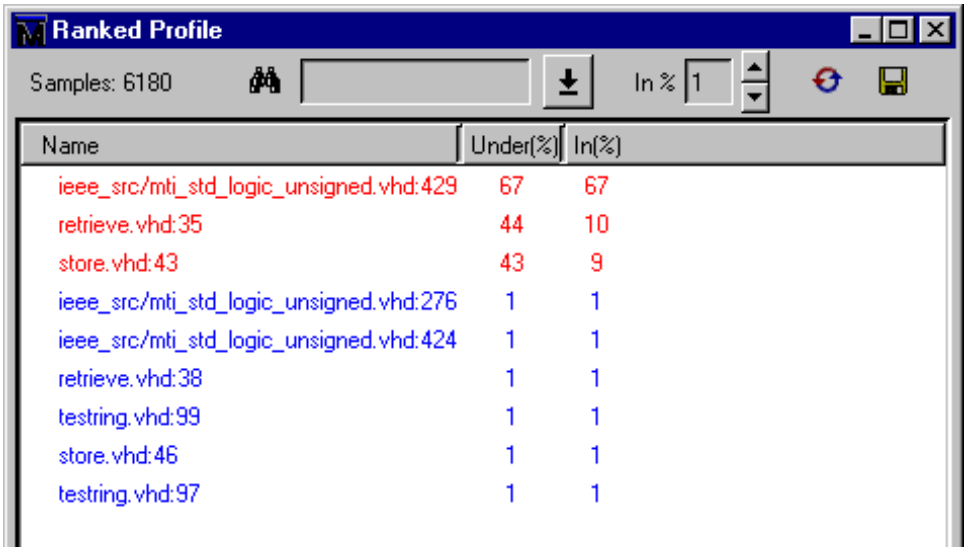
The Performance Analyzer provides two views of the collected data – a *hierarchical* and a *ranked* view. The hierarchical view is accessed by clicking **View** > **Other** > **Hierarchical Profile** (Main window). The ranked view is accessed by selecting **View** > **Other** > **Ranked Profile**.



The Hierarchical view can also be invoked by entering **view_profile** 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 a 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, two lines (retrieve.vhd:35 and store.vhd:43) are taking the majority of the simulation time.



The Ranked view can be invoked by entering **view_profile_ranked**. The modules and code lines are ranked in order of the amount of simulation time used. The two lines that are taking up most of the simulation time – retrieve.vhd:35 and store.vhd:43 – appear at the top of the list under the VHDL module that contains them.



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.

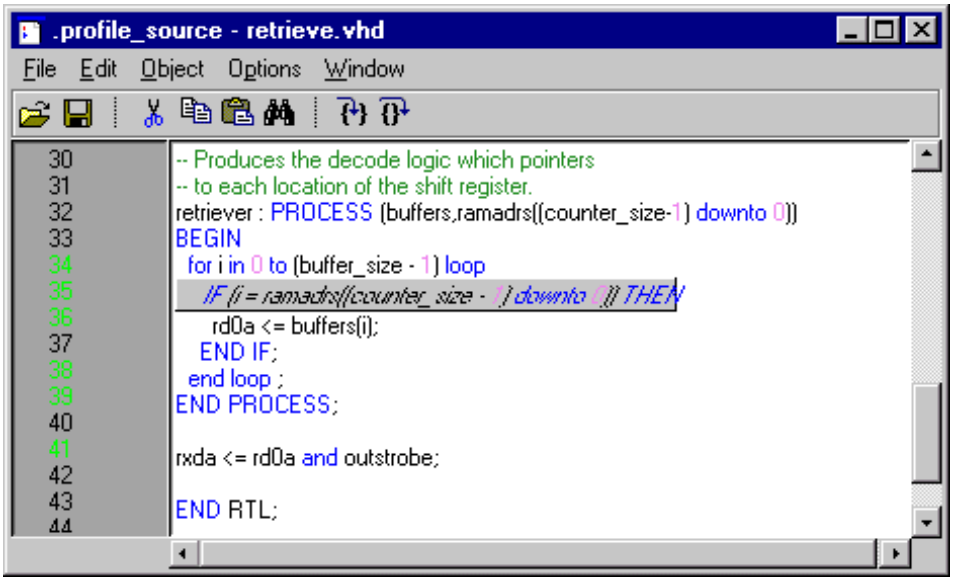
The *hierarchical view* opens with all levels displayed. You can collapse the hierarchical view by clicking the boxes next to the high-level names. At this time, the *hierarchical* view will not remember which levels are opened or closed when data is reloaded. By default, hierarchical levels are opened 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. For the *retrieve.vhd:35* entry in the hierarchical and ranked views above, Under(%) is 44 and In(%) is 10. "Under(%)" means that this particular line and all support routines it needed took 44% of total simulation time. "In(%)" means that 10% of the total simulation time was actually spent executing this line of VHDL code.

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. In the diagram below, *retrieve.vhd:35* was selected in the Hierarchical Profile and, consequently, is highlighted in the Source window.



The actual line of VHDL code for *retrieve.vhd:35* is:

```
IF (i=ramadrs((counter_size-1)downto 0))THEN
```

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. For example, in the hierarchical view above, the line *store.vhd:43* calls *ieee_src/mti_std_logic_unsigned.vhd:429*.

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 compute 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.

Ranked/Hierarchical Profile Window Features

The Ranked and Hierarchical Profile windows have a number of features that can manipulate the data displayed. Most of these features are contained in a toolbar in the header of the window, which displays an icon for each feature. Placing the mouse over an icon causes its function to be displayed.

The **Find Entry** icon 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.

The **Under%** filter allows you to specify 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.

The **hierCutoff** and **rankCutoff** variables provide a similar function. See "Setting preferences with Tcl variables" (9-290).

The **Update Data** icon 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 automatically updated. You should click on this button to update the data being displayed in these windows.

The **Save to File** icon allows the data to be saved to disk. You will be prompted for the output file name.

The **profile report** command (CR-158) provides another way to save profile data.

The report option

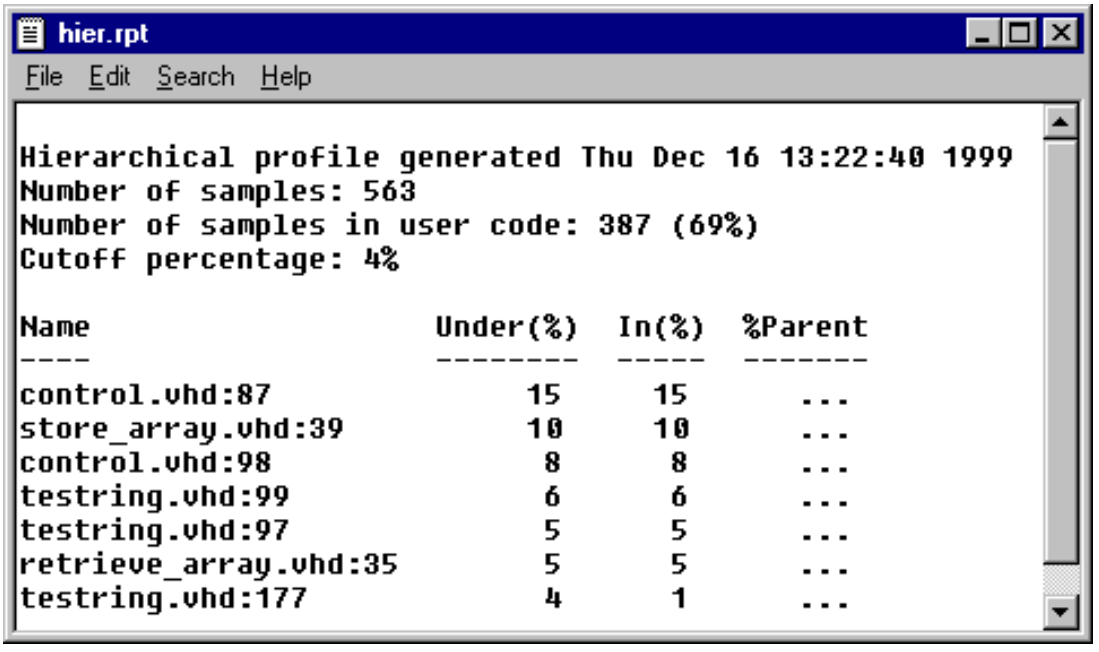
You can also use the **profile report** command (CR-158) to save the Performance Analyzer results.

```
profile report [<option>]
```

The arguments to the command are [-hierarchical | -ranked] [-file<filename>] [-cutoff <percentage>]. 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.



Setting preferences with Tcl variables

Various Tcl variables control how the Hierarchical Profile and Ranked Profile windows are displayed.You can set these preference variables by selecting **Options > 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 http://www.model.com/resources/pref_variables/frameset.htm for more information on the individual variables.

Performance Analyzer commands

The table below provides a brief description of the profile commands; follow the links for complete command syntax.

See the *ModelSim Command Reference* for complete command details.

Command	Description
profile clear (CR-153)	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-154)	selects the frequency with which the profiler collects samples during a run command
profile off (CR-155)	disables runtime profiling
profile on (CR-156)	enables runtime analysis of where your simulation is spending its time
profile option (CR-157)	changes various profiling options
profile report (CR-158)	produces a textual output of the profiling statistics that have been gathered up to this point

10 - Code Coverage

Chapter contents

Code Coverage gives you graphical and report file feedback on how your source code is being executed. This integrated feature provides three important benefits to the Model*Sim* user:

1. Because it's integrated into the Model*Sim* engine, it is totally non-intrusive – it doesn't require instrumented HDL code as do third-party code coverage products.

2. It has very little impact on simulation performance (typically less than 5%).

3. There is no need to recompile to obtain code coverage statistics. Model*Sim* version 5.3 and later libraries fully support this feature.

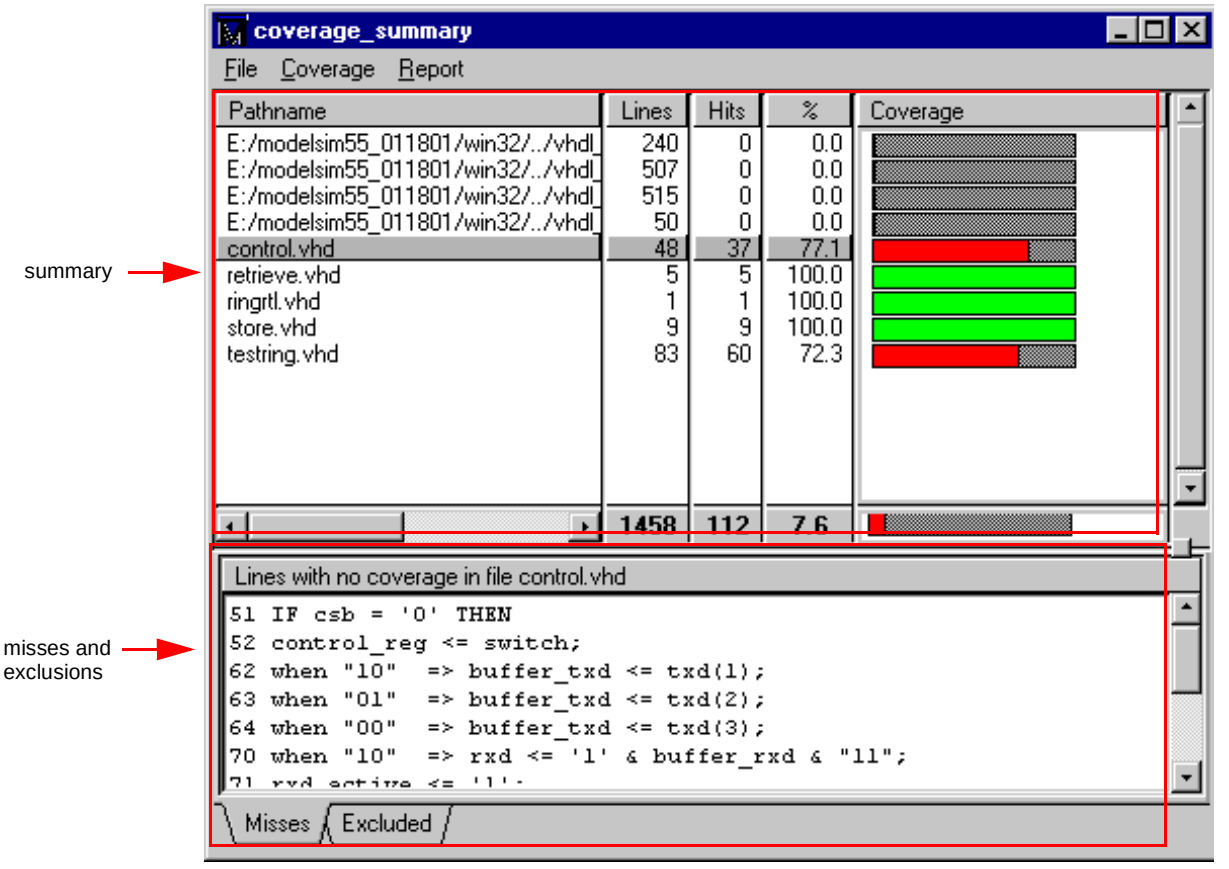
Enabling Code Coverage

To enable code coverage, begin simulation with the **-coverage** option to the **vsim** command (CR-258). With coverage enabled, ModelSim counts how many times each executable line is executed during simulation (number of "hits"). The information is then displayed in the coverage_source and coverage_summary windows. Or, you can save the information in several different text reports (see below for details).

Note: To view the maximum number of lines while doing code coverage, use the **-O0** (capital O zero) argument when you compile your design files. This argument minimizes compiler optimizations.

The coverage_summary window

The coverage_summary window provides a graphical view of code coverage. To display the coverage_summary window, select **View** > **Other** > **Source Coverage** (Main window) or enter **view_coverage** at the VSIM prompt.



The window is split into two panes: the top pane displays Summary information (10-293) on a per file basis; the bottom pane displays lines misses on the Misses tab (10-293) and file or line exclusions on the Exclusions tab (10-293).

The coverage_summary window is linked to The coverage_source window (10-296). When you select a file in the top pane, that file displays in the coverage_source window. Likewise, if you select a line number in the bottom pane, that line is scrolled to in the coverage_source window. In addition, any exclusions you make in the coverage_summary window automatically show up in the coverage_source window and vice versa.

Summary information

The top pane of the coverage_summary window shows all of the design files that have executable lines of code. The columns of information include:

- The Pathname column shows the path and file name.
- The Lines column contains the number of executable lines in the file.
- The Hits column indicates the number of executable lines that have been executed in the current simulation.
- The Percentage column is the current ratio of Hits to Lines. There is also a bar chart that graphically displays this percentage. If the coverage percentage is below 90%, the bar chart is displayed in red (you can change the percentage by editing the PrefCoverage(cutoff) preference variable).

By default, the summary information is sorted by Pathname. You can sort by another column by clicking on the column heading (i.e., Lines, Hits, %).

A totals row at the bottom of the summary information shows coverage statistics for all of the files combined.

Misses tab

The Misses tab lists lines from the current file with no hits. Select a file in the top pane of the coverage_summary window to see that file's missed lines.

This tab also lets you select lines to exclude. Select the line(s) you want to exclude, click your right mouse button, and select **Exclude Selected Lines**. The lines you exclude will be shown in the Exclusions tab and also marked with a green "X" in The coverage_source window (10-296).

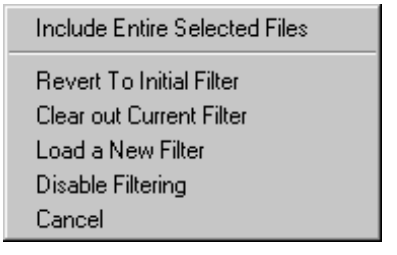
Exclusions tab

The Exclusions tab lists all file and line exclusion filters for the current simulation. This includes line or file exclusions made in the Misses tab or in the coverage_source window.

The Exclusions tab offers several commands via a context menu. Click anywhere within the tab with your right mouse button to get the following context menu:

The menu has the following options:

- **Include Entire Selected Files**
  Adds selected lines or files back into the coverage statistics. If you have multiple lines excluded in one file, it will add back all of them. To add back individual lines, use the coverage_source window.



- **Revert To Initial Filter**
  Returns filtering to the default exclusion filter file
- **Clear Out Current Filter**
  Clears active exclusion filters
- **Load a New Filter**
  Opens a different exclusion filter file
- **Disable/Enable Filtering**
  Disables/enables filtering. Acts as a toggle. Allows you to temporarily turn off filtering to see raw code coverage statistics.
- **Cancel**
  Closes the context menu

The coverage_summary window menu bar

The coverage_summary window has three menus: File, Coverage, and Report. Brief descriptions of each command are given below.

File menu

Open > Coverage > Merge Coverage	Merges saved reports into the current analysis. See "Merging coverage report files" (10-298) for more details
Open > Coverage > Apply a Previous Coverage	Clears the current coverage statistics and loads a previously saved coverage report
Open > Load a New Filter	Loads an exclusion filter file. See "Exclusion filter files" (10-299) for more details
Save > Line Coverage	Saves a textual report of the source file summary data and details for each executable line in the file
Save > Current Filter	Saves the current exclusion filter to a file that can be reloaded later. See "Exclusion filter files" (10-299) for more details
Close	Closes the view_coverage window

Coverage menu

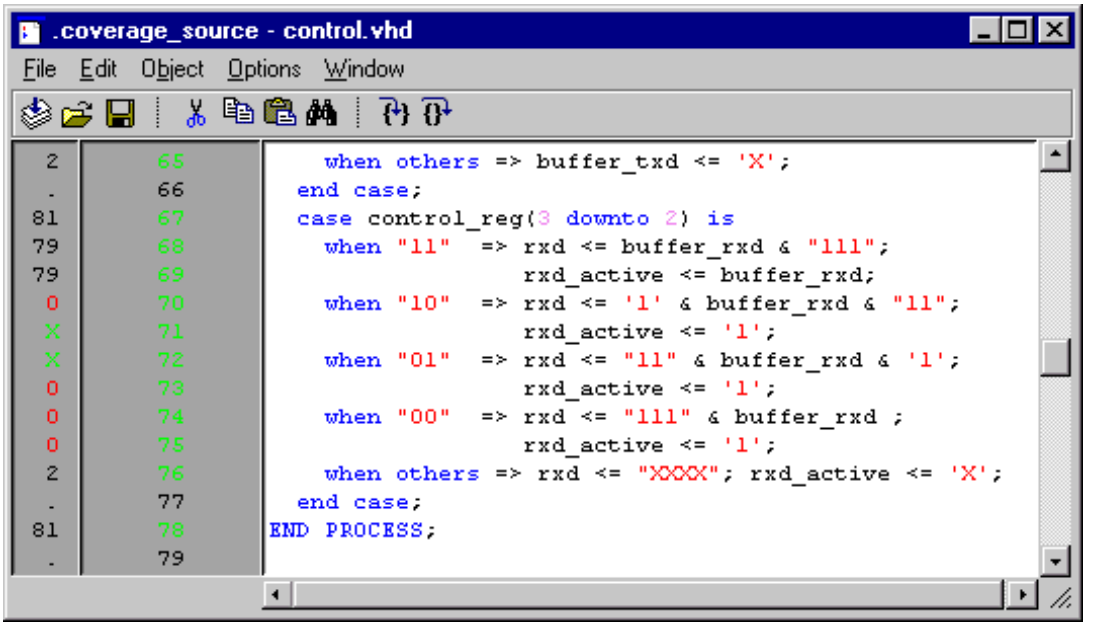
Clear Current Coverage	Clears the current coverage statistics
Revert To Initial Filter	Returns filtering to the default exclusion filter file
Clear out Current Filter	Clears active exclusion filters
Disable/Enable Filtering	Disables/Enables filtering. Acts as a toggle.

Report menu

Save Summary Coverage	Saves a textual report of the summary lines, hits, and percentages for each source file being analyzed
Save Line Coverage	Saves a textual report of the source file summary data and details for each executable line in the file
Save Excluded Lines	Saves a textual report of the lines and files that are currently being excluded from the coverage statistics
Save Zeroed Lines	Saves a textual report like the Line Coverage report but only includes those lines that have zero coverage
Save Totals	Saves a one line text report of the total files, lines, hits and overall percentage for the current analysis
Save As	Lets you choose from the above reports in one dialog

The coverage_source window

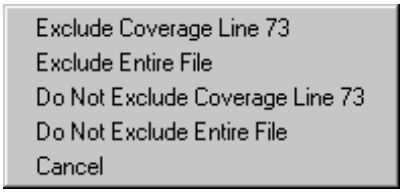
You can open the coverage_source window by selecting a file in the pathname column of "The coverage_summary window" (10-292). The coverage_source window is an enhanced version of the standard Source window (8-201). When code coverage is enabled, an additional column appears on the left side of the window. This column identifies how many times each executable line of code has been executed during simulation (lines that are not executed are highlighted with a red zero); and it marks with a green "X" lines that have been excluded from the code coverage statistics.



You can skip to "missed lines" using the **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** commands, or by pressing <Shift> - <Tab> (previous miss) or Tab (next miss).

Excluding lines and files

There may be certain lines or files that you do not want to include in the code coverage statistics. In the coverage_source window, click your right mouse button in the far-left column (the one with the hit counts) to display the following context menu:



The menu has the following options:

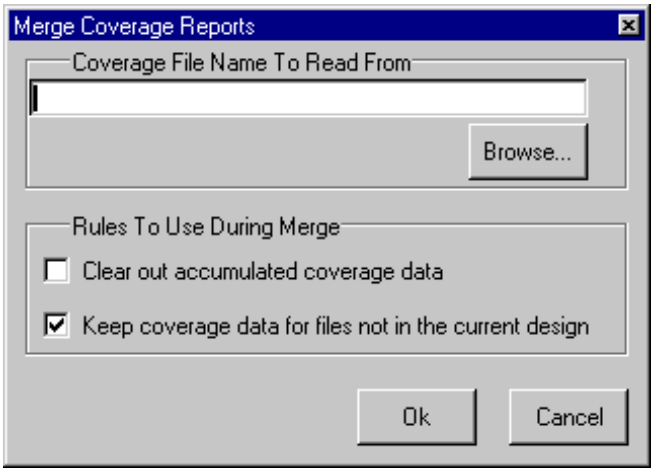
- **Exclude Coverage Line #**
  Excludes the specified line number from the code coverage statistics.
- **Exclude Entire File**
  Excludes the entire file from the code coverage statistics.

- **Do Not Exclude Coverage Line #**
  Adds the specified line number back into the code coverage statistics .
- **Do Not Exclude Entire File**
  Adds the file back into the code coverage statistics.

Any exclusions you make in the coverage_source window will show up in the Excluded tab of The coverage_summary window (10-292).

Merging coverage report files

You can merge the results from two or more analyses. Select **File > Open > Coverage > Merge Coverage** from the coverage_summary window.



The Merge Coverage Reports dialog has the following options:

- **Coverage File Name to Read From**
  Specify one or more saved coverage reports that you want to merge into the current analysis
- **Clear out accumulated coverage data**
  When checked, clears coverage statistics from the current analysis before merging in saved coverage reports
- **Keep coverage data for files not in the current design**
  When checked, includes coverage data from all files you are merging in, even if they are not part of the current design. If you then select one of those included files in the coverage_source window, it will pop-up an Open Source dialog so you can point to the location of the file.

Exclusion filter files

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 coverage_source window by selecting **File > Save > Current Filter**. To load the filter during a future analysis, select **File > Open > Load a New Filter**.

Syntax

```
<filename> [[<range> ...] [<line#> ...]] | all
...
```

Arguments

`<filename>`
The name of the file you want to exclude. Required. The filter file may include an unlimited number of filename entries, each on its own line.

`<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
```

Default filter file

The Tcl preference variable **PrefCoverage(pref_InitFilterFrom)** specifies a default filter file and path to read when a design is loaded with the -coverage switch. By default this variable is set to "Exclude.cov". See "Code Coverage preference variables" (10-300) for details on changing this variable.

Code Coverage preference variables

Various Tcl variables control how the coverage data is displayed. You can set these preference variables by selecting **Options > Edit Preferences > By Name > Coverage** (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 http://www.model.com/resources/pref_variables/frameset.htm for more information on the individual variables.

Code Coverage commands

The commands below are available once Code Coverage is active. Enable code coverage with the **-coverage** option to the **vsim** command (CR-258).

The table below provides a brief description of the coverage commands; follow the links for complete command syntax.

See the ModelSim *Command Reference* for complete command details.

Command	Description
coverage clear (CR-92)	clears all coverage data obtained during previous run commands
coverage reload (CR-93)	merges coverage statistics with the output of a previous **coverage report** command
coverage report (CR-94)	used to produce a textual output of the coverage statistics that have been gathered up to this point

11 - Waveform Comparison

Chapter contents

Introducing Waveform Comparison

The ModelSim Waveform Comparison feature allows you to compare the current live simulation against a reference wave logfile or dataset (.wlf file), compare two saved 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 Comparison 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, and
- step through a succession of timing differences via buttons in the Wave window.

By default, Waveform Comparison computes the timing differences between test signals and reference signals from time zero to the end of the shortest dataset, or to the end of the current live simulation. But you can also specify an optional start time and end time, or you can limit the comparison to a specific number of encountered timing differences. In addition, you can exclude windows of time with **-when** conditions in either the clock definitions or in the **compare add** command (CR-63). The display will indicate intervals of time during which no attempt was made to compute differences.

All waveform differences encountered in the waveform 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 (11-316) and List window display (11-322)). Icons in the toolbar of the Wave window allow you to step forward and backward through successive differences. Or, you can use the Tab and Shift-Tab keys on your keyboard to move to the next or previous difference of a selected signal.

You can also write a list of the differences to a file using the **compare info** command (CR-73).

Two Modes of Comparison

The Waveform Comparison 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

In the clocked mode, also called strobed comparison, one or more clocks are defined. A test signal is then 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 Comparison 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-63) 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-63) 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-63) to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

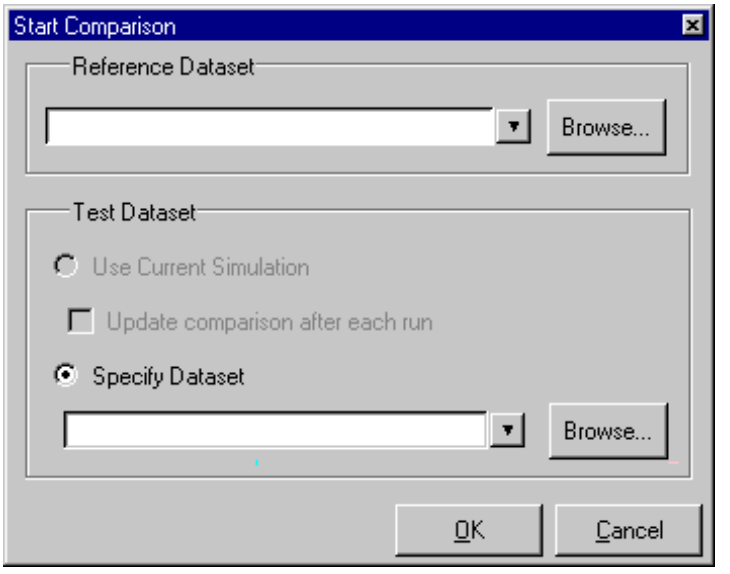
If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Comparison feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Comparison will let you know.

Graphical Interface to Waveform Comparison

Waveform Comparison is initiated from either the Main or Wave window by selecting **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.

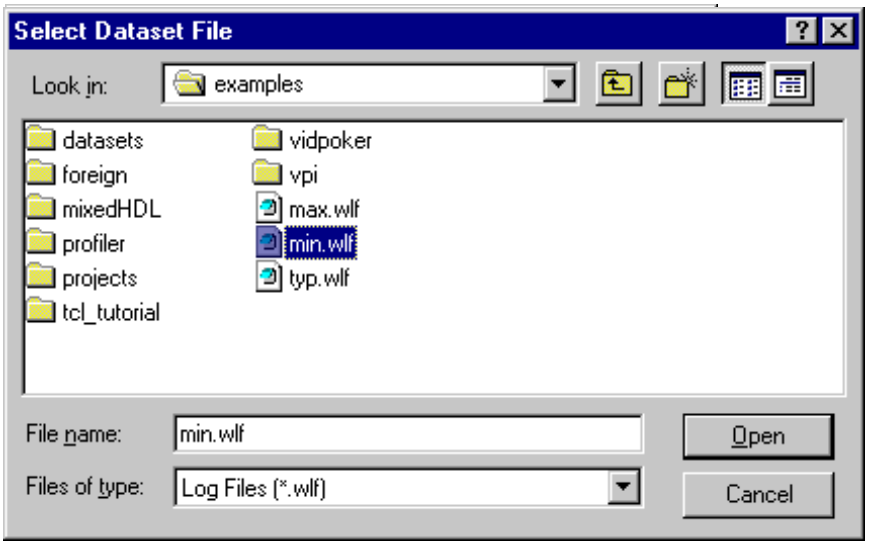
Test Dataset

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.

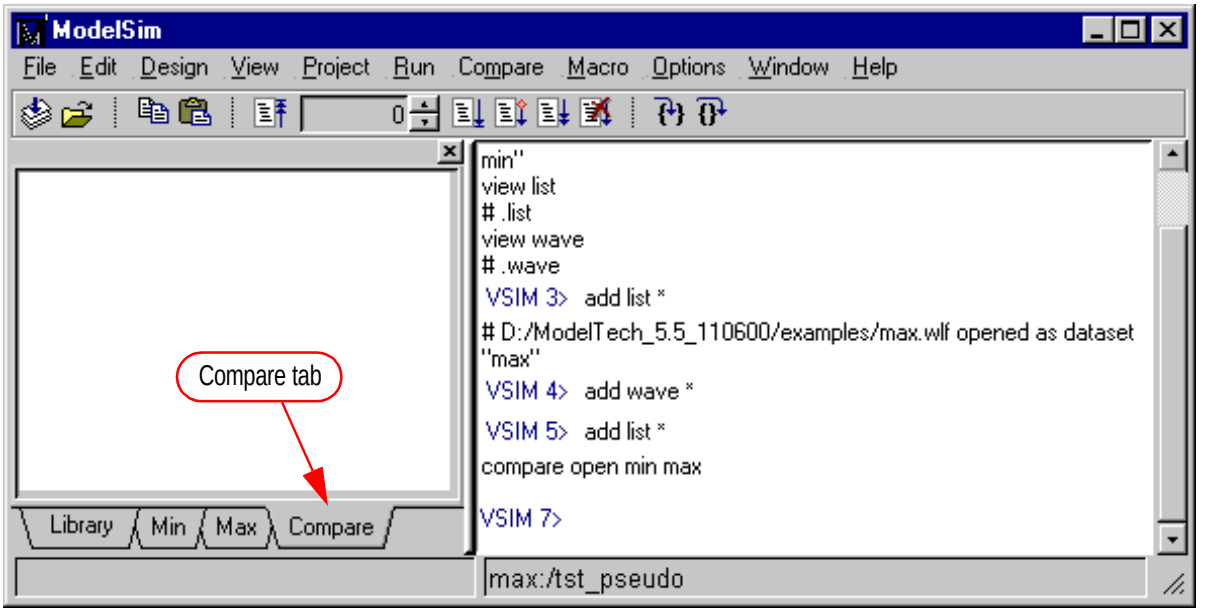
- **Use Current Simulation**
  Selects the current simulation to be used as the Test Dataset. Provides for an optional update on the comparison after each simulation run.
- **Specify Dataset**
  Allows you to select any saved .wlf file to be used as the Test Dataset.

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.

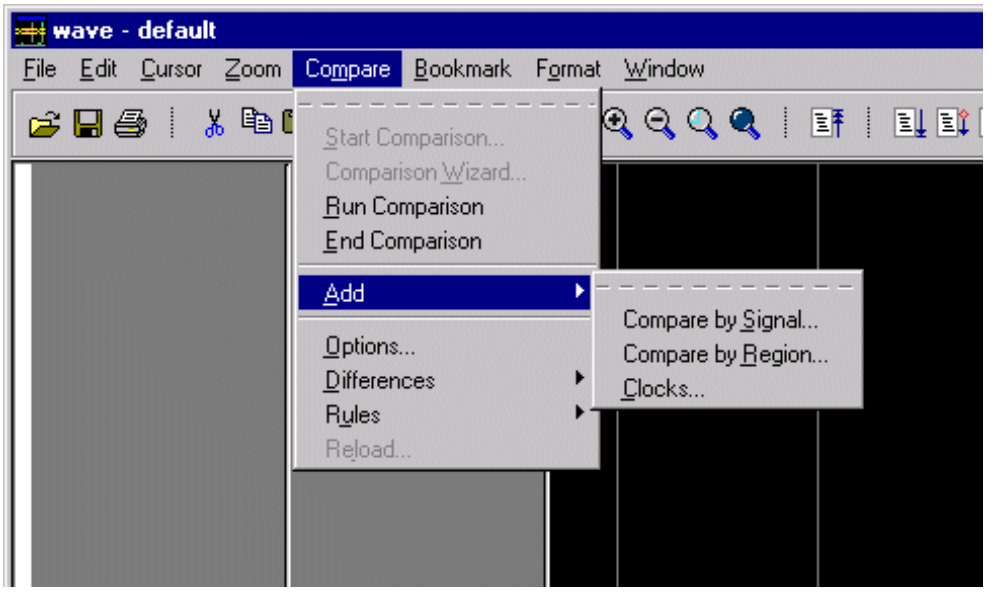


Once the Reference and Test Datasets have been specified, clicking "OK" in the Compare Dataset dialog box will place a Compare tab in the project pane of the Main window. After adding the signals, regions and/or clocks you want to use in the comparison (see "Adding Signals, Regions and/or Clocks" (11-307)) you'll be able to drag compare objects from this project tab into the Wave and List windows.



Adding Signals, Regions and/or Clocks

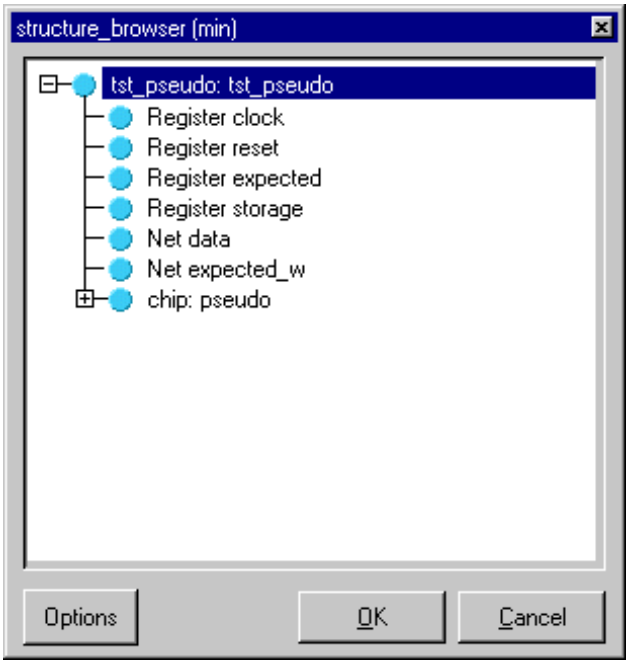
To designate the signals, regions and/or clocks to be used in the comparison, click **Compare > Add** in the Main or Wave window, then make a selection (Compare by Signal (11-307), Compare by Region (11-311), Clocks) from the popup menu.



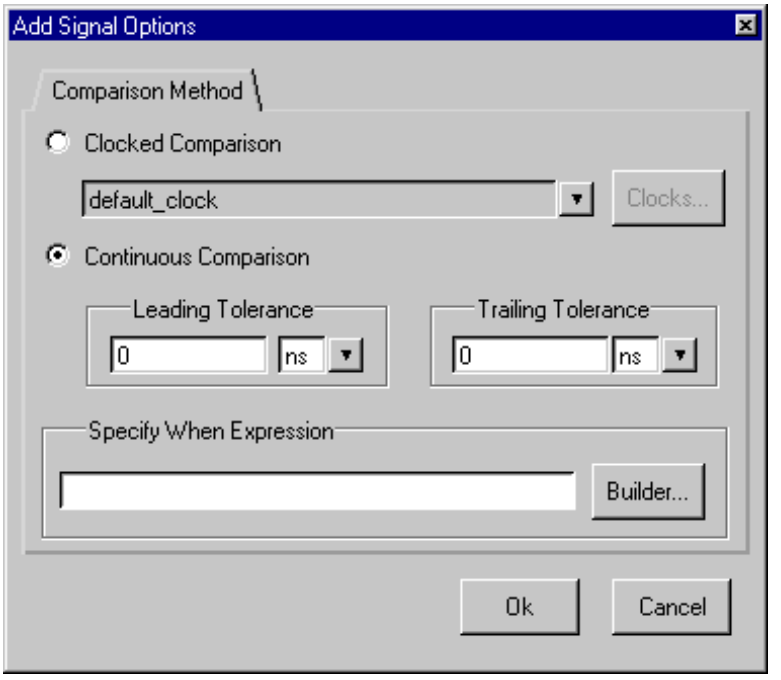
Compare by Signal

Clicking **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, which opens the Add Signal Options dialog box.



- **Add Signal Options**
  The Add Signal Options dialog allows you to select the Waveform Comparison method to be used – Clocked (Strobed) or Continuous – and to specify a **when** expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective. A **when** expression can be built using "The GUI Expression Builder" (8-275), which is accessed by clicking the Builder button.

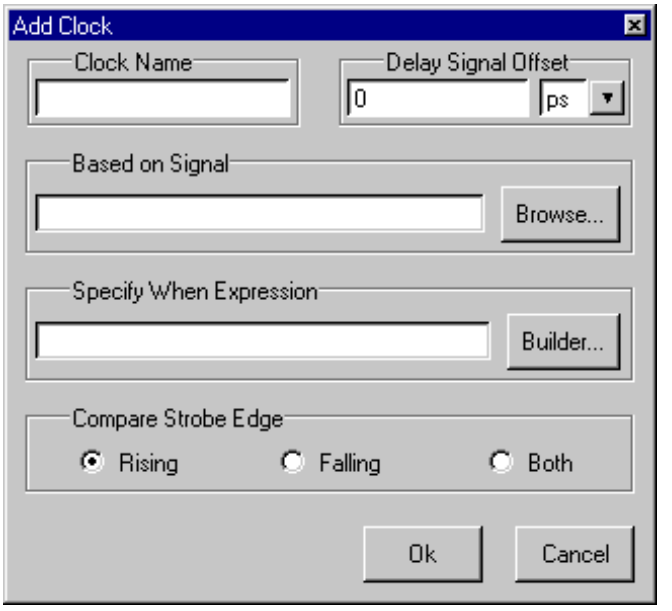


Clocked Comparison

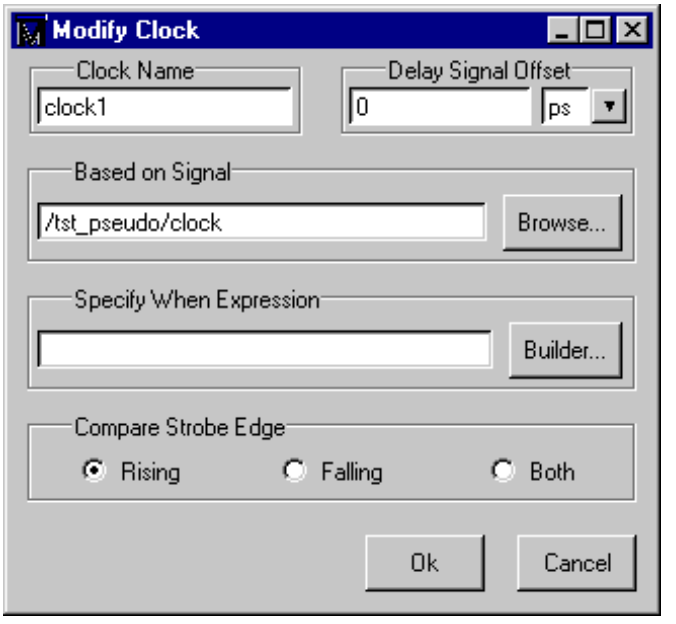
If the Clocked Comparison method is chosen, you can select a clock from the drop-down history of past clock selections or click the Clocks button to add a new clock.

Clicking the Clocks button opens the Comparison Clocks dialog box where you can add, modify or delete signals.

The Add button opens the Add Clock dialog, 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 GUI Expression Builder" (8-275) to specify a **when** expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.

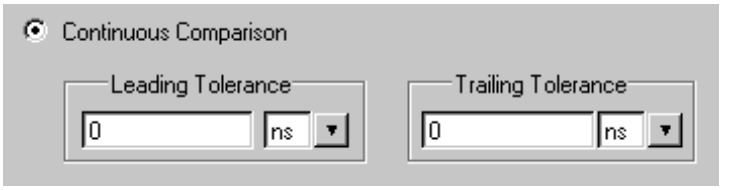


Clicking the Modify button in the Comparison Clocks dialog opens the Modify Clock dialog. This dialog provides the same functionality as the Add Clock dialog.

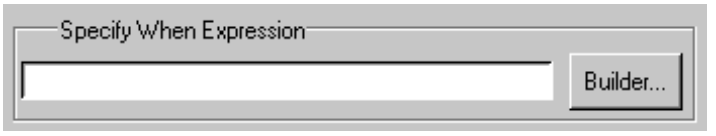


Continuous Comparison

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.

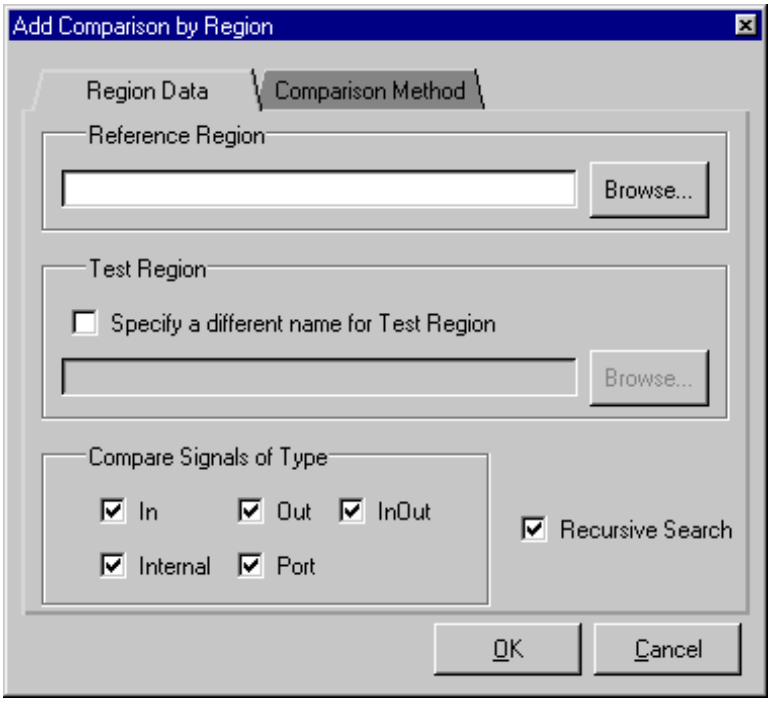


With Continuous Comparison, you can also use "The GUI Expression Builder" (8-275) to specify a **when** expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.



Compare by Region

Clicking **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.

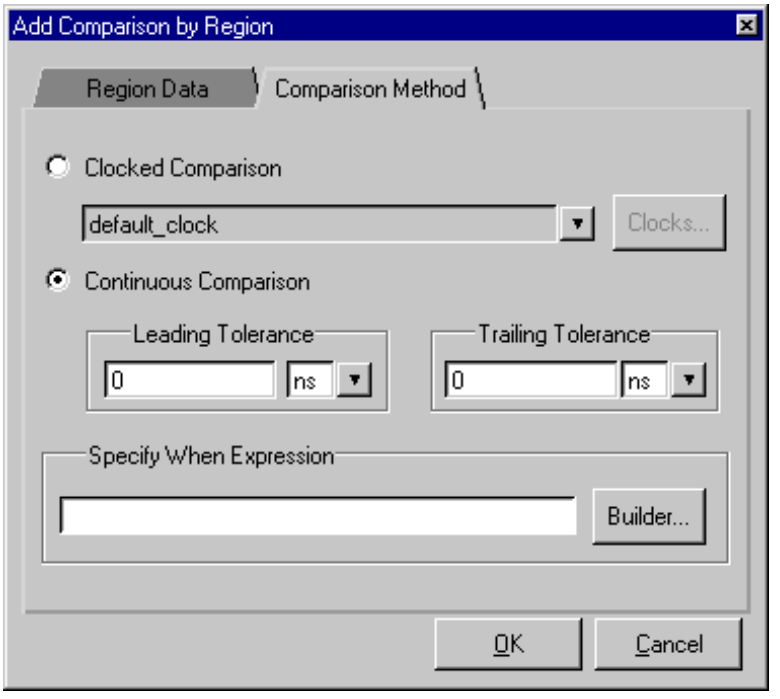


Region Data Tab

- **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

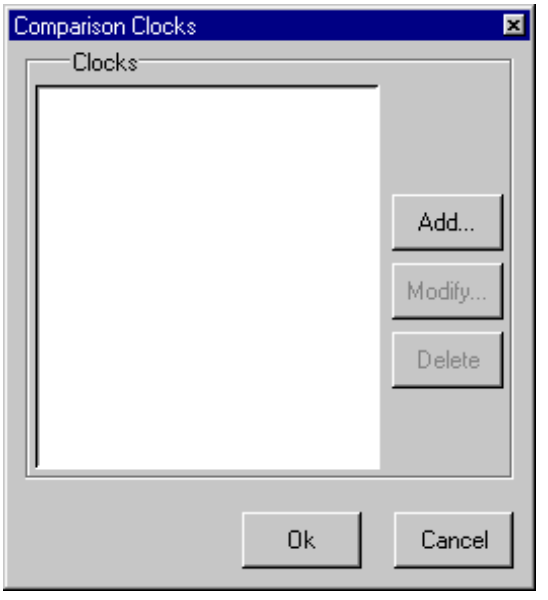
Allows you to select clocked or continuous comparison, and provides the capability to specify a "When" expression.



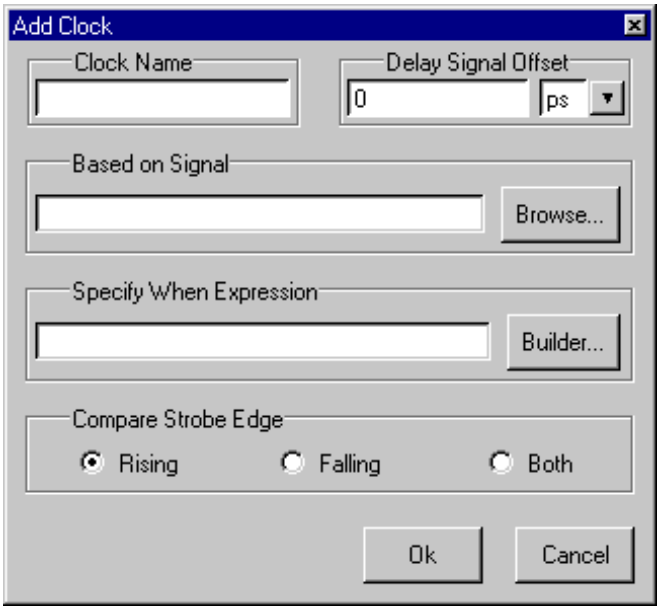
- **Clocked Comparison**
  Allows you can select a clock from the drop-down history of past clock selections. Or, you can click the Clocks button to add a new clock.

  Clicking the Clocks button opens the Comparison Clocks dialog box.

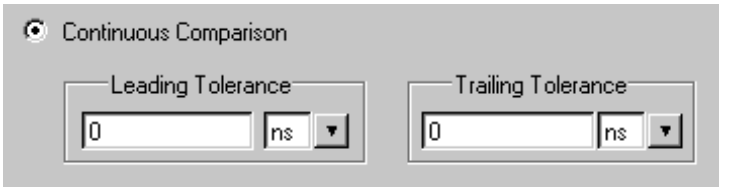
  To add a signal, click the Add button to open the Add Clock dialog box, where you can define a clock signal name, a delay signal offset, the signal upon which the clock will be based, and whether the compare strobe edge will be the rising or falling edge or both. You can also use the Expression Builder to Specify



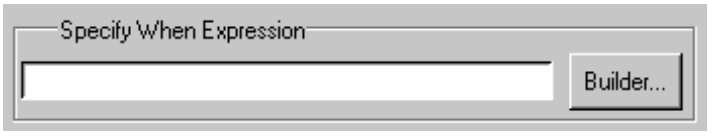
a When Expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.



- **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" (8-275) to specify a **when** expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.



Setting Compare Options

Selecting **Compare > Options** in either the Main or Wave windows provides access to the Add Signal Options dialog box. This dialog is divided into two tabs – the General Options tab and the Comparison Method tab.

- **General Options**

Comparison Limit Count — Allows you to limit the waveform 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 the VHDL X and Z values.

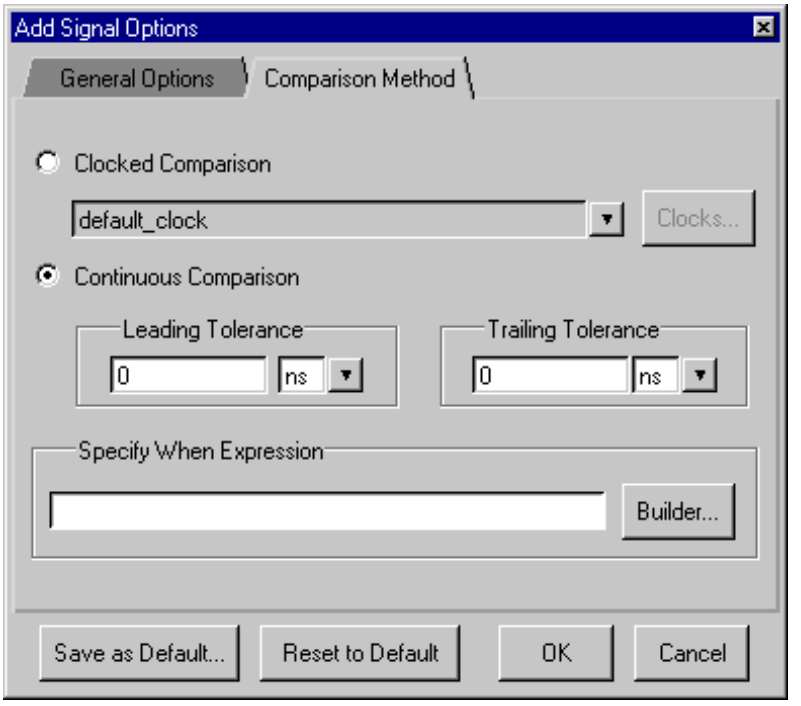
Verilog Matching — Allows you to designate which Verilog signal values will match the Verilog X and Z values. It also allows you to ignore the strength of the Verilog signal and consider only logic values.

Save as Default — Allows you to save all changes as the new default settings for subsequent waveform comparisons.

Reset to Default — Resets all settings to original default values.

Automatically add comparisons to the wave window?— Specifies whether new signal comparison objects are added automatically to the Wave window.

- **Comparison Method**



Clocked (Strobed) Comparison — Allows you to select a default reference clock signal via a selection history or a browse button.

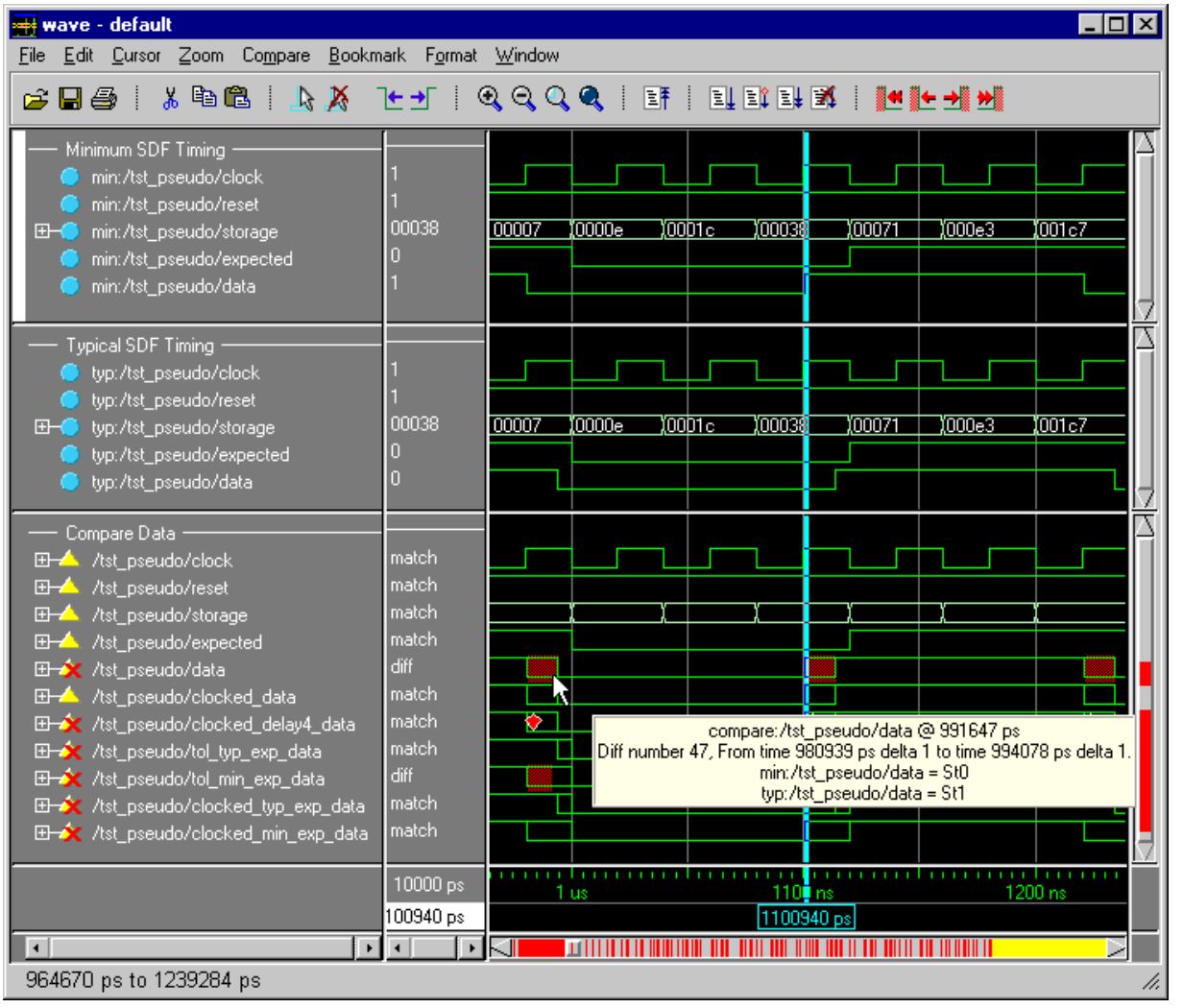
Continuous Comparison — Allows you to set leading and trailing edge tolerances for the waveform comparison. The leading edge tolerance specifies how much earlier the test signal edge may occur before the reference signal edge. The trailing edge tolerance specifies how much later the test signal edge may occur after the reference signal edge. The default value for both tolerances is zero.

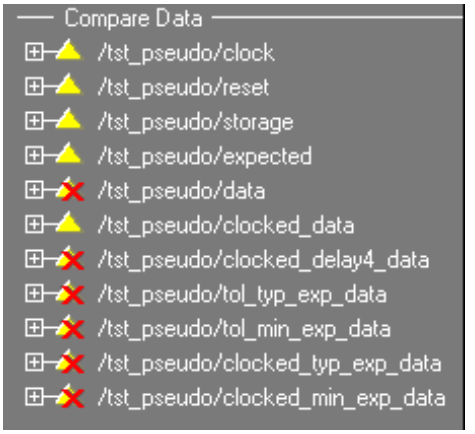
Specify When Expression — Allows you to specify a **when** expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective. Clicking the Builder button will give you access to "The GUI Expression Builder" (8-275).

Save as Default — Allows you to save all changes as the new default settings for subsequent waveform comparisons.

Reset to Default — Resets all settings to original default values.

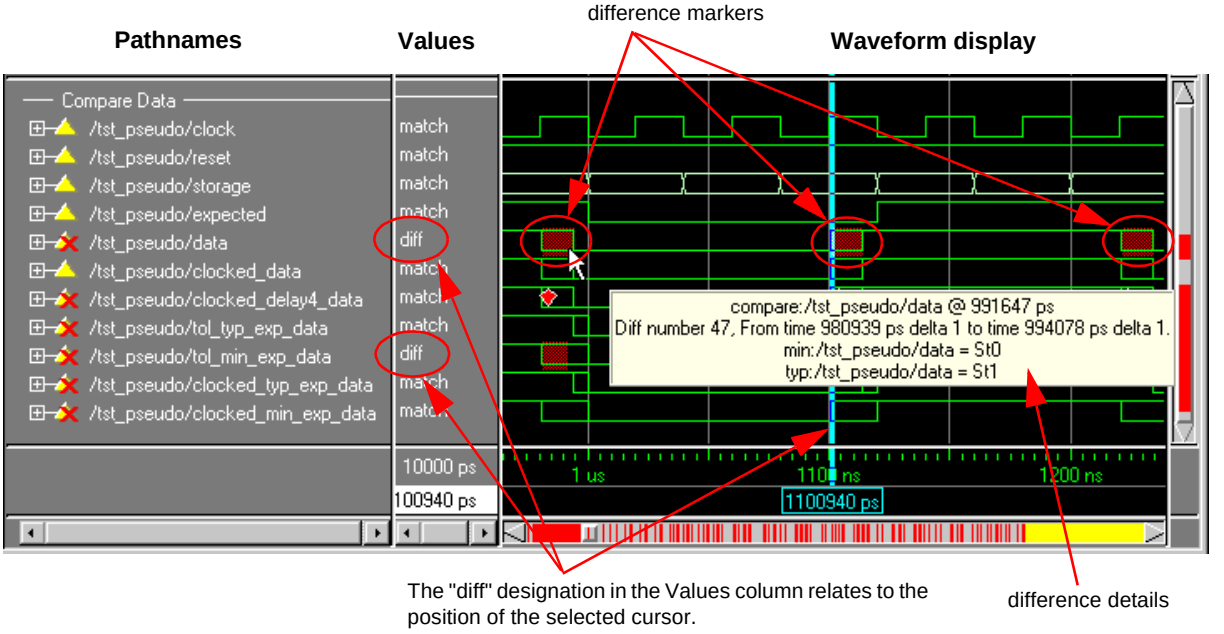
Wave window display





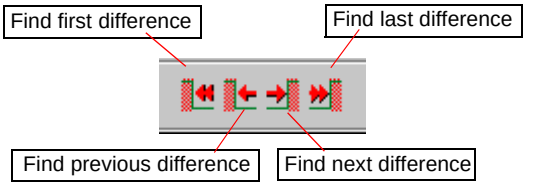
The Wave window provides a graphic display of waveform comparison results. Pathnames of all test signals included in the waveform comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.

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.



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.

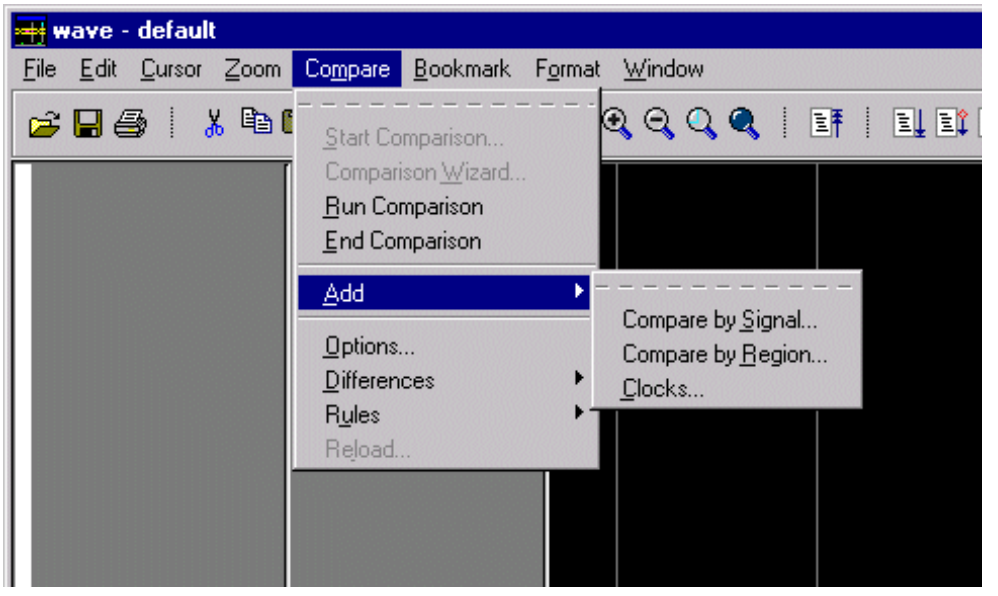
Compare icons



The Wave window includes four waveform comparison icons that enable you to quickly locate the first and last waveform difference and move the cursor in steps to the previous or next difference. The next and previous icons move between differences on all signals in the Wave window. If you want to move between differences for the selected signal only, use <tab> (next) or <shift>-<tab> (previous).

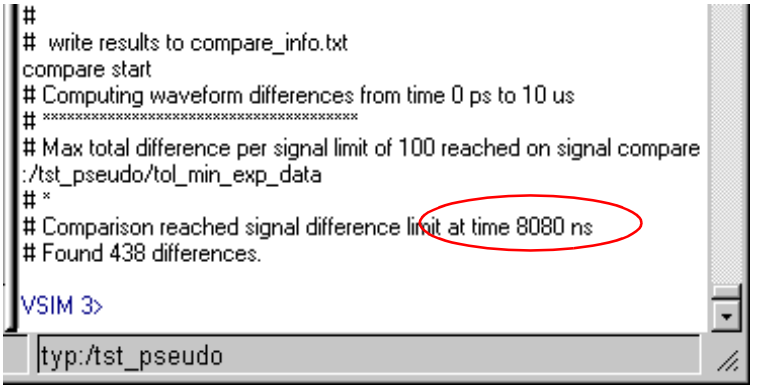
Compare menu

The Compare menu provides a number of options for controlling waveform comparisons.



- **Start Comparison**
  Opens the Compare Dataset dialog box (page 11-305) 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 and saved to the compare_info.txt file. It is equivalent to the **compare run command** (CR-80).



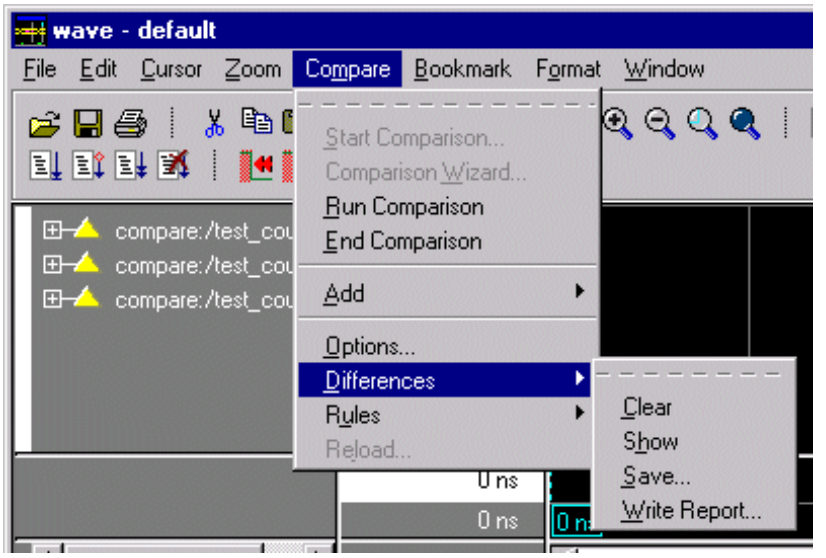
- **End Comparison**
  Stops difference computation and closes the currently open comparison.
- **Add**

  Compare by Signal — Opens the structure_browser dialog box (page 11-307) and allows you to designate signals for comparison.

  Compare by Region — Opens the Add Comparison by Region dialog box (page 11-311) 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 (page 11-308) and allows you to define clocks to be used in the comparison.
- **Options**
  Opens the Add Signal Options dialog box (page 11-314), which allows you to define a number of waveform comparison options.

- **Differences**



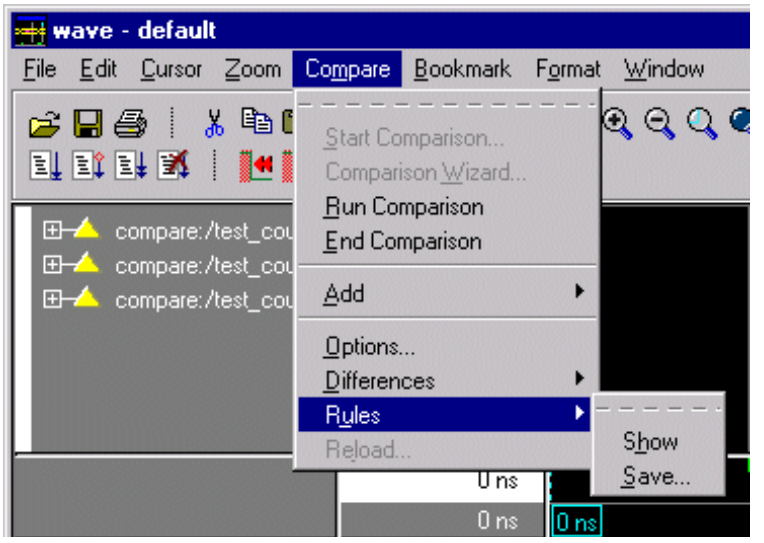
Clear — Clears all differences from the Wave window and resets the waveform comparison function. It is equivalent to the **compare reset** command (CR-79).

Show — Displays the difference in text format in the transcript area of the Main window. It is equivalent to the **compare info** command (CR-73).

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.

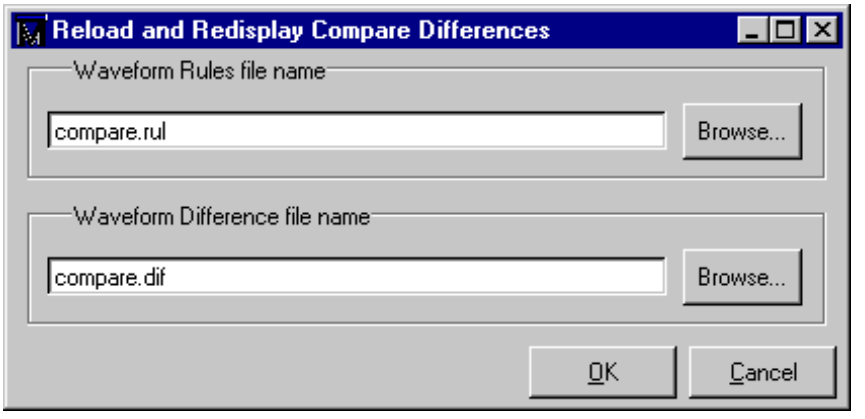
- **Rules**



Show — Displays the rules or instructions used to set up the waveform compare. It is equivalent to the **compare list** command (CR-74).

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.

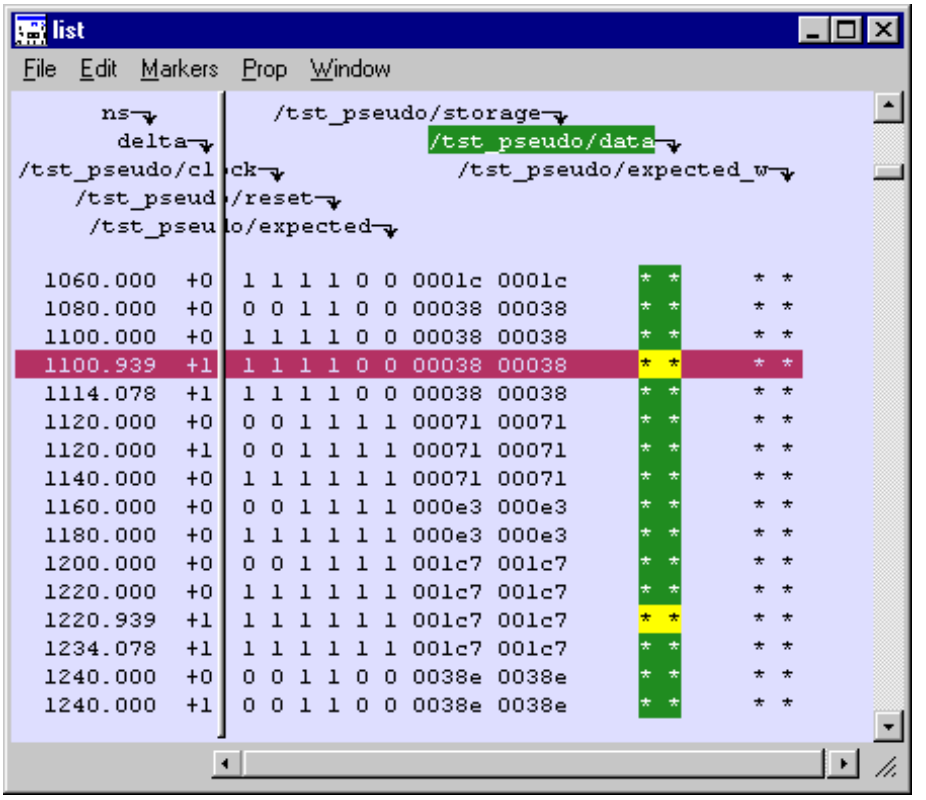


Printing compare differences

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" (8-245) for details.

List window display

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.



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.

Command-line interface to Waveform Comparison

Preference Variables

Various Tcl variables control the default options of the Waveform Comparison feature. See http://www.model.com/resources/pref_variables/frameset.htm for details on how to set these variables.

Compare commands

The table below provides a brief description of the compare commands. Follow the links for complete command syntax.

See *ModelSim Commands* for complete command details.

Command	Description
compare add (CR-63)	defines a comparison between the signals in a specified reference design and the signals in a specified test design
compare annotate (CR-66)	allows a difference to be flagged as **ignore**, or an additional text string to be attached
compare clock (CR-67)	defines a clock for clocked comparison; or, if -delete is specified, deletes a previously-defined clock
compare delete (CR-71)	deletes a signal or region from the current open comparison.
compare end (CR-72)	destroys the compare data structures and forgets clock definitions and signals selected for comparison
compare info (CR-73)	writes out results of the comparison; writes to the transcript unless the -write option is specified
compare list (CR-74)	shows all the **compare region** and **compare signal** commands currently in effect
compare options (CR-75)	sets values for various compare options on the Tcl parser side; when subsequent commands are called, these values become the defaults
compare reset (CR-79)	clears the current compare differences, allowing another **compare start** to be executed
compare reload (CR-78)	reloads comparison differences to allow viewing without recomputation
compare run (CR-80)	registers required callbacks and runs the difference computation on the signals selected for comparison; reports the total number of errors found
compare savediffs (CR-81)	saves the comparison result differences in a form that can be reloaded later
compare saverules (CR-82)	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

Command	Description
compare see command (CR-83)	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-84)	initializes internal data structures for waveform compare
compare stop command (CR-86)	used internally by the **compare stop** button to suspend comparison computations in progress
compare update command (CR-87)	used internally to update the comparison differences when comparing a live simulation against a .wlf file

12 - Standard Delay Format (SDF) Timing Annotation

Chapter contents

This chapter discusses Model*Sim*'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 Model*Sim*.

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: In order to conserve disk space, Model*Sim* will read sdf files that were compressed using the standard unix/gnu file compression algorithm. The filename must end with the suffix ".Z" for the decompress to work.

Specifying SDF files for simulation

Model*Sim* 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-258) 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,

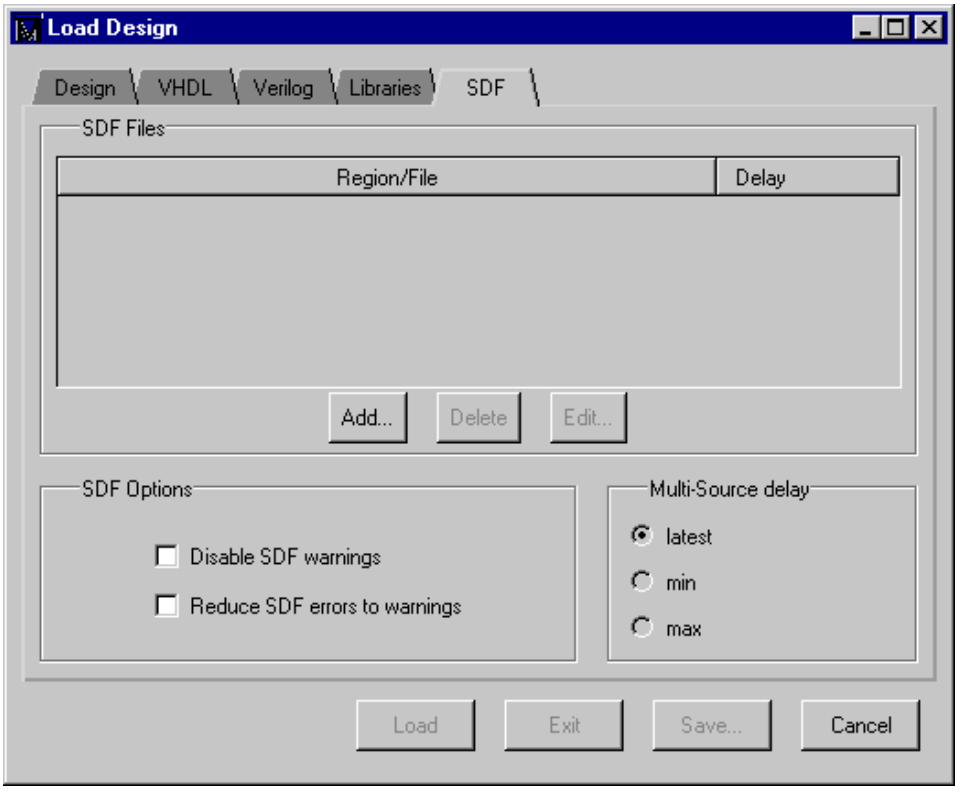
```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

One exception to the rule of never omitting the instance name occurs when your SDF file contains only one instance. In this case, you can omit the instance name. For example, if *myasic.sdf* has only one instance of *u1*, the first command above would look as follows:

```
vsim -sdfmax myasic.sdf testbench
```

SDF specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Load Design** dialog box under the SDF tab.



You can access this dialog by invoking the simulator without any arguments or by selecting **Design** > **Load Design** (Main window). For Verilog designs, you can also specify SDF files by using the **$sdf_annotate** system task. See "The $sdf_annotate system task" (12-330) 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-258) 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** page from the **Load Design** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn, or +nosdfwarn) to disable warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See "Troubleshooting" (12-337) 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, we provide the following summary to help you understand simulator error messages. For additional VITAL specification information, see "Obtaining the VITAL specification and source code" (4-65).

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: 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-258) with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see "Troubleshooting" (12-337).

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 it applies to:

```
$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 port:

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. 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. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(data, edge[01, 0x] clk, 0);

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-258) 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. With Verilog designs, Model*Sim* can model single interconnect delays or multisource interconnect delays. See "Arguments, Verilog" (CR-265) under the **vsim** command for more information on the relevant command-line switches.

Per VHDL VITAL '95, there is no convenient way to handle interconnect delays from multiple outputs to a single input. Interconnect delay is modeled in the receiving device as a single delay from an input port to an internal node. (The node is explicitly declared.) The default is to use the value of the maximum encountered delay in the SDF file. Alternatively, you can choose the minimum or latest value of the multiple delays with the **vsim** command (CR-258) **-multisource_delay** option.

```
-multisource_delay min|max|latest
```

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.

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. (One exception is when you have a single instance in the SDF file.) See "Instance specification" (12-326) for an example.

A common example for both VHDL and Verilog test benches 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-114). 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: myasic.sdf:
The design does not have an instance named '/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: myasic.sdf:
Failed to find INSTANCE '/testbench/u1'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u2'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u3'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u4'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u5'

WARNING: myasic.sdf:
This file is probably applied to the wrong instance.

WARNING: 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: myasic.sdf:
Failed to find any of the 358 instances from this file.

WARNING: 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" (12-329) for specific VHDL VITAL SDF troubleshooting.

Obtaining the SDF specification

The SDF specification is available from Open Verilog International:

Lynn Horobin
phone: (408)358-9510
fax: (408)358-3910
email: info@ovi.org
home page: http://www.ovi.org

13 - Value Change Dump (VCD) Files

Chapter contents

This chapter explains Model Technology's Verilog VCD implementation for Model*Sim*.

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. Model*Sim* provides simulator command equivalents for these system tasks and extends VCD support to VHDL designs; the Model*Sim* commands can be used on either VHDL or Verilog designs.

VHDL VCD files can be used for resimulation with the **vsim -vcdread** command. See "Resimulating a VHDL design from a VCD file" (13-344).

Note: If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

Model*Sim* 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-198)	$dumpvars
vcd checkpoint (CR-199)	$dumpall
vcd file (CR-208)	$dumpfile
vcd flush (CR-212)	$dumpflush
vcd limit (CR-213)	$dumplimit
vcd off (CR-214)	$dumpoff
vcd on (CR-215)	$dumpon

Model*Sim* versions 5.5 and later support 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-198) `-file <filename>`	$fdumpvars
vcd checkpoint (CR-199) `<filename>`	$fdumpall
vcd files (CR-210) `<filename>`	$fdumpfile
vcd flush (CR-212) `<filename>`	$fdumpflush
vcd limit (CR-213) `<filename>`	$fdumplimit
vcd off (CR-214) `<filename>`	$fdumpoff
vcd on (CR-215) `<filename>`	$fdumpon

Model*Sim* versions 5.5 and later also support dumpports system tasks. The table below maps the VCD dumpports commands to their associated tasks.

VCD dumpports commands	VCD system tasks
vcd dumpports (CR-201)	$dumpports
vcd dumpportsall (CR-203)	$dumpportsall
vcd dumpportsflush (CR-204)	$dumpportsflush
vcd dumpportslimit (CR-205)	$dumpportslimit
vcd dumpportsoff (CR-206)	$dumpportsoff
vcd dumpportson (CR-207)	$dumpportson

Resimulating a VHDL design from a VCD file

A VCD file intended for resimulation is created by capturing the ports of a VHDL design unit instance within a testbench or design. The following discussion shows you how to prepare a VCD file for resimulation. Note that the preparation varies depending on your design. Also note that you cannot resimulate with VCD stimulus in a Verilog or mixed-language design.

Extracting the proper stimulus for bidirectional ports

To extract the proper stimulus for bidirectional ports, the **splitio** command (CR-185) must be used before creating the VCD file. This splits bidirectional ports into separate signals that mirror the output driving contributions of their related ports. By recording in the VCD file both the resolved value of a bidirectional port and its output driving contribution, an appropriate stimulus can be derived by **vsim -vcdread**. The **splitio** command (CR-185) operates on a bidirectional port and creates a new signal having the same name as the port suffixed with "__o". This new signal must be captured in the VCD file along with its related bidirectional port. See the description of the **splitio** command (CR-185) for more details.

Note: When using the splitio command in conjunction with VCD files, be aware that VCD file output will vary between a model coded in VHDL and the same model coded in Verilog with timing wrapped in VHDL. The difference occurs because splitio generates Extended VCD stimulus files, and the Extended VCD format is supported only for pure VHDL designs.

Specifying a filename and state mappings

After using splitio, the VCD filename and state mapping are specified using the **vcd files** command (CR-210) with the **-nomap -direction** options.

Note that the **-nomap** option is not necessary if the port types on the top-level design are bit or bit_vector. It is required, however, for std_logic ports because it records the entire std_logic state set. This allows the **-vcdread** option to duplicate the original stimulus on the ports.

The default VCD file is *dump.vcd*, but you can specify a different filename with **vcd files**.

Creating the VCD file

After invoking **vcd files** you can create the new VCD file by executing **vcd add** (CR-198) at the time you wish to begin capturing value changes. To dump everything in a design to a dump file you might use a command like this:

```
vcd add -r /*
```

At a minimum, the VCD file must contain the in and inout ports of the design unit. Value changes on all other signals are ignored by **-vcdread**. This also means that the simulation results are not checked against the VCD file.

After the VCD file is created, it can be input to **vsim** (CR-258) with the **-vcdread** option to resimulate the design unit stand-alone.

Example

The following example illustrates a typical sequence of commands to create a VCD file for input to **-vcdread**. Assume that a VHDL testbench named *testbench* instantiates *dut* with an instance name of *u1*, and that you would like to simulate *testbench* and later be able to resimulate *dut* stand-alone:

```
vsim -c -t ps testbench
VSIM 1> splitio /u1/*
VSIM 2> vcd files -nomap -direction
VSIM 3> vcd add -ports /u1/*
VSIM 4> run 1000
VSIM 5> quit
```

Now, to resimulate using the VCD file:

```
vsim -c -t ps -vcdread dump.vcd dut
VSIM 1> run 1000
VSIM 2> quit
```

Note: You must manually invoke the **run** command (CR-176) even when using **-vcdread**.

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 and quits the simulator at time 1200:

```
vcd files 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
```

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

```
$comment
    File created using the following
command:
        vcd files output.vcd
$date
    Fri Jan 12 09:07:17 2000
$end
$version
    ModelSim EE/PLUS 5.4
$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 vendor's 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 Model*Sim* use the **vcd dumpports** command (CR-201) 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

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 tri-state
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
C unknown (input driving unknown and output driving low)
b unknown (input driving high and output driving unknown)
B unknown (input driving high and output driving low)
c unknown (input driving unknown and output driving high)

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 dumpports results/dump1
$end
$date
Tue Aug 20 13:33:02 2000
$end
$version
ModelSim Version 5.4c
$end
$timescale
1ns
$end
$scope module top1 $end
$scope module u1 $end
$var port 1 <0 a $end
$var port 1 <1 b $end
$var port 1 <2 c $end
$upscope $end
$upscope $end
$enddefinitions $end
#0
$dumpports
pN 6 6 <0
pX 6 6 <1
p? 6 6 <2
$end
#10
pX 6 6 <1
pN 6 6 <0
p? 6 6 <2
#20
pL 6 0 <1
pD 6 0 <0
pa 6 6 <2
#30
pH 0 6 <1
pU 0 6 <0
pb 6 6 <2
#40
pT 0 0 <1
pZ 0 0 <0
pX 6 6 <2
#50
pX 5 5 <1
pN 5 5 <0
p? 6 6 <2
#60
pL 5 0 <1
pD 5 0 <0
pa 6 6 <2
#70
pH 0 5 <1
pU 0 5 <0
pb 6 6 <2
#80
pX 6 6 <1
pN 6 6 <0
p? 6 6 <2
```

14 - Logic Modeling SmartModels

Chapter contents

The Logic Modeling SWIFT-based SmartModel library can be used with Model*Sim* 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 Model*Sim*.

Note: The SmartModel library must be obtained from Logic Modeling along with the SmartModel library documentation that describes how to use it. This chapter only describes the specifics of using the library with Model*Sim* SE.

VHDL SmartModel interface

Model*Sim* 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 Model*Sim* 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 Model*Sim* 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 Model*Sim* 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 Model*Sim* installation directory indicated by the **MODEL_TECH** environment variable. Model*Sim* 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 SmartModel library documentation for details.

Creating foreign architectures with sm_entity

The Model*Sim* **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 Model*Sim* 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 Model*Sim* 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. Model*Sim* 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-114)). 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>"
```

Once again, consult your SmartModel library documentation for details on these commands.

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 for details on this feature. The simulator interface to this feature is described below.

Window name syntax is important. Beginning in version 5.3c of Model*Sim*, 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\
```

As with all extended identifiers, case is important.

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 can then be referenced in other simulator commands just like any other signal (the **add list** command (CR-28) is 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.

LMTV usage documentation

The *SmartModel Library Simulator Interface Manual* is installed with Logic Modeling's software. Look for the file: *<LMC_install_dir>/doc/smartmodel/manuals/slim.pdf*. This document is written with Cadence Verilog in mind, but mostly applies to Model*Sim* Verilog. **Make sure you follow the instructions below for linking the LMTV interface to the simulator.**

Linking the LMTV interface to the simulator

Model Technology ships a dynamically loadable library that links Model*Sim* to the LMTV interface. To link to the LMTV all you need to do is add *libswiftpli.sl* to the **Veriuser** line in *modelsim.ini* as in the example below:

```
Veriuser = $MODEL_TECH/libswiftpli.sl
```

Note: On Windows platforms, the above file should be named libswiftpli.dll.

Compiling Verilog shells

Once *libswiftpli.sl* is in the *modelsim.ini* file you can compile the Verilog shells provided by Logic Modeling. You compile them just like any other Verilog modules in Model*Sim* Verilog. Details on the Verilog shells are in the *SmartModel Library Simulator Interface Manual* as well. The command line plus options and LMTV system tasks described in that document also apply to Model*Sim*.

15 - Logic Modeling Hardware Models

Chapter contents

Logic Modeling hardware models can be used with Model*Sim* 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 Model*Sim*.

Note: Please refer to the Logic Modeling documentation for details on using the hardware modeler. This chapter only describes the specifics of using hardware models with Model*Sim* SE.

VHDL Hardware Model interface

Model*Sim* 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 Model*Sim* 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 Model*Sim* 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
```

The **libhm** entry points to the Model*Sim* 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 Model*Sim* installation directory indicated by the MODEL_TECH environment variable. Model*Sim* 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 the Logic Modeling documentation.

Creating foreign architectures with hm_entity

The Model*Sim* **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 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 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 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.

16 - Tcl and Model*Sim*

Chapter contents

This chapter provides an overview of Tcl (tool command language) as used with Model*Sim*. Additional Tcl and Tk (Tcl's toolkit) can be found through several Tcl online references (16-370).

Tcl is a scripting language for controlling and extending Model*Sim*. Within Model*Sim* 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 Model*Sim*. In addition, if Model*Sim* does not provide the command you need, you can use Tcl to create your own commands.

Tcl features within Model*Sim*

Using Tcl with Model*Sim* 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)

Note: Model*Sim* PE *does not* support Tk. You must be using Model*Sim* SE to customize the interface.

Tcl References

Tcl printed references

Two sources of information 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.

Tcl online references

The following are a few of the many Tcl references available:

- Select **Help** > **Tcl Man Pages** (Main window).
- Tcl man pages are also available at: http://dev.scriptics.com/man/tcl8.1
- Tcl/Tk general information is available from the Tcl/Tk Consortium: www.tclconsortium.org
- The Scriptics Corporation, John Ousterhout's company (the original Tcl developer): www.scriptics.com.

Tcl tutorial

For some hands-on experience using Tcl with Model*Sim*, see the "Tcl/Tk and Model*Sim*" lesson in the *ModelSim SE Tutorial*.

Tcl commands

The Tcl commands are listed below. For complete information on Tcl commands, select **Help > Tcl Man Pages** (Main window) or refer to one of the Tcl/Tk resources noted above. Also see "Preference variables located in TCL files" (B-406) for information on Tcl variables.

append	array	break	case	catch
cd	close	concat	continue	eof
error	eval	exec	expr	file
flush	for	foreach	format	gets
glob	global	history	if	incr
info	insert	join	lappend	list
llength	lindex	lrange	lreplace	lsearch
lsort	open	pid	proc	puts
pwd	read	regexp	regsub	rename
return	scan	seek	set	split
string	switch	tell	time	trace
source	unset	uplevel	upvar	while

Note: Model*Sim* command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Previous Model*Sim* command	Command changed to (or replaced by)
continue	**run** (CR-176) with the **-continue** option
format list | wave	**write format** (CR-277) with either list or wave specified
if	replaced by the Tcl **if** command, see "if command syntax" (16-374) for more information
list	**add list** (CR-28)
nolist | nowave	**delete** (CR-99) with either list or wave specified
set	replaced by the Tcl **set** command, see "set command syntax" (16-375) for more information
source	**vsource** (CR-270)
wave	**add wave** (CR-37)

Tcl command syntax

The former Model*Sim* commands, **if** and **set** are now Tcl commands. You should understand Tcl command syntax before using these commands. The syntax, especially for the **if** command, may be unfamiliar.

The following rules define the syntax and semantics of the Tcl language. Details on if command syntax (16-374) and set command syntax (16-375) follow the general discussion of Tcl command syntax.

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 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.
\xhh	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.

More Tcl commands

All Tcl commands are documented from within Model*Sim*. Select **Help > Tcl Man Page** (Main window).

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, 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 Model*Sim* 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" (B-408) for more information about Model*Sim*-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 Model*Sim* Tcl command: **lecho** (CR-128)

Model*Sim* Tcl commands

These additional commands enhance the interface between Tcl and Model*Sim*. Only brief descriptions are provided here; for more information and command syntax see the "ModelSim Commands" (CR-9).

Command	Description
alias (CR-41)	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find (CR-119)	locates incrTcl classes and objects
lecho (CR-128)	takes one or more Tcl lists as arguments and pretty-prints them to the Main window
lshift (CR-133)	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist (CR-134)	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv (CR-152)	echoes to the Main window the current names and values of all environment variables

Model*Sim* Tcl time commands

Model*Sim* 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 Model*Sim* 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

Example 1

The following Tcl/Model*Sim* example for UNIX 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.

Note: In a Windows environment, the Tcl **exec** command shown below will execute compiled files only, not system commands.

(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 [exec date]
    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 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 ""
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 ""
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 ""
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 ""
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 ""
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

The last example is of the Tcl **switch** command:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

Example 2

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 example works in Model*Sim* SE only.

```
##  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

##  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 dialog_prompt vsimPriv
    set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
    set dialog_prompt(result) $defaultName
    set windowName [GetValue . "Create Named Wave Window:" ]
    ## 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"}} {
    global vsimPriv
    view -new wave
    set newWave [lindex $vsimPriv(WaveWindows) [expr [llength \
    $vsimPriv(WaveWindows)] - 1]]
    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
    set fileId [open windowSet_$fileroot.do w 755]
    foreach w $vsimPriv(WaveWindows) {
        echo "Saving: [wm title $w]"
        set filename $fileroot$n.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
    }
```

```
    if {![catch {glob $fileroot\[$n-9\].do}]} {
        foreach f [lsort [glob $fileroot\[$n-9\].do]] {
            echo "Removing:  $f"
            exec rm $f
        }

    }

}

## Provide file root argument and load_wave restores all saved widows.
## Default file root is "wave".

proc load_wave {{fileroot "wave"}} {
    global vsimPriv
    foreach f [lsort [glob $fileroot\[1-9\].do]] {
        echo "Loading: $f"
        view -new wave
        do $f
    }
    if {[file exists windowSet_$fileroot.do]} {
        do windowSet_$fileroot.do
    }
}
```

A - Technical Support, Updates, and Licensing

Appendix contents

Technical support - electronic

Model Technology customers

Support questions may be submitted through the Model Technology online support form at: www.model.com. Model Technology customers may also email test cases to support@model.com; please provide the following information, in this format, in the body of your email message:

- Your name:
  Company:
  Email address (if different from message address):
  Telephone:
  FAX (optional):
- Model*Sim* product (SE, EE or PE, and VHDL, VLOG, or PLUS):
- Model*Sim* Version:
  (Use the Help About dialog box with Windows; type **vcom** for UNIX workstations.)
- Host operating system version:
- PC hardware security key authorization number:
- Ethernet card address if used for authorization:
- Host ID of license server for workstations:
- Description of the problem (please include the exact wording of any error messages):

Note: Model Technology customers in Europe should contact their distributor for support. See www.model.com/contact_us.asp for distributor contact information.

Mentor Graphics customers

Mentor Graphics Customer Support offers a SupportNet-Email server for North American and European companies that lets customers find product information or submit service requests (call logs) to the SupportCenter 24 hours a day, 365 days a year. The server will return a call log number within about 15 minutes. CAEs follow up on the call logs submitted through SupportNet-Email using the same process as if a customer had phoned the SupportCenter. For more information about using the SupportNet-Email server, send a blank e-mail message to the following address: support_net@mentor.com.

Additionally, customers can open call logs or search TechNotes and AppNotes to try to find the answers to their questions by logging onto Mentor Graphics' Customer Support web home page at www.mentor.com/supportnet.

If you are not yet registered for SupportNet and have an active support contract with Mentor Graphics, you may do so by clicking **Request Log-In** and filling out the information at: www.mentor.com/supportnet_register/

While all contract customers worldwide are invited to obtain a SupportNet Log-In, SupportNet services are currently limited to customers who receive support from Mentor support offices in North America or Europe. If you receive support from Mentor offices outside of North America or Europe, please contact your local field office to obtain assistance for a technical-support issue.

Technical support - telephone

Model Technology customers worldwide

For customers who purchased from Model Technology, please contact Model Technology via the support line at 1-503-641-1340 from 8:00 AM to 5:00 PM Pacific Time, Monday through Friday, excluding holidays. Be sure to have your server hostID, ethernet card address, or hardware security key authorization number handy.

Note: Model Technology customers in Europe should contact their distributor for support. See www.model.com/contact_us.asp for distributor contact information.

Mentor Graphics customers in North America

For customers who purchased products from Mentor Graphics in North America, and are under a current support contract, technical telephone support is available from the central SupportCenter by calling toll-free 1-800-547-4303. The coverage window is from 5:30am to 5:30pm Pacific Time, Monday through Friday, excluding Mentor Graphics holidays.

The more details you can supply about a problem or issue, the sooner a Corporate Application Engineer can supply you with a solution or workaround. Be prepared to provide the following important information:

- The priority of the call (critical, high, medium, low)
- The product about which you are calling
- Your operating system and software version numbers (accuracy is very important here)
- The steps that led to the problem or crash
- If it is a crash, the first few lines of a traceback
- Any non-Mentor Graphics tools or customized software that may be involved

Mentor Graphics customers outside North America

Customers who purchased products from Mentor Graphics outside of North America, should contact their local support organization. A list of local Mentor Graphics support and sales offices can be found at www.mentor.com/supportnet/support_offices.html.

Technical support - other channels

For customers who purchased Model*Sim* as part of a bundled product from an OEM, or VAR, please refer to the www.model.com/partners/default.asp on the Model Technology website for contact information.

Updates

Model Technology customers

You can ftp the latest version of the software from the web site at ftp://ftp.model.com. Instructions are there as well.

Mentor Graphics customers

You can ftp the latest SE or PE version of the software from the SupportNet site at ftp://supportnet.mentor.com/pub/mentortech/modeltech/. Instructions are there as well. A valid license file from Mentor Graphics is needed to uncompress the Model*Sim* files.

Online References

The Model Technology web site (www.model.com) includes links to support, software downloads, and many EDA information sources. Check the links below for the most current information.

Latest version email

Place your name on our list for email notification of new releases and updates.
model.com/support/register_news_list.asp

News

Current news of Model Technology within the EDA industry.
model.com/news_events/default.asp

Partners

Model Technology's value added partners, OEM partners, FPGA partners, ASIC partners, and training partners.
model.com/partners/default.asp

Products

A complete collection of Model Technology product information.
model.com/products/default.asp

Technical Documents

Technical notes, application notes, FAQs.
model.com/resources/techdocs.asp

Sales

Locate Model*Sim* sales contacts anywhere in the world.
model.com/contact_us.asp

Support

Model Technology email support and software downloads.
model.com/support/default.asp

FLEXlm Licenses

ModelSim uses Globetrotter's FLEXlm license manager and files. Globetrotter FLEXlm license files contain lines that can be referred to by the word that appears first on the line. Each kind of line has a specific purpose and there are many more kinds of lines that MTI does not use.

Mentor Graphics customers

Mentor Graphics provides licensing information in the *Mentor Graphics Licensing* chapter in the Managing Mentor Graphics Software document. In addition, Model Technology provides some basic Mentor Graphics licensing files. See the readme file in the MGLS-related directory at ftp.model.com/pub/SE for more information.

Where to obtain your license

Mentor Graphics customers must contact their Mentor Graphics salesperson for ModelSim licensing. All other customers may obtain ModelSim licenses from Model Technology. Please contact Model Technology at license@model.com.

If you have trouble with licensing

Contact your normal technical support channel:

Technical support - electronic (A-386)

Technical support - telephone (A-387)

Technical support - other channels (A-387)

Maintenance renewals and licenses

When maintenance is renewed, a new license file that incorporates the new maintenance expiration date will be automatically sent to you. If maintenance is not renewed, the current license file will still permit the use of software versions built before maintenance expired until the stop date is reached.

License transfers and server changes

Model Technology and Mentor Graphics both charge a fee for server changes or license transfers. Contact sales@model.com for more information from Model Technology, or contact your local Mentor Graphics sales office for Mentor Graphics purchases.

Additional licensing details

A complete discussion of licensing is located in the *Start Here for ModelSim* guide. For an online version of *Start Here*, check the ModelSim Main window Help menu for SE Documentation.

B - Model*Sim* Variables

Appendix contents

This appendix documents the following types of Model*Sim* variables:

- **environment variables**
  Variables referenced and set according to operating system conventions. Environment variables prepare the Model*Sim* environment prior to simulation.
- **Model*Sim* preference variables**
  Variables used to control compiler or simulator functions (usually in .tcl files) and modify the appearance of the Model*Sim* GUI (usually in INI files).
- **simulator state variable**s
  Variables that provide feedback on the state of the current simulation.

Variable settings report

The **report** command (CR-168) returns a list of current settings for either the simulator state, or simulator control variables. Use the following commands at either the Model*Sim* or VSIM prompt:

```
report simulator state
report simulator control
```

Personal preferences

There are several preferences stored by Model*Sim* 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\Model*Sim*.

- **cwd**
  History of the last five working directories (pwd). This history appears in the Main window File menu.
- **phst**
  Project History
- **pinit**
  Project Initialization state (one of: Welcome | OpenLast | NoWelcome). This determines whether the Welcome To Model*Sim* dialog box appears when you invoke the tool.
- **printersetup**
  All setup parameters related to Printing (i.e., current printer, etc.)

The HKEY_CURRENT_USER key is unique for each user Login on Windows NT.

Returning to the original Model*Sim* defaults

If you would like to return Model*Sim*'s interface to its original state, simply rename or delete the existing *modelsim.tcl* and *modelsim.ini* files. Model*Sim* will use *pref.tcl* for GUI preferences and make a copy of *<install_dir>/modeltech/modelsim.ini* to use the next time Model*Sim* 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 95/98 machines, and set through the System control panel on NT machines. For UNIX, the variables are typically found in the .login script. The LM_LICENSE_FILE variable is required, all others are optional.

Model*Sim* Environment Variables

Variable	Description
DOPATH	used by Model*Sim* to search for simulator command files (do files); consists of a colon-separated (semi-colon for Windows) list of paths to directories; optional; this variable can be overridden by the DOPATH .tcl file variable
EDITOR	specifies the editor to invoke with the **edit** command (CR-110)
HOME	used by Model*Sim* to look for an optional graphical preference file and optional location map file; see: "Preference variables located in INI files" (B-396) and "Using location mapping" (E-437)
LM_LICENSE_FILE	used by the Model*Sim* 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; see: "Using the FLEXlm License Manager" (D-417)
MODEL_TECH	set by all Model*Sim* tools to the directory in which the binary executables reside; **YOU SHOULD NOT SET THIS VARIABLE**
MODEL_TECH_TCL	used by Model*Sim* to find Tcl libraries for: Tcl/Tk 8.0, Tix, and vsim; defaults to /modeltech/../tcl; may be set to an alternate path
MGC_LOCATION_MAP	used by Model*Sim* tools to find source files based on easily reallocated "soft" paths; optional; see: "Using location mapping" (E-437); also see the Tcl variables: SourceDir and SourceMap
MODELSIM	used by all Model*Sim* tools to find the *modelsim.ini* file; consists of a path including the file name; optional. 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 Model*Sim* tools.
MODELSIM_TCL	used by Model*Sim* to look for an optional graphical preference file; can be a colon-separated (UNIX) or semi-colon (Windows) separated list of file paths
MTI_TF_LIMIT	limits the size of the VSOUT temp file (generated by the Model*Sim* 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
MTI_USELIB_DIR	specifies the directory into which object libraries are compiled when using the -compile_uselibs argument to the **vlog** command (CR-250)

Variable	Description
PLIOBJS	used by Model*Sim* to search for PLI object files for loading; consists of a space-separated list of file or path names; optional
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	specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel; optional

Setting 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 95/98

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

Right-click the My Computer icon and select Properties, then select the Environment tab of the System Properties control panel. Add the new variable to these fields: Variable:MY_PATH and Value:\temp\work.

Click Set and Apply to initialize the variable (you don't need to restart NT).

Library mapping with environment variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command (CR-257) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you're using **vmap** from Model*Sim*/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 Model*Sim*/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
```

Note: 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 Model*Sim*

There are two ways to reference environment variables within Model*Sim*. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
entity test is end;
use std.textio.all;

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 Model*Sim* command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

Removing temp files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the Model*Sim* GUI. In normal circumstances the file is deleted when the simulator exits. If Model*Sim* 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.

Note: 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** compiler or **vsim** simulator was invoked. **MODEL_TECH** is used by the other Model Technology tools to find the libraries.

Preference variables located in INI files

Model*Sim* initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings.

The following tables list the variables by section, and in order of their appearance within the INI file:

[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 /modeltech/../ieee
std	any valid path; may include environment variables	sets the path to the VHDL STD library; the default is /modeltech/../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 /modeltech/../std_developerskit
synopsys	any valid path; may include environment variables	sets the path to the accelerated arithmetic packages; the default is /modeltech/../synopsys
verilog	any valid path; may include environment variables	sets the path to the library containing VHDL/Verilog type mappings; the default is /modeltech/../verilog

[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	off (0)

Variable name	Value range	Purpose	Default
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)
NoOthersStaticError	0, 1	if 1, disables errors caused by aggregates that are not locally static	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 IEEE std_logic_1164 package	on (1)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
RequireConfigForAllDefaultBinding	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)
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	if 1, turns on VHDL-1993	off (0)

[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 vars)	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)
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_source	0, 1	if 1, shows source line containing error	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 options" (5-79)	off (0)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
AssertFile	any valid filename	alternative file for storing assertion messages	transcript
AssertionFormat	see purpose	sets the message to display after a break on assertion; message formats include: %S - severity level %R - report message %T - time of assertion %D - delta %I - instance or region pathname (if available) %% - print '%' character	"** %S: %R\n Time: %T Iteration: %D%I\n"
BreakOnAssertion	0-4	defines severity of assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal)	3
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format	on (1)
CommandHistory	any valid filename	sets the name of a file in which to store the Main window command history	commented out (;)

Variable name	Value range	Purpose	Default
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 single character	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	drive for resolved signals; freeze for unresolved signals
DefaultRadix	symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii	any radix may be specified as a number or name (i.e., binary can be specified as binary or 2)	symbolic
DefaultRestartOptions	one or more of: -force, -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	off (0)
GenerateFormat	Any non-quoted string containing at a minimum a %s followed by a %d	control the format of a generate statement label (don't quote it)	%s__%d
IgnoreError	0,1	if 1, ignore assertion errors	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures	off (0)
IgnoreNote	0,1	if 1, ignore assertion notes	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings	off (0)
IterationLimit	positive integer	limit on simulation kernel iterations during one time delta	5000

Variable name	Value range	Purpose	Default
License	any single <license_option>	if set, controls Model*Sim* 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-258) <license_option>	search all licenses
LockedMemory	positive integer; mb of memory to lock	for HP-UX 10.2 use only; enables memory locking to speed up large designs (> 500mb memory footprint); see "Accelerate simulation by locking memory under HP-UX 10.2" (E-439)	disabled
NumericStdNoWarnings	0, 1	if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed	off (0)
PathSeparator	any single character	used for hierarchical path names; must not be the same character as DatasetSeparator	/
Resolution	fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100	simulator resolution; this value must be less than or equal to the UserTimeUnit specified below; NOTE - if your delays are truncated, set the resolution smaller; no space between value and units (i.e., 10ps, not 10 ps)	ns
RunLength	positive integer	default simulation length in units specified by the UserTimeUnit variable	100
Startup	= do <DO filename>; any valid macro (do) file	specifies the Model*Sim* startup macro; see the **do** command (CR-104)	commented out (;)
StdArithNoWarnings	0, 1	if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed	off (0)
TranscriptFile	any valid filename	file for saving command transcript; environment variables may be included in the path name	transcript

Variable name	Value range	Purpose	Default
UnbufferedOutput	0, 1	controls VHDL and Verilog files open for write; 0 = Buffered, 1 = Unbuffered	0
UserTimeUnit	fs, ps, ns, us, ms, sec, or default	specifies the default units to use for the "<timesteps> [<time_units>]" argument to the **run** command (CR-176); NOTE - the value of this variable must be set equal to, or larger than, the current simulator resolution specified by the Resolution variable shown above	ns
Veriuser	one or more valid shared objects	list of dynamically loadable objects for Verilog PLI/VPI applications; see "Using the Verilog PLI/VPI" (5-108)	commented out (;)
WaveSignalNameWidth	0, positive integer	controls the number of visible hierarchical regions of a signal name shown in the Wave window (8-216); 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
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 - n 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 - n	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

Model*Sim*'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" (14-354) and "VHDL Hardware Model interface" (15-364) respectively.

Spaces in path names

For the Src_Files and Work_Libs variables, each element in the list is enclosed within curly braces ({}). This allows spaces inside elements (since Windows allows spaces inside path names). For example a source file list might look like:

```
Src_Files = {$MODELSIM_PROJECT/counter.v} {$MODELSIM_PROJECT/tb counter.v}
```

Where the file *tb counter.v* contains a space character between the "b" and "c".

Setting variables in INI files

Edit the initialization file directly with any text editor to change or add a variable. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

Note: The **vmap** command (CR-257) automatically modifies library mappings in the current INI file.

Reading variable values from the INI file

These Tcl functions allow you to read values from the *modelsim.ini* file.

`GetIniInt <var_name> <default_value>`
Reads the integer value for the specified variable.

`GetIniReal <var_name> <default_value>`
Reads the real value for the specified variable.

`GetProfileString <section> <var_name> [<default>]`
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 [GetIniInt "CheckpointCompressMode" 1]

set PrefMain(file) [GetProfileString vsim TranscriptFile ""]
```

Variable functions

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.

Examples

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

Tip:

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 compiler 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 Model*Sim* 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.

Examples

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Tip:

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.

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 Model*Sim* history. The size of this file can be controlled with the MTI_TF_LIMIT (B-393) variable.

```
; 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 Model*Sim* 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-104) 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
```

Messages may also be turned off with Tcl variables; see "Preference variables located in TCL files" (B-406).

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
```

Warnings may also be turned off with Tcl variables; see "Preference variables located in TCL files" (B-406).

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.

VHDL93

You can make the VHDL93 standard the default by including the following line in the *INI* file:

```
[vcom]
; Turn on VHDL-1993 as the default. Default is off (VHDL-1987).
VHDL93 = 1
```

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 will affect all windows. For complete documentation on Tcl preference variables, see the following URL:

http://www.model.com/resources/pref_variables/frameset.htm

When ModelSim is invoked for the first time, default preferences are loaded from the *pref.tcl* file. Customized variable settings may be set from within the ModelSim GUI, on the ModelSim command line, or by directly editing the preference file.

The default file for customized preferences is *modelsim.tcl*. If your preference file is not named *modelsim.tcl*, you must refer to it with the MODELSIM_TCL (B-393) environment variable.

User-defined variables

Temporary, user-defined variables can be created with the Tcl **set** command. 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* and MPF files, see "Preference variables located in INI files" (B-396).

Preference variable loading order

ModelSim .tcl, .ini, and .mpf files all contain variables that are loaded when you start ModelSim. The files are evaluated for variable settings in the order below.

.tcl file variables are evaluated before the design is loaded

ModelSim evaluates .tcl files prior to loading a design for simulation. Any window user_hook_variables are evaluated after the associated window type is created.

1. *The <install_dir>/modeltech/tcl/vsim/pref.tcl* file is always loaded.
2. The file specified by the MODELSIM_TCL (B-393) environment variable is loaded next.
3. If MODELSIM_TCL does not exist, the *modelsim.tcl* in the current directory is evaluated.
4. If MODELSIM_TCL and *./modelsim.tcl* do not exist, the file specified by the HOME (B-393) environment variable is used.

.ini and .mpf file variables are evaluated after the design is loaded

After the design is loaded, .ini or .mpf file variables are found in these locations:

1. First the location specified by the MODELSIM (B-393) environment variable,

   If no MODELSIM variable exists, Mod*elSi*m looks for .mpf and .ini files in the locations shown below. Project files (.mpf) are evaluated first, if no project file is found, Model*Sim* looks for an .ini file in the same location.
2. Next in the current directory if no MODELSIM variable exists.
3. Then in the directory where the executable exists (*/<install_dir>/modeltech/<platform>)*.
4. Finally in the parent of the directory where the executable is (*/<install_dir>/modeltech*).

Note: The MODELSIM variable is generally set to an .ini file. Setting the variable to an MPF file is not recommended since the file would contain project-specific information. Setting the MODELSIM variable to an .mpf file is only recommended for batch-mode usage.

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 Model*Sim* DO files (macros).

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 a configuration or 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	returns the current simulation time expressed in the current time resolution (e.g., 1000 ns)
now	returns the current simulation time as an absolute number of time steps (e.g., 1000)
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 $ 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 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.

C - Model*Sim* Shortcuts

Appendix contents

This appendix is a collection of the keyboard and command shortcuts available in the Model*Sim* GUI.

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 - 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

Keystroke	Action
i I or +	zoom in
o O or -	zoom out
f or F	zoom full; mouse pointer must be over the the cursor or waveform panes
l or L	zoom last
r or R	zoom range
<arrow up>	scroll waveform display up by selecting the item above the currently selected item
<arrow down>	scroll waveform display down by selecting the item below the currently selected item
<arrow left>	scroll waveform display left
<arrow right>	scroll waveform display right
<page up>	scroll waveform display up by a page
<page down>	scroll waveform display 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 pane for text strings

List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Key	Action
<arrow up>	scroll listing up (selects and highlights the line above the currently selected line)
<arrow down>	scroll listing down (selects and highlights the line below the currently selected line)
<arrow left>	scroll listing left
<arrow right>	scroll listing right
<page up>	scroll listing up by page
<page down>	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)
<control-f> Windows <control-s> UNIX	opens the find dialog box; finds the specified item label within the list display

Command shortcuts

You may abbreviate command syntax, but there's a catch. The minimum 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 Model*Sim* does not allow command name abbreviations in macro files. This minimizes your need to maintain macro files as new commands are added.

Command history shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the Model*Sim*/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)

Mouse and keyboard shortcuts in the Transcript and Source windows

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 Model*Sim* 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 >	none	paste clipboard
< middle-button - press > + drag	none	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
< alt >		activate or inactivate menu bar mode

Keystrokes - UNIX	Keystrokes - Windows	Result
< alt > < F4 >		close active window
< control - a >, < home >	< home >	move cursor to the beginning of the line
< control - b >		move cursor left
< control - d >		delete character to the right
< control - e >, < end >	< end >	move cursor to the end of the line
< control - f >		move cursor right one character
< control - k >		delete to the end of line
< control - n >		move cursor one line down (Source window only under Windows)
< control - o >	none	insert a newline character in front of the cursor
< control - p >		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 to the right of the cursor
< control - u >		delete line
< control - v >	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	move cursor up one screen
< Meta - w>	< control - c >	copy selection
< F8 >		search for the most recent command that matches the characters typed (Main window only)

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.

Right mouse button

The right mouse button provides shortcut menus in the Main and Wave windows. In the Source window, the button gives you feedback on any HDL item under the cursor. See *Chapter 8 - ModelSim Graphic Interface* for menu descriptions.

D - Using the FLEXlm License Manager

Appendix contents

This appendix covers Model Technology's application of FLEXlm for Model*Sim* licensing.

Globetrotter Software's Flexible License Manager (FLEXlm) is a network floating licensing package that allows the application to be licensed on a concurrent usage basis, as well as on a per-computer basis.

FLEXlm user's manual

The content of this appendix is limited to the use of FLEXlm with Model Technology's software. For more information, see the FLEXlm user's manual that is available from Globetrotter Software's web site:

http://www.globetrotter.com/manual.htm

Starting the license server daemon

Locating the license file

When the license manager daemon is started, it must be able to find the license file. The default location is */usr/local/flexlm/licenses/license.dat* for Unix or *c:\flexlm\license.dat* for Windows. You can change where the daemon looks for the license file using one of two methods:

- By starting the license manager using the **-c <pathname>** option.
- By setting the LM_LICENSE_FILE (B-393) environment variable to the path of the file.

More information about installing Model*Sim* and using a license file is available in Model Technology's *Start Here for ModelSim* guide, see "Where to find our documentation" (1-21), or email us at license@model.com.

Controlling the license file search

By default, Model*Sim* checks for the existence of both Model Technology and Mentor Graphics generated licenses. When vsim is invoked it will first locate and use any available MTI licenses, then search for MGC licenses as needed. The following **vsim** command (CR-258) switches narrow the search to exclude or include specific licenses:

license option	Description
-lic_nomgc	excludes any MGC licenses from the search
-lic_nomti	excludes any MTI licenses from the search
-lic_noqueue	do not wait in license queue if no licenses are available
-lic_plus	searches only for PLUS licenses
-lic_vlog	searches only for VLOG licenses
-lic_vhdl	searches only for VHDL licenses
-lic_viewsim	accepts a simulator license rather than being queued for a viewer license

The options may also be specified with the License (B-400) variable in the *modelsim.ini* file; see the [vsim] simulator control variables (B-398). Note that settings made from the command line are additive to options set in the License variable. For example, if you set the License variable to nomgc and use the -lic_plus option from the command line, vsim will search only for MTI SE/PLUS licenses.

Manual start

Unix

To start the license manager daemon, place the license file in the `/<install_dir>/modeltech/<platform>` directory and enter the following commands:

```
cd /<install_dir>/modeltech/<platform>
lmgrd -c license.dat >& report.log
```

where <platform> can be sunos5, sunos5v9, hp700, hppa64, rs6000, rs64, or linux.

This can be done by an ordinary user; you should not be logged in as root.

Windows

To start the license manager daemon in Windows, place the license file in the *modeltech* installation directory and enter the following commands:

```
cd \<install_dir>\modeltech\win32
lmgrd -app -c license.dat
```

Automatic start at boot time

Unix

You can cause the license manager daemon to start automatically at boot time by adding the following line to the file */etc/rc.boot* or to */etc/rc.local*:

```
/<install_dir>/modeltech/<platform>/lmgrd -c /<install_dir>/license.dat &
```

Windows

You can use the FLEXlm Control Panel to enact an automatic start. See the *FLEXlm End User's Manual* for more information.

What to do if another application uses FLEXlm

If you have other applications that use FLEXlm, you can handle any conflict in one of the following ways:

Case 1: All the license files use the same license server nodes

You can combine the license files by taking the set of SERVER lines from one license file, and adding the DAEMON, FEATURE, and FEATURESET lines from all of the license files. This combined file can be copied to */<install_dir>/license/license.dat* and to any location required by the other applications.

Case 2: The applications use different license server nodes

You cannot combine the license files if the applications use different servers. Instead, set the LM_LICENSE_FILE (B-393) environment variable to be a list of files, as follows:

```
setenv LM_LICENSE_FILE \
    lic_file1:lic_file2:/<install_dir>/license.dat
```

In Windows use semi-colons (;) to separate the file names.

Do not use the **-c** option when you start the license manager daemon. For example:

```
lmgrd > report.log
```

Format of the license file

Model*Sim* license files contain three types of lines: SERVER lines, DAEMON lines, and FEATURE lines. For example:

```
SERVER hostname hostid [TCP_portnumber]
DAEMON daemon-name path-to-daemon [path-to-options-file]
FEATURE name daemon-name version exp_date #users_code \
    “description” [hostid]
```

Only the following items may be modified:

- the hostname on SERVER lines
- the TCP_portnumber on SERVER lines
- the path-to-daemon on DAEMON lines
- the path-to-options-file on DAEMON lines
- anything in the daemon options file (described in the following section)

Format of the daemon options file

You can customize your Model*Sim* licensing with the daemon options file. This options file allows you to reserve licenses for specified users or groups of users, to determine which users have access to Model*Sim* software, to set software time-outs, and to log activity to an optional report writer.

RESERVE
: Ensures that Model*Sim* will always be available to one or more users on one or more host computers.

INCLUDE
: Allows you to specify a list of users who are allowed access to the Model*Sim* software.

EXCLUDE
: Allows you to disallow access to Model*Sim* for certain users.

GROUP
: Allows you to define a group of users for use in the other commands.

NOLOG
: Causes messages of the specified type to be filtered out of the daemon’s log output.

To use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the line that begins with DAEMON modeltech.

A daemon options file consists of lines in the following format:

```
RESERVE number feature {USER | HOST | DISPLAY | GROUP} name
INCLUDE feature {USER | HOST | DISPLAY | GROUP} name
EXCLUDE feature {USER | HOST | DISPLAY | GROUP} name
GROUP name <list_of_users>
NOLOG {IN | OUT | DENIED | QUEUED}
REPORTLOG file
```

Lines beginning with the number character (#) are treated as comments. If the filename in the REPORTLOG line starts with a plus (+) character, the old report logfile will be opened for appending.

For example, the following options file would reserve one copy of the feature **vsim** for the user walter, three copies for the user john, one copy for anyone on a computer with the hostname of bob, and would cause QUEUED messages to be omitted from the logfile. The user rita would not be allowed to use the vsim feature.

```
RESERVE 1 vsim USER walter
RESERVE 3 vsim USER john
RESERVE 1 vsim HOST bob
EXCLUDE vsim USER rita
NOLOG QUEUED
```

If this data were in the file named:

```
/usr/local/options
```

modify the license file DAEMON line as follows:

```
DAEMON modeltech /<install_dir>/<platform>/modeltech \
       /usr/local/options
```

License administration tools

lmstat

License administration is simplified by the **lmstat** utility. **lmstat** allows a user of FLEXlm to instantly monitor the status of all network licensing activities. **lmstat** allows a system administrator at a user site to monitor license management operations, including:

- which daemons are running;
- which users are using individual features; and
- which users are using features served by a specific DAEMON.

The case-sensitive syntax is shown below:

Syntax

```
lmstat
  -a -A
  -S <daemon>
  -c <license_file>
  -f <feature_name>
  -s <server_name>
  -t <value>
```

Arguments

`-a`
Displays everything.

`-A`
Lists all active licenses.

`-S <daemon>`
Lists all users of the specified daemon's features.

`-c <license_file>`
Specifies that the specified license file is to be used.

`-f <feature_name>`
Lists users of the specified feature(s).

`-s <server_name>`
Displays the status of the specified server node(s).

`-t <value>`
Sets the lmstat time-out to the specified value.

lmdown

The **lmdown** utility allows for the graceful shutdown of all license daemons (both **lmgrd** and all vendor daemons) on all nodes.

Syntax

```
lmdown
  -c [<license_file_path>]
```

If not supplied here, the license file used is in either */user/local/flexlm/licenses/license.dat*, or the license file pathname in the environment variable LM_LICENSE_FILE (B-393).

The system administrator should protect the execution of **lmdown**, since shutting down the servers will cause loss of licenses.

lmremove

The **lmremove** utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **lmremove** will allow the license to return to the pool of available licenses.

Syntax

```
lmremove
  -c <file> <feature> <user> <host> <display>
```

lmremove removes all instances of *user* on the node *host* (on the display, if specified) from usage of *feature*. If the optional **-c <file>** switch is specified, the indicated file will be used as the license file. The system administrator should protect the execution of **lmremove**, since removing a user's license can be disruptive.

lmreread

The **lmreread** utility causes the license daemon to reread the license file and start any new vendor daemons that have been added. In addition, all preexisting daemons will be signaled to reread the license file for changes in feature licensing information.

Syntax

```
lmreread [daemon]
  [-c <license_file>]
```

Note: If the **-c** option is used, the license file specified will be read by the daemon, not by **lmgrd**. **lmgrd** rereads the file it read originally. Also, **lmreread** cannot be used to change server node names or port numbers. Vendor daemons will not reread their option files as a result of **lmreread**.

Administration tools for Windows

All of the Unix administration tools listed above may be used on Windows platforms as well. However, in Windows, all of the tools are launched via the program "*lmutil.*" For example, if you want to run *lmstat*, you would type the following at a command prompt:

```
lmutil lmstat [-args]
```

The arguments for Windows are the same as those listed above for Unix.

E - Tips and Techniques

Appendix contents

This appendix contains various tips and techniques collected from several parts of the manual and from answers to questions received by tech support. Your suggestions, tips, and techniques for this section would be appreciated.

How to use checkpoint/restore

The **checkpoint** (CR-62) and **restore** (CR-172) commands will save and restore the simulator state within the same invocation of **vsim** or between **vsim** sessions.

If you want to **restore** while running **vsim**, use the **restore** command (CR-172); we call this a "warm restore". If you want to start up **vsim** and restore a previously-saved checkpoint, use the **-restore** switch with the **vsim** command (CR-258); we call this a "cold restore".

Note: Checkpoint/restore allows a cold restore, followed by simulation activity, followed by a warm restore back to the original cold-restore checkpoint file. Warm restores to checkpoint files that were not created in the current run are not allowed except for this special case of an original cold restore file.

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

- 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

Things that are NOT restored are:

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

In order to save the simulator state, use the command

```
checkpoint <filename>
```

To restore the simulator state during the same session as when the state was saved, use the command:

```
restore <filename>
```

To restore the state after quitting Model*Sim*, invoke **vsim** as follows:

```
vsim -restore <filename> [-nocompress]
```

The checkpoint file is normally compressed. If there is a need to turn off the compression, you can do so by setting a special Tcl variable. Use:

```
set CheckpointCompressMode 0
```

to turn compression off, and turn compression back on with:

```
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>
```

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 for more information.

The difference between checkpoint/restore and restarting

The **restart** (CR-170) 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. But with **restart** you don't have to save the checkpoint and the **restart** is likely to be faster. But when you need to set the state to anything other than time zero, you will need to use **checkpoint/restore**.

Using macros with restart and checkpoint/restore

The **restart** (CR-170) 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 Model*Sim*, that is, doing a **checkpoint** (CR-62) and later in the same session doing a **restore** (CR-172) 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.

Running command-line and batch-mode simulations

The typical method of running Model*Sim* is interactive: you push buttons and/or pull down menus in a series of windows in the GUI (graphic user interface). But there are really three specific modes of Model*Sim* operation: GUI, command line, and batch. Here are their characteristics:

- **GUI mode**
  This is the usual interactive mode; it has graphical windows, push-buttons, menus, and a command line in the text window. This is the default mode.
- **Command-line mode**
  This an operational mode that has only an interactive command line; no interactive windows are opened. To run **vsim** in this manner, invoke it with the **-c** option as the first argument from either the UNIX prompt or the DOS prompt in Windows 95/98/2000/NT.
- **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. Batch mode does not require the **-c** option. In a Windows environment, **vsim** is run from a Windows 95/98/2000/NT DOS prompt and standard input and output are re-directed to and from files. An example is:

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

Here is another example of batch mode, this time using a file as input:

```
vsim counter < yourfile
```

From a user's point of view, command-line mode can look like batch mode if you use the **vsim** command (CR-258) with the **-do** option to execute a macro that does a **quit -f** (CR-165) before returning, or if the startup.do macro does a **quit -f** before returning. But technically, that mode of operation is still command-line mode because stdin is still operating from the terminal.

The following paragraphs describe the behavior defined for the batch and command-line modes.

Command-line mode

In command-line mode Model*Sim* executes any startup command specified by the Startup (B-400) variable in the *modelsim.ini* file. If **vsim** (CR-258) 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-194) after the design loads (see the example below). The **transcript on** command will write all of the commands you invoke to the transcript file. For example, the following series of commands will result in a transcript file that can be used for command input if *top* is resimulated (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
```

Note: 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 "#".

Note: Stand-alone tools will 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 will 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

In batch mode Model*Sim* behaves much as in command-line mode except that there are no prompts, and commands from re-directed stdin are not echoed to stdout. Do not use the **-c** argument with **vsim** for batch mode simulations because **-c** invokes the command-line mode, which supplies the prompts and echoes the commands.

Tcl user_hook_variables may also be used for Tcl customization during batch-mode simulation; see http://www.model.com/resources/pref_variables/frameset.htm.

Using macros (DO files)

ModelSim macros (also called DO files) are scripts that contain ModelSim and, optionally, Tcl commands. You invoke DO files with the **Macro > Execute Macro** (Main window) menu selection or the **do** command (CR-104).

Creating DO files

You can create DO files by typing the required commands in any editor and saving the file. Alternatively, you can save the Main window transcript to a DO file (see "Saving the Main window transcript file" (8-159)).

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
```

You can write more complex macros using the Tcl scripting language. See *Chapter 16 - Tcl and ModelSim* for more information.

Using Parameters with DO files

You can increase the flexibility of DO files using parameters. Parameters specify values that are passed to the corresponding parameters $1 through $9 in the macro file. For example,

```
do  testfile  design.vhd  127
```

If the macro file *testfile* contains the line **bp** $1 $2, this command would place a breakpoint in the source file named *design.vhd* at line 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-183) to see the other parameters.

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, Model*Sim* 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-176) **-continue**	continue as if the breakpoint had not been executed, completes the **run** (CR-176) that was interrupted
resume (CR-173)	continue running the macro
onbreak (CR-144)	specify a command to run when you hit a breakpoint within a macro
onElabError (CR-145)	specify a command to run when an error is encountered during elaboration
onerror (CR-146)	specify a command to run when an error is encountered within a macro
status (CR-186)	get a traceback of nested macro calls when a macro is interrupted
abort (CR-25)	terminate a macro once the macro has been interrupted or paused
pause (CR-147)	cause the macro to be interrupted, the macro can be resumed by entering a **resume** command (CR-173) via the command line
transcript (CR-194)	control echoing of macro commands to the Main window transcript

Note: You can also set the OnErrorDefaultAction Tcl variable in the pref.tcl file to dictate what action Model*Sim* takes when an error occurs.

Error action in DO files

If a command in a macro returns an error, Model*Sim* does the following:

1. If an **onerror** (CR-146) command has been set in the macro script, Model*Sim* executes that command.
2. If no onerror command has been specified in the script, Model*Sim* checks the OnErrorDefaultAction Tcl variable. If the variable is defined, it 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.

See also

See the **do** command (CR-104). Also see the DOPATH (B-393) variable for adding a DO file path to your environment.

Source code security and -nodebug

The **-nodebug** option on both **vcom** (CR-217) and **vlog** (CR-250) 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: Model*Sim*'s **-nodebug** compiler option provides protection for proprietary model information. The Verilog **protect** compiler directive provides similar protection, but uses a Cadence encryption algorithm that is unavailable to Model Technology.

If a design unit is compiled with **-nodebug** the Source window will not display the design unit's source code, the Structure window will not display the internal structure, the Signals window will not display internal signals (it still displays ports), 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 Model*Sim* commands.

Even with the data hiding of **-nodebug**, there remains some visibility into models compiled with **-nodebug**. The names of all design units comprising your model are visible in the library, and you may invoke **vsim** (CR-258) directly on any of these design units and see the ports. Design units or modules compiled with **-nodebug** can only instantiate design units or modules that are also compiled **-nodebug**.

To restrict visibility into the lower levels of your design you can use the following **-nodebug** switches 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 (or =pli+ports)	combines the functions of -nodebug=ports and -nodebug=pli

Note: Don't use the **=ports** switch on a design without hierarchy, or on the top level of a hierarchical design; if you do, no ports will be visible for simulation. To properly use the switch, compile all lower portions of the design with -nodebug=ports first, then compile the top level with -nodebug alone.

Also note the =pli switch will not work with vcom (the VHDL compiler). PLI functions are valid only for Verilog design units.

Saving and viewing waveforms

You can run **vsim** as a batch job, but view the resulting waveforms later.

1. When you invoke **vsim** the first time, use the **-wlf** option to rename the logfile, and redirect stdin to invoke the batch mode. The command should look like this:

```
vsim -wlf wavesav1.wlf counter < command.do
```

   Within your *command.do* file, use the **log** command (CR-131) to save the waveforms you want to look at later, run the simulation, and quit.

   When **vsim** runs in batch mode, it does not write to the screen, and can be run in the background.

2. When you return to work the next day after running several batch jobs, you can start up **vsim** in its viewing mode with this command and the appropriate *.wlf* files:

```
vsim -view wavesav1.wlf
```

   Now you will be able to use the Waveform and List windows normally.

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 Model*Sim* 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
```

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

It is possible with Model*Sim* to maintain 64-bit and 32-bit versions of a design in the same library. To do this, you must compile the design with one of the versions (64-bit or 32-bit), and "refresh" the design with the other version. For example:

Using the 32-bit version of Model*Sim*:

```
vcom file1.vhd
vcom file2.vhd
```

Next, using the 64-bit version of Model*Sim*:

```
vcom -refresh
```

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

Bus contention checking

Bus contention checking detects bus fights on nodes that have multiple drivers. A bus fight occurs when two or more drivers drive a node with the same strength and that strength is the strongest of all drivers currently driving the node. The following table provides some examples for two drivers driving a std_logic signal:

driver 1	driver 2	fight
Z	Z	no
0	0	yes
1	Z	no
0	1	yes
L	1	no
L	H	yes

Detection of a bus fight results in an error message specifying the node and its drivers' current driving values. If a node's drivers later change value and the node is still in contention, a message is issued giving the new values of the drivers. A message is also issued when the contention ends. The bus contention checking commands can be used on VHDL and Verilog designs.

These bus checking commands are in "ModelSim Commands" (CR-9):

- **check contention add** (CR-54)
- **check contention config** (CR-55)
- **check contention off** (CR-56)

Bus float checking

Bus float checking detects nodes that are in the high impedance state for a time equal to or exceeding a user-defined limit. This is an error in some technologies. Detection of a float violation results in an error message identifying the node. A message is also issued when the float violation ends. The bus float checking commands can be used on VHDL and Verilog designs.

These bus float checking commands are in "ModelSim Commands" (CR-9):

- **check float add** (CR-57)
- **check float config** (CR-58)
- **check float off** (CR-59)

Design stability checking

Design stability checking detects when circuit activity has not settled within a period you define for synchronous designs. You specify the clock period for the design and the strobe time within the period during which the circuit must be stable. A violation is detected and an error message is issued if there are pending driver events at the strobe time. The message identifies the driver that has a pending event, the node that it drives, and the cycle number. The design stability checking commands can be used on VHDL and Verilog designs.

These design stability checking commands are in "ModelSim Commands" (CR-9):

- **check stable on** (CR-61)
- **check stable off** (CR-60)

Toggle checking

Toggle checking counts the number of transitions to 0 and 1 on specified nodes. Once the nodes have been selected, a toggle report may be requested at any time during the simulation. The toggle commands can be used on VHDL and Verilog designs.

These toggle checking commands are in "ModelSim Commands" (CR-9):

- **toggle add** (CR-190)
- **toggle reset** (CR-192)
- **toggle report** (CR-191)

Detecting infinite zero-delay loops

Simulations use steps that advance simulated time, and steps that do not advance simulated time. Steps that do not advance simulated time are called "delta cycles". Delta cycles are used when signal assignments are made with zero time delay.

If a large number of delta cycles 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, Model*Sim* defines a limit, the “iteration_limit", on the number of successive delta cycles that can occur. When the iteration_limit is exceeded, **vsim** stops the simulation and gives a warning message.

You can set the iteration_limit from the **Options > Simulation** menu, by modifying the *modelsim.ini* file, or by setting a Tcl variable called IterationLimit (B-399).

The iteration_limit default value is 5000.

When you get an iteration_limit warning, first increase the iteration limit and try to continue simulation. If the problem persists, look for zero-delay loops.

One approach to finding zero-delay loops is to increase the iteration limit again and start single stepping. You should be able to see the assignment statements or processes that are looping. Looking at the Process window will also help you to see the active looping processes.

When the loop is found, you will need to change the design to eliminate the unstable loop.

See "Projects and system initialization" (2-25) for more information on modifying the *modelsim.ini* file. And see "Preference variables located in TCL files" (B-406) for more information on Tcl variables. Also see the Main window Help menu for Tcl Help and man pages.

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.

Model*Sim* 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.

Model*Sim* tools open the location map file on invocation if the MGC_LOCATION_MAP (B-393) environment variable is set. If MGC_LOCATION_MAP is not set, Model*Sim* 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 Model*Sim* binaries
- the Model*Sim* 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, Model*Sim* expects an environment variable to be set for each logical pathname (with the same name). Model*Sim* 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, Model*Sim* 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, Model*Sim* 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 http://www.model.com/resources/pref_variables/frameset.htm.

Accelerate simulation by locking memory under HP-UX 10.2

Model*Sim* 5.3 and later versions contain a feature to allow HP-UX 10.2 to use locked memory. This feature provides significant acceleration of simulation time for large designs – i.e. with a memory footprint > 500Mb. (Test cases showed 2x acceleration of large simulations.) The following steps show how to set up the HP-UX 10.2 so memory can be locked.

1. Allow the average-user to lock memory. By default, this privilege is not allowed, so it has to be enabled. To allow everyone MLOCK privileges, the administrator needs to execute this command on the machine that will be running Model*Sim*:

   ```
   /usr/sbin/setprivgrp -g MLOCK
   ```

   To only allow a particular group MLOCK privileges, use the command:

   ```
   /usr/sbin/setprivgrp <group-name> MLOCK
   ```

   This allows you to lock memory. No other privileges are enabled.

2. Once the MLOCK privilege is enabled, you merely have to modify the modelsim.ini file, and add the following entry to the [vsim] section:

   ```
   LockedMemory = <some-value>
   ```

   Where <some-value> is an integer representing the number of megabytes of memory to be locked. Once this is done, the memory will be locked when vsim invokes (using this .ini file).

   Model*Sim* will not lock more memory than is available in the system. The maximum memory that can be locked is: system physical memory (RAM) - 100 Mb = locked memory

   When Model*Sim* locks memory, other processes will not have access to it. Therefore, you should consider how much memory is locked on a per-design basis to avoid locking more than is needed.

   System parameters used for shared/locked memory may not be set (by default) high enough to take full advantage of this feature in later generations of HP-UX. Using the "sam" program, go to the "Configurable Parameters" window (under "Kernel Configuration"). There are several values that may need to be increased.

   First, enable shared memory. The value for "shmem" should be equal to 1. Set the value for "shmmax" as large as possible. The defaults for the values of "shmmin" and "shmseg" should be ok. To change these parameters, you have to rebuild the kernel and reboot.

Modeling memory in VHDL

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 more 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.

A simple alternative implementation 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

The trick is to model memory using variables instead of signals.

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".

```
use std.standard.all;
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;
                 data_out <= ram(address);
            else
                 data_out <= ram(address);
            end if;
        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;
                 data_out <= ram(address);
            else
                 data_out <= ram(address);
            end if;
        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;

------------------------------------------------------------
------------------------------------------------------------
use std.standard.all;
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;
```

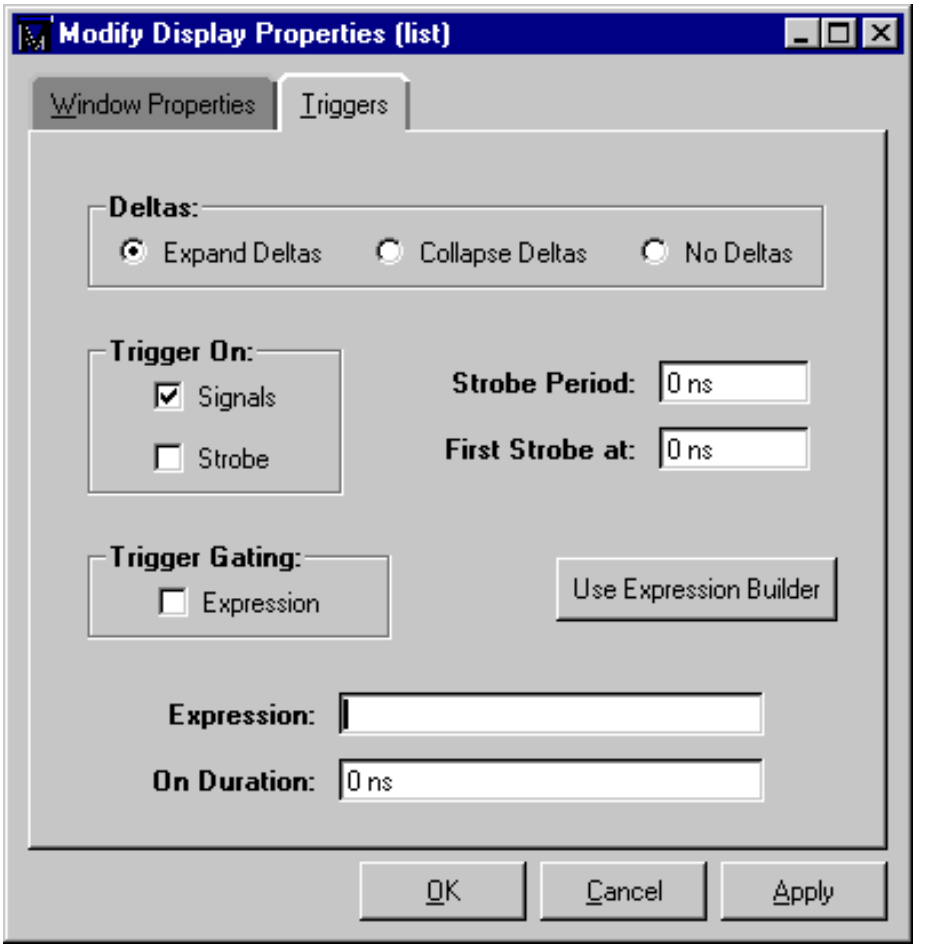
```
    begin
        for i in x'reverse_range loop
            if (tempn mod 2) = 1 then
                x(i) := '1';
            end if;
            tempn := tempn / 2;
        end loop;
        return x;
    end natural_to_sulv;

end conversions;
```

Setting up a List trigger with Expression Builder

This example shows you how to set a List window trigger based on a gating expression created with the Model*Sim* Expression Builder.

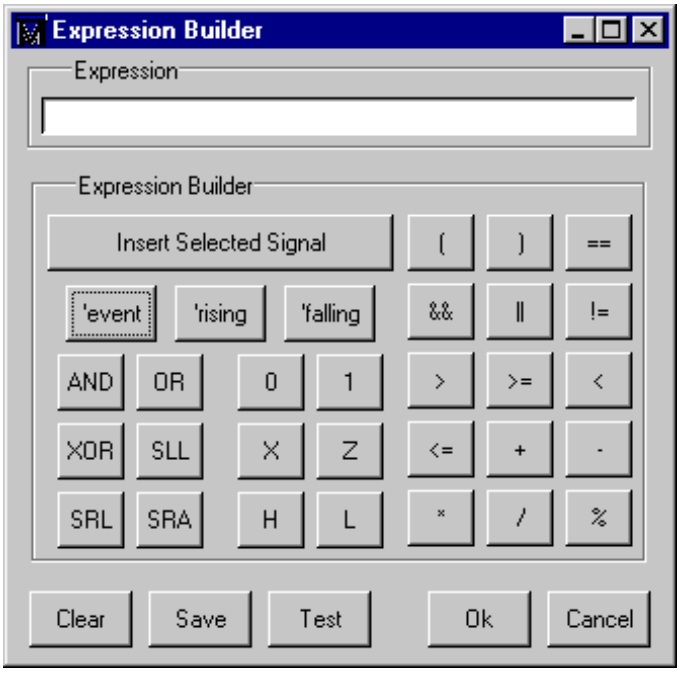
If you want to look at a set of signal values ONLY during the simulation cycles during which an enable signal rises, you would need to use the List window Trigger Gating feature. The gating feature suppresses all display lines except those for which a specified gating function evaluates to true.

Select **Prop > Display Props** (List window) to access the Triggers page.



Check the **Trigger Gating: Expression** check box. Then click on **Use Expression Builder**. 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. Then click **Insert Selected Signal** and **'rising** in the Expression Builder.



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. (Leave the **On Duration** field zero for now.) Click the **OK** button.

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**. Otherwise, leave it at zero, and select **Apply** again. When everything is correct, click **OK** to close the Modify Display Properties dialog box.

When you save the List window configuration, the list gating parameters will be saved as well, and can be set up again by reading in that macro. You can take a look at the macro to see how the gating can be set up using macro commands.

F - What's new in Model*Sim*

Appendix contents

Model*Sim* 5.5 includes many new features and enhancements that are described in the tables below. Links within the groups will connect you to more detail. GUI changes are described toward the end of the appendix.

New features

What	Description	Where (select a link)	Model*Sim* release
waveform comparison	compare simulations and datasets	*Chapter 11 - Waveform Comparison*	5.5
Model*Sim* projects	projects have been completely revamped to ease getting started with Model*Sim*	Projects and system initialization (2-25)	5.5
gate-level optimizations	gate-level Verilog designs can now be optimized using -fast	Compiling for faster performance (5-90)	5.5
VCD file enhancements	support multiple VCD files and dumpports tasks	ModelSim VCD commands and VCD tasks (13-342)	5.5
enhanced Code Coverage feature	new interface and ability to exclude files and lines	*Chapter 10 - Code Coverage*	5.5
vcd2wlf	new utility converts VCD files to WLF files	vcd2wlf (CR-216)	5.5
bookmarks	save zoom and scroll settings in Wave window	Saving zoom range and scroll position with bookmarks (8-241)	5.5
Workspace	new Main window eases working with design units and datasets	Workspace (8-158)	5.5
find and replace in Source window	Source window now supports search and replace for text and regular expressions	Finding and replacing in the Source window (8-208)	5.5
breakpoints dialog	manage breakpoints via dialog boxes	Setting signal breakpoints (8-236)	5.5
import library wizard	imports FPGA libraries	Importing FPGA libraries (3-53)	5.5

Command and variable changes

What	Description	Where (select a link)	ModelSim release
-compile_uselibs argument for vlog	eases use of **'uselib** directives	-compile_uselibs argument (5-82)	5.5
-lint argument for vlog	enables lint-style checks	`-lint` (CR-252)	5.5
middle mouse button pasting control	enables/disables middle mouse button pasting	Middle Mouse Button Paste (8-203)	5.5
init_signal_spy utility	reference signals, registers, or wires at any level of hierarchy	init_signal_spy() (4-69) and $init_signal_spy (5-104)	5.5
get_resolution function	returns the current simulator resolution as a real	get_resolution() (4-68)	5.5
to_real function	converts the physical type time to the type real	to_real() (4-70)	5.5
to_time function	converts the type real to the physical type time	to_time() (4-71)	5.5
compare commands	several commands for doing waveform comparisons	Compare commands (11-323)	5.5
bookmark commands	several commands for saving/ editing bookmarks	bookmark add wave (CR-44)	5.5
PrefCompare Tcl variables	Tcl preference variables for waveform comparisons	Preference variable database	5.5
-delay argument for virtual signal and virtual function	assign delay to signals within a virtual command	virtual function (CR-233) & virtual signal (CR-245)	5.5
-keeploaded and -keeploadedrestart arguments for vsim	leaves FLI/PLI/VPI shared libraries loaded during a restart or design load	`-keeploaded` (CR-260) and `-keeploadedrestart` (CR-260)	5.5
vsim arguments related to WLF files	four arguments control WLF file creation	`-wlf <filename>` (CR-263), `-wlfslim <size>` (CR-263), `-wlftlim` <duration> (CR-263), and `-wlfnocompress` (CR-264)	5.5
delay in GUI_expression_format	assign delay to signals in a GUI_expression	Signal attributes (CR-303)	5.5
acc_fetch_paramval_str() function in PLI	allows fetching of a string on 64-bit platforms	64-bit support in the PLI (5-125)	5.5
WLF file control variables	new vsim control variables configure WLF file creation	Setting default simulation options (8-265)	5.5

Documentation changes

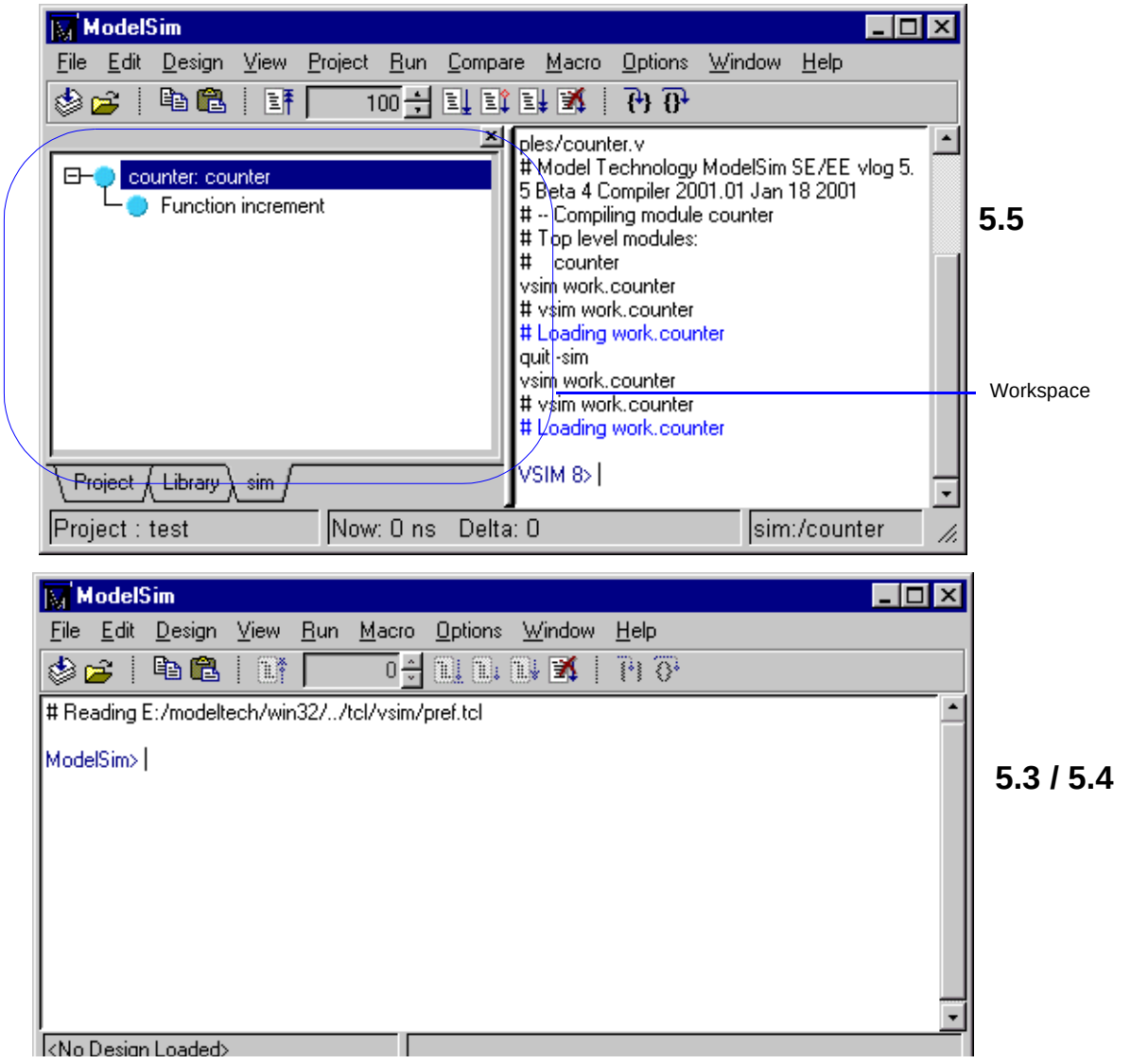
What	Description	Where (select a link)	Model*Sim* release
New Foreign Language Interface Reference manual	new manual provides detailed documentation of FLI including code examples	FLI Reference Manual	5.5
FLI chapter has been eliminated	replaced by FLI reference manual		5.5
new chapter on waveform comparison	describes new waveform comparison feature	*Chapter 11 - Waveform Comparison*	5.5
new tutorial on waveform comparison	practice using the new waveform comparison feature	*ModelSim Tutorial*	5.5

GUI changes in version 5.5

This section identifies differences between the version 5.3/5.4 GUI and the 5.5 GUI.

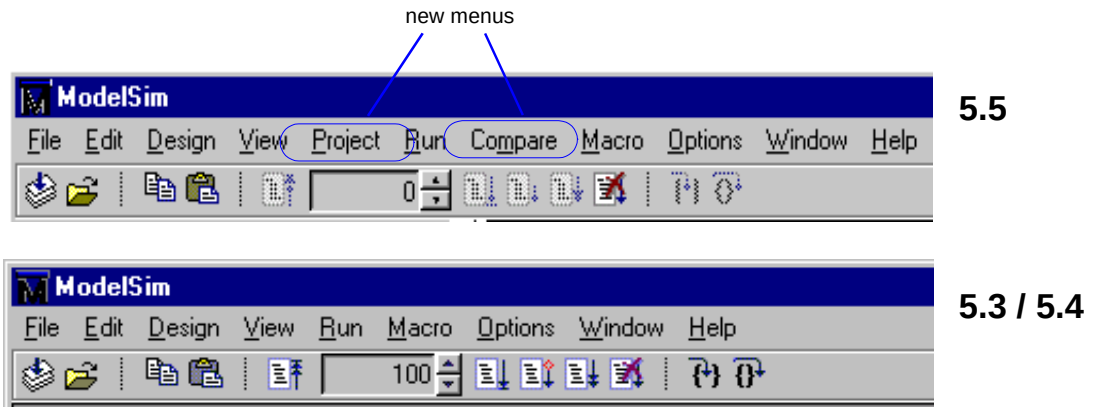
Main window changes

The most obvious change in the version 5.5 Main window is the addition of the workspace. See "Workspace" (8-158) for full details.



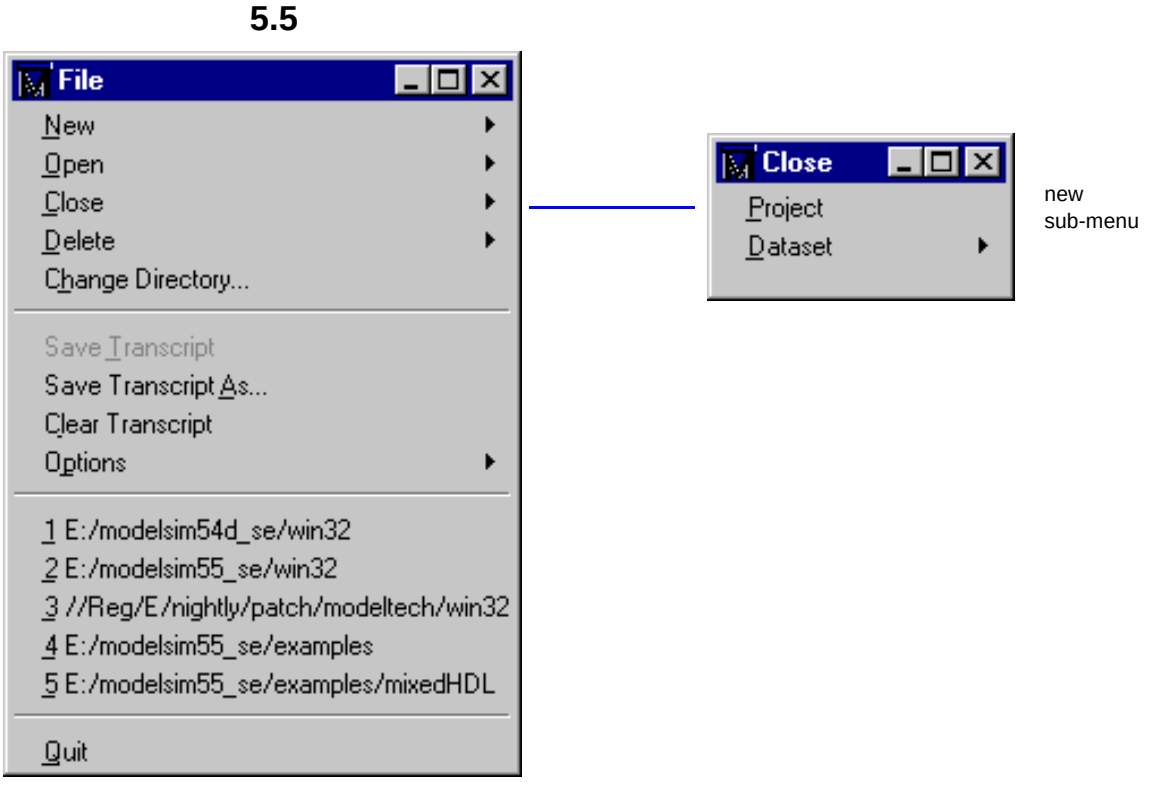
Menu bar and toolbar

The Main window toolbar in version 5.5 has not changed from version 5.3 / 5.4. The Main window menu bar has new Project and Compare menus. See the following pages for additional menu changes.

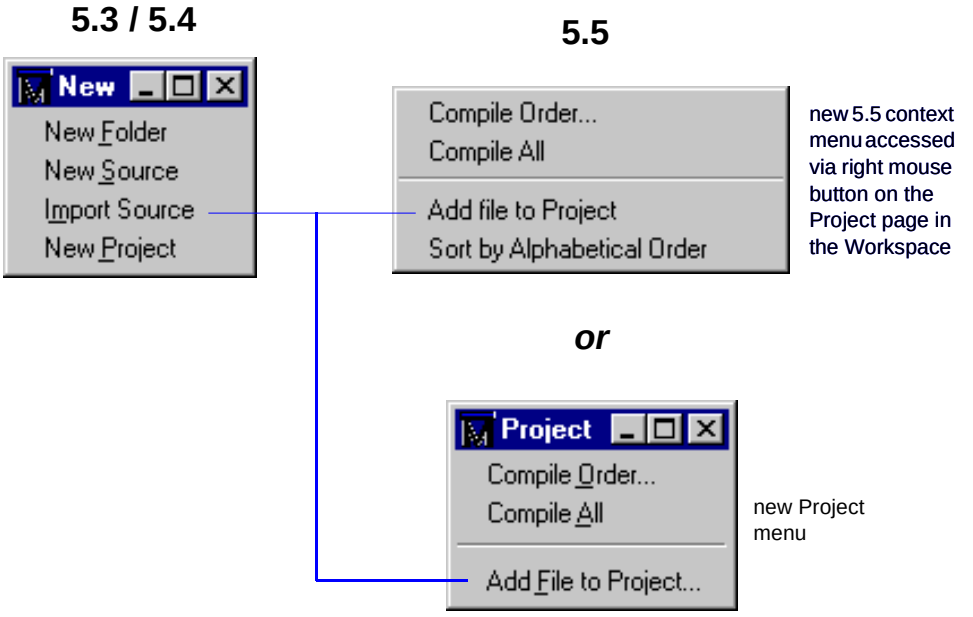


File menu

As shown below, the version 5.5 Main window File menu contains two additions. See "The Main window menu bar" (8-160) for complete menu option details.

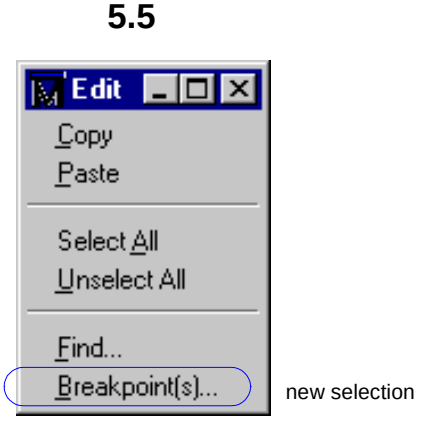


The graphic below shows the new menu command for importing (adding) a source file to a project. See "Step 2 — Add files to the project" (2-31) for details.



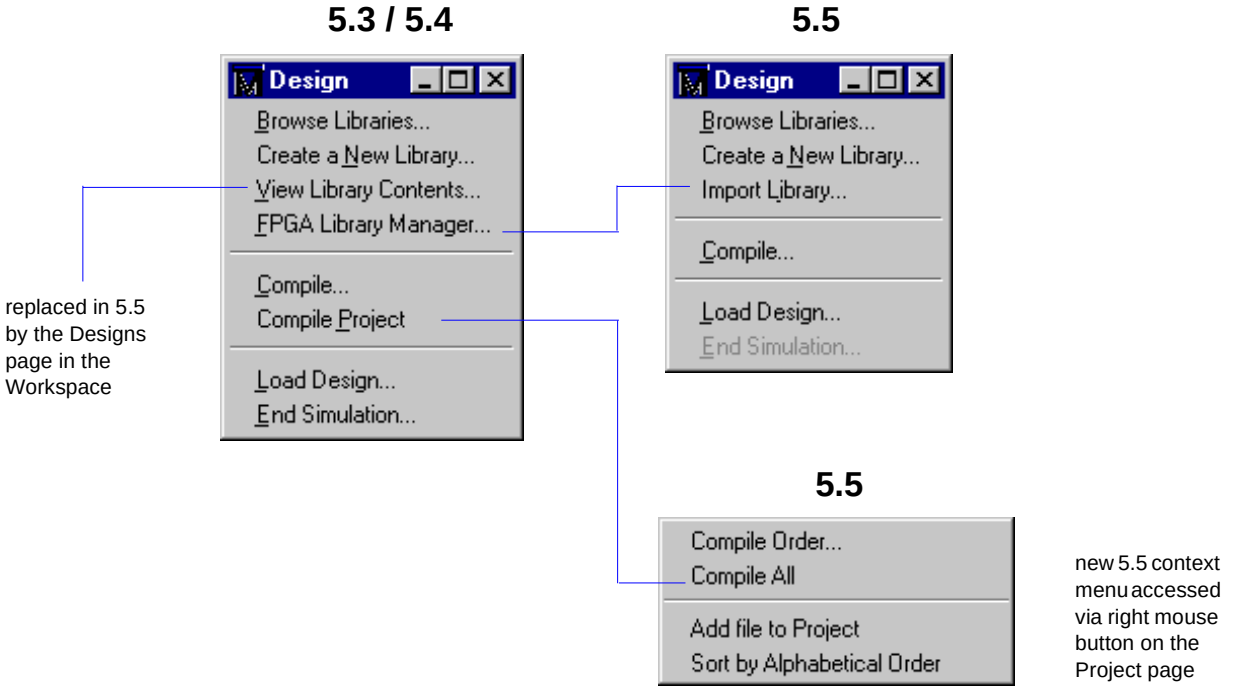
Edit menu

See "The Main window menu bar" (8-160) for complete menu option details.



Design menu

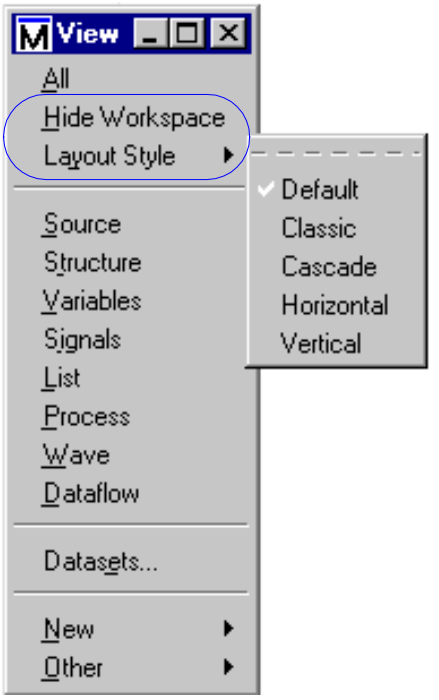
See "The Main window menu bar" (8-160) for complete menu option details.



View menu

See "The Main window menu bar" (8-160) for complete menu option details.

5.5

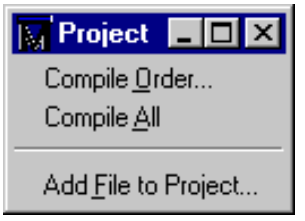


new selections and sub-menu

Project menu

The Project menu is new in version 5.5. See "What are projects?" (2-26) for details.

5.5

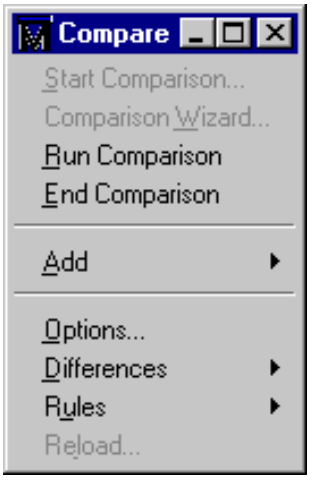


new menu

Compare menu

The Compare menu is new in version 5.5. See *Chapter 11 - Waveform Comparison* for details on waveform comparisons. See also "The Main window menu bar" (8-160) for complete menu option details..

5.5

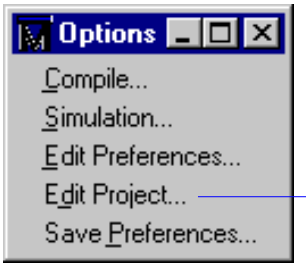


new menu

Options menu

See "The Main window menu bar" (8-160) for complete menu option details. See also "What are projects?" (2-26) for details on Project operations.

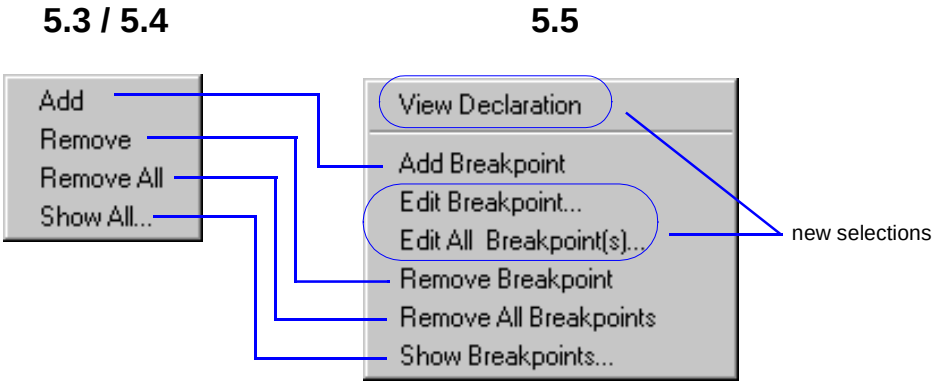
5.3 / 5.4



no equivalent in 5.5; all Project editing is done from the Project page in the Workspace

Signals window changes

The menus accessed from the Signals menu bar are the same in version 5.5 as they were in version 5.3 / 5.4. However, the context menu (accessed with a right mouse click in the Signals window) has changed. See "Setting signal breakpoints" (8-198) for complete details on this context menu.



Source window changes

Edit menu

See "The Source window menu bar" (8-202) for complete menu option details.

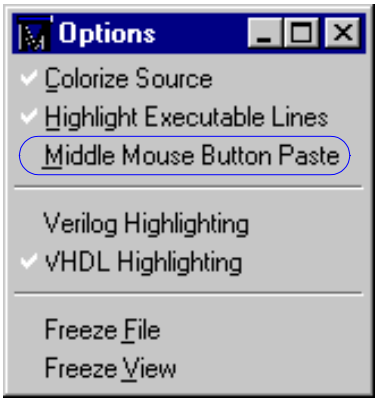
5.5

new selection

Options menu

See "The Structure window menu bar" (8-211) for complete menu option details.

5.5

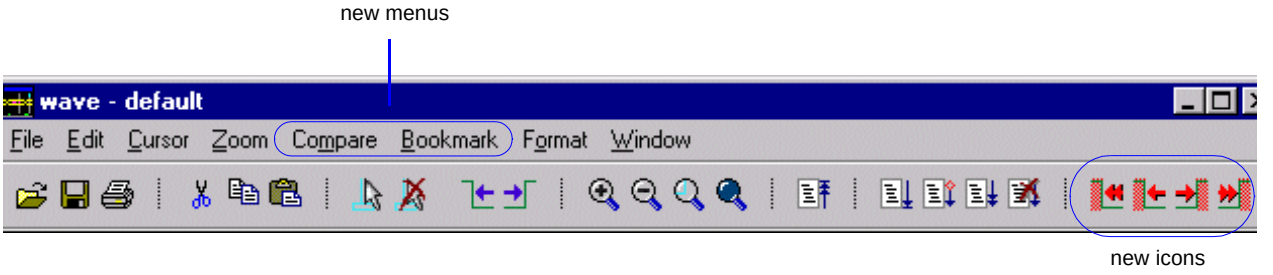


new selection

Wave window changes

Menu bar and toolbar

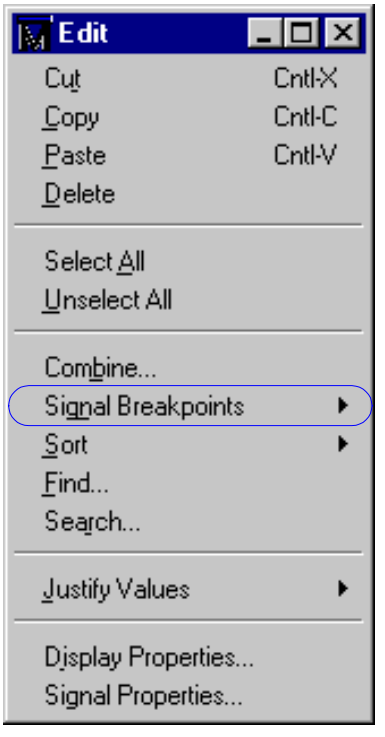
The version 5.5 Wave window menu bar has two new menus, and the toolbar has four new icons. See "The Wave window menu bar" (8-220) for complete menu and toolbar option details.



Edit menu

See "The Wave window menu bar" (8-220) for complete menu option details.

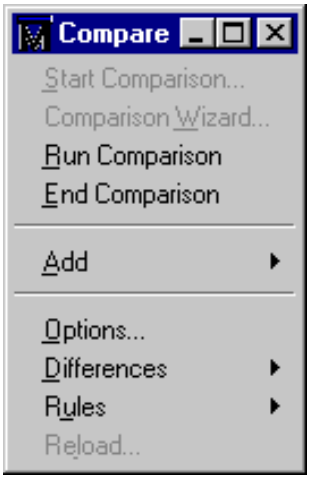
5.5



Compare menu

The Compare menu is new in version 5.5. See *Chapter 11 - Waveform Comparison* for details on waveform comparisons. See also "The Wave window menu bar" (8-220) for complete menu option details.

5.5



new menu

Bookmark menu

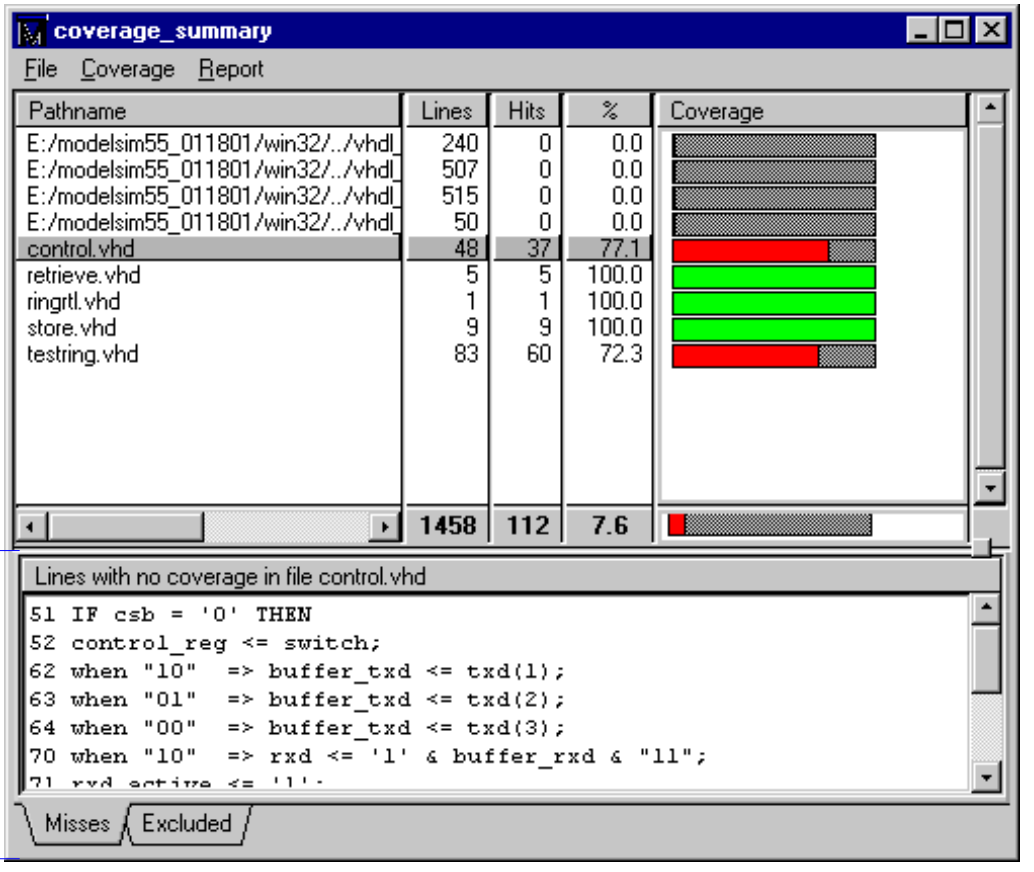
The Bookmark menu is new in version 5.5. See "Saving zoom range and scroll position with bookmarks" (8-241) for details on bookmarks. See also "The Wave window menu bar" (8-220) for complete menu option details.

5.5

new menu

Coverage_summary window changes

The coverage_summary window has been enhanced to show line misses and exclusions below the summary information.



new half of window shows line misses and exclusions

License Agreement

IMPORTANT – USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS

CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE

This license is a legal “Agreement” concerning the use of Software between you, the end user, either individually or as an authorized representative of the company purchasing the license, and Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Mentor Graphics (Singapore) Private Limited, and their majority-owned subsidiaries (“Mentor Graphics”). USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within 10 days after receipt of Software and receive a full refund of any license fee paid.

END USER LICENSE AGREEMENT

1. GRANT OF LICENSE. The software programs you are installing, downloading, or have acquired with this Agreement, including any updates, modifications, revisions, copies, and documentation (“Software”) are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics or its authorized distributor grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; and (c) on the computer hardware or at the site for which an applicable license fee is paid, or as authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics’ then-current standard policies, which vary depending on Software, license fees paid or service plan purchased, apply to the following and are subject to change: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be communicated and technically implemented through the use of authorization codes or similar devices); (c) eligibility to receive updates, modifications, and revisions; and (d) support services provided. Current standard policies are available upon request.

2. ESD SOFTWARE. If you purchased a license to use embedded software development (“ESD”) Software, Mentor Graphics or its authorized distributor grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate or incorporate copies of Mentor Graphics' real-time operating systems or other ESD Software, except those explicitly granted in this section, into your products without first signing a separate agreement with Mentor Graphics for such purpose.

3. BETA CODE.

3.1 Portions or all of certain Software may contain code for experimental testing and evaluation (“Beta Code”), which may not be used without Mentor Graphics’ explicit authorization. Upon Mentor Graphics’ authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

3.2 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.

3.3 You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceives or makes 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 your employer's 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 the European Union Software Directive or 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 without Mentor Graphics’ prior written consent. The provisions of this section 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 LOANED TO YOU FOR A LIMITED TERM OR 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 OR IMPLIED, 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.

6. LIMITATION OF LIABILITY. EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE STATUTE OR REGULATION, 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. 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 SUCH USE.

8. INFRINGEMENT.

8.1 Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright in the United States, Canada, Japan, Switzerland, Norway, Israel, Egypt, or the European Union. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the claim, provided that you: (a) notify Mentor

Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the claim; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the claim.

8.2 If an infringement claim is made, Mentor Graphics may, at its option and expense, either (a) replace or modify Software so that it becomes noninfringing, or (b) procure for you the right to continue using Software. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

8.3 Mentor Graphics has no liability to you if the alleged 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 design or market; (f) any Beta Code contained in Software; or (g) any Software provided by Mentor Graphics' licensors which do not provide such indemnification to Mentor Graphics' customers.

8.4 THIS SECTION 8 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 BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

9. 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.

10. 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 in any manner any Software or direct product of Software, without first obtaining all necessary approval from appropriate local and United States government agencies.

11. RESTRICTED RIGHTS NOTICE. Software has been 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 Boeckman Road,

Wilsonville, Oregon 97070-7777 USA.

12. 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.

13. CONTROLLING LAW. This Agreement shall be governed by and construed under the laws of Ireland if the Software is licensed for use in Israel, Egypt, Switzerland, Norway, South Africa, or the European Union, the laws of Japan if the Software is licensed for use in Japan, the laws of Singapore if the Software is licensed for use in Singapore, People's Republic of China, Republic of China, India, or Korea, and the laws of the state of Oregon if the Software is licensed for use in the United States of America, Canada, Mexico, South America or anywhere else worldwide not provided for in this section.

14. 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.

15. MISCELLANEOUS. This Agreement contains the entire understanding between the parties 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. This Agreement may only be modified by a physically signed writing between you and an authorized agent of Mentor Graphics. 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. 03/00

Index

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z