



ModelSim® SE User's Manual

Software Version 10.1

**© 1991-2011 Mentor Graphics Corporation
All rights reserved.**

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

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

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

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

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

RESTRICTED RIGHTS LEGEND 03/97

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

Contractor/manufacturer is:

Mentor Graphics Corporation

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

Telephone: 503.685.7000

Toll-Free Telephone: 800.592.2210

Website: www.mentor.com

SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

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

Table of Contents

Chapter 1

Introduction	57
Operational Structure and Flow.....	57
Simulation Task Overview.....	58
Basic Steps for Simulation.....	60
Step 1 — Collect Files and Map Libraries.....	60
Step 2 — Compile the Design.....	62
Step 3 — Load the Design for Simulation.....	63
Step 4 — Simulate the Design.....	63
Step 5 — Debug the Design.....	63
Modes of Operation.....	64
Command Line Mode.....	64
Batch Mode.....	66
Definition of an Object.....	66
Graphic Interface Overview.....	67
Standards Supported.....	67
Assumptions.....	68
Text Conventions.....	69
Installation Directory Pathnames.....	69
Where to Find ModelSim Documentation.....	69
Mentor Graphics Support.....	70
Deprecated Features, Commands, and Variables.....	71

Chapter 2

Graphical User Interface	73
Design Object Icons and Their Meaning.....	76
Setting Fonts.....	77
Using the Find and Filter Functions.....	78
Using the Find Options Popup Menu.....	81
User-Defined Radices.....	81
Using the radix define Command.....	82
Saving and Reloading Formats and Content.....	86
Active Time Label.....	86
Window Specific Keyboard Shortcuts.....	87
User Defined Keyboard Shortcuts.....	88
The Keyboard Shortcuts Dialog Box.....	89
Bookmarks.....	91
Working with Bookmarks.....	91
Managing Your Bookmarks.....	92
Main Window.....	95
Elements of the Main Window.....	95
Selecting the Active Window.....	102

Rearranging the Main Window	102
Navigating in the Main Window	104
Main Window Menu Bar	104
Main Window Toolbar	114
Assertions Window	140
Assertions Window Tasks	140
GUI Elements of the Assertions Window	140
ATV Window	145
ATV Window Tasks	146
GUI Elements of the ATV Window	146
Call Stack Window	148
Call Stack Window Tasks	149
Related Commands of the Call Stack Window	150
GUI Elements of the Call Stack Window	150
Capacity Window	150
GUI Elements of the Capacity Window	151
Class Graph Window	153
Class Graph Window Tasks	153
GUI Elements of the Class Graph Window	154
Class Instances Window	155
GUI Elements of the Class Instances Window	156
Class Tree Window	156
GUI Elements of the Class Tree Window	157
Code Coverage Analysis Window	158
Viewing Code Coverage Data and Current Exclusions	160
Coverage Details Window	162
Cover Directives Window	166
GUI Elements of the Cover Directives Window	167
Cover Directives Window Tasks	168
Covergroups Window	169
GUI Elements of the Covergroups Window	170
Covergroups Window Tasks	171
Dataflow Window	172
Dataflow Window Tasks	174
Files Window	176
GUI Elements of the Files Window	177
FSM List Window	179
GUI Elements of the FSM List Window	180
FSM Viewer Window	181
FSM Viewer Window Tasks	182
GUI Elements of the FSM Viewer Window	186
Instance Coverage Window	188
Instance Coverage Window Tasks	189
GUI Elements of the Instance Coverage Window	190
Library Window	194
GUI Elements of the Library Window	195
List Window	196
List Window Tasks	197
GUI Elements of the List Window	198

Table of Contents

Locals Window	200
Locals Window Tasks.	201
GUI Elements of the Locals Window.	201
Memory List Window	203
Memory List Window Tasks	205
GUI Elements of the Memory List Window.	206
Memory Data Window	207
Memory Data Window Tasks.	208
GUI Elements of the Memory Data Window	209
Message Viewer Window	210
Message Viewer Window Tasks.	212
GUI Elements of the Message Viewer Window.	213
Objects Window.	217
Objects Window Tasks	218
GUI Elements of the Objects Window	221
Processes Window	223
Processes Window Tasks	223
GUI Elements of the Processes Window	226
Profiling Windows.	228
GUI Elements of the Profile Windows	231
Schematic Window	232
Schematic Window Tasks.	234
GUI Elements of the Schematic Window.	239
Source Window	242
Opening Source Files	244
Displaying Multiple Source Files	244
Dragging and Dropping Objects into the Wave and List Windows	245
Setting your Context by Navigating Source Files.	245
Coverage Data in the Source Window	247
Debugging with Source Annotation	250
Accessing Textual Connectivity Information.	252
Using Language Templates.	253
Setting File-Line Breakpoints with the GUI.	256
Adding File-Line Breakpoints with the bp Command	257
Editing File-Line Breakpoints.	258
Setting Conditional Breakpoints.	260
Checking Object Values and Descriptions	262
Marking Lines with Bookmarks	263
Performing Incremental Search for Specific Code	263
Customizing the Source Window	264
Structure Window	265
Viewing the Structure Window	266
Structure Window Tasks.	267
GUI Elements of the Structure Window.	269
Code Coverage in the Structure Window	274
Verification Management Browser Window	274
Controlling the Verification Browser Columns	275
Saving Verification Browser Column and Filter Settings	276
GUI Elements of the Verification Browser Window	276

Verification Results Analysis Window	278
Verification Test Analysis Window.....	278
Verification Tracker Window	279
Verification Trender Window	279
Transaction View Window	279
Transcript Window	279
Displaying the Transcript Window.....	279
Viewing Data in the Transcript Window	279
Saving the Transcript File.....	280
Colorizing the Transcript	280
Disabling Creation of the Transcript File.....	281
Performing an Incremental Search	281
Using Automatic Command Help.....	282
Using Transcript Menu Items	282
Watch Window	283
Adding Objects to the Watch Window.....	285
Expanding Objects to Show Individual Bits.....	285
Grouping and Ungrouping Objects.....	286
Saving and Reloading Format Files	287
Wave Window	287
Add Objects to the Wave Window	288
Wave Window Panes	289
Objects You Can View in the Wave Window	296
Wave Window Toolbar.....	298
Chapter 3	
Protecting Your Source Code	299
Creating Encryption Envelopes	299
Configuring the Encryption Envelope	300
Protection Expressions	302
Using the `include Compiler Directive (Verilog only).....	304
Compiling with +protect	307
The Runtime Encryption Model	308
Language-Specific Usage Models	309
Usage Models for Protecting Verilog Source Code	309
Usage Models for Protecting VHDL Source Code.....	314
Proprietary Source Code Encryption Tools.....	321
Using Proprietary Compiler Directives	322
Protecting Source Code Using -nodebug	323
Encryption Reference.....	324
Encryption and Encoding Methods.....	325
How Encryption Envelopes Work	326
Using Public Encryption Keys	327
Using the Mentor Graphics Public Encryption Key	327
.....	329

Chapter 4

Optimizing Designs with vopt 331

- Optimization Flows 331
 - Three-Step Flow 331
 - Two-Step Flow 334
 - Using vopt and the -O Optimization Control Switches 335
 - Inlining and the Implications of Coverage Settings 336
- Optimizing Parameters and Generics. 336
- Preoptimizing Regions of Your Design. 337
 - Simulating Designs with Several Different Test Benches 338
 - Using Configurations with Preoptimized Design Units 339
 - Resolving Preoptimized Design Unit Loading Errors 342
- Specifying Coverage During Optimization 343
- Alternate Optimization Flows 343
 - Creating Locked Libraries for Multiple-User Simulation Environments 344
 - Optimizing Liberty Cell Libraries for Debugging 345
 - Creating an Environment for Optimized and Unoptimized Flows 345
- Preserving Design Visibility with the Learn Flow 346
 - Description of Learn Flow Control Files 348
- Controlling Optimization from the GUI 348
- Optimization Considerations for Verilog Designs. 348
 - Design Object Visibility for Designs with PLI. 349
 - Performing Optimization on Designs Containing SDF 350
 - Reporting on Gate-Level Optimizations. 351
 - Using Pre-Compiled Libraries 351
 - Event Order and Optimized Designs 351
 - Timing Checks in Optimized Designs 351

Chapter 5

Projects..... 353

- What are Projects? 353
 - What are the Benefits of Projects? 353
 - Project Conversion Between Versions 354
- Getting Started with Projects 354
 - Step 1 — Creating a New Project 355
 - Step 2 — Adding Items to the Project 356
 - Step 3 — Compiling the Files. 357
 - Step 4 — Simulating a Design 360
 - Other Basic Project Operations. 362
- The Project Window 362
 - Sorting the List 363
- Creating a Simulation Configuration. 363
 - Optimization Configurations 365
- Organizing Projects with Folders. 365
 - Adding a Folder 365
- Specifying File Properties and Project Settings. 367
 - File Compilation Properties 367
 - Project Settings 369

Accessing Projects from the Command Line	370
Chapter 6	
Design Libraries	371
Design Library Overview	371
Design Unit Information	371
Working Library Versus Resource Libraries	371
Archives	372
Working with Design Libraries	372
Creating a Library	373
Managing Library Contents	373
Assigning a Logical Name to a Design Library	375
Moving a Library	376
Setting Up Libraries for Group Use	377
Specifying Resource Libraries	377
Verilog Resource Libraries	377
VHDL Resource Libraries	377
Predefined Libraries	378
Alternate IEEE Libraries Supplied	378
Regenerating Your Design Libraries	379
Maintaining 32- and 64-bit Versions in the Same Library	379
Importing FPGA Libraries	380
Protecting Source Code	381
Chapter 7	
VHDL Simulation	383
Basic VHDL Usage	383
Compilation and Simulation of VHDL	383
Creating a Design Library for VHDL	383
Compiling a VHDL Design—the vcom Command	384
Simulating a VHDL Design	388
Naming Behavior of VHDL For Generate Blocks	389
Differences Between Versions of VHDL	390
Foreign Language Interface	393
Simulator Resolution Limit for VHDL	393
Default Binding	394
Delta Delays	395
Using the TextIO Package	397
Syntax for File Declaration	398
Using STD_INPUT and STD_OUTPUT Within ModelSim	398
TextIO Implementation Issues	399
Writing Strings and Aggregates	399
Reading and Writing Hexadecimal Numbers	400
Dangling Pointers	400
The ENDLIN Function	400
The ENDFILE Function	401
Using Alternative Input/Output Files	401
Flushing the TEXTIO Buffer	401

Table of Contents

Providing Stimulus	401
VITAL Usage and Compliance	402
VITAL Source Code	402
VITAL 1995 and 2000 Packages	402
VITAL Compliance	403
Compiling and Simulating with Accelerated VITAL Packages	404
Compiler Options for VITAL Optimization	404
VHDL Utilities Package (util)	405
get_resolution	405
init_signal_driver()	406
init_signal_spy()	406
signal_force()	406
signal_release()	406
to_real()	406
to_time()	407
Modeling Memory	408
Examples of Different Memory Models	409
Affecting Performance by Cancelling Scheduled Events	418

Chapter 8

Verilog and SystemVerilog Simulation	419
Standards, Nomenclature, and Conventions	420
Alternative One-Step Flow	421
Basic Verilog Usage	421
Verilog Compilation	422
Creating a Working Library	422
Invoking the Verilog Compiler	422
Initializing enum Variables	425
Incremental Compilation	426
Library Usage	428
SystemVerilog Multi-File Compilation	430
Verilog-XL Compatible Compiler Arguments	431
Verilog-XL uselib Compiler Directive	432
Verilog Configurations	434
Verilog Generate Statements	435
Initializing Registers and Memories	436
Verilog Simulation	439
Simulator Resolution Limit (Verilog)	439
Event Ordering in Verilog Designs	441
Debugging Event Order Issues	444
Debugging Signal Segmentation Violations	446
Negative Timing Checks	448
Force and Release Statements in Verilog	457
Verilog-XL Compatible Simulator Arguments	457
Using Escaped Identifiers	457
Cell Libraries	459
SDF Timing Annotation	459
Delay Modes	459

Approximating Metastability	460
System Tasks and Functions	461
IEEE Std 1364 System Tasks and Functions	462
SystemVerilog System Tasks and Functions	464
Simulator-Specific System Tasks and Functions	466
Verilog-XL Compatible System Tasks and Functions	471
Compiler Directives	474
IEEE Std 1364 Compiler Directives	475
Compiler Directives for vlog	475
Verilog-XL Compatible Compiler Directives	477
Sparse Memory Modeling	478
Manually Marking Sparse Memories	479
Automatically Enabling Sparse Memories	479
Combining Automatic and Manual Modes	479
Priority of Sparse Memories	479
Determining Which Memories Were Implemented as Sparse	480
Verilog PLI/VPI and SystemVerilog DPI	481
Standards, Nomenclature, and Conventions	481
Extensions to SystemVerilog DPI	481
OVM-Aware Debug	482
Preparing Your Simulation for OVM-Aware Debug	482
OVM-Aware Debugging Tasks	483
OVM-Aware Debug Windows	484
UVM-Aware Debug	485
Preparing Your Simulation for UVM-Aware Debug	486
Simulating With UVM-Aware Debug Enabled	486
Chapter 9	
SystemC Simulation	489
Supported Platforms and Compiler Versions	489
Building gcc with Custom Configuration Options	490
Usage Flow for SystemC-Only Designs	491
Recommendations for using sc_main at the Top Level	491
Creating Shared Object Files for SystemC Code	495
Binding to Verilog or SystemVerilog Designs	496
Limitations of Bind Support for SystemC	496
Distributing SystemC IP	496
Compiling SystemC Files	497
Creating a Design Library for SystemC	497
Invoking the SystemC Compiler	498
Compiling Optimized and/or Debug Code	498
Specifying an Alternate g++ Installation	499
Maintaining Portability Between OSCI and the Simulator	499
Using sccom in Addition to the Raw C++ Compiler	500
Compiling Changed Files Only (Incremental Compilation)	500
Issues with C++ Templates	502
Linking the Compiled Source	511
Simulating SystemC Designs	511

Table of Contents

Loading the Design	511
Running Simulation	512
SystemC Time Unit and Simulator Resolution	512
Initialization and Cleanup of SystemC State-Based Code	514
Debugging the Design	515
Viewable SystemC Types	515
Viewable SystemC Objects	516
Waveform Compare with SystemC	517
Debugging Source-Level Code	517
SystemC Object and Type Display	521
Support for Globals and Statics	521
Support for Aggregates	522
SystemC Dynamic Module Array	523
Viewing FIFOs	523
Viewing SystemC Memories	524
Properly Recognizing Derived Module Class Pointers	525
Custom Debugging of SystemC Channels and Variables	526
Modifying SystemC Source Code	531
Code Modification Examples	532
Differences Between the Simulator and OSCI	534
Fixed-Point Types	535
Algorithmic C Datatype Support	536
Support for cin	536
OSCI 2.2 Feature Implementation Details	537
Support for OSCI TLM Library	537
Phase Callback	537
Accessing Command-Line Arguments	537
sc_stop Behavior	538
Construction Parameters for SystemC Types	538
Troubleshooting SystemC Errors	540
Unexplained Behaviors During Loading or Runtime	540
Errors During Loading	540
Chapter 10	
Mixed-Language Simulation	545
Basic Mixed-Language Flow	545
Separate Compilers with Common Design Libraries	546
Using Hierarchical References	546
Using SystemVerilog bind Construct in Mixed-Language Designs	548
Syntax of bind Statement	549
What Can Be Bound	549
Hierarchical References to a VHDL Object from a Verilog/SystemVerilog Scope	550
Mapping of Types	551
Using SV Bind With or Without vopt	551
Binding to VHDL Enumerated Types	552
Binding to a VHDL Instance	554
Limitations to Bind Support for SystemC	558
Optimizing Mixed Designs	558

Simulator Resolution Limit	558
Runtime Modeling Semantics	559
Hierarchical References to SystemVerilog	559
Hierarchical References In Mixed HDL and SystemC Designs	559
Signal Connections Between Mixed HDL and SystemC Designs	561
Mapping Data Types	562
Verilog and SystemVerilog to VHDL Mappings	562
VHDL To Verilog and SystemVerilog Mappings	567
Verilog or SystemVerilog and SystemC Signal Interaction And Mappings	574
VHDL and SystemC Signal Interaction And Mappings	582
VHDL Instantiating Verilog or SystemVerilog	588
Verilog/SystemVerilog Instantiation Criteria Within VHDL	588
Component Declaration for VHDL Instantiating Verilog	589
vgencomp Component Declaration when VHDL Instantiates Verilog	590
Modules with Bidirectional Pass Switches	591
Modules with Unnamed Ports	591
Verilog or SystemVerilog Instantiating VHDL	592
VHDL Instantiation Criteria Within Verilog	592
Entity and Architecture Names and Escaped Identifiers	594
Named Port Associations	594
Generic Associations	594
SDF Annotation	594
Sharing User-Defined Types	595
SystemC Instantiating Verilog or SystemVerilog	600
Verilog Instantiation Criteria Within SystemC	600
SystemC Foreign Module (Verilog) Declaration	601
Parameter Support for SystemC Instantiating Verilog	602
Verilog or SystemVerilog Instantiating SystemC	607
SystemC Instantiation Criteria for Verilog	607
Exporting SystemC Modules for Verilog	607
Parameter Support for Verilog Instantiating SystemC	608
SystemC Instantiating VHDL	610
VHDL Instantiation Criteria Within SystemC	610
SystemC Foreign Module (VHDL) Declaration	611
Generic Support for SystemC Instantiating VHDL	612
VHDL Instantiating SystemC	617
SystemC Instantiation Criteria for VHDL	617
Component Declaration for VHDL Instantiating SystemC	617
vgencomp Component Declaration when VHDL Instantiates SystemC	618
Exporting SystemC Modules for VHDL	618
Passing Generics From VHDL or Verilog Down to SystemC	619
SystemC Procedural Interface to SystemVerilog	620
Definition of Terms	621
SystemC DPI Usage Flow	621
SystemC Import Functions	621
Calling SystemVerilog Export Tasks / Functions from SystemC	626
SystemC Data Type Support in SystemVerilog DPI	626
SystemC Function Prototype Header File (sc_dpiheader.h)	629
Support for Multiple SystemVerilog Libraries	629

Table of Contents

SystemC DPI Usage Example	630
Chapter 11	
Advanced Simulation Techniques	633
Checkpointing and Restoring Simulations	633
Checkpoint File Contents	633
Controlling Checkpoint File Compression	634
The Difference Between Checkpoint/Restore and Restart	634
Using Macros with Restart and Checkpoint/Restore	634
Checkpointing Foreign C Code That Works with Heap Memory	635
Checkpointing a Running Simulation	635
Simulating with an Elaboration File	637
Why an Elaboration File?	637
Elaboration File Flow	637
Creating an Elaboration File	638
Loading an Elaboration File	638
Modifying Stimulus	639
Using With the PLI or FLI	640
Chapter 12	
Recording and Viewing Transactions	641
Transaction Background	641
What is a Transaction?	642
About the Source Code for Transactions	642
About Transaction Streams	643
Viewing Transactions in the GUI	645
Transaction Viewing Commonalities	645
Viewing Transaction Objects in the Structure Window	647
Viewing Transactions in the Wave Window	647
Viewing a Transaction in the List Window	654
Viewing a Transaction in the Objects Window	655
Debugging with Tcl	656
Transactions in Designs with Questa Verification IP	657
Transaction Recording Flow	657
Transaction Recording Guidelines	660
Names of Streams and Substreams	661
Stream Logging	661
Transaction UIDs	662
Attribute Type	662
Multiple Uses of the Same Attribute	662
Anonymous Attributes	663
Definition of Relationship in Transactions	663
The Life-cycle of a Transaction	664
Transaction Handles and Memory Leaks	665
Transaction Recording Procedures	665
Recording Transactions in Verilog and VHDL	665
Recording Transactions in SystemC	669
SCV Limitations	676

CLI Debugging Command Reference	676
Verilog and VHDL API System Task Reference	677
add_attribute	677
add_relation	678
begin_transaction	679
create_transaction_stream	680
delete_transaction	681
end_transaction	682
free_transaction	683
Chapter 13	
Verifying Designs with	
Questa Verification IP Library Components.....	685
What is Questa Verification IP?	685
What is a Questa Verification IP Transaction?	686
Questa Verification IP Transaction Relationships	686
Questa Verification IP Transaction Viewing in the GUI	687
Questa Verification IP Objects in the GUI	687
Arrays in Questa Verification IP	689
Viewing Questa Verification IP Transactions in the Wave Window	690
Viewing Questa Verification IP Transactions in Objects Window	696
Viewing Questa Verification IP Transactions in List Window	696
Questa Verification IP Transaction Debug	697
Debugging Using Relationships	698
Questa Verification IP Transaction Details in Transaction View Window	699
Chapter 14	
Recording Simulation Results With Datasets.....	703
Saving a Simulation to a WLF File	704
WLF File Parameter Overview	705
Limiting the WLF File Size	706
Multithreading on Linux Platforms	707
Opening Datasets	708
Viewing Dataset Structure	708
Structure Tab Columns	709
Managing Multiple Datasets	710
Managing Multiple Datasets in the GUI	710
Command Line	710
Restricting the Dataset Prefix Display	711
Saving at Intervals with Dataset Snapshot	712
Collapsing Time and Delta Steps	713
Virtual Objects	714
Virtual Signals	715
Virtual Functions	716
Virtual Regions	717
Virtual Types	717

Chapter 15

Waveform Analysis..... 719

- Objects You Can View 719
- Wave Window Overview..... 720
 - Wave Window Panes 720
- List Window Overview 722
- Adding Objects to the Wave or List Window 722
 - Adding Objects with Mouse Actions 723
 - Adding Objects with Menu Selections 723
 - Adding Objects with a Command..... 723
 - Adding Objects with a Window Format File 724
- Working with Cursors 724
 - Adding Cursors..... 726
 - Jumping to a Signal Transition..... 726
 - Measuring Time with Cursors in the Wave Window 727
 - Syncing All Active Cursors 727
 - Linking Cursors 728
 - Understanding Cursor Behavior 728
 - Shortcuts for Working with Cursors..... 729
- Setting Time Markers in the List Window 729
 - Working with Markers 730
- Expanded Time in the Wave and List Windows 730
 - Expanded Time Terminology 731
 - Recording Expanded Time Information 731
 - Viewing Expanded Time Information in the Wave Window..... 732
 - Selecting the Expanded Time Display Mode 736
 - Switching Between Time Modes 737
 - Expanding and Collapsing Simulation Time 737
 - Expanded Time Viewing in the List Window 738
- Zooming the Wave Window Display 740
 - Zooming with the Menu, Toolbar and Mouse 741
 - Saving Zoom Range and Scroll Position with Bookmarks..... 742
- Searching in the Wave and List Windows..... 743
 - Searching for Values or Transitions 744
 - Using the Expression Builder for Expression Searches 745
- Filtering the Wave Window Display 747
- Formatting the Wave Window..... 747
 - Setting Wave Window Display Preferences..... 747
 - Formatting Objects in the Wave Window 751
 - Dividing the Wave Window 753
 - Splitting Wave Window Panes 755
- Wave Groups 756
 - Creating a Wave Group 756
 - Deleting or Ungrouping a Wave Group 758
 - Adding Items to an Existing Wave Group 758
 - Removing Items from an Existing Wave Group..... 758
 - Miscellaneous Wave Group Features 759
- Composite Signals or Buses 759

Formatting the List Window	760
Setting List Window Display Properties.....	760
Formatting Objects in the List Window	761
Saving the Window Format	763
Exporting Waveforms from the Wave window.....	764
Exporting the Wave Window as a Bitmap Image.....	764
Printing the Wave Window to a Postscript File	765
Printing the Wave Window on the Windows Platform	765
Saving Waveforms Between Two Cursors.....	765
Saving List Window Data to a File	767
Viewing SystemVerilog Class Objects	768
Combining Objects into Buses	770
Creating a Virtual Signal	771
Configuring New Line Triggering in the List Window.....	772
Using Gating Expressions to Control Triggering.....	775
Sampling Signals at a Clock Change	776
Miscellaneous Tasks	777
Examining Waveform Values.....	777
Displaying Drivers of the Selected Waveform.....	777
Sorting a Group of Objects in the Wave Window	777
Creating and Managing Breakpoints	777
Signal Breakpoints	778
File-Line Breakpoints	780
Saving and Restoring Breakpoints	782
Waveform Compare.....	782
Mixed-Language Waveform Compare Support	783
Three Options for Setting up a Comparison.....	783
Setting Up a Comparison with the GUI	785
Starting a Waveform Comparison	785
Adding Signals, Regions, and Clocks.....	786
Specifying the Comparison Method	788
Setting Compare Options	789
Viewing Differences in the Wave Window	790
Viewing Differences in the List Window.....	792
Viewing Differences in Textual Format	793
Saving and Reloading Comparison Results	793
Comparing Hierarchical and Flattened Designs	794
Chapter 16	
Schematic Window	795
Schematic Window Usage Flow	796
Post Simulation Schematic Debug Flow	797
Two Schematic Views	797
Common Tasks for Schematic Debugging	798
Adding Objects to the Incremental View	798
Display a Structural Overview in the Full View.....	799
Exploring the Connectivity of the Design	800
Folding and Unfolding Instances in the Incremental View	807

Table of Contents

Exploring Designs with the Embedded Wave Viewer	808
Tracing Events in the Incremental View	809
Tracing the Source of an Unknown State (StX)	814
Finding Objects by Name in the Schematic Window	816
Schematic Concepts	816
Symbol Mapping	817
Schematic Window Graphic Interface Reference	819
What Can I View in the Schematic Window?	820
How is the Schematic Window Linked to Other Windows?	820
How Can I Print and Save the Display?	821
How do I Configure Window Options?	822
How do I Zoom and Pan the Display?	824
How do I Use Keyboard Shortcuts?	826
Chapter 17	
Debugging with the Dataflow Window	829
Dataflow Window Overview	829
Dataflow Usage Flow	830
Post-Simulation Debug Flow Details	830
Common Tasks for Dataflow Debugging	832
Adding Objects to the Dataflow Window	832
Exploring the Connectivity of the Design	833
Exploring Designs with the Embedded Wave Viewer	837
Tracing Events	839
Tracing the Source of an Unknown State (StX)	839
Finding Objects by Name in the Dataflow Window	841
Automatically Tracing All Paths Between Two Nets	841
Dataflow Concepts	843
Symbol Mapping	843
Current vs. Post-Simulation Command Output	845
Dataflow Window Graphic Interface Reference	846
What Can I View in the Dataflow Window?	846
How is the Dataflow Window Linked to Other Windows?	846
How Can I Print and Save the Display?	847
How Do I Configure Window Options?	849
Chapter 18	
Source Window	851
Creating and Editing Source Files	851
Creating New Files	851
Opening Existing Files	852
Editing Files	853
Saving Files	856
Searching for Code in the Source Window	856
Navigating Through Your Design	857
Data and Objects in the Source Window	858
Determining Object Values and Descriptions	859
Displaying Object Values with Source Annotation	859

Source Window Debugging and Textual Connectivity	861
Dragging Source Window Objects Into Other Windows	863
Highlighted Text in the Source Window	864
Hyperlinked Text in the Source Window	865
Code Coverage Data in the Source Window	865
Breakpoints	867
Setting Individual Breakpoints in a Source File	868
Setting Breakpoints with the bp Command	868
Setting SystemC Breakpoints	869
Editing Breakpoints	869
Saving and Restoring Breakpoints	871
Setting Conditional Breakpoints	872
Run Until Here	874
Source Window Bookmarks	875
Setting and Removing Bookmarks	875
Setting Source Window Preferences	875
Chapter 19	
Using Causality Traceback	877
Usage Flow for Causality Traceback	877
Post-sim Debug	879
Initiating Causality Traceback from the GUI	879
Tracing to the First Sequential Process	880
Tracing to the Immediate Driving Process	885
Tracing to the Root Cause	887
Tracing to the Root Cause of an 'X'	889
Finding All Possible Drivers	889
Tracing from a Specific Time	890
Handling Multiple Drivers	891
Viewing Causality Path Details	891
Initiating Causality Traceback from the Command Line	894
Setting the Report Destination with Command Line Options	895
Setting Causality Traceback Preferences	896
Chapter 20	
Code Coverage	899
Overview of Code Coverage Types	900
Language and Datatype Support	900
Usage Flow for Code Coverage Collection	901
Specifying Coverage Types for Collection	902
Enabling Simulation for Code Coverage Collection	903
Saving Code Coverage in the UCDB	904
Code Coverage in the UCDB	905
Code Coverage in the Graphic Interface	906
Understanding Unexpected Coverage Results	908
Code Coverage Types	908
Statement Coverage	909
Branch Coverage	909

Table of Contents

Case and Branches	910
AllFalse Branches	910
Missing Branches in VHDL and Clock Optimizations	911
Condition and Expression Coverage	911
Effect of Short-circuiting on Expression and Condition Coverage	912
Reporting Condition and Expression Coverage	913
FEC Coverage Detailed Examples	914
UDP Coverage Details and Examples	920
VHDL Condition and Expression Type Support	923
Verilog/SV Condition and Expression Type Support	923
Toggle Coverage	924
Toggle Coverage and Performance Considerations	924
VHDL Toggle Coverage Type Support	925
Verilog/SV Toggle Coverage Type Support	925
Toggle Ports Only Flow	926
Viewing Toggle Coverage Data in the Objects Window	927
Understanding Toggle Counts	928
Limiting Toggle Coverage	932
Finite State Machine Coverage	933
Coverage Exclusions	933
What Objects can be Excluded?	933
Auto Exclusions	934
Methods for Excluding Objects	934
Toggle Exclusion Management	945
Exclude Nodes from Toggle Coverage	946
FSM Coverage Exclusions	949
Saving and Recalling Exclusions	952
Coverage Reports	954
Using the coverage report Command	956
Using the toggle report Command	957
Using the Coverage Report Dialog	958
Setting a Default Coverage Reporting Mode	958
XML Output	958
HTML Output	959
Coverage Reporting on a Specific Test	959
Notes on Coverage and Optimization	959
Customizing Optimization Level for Coverage Runs	960
Interaction of Optimization and Coverage Arguments	961

Chapter 21

Finite State Machines	963
FSM Recognition	963
Unsupported FSM Design Styles	964
FSM Design Style Examples	965
FSM Coverage	967
FSM Multi-State Transitions	968
Collecting FSM Coverage Metrics	968
Reporting Coverage Metrics for FSMs	970

Viewing FSM Information in the GUI	971
FSM Coverage Metrics Available in the GUI	973
Advanced Command Arguments for FSMs	975
Consolidated FSM Recognition Arguments	975
Recognized FSM Note	976
FSM Recognition Info Note	976
FSM Coverage Text Report	978
Chapter 22	
Verification with Assertions and Cover Directives	981
Overview of Assertions and Cover Directives	981
Assertion Coding Guidelines	982
Processing Assume Directives	986
Configuring Assertions	987
Simulating Assertions	997
Analyzing Assertions and Cover Directives	1002
Saving Metrics to the UCDB	1012
Excluding Assertions and Cover Directives	1012
Creating Assertion Reports	1012
PSL Assertions and Cover Directives	1014
Using PSL Directives in Procedural Blocks	1015
PSL Limitations in 0-In	1015
Common PSL Assertions Coding Tasks	1015
Compiling and Simulating PSL Assertions	1025
PSL Limitations	1026
Using SVA Assertions and Cover Directives	1027
Assertions and Action Blocks in SVA	1027
Deferred Assertions and Cover Directives	1027
SystemVerilog Cover Directives	1027
SVA Usage Flow for Assertions and Cover Directives	1028
Using -assertdebug to Debug with Assertions and Cover Directives	1029
Viewing Debugging Information	1029
Enabling ATV Recording	1030
Saving Assertion and Cover Directive Metrics	1032
Viewing Assertion Threads in the ATV Window	1036
Navigating Inside an ATV Window	1039
Actions in the ATV Window	1043
Chapter 23	
Verification with Functional Coverage	1049
Functional Coverage Flow	1049
Configuring Covergroups, Coverpoints, and Crosses	1050
Functional Coverage Computation	1051
Predefined Coverage Methods	1051
Predefined Coverage System Function	1051
SystemVerilog Functional Coverage Terminology	1051
IEEE Std 1800-2009 Option Behavior	1051
Type-Based Coverage With Constructor Parameters	1056

Table of Contents

Viewing Functional Coverage Statistics in the GUI	1059
Functional Coverage Statistics in the Covergroups Window	1060
Functional Coverage Aggregation in Structure Window	1060
Reporting on Functional Coverage	1061
Creating Text Reports Via the GUI	1061
Creating HTML Reports Via the GUI	1063
Covergroup Bin Reporting and Timestamps	1063
Filtering Functional Coverage Data	1064
Reporting Via the Command Line	1066
Assertion/Cover Directive Naming Conventions	1067
Covergroup Naming Conventions	1068
Covergroup in a Class	1068
Saving Functional Coverage Data	1069
Loading a Functional Coverage Database into Simulation	1070
Excluding Functional Coverage	1072
Merging Databases	1074
Chapter 24	
Verification with Constrained	
Random Stimulus	1075
Building Constrained Random Test Benches on SystemVerilog Classes	1076
Generating New Random Values with randomize()	1077
Using Attributes	1078
Inheriting Constraints	1082
Examining Solver Failures	1083
Setting Compatibility with a Previous Release	1084
Seeding the Random Number Generator (RNG)	1084
Using Program Blocks	1085
Chapter 25	
Coverage and Verification Management in the UCDB	1087
Coverage and Verification Overview	1088
What is the Unified Coverage Database?	1089
Calculation of Total Coverage	1090
Coverage and Simulator Use Modes	1095
Coverage View Mode and the UCDB	1095
Running Tests and Collecting Data	1096
Collecting and Saving Coverage Data	1096
Understanding Stored Test Data in the UCDB	1098
Rerunning Tests and Executing Commands	1101
Managing Test Data in UCDBs	1104
Merging Coverage Test Data	1105
Warnings During Merge	1108
Ranking Coverage Test Data	1109
Modifying UCDBs	1111
About the Merge Algorithm	1113
Merge Usage Scenarios	1118
Viewing and Analyzing Verification Data	1119

Storing User Attributes in the UCDB	1120
Viewing Test Data in the GUI	1120
Viewing Test Data in the Browser Window	1120
Generating Coverage Reports	1122
Filtering Data in the UCDB	1129
Filtering Results by User Attributes	1131
Retrieving Test Attribute Record Content	1133
Analysis for Late-stage ECO Changes	1133
Chapter 26	
C Debug	1135
Supported Platforms and gdb Versions	1135
Running C Debug on Windows Platforms	1136
Setting Up C Debug	1136
Running C Debug from a DO File	1137
Setting Breakpoints	1137
Stepping in C Debug	1139
Debugging Active or Suspended Threads	1140
Known Problems With Stepping in C Debug	1140
Quitting C Debug	1141
Finding Function Entry Points with Auto Find bp	1141
Identifying All Registered Function Calls	1141
Enabling Auto Step Mode	1142
Auto Find bp Versus Auto Step Mode	1144
Debugging Functions During Elaboration	1144
FLI Functions in Initialization Mode	1146
PLI Functions in Initialization Mode	1146
VPI Functions in Initialization Mode	1148
Completing Design Load	1148
Debugging Functions when Quitting Simulation	1148
C Debug Command Reference	1149
Chapter 27	
Profiling Performance and Memory Use	1151
Introducing Performance and Memory Profiling	1151
Statistical Sampling Profiler	1152
Memory Allocation Profiler	1152
Getting Started with the Profiler	1152
Enabling the Memory Allocation Profiler	1152
Enabling the Statistical Sampling Profiler	1154
Collecting Memory Allocation and Performance Data	1154
Running the Profiler on Windows with PLI/VPI Code	1155
Interpreting Profiler Data	1155
Viewing Profiler Results	1155
Ranked Window	1156
Design Units Window	1157
Calltree Window	1157
Structural Window	1159

Table of Contents

Viewing Profile Details	1160
Integration with Source Windows	1161
Analyzing C Code Performance	1162
Searching Profiler Results	1163
Reporting Profiler Results	1163
Capacity Analysis	1165
Enabling or Disabling Capacity Analysis	1166
Levels of Capacity Analysis	1168
Obtaining a Graphical Interface (GUI) Display	1169
Writing a Text-Based Report	1170
Chapter 28	
Signal Spy	1175
Signal Spy Formatting Syntax	1176
Signal Spy Supported Types	1176
disable_signal_spy	1177
enable_signal_spy	1179
init_signal_driver	1181
init_signal_spy	1185
signal_force	1189
signal_release	1193
Chapter 29	
Monitoring Simulations with JobSpy	1195
Basic JobSpy Flow	1195
Starting the JobSpy Daemon	1196
Running JobSpy from the Command Line	1197
Simulation Commands Available to JobSpy	1197
Example Session	1198
Running the JobSpy GUI	1199
Starting Job Manager	1199
Invoking Simulation Commands in Job Manager	1199
View Commands and Pathnames	1200
Viewing Results During Active Simulation	1201
Viewing Waveforms from the Command Line	1201
Licensing and Job Suspension	1202
Checkpointing Jobs	1202
Connecting to Load-Sharing Software	1203
Checkpointing with Load-Sharing Software	1203
Chapter 30	
Generating Stimulus with Waveform Editor	1205
Getting Started with the Waveform Editor	1205
Using Waveform Editor Prior to Loading a Design	1205
Using Waveform Editor After Loading a Design	1206
Creating Waveforms from Patterns	1207
Creating Waveforms with Wave Create Command	1208
Editing Waveforms	1209

Selecting Parts of the Waveform	1210
Stretching and Moving Edges	1212
Simulating Directly from Waveform Editor	1212
Exporting Waveforms to a Stimulus File	1212
Driving Simulation with the Saved Stimulus File	1213
Signal Mapping and Importing EVCD Files	1214
Using Waveform Compare with Created Waveforms	1214
Saving the Waveform Editor Commands	1215
Chapter 31	
Standard Delay Format (SDF) Timing Annotation	1217
Specifying SDF Files for Simulation	1217
Instance Specification	1218
SDF Specification with the GUI	1218
Errors and Warnings	1219
Compiling SDF Files	1219
Simulating with Compiled SDF Files	1219
Using \$sdf_annotate() with Compiled SDF	1220
VHDL VITAL SDF	1220
SDF to VHDL Generic Matching	1220
Resolving Errors	1221
Verilog SDF	1221
\$sdf_annotate	1223
SDF to Verilog Construct Matching	1224
Retain Delay Behavior	1227
Optional Edge Specifications	1229
Optional Conditions	1230
Rounded Timing Values	1230
SDF for Mixed VHDL and Verilog Designs	1231
Interconnect Delays	1231
Disabling Timing Checks	1231
Troubleshooting	1232
Specifying the Wrong Instance	1232
Matching a Single Timing Check	1233
Mistaking a Component or Module Name for an Instance Label	1233
Forgetting to Specify the Instance	1234
Chapter 32	
Value Change Dump (VCD) Files	1235
Creating a VCD File	1235
Four-State VCD File	1235
Extended VCD File	1236
VCD Case Sensitivity	1236
Checkpoint/Restore and Writing VCD Files	1236
Using Extended VCD as Stimulus	1237
Simulating with Input Values from a VCD File	1237
Replacing Instances with Output Values from a VCD File	1238
VCD Commands and VCD Tasks	1240

Table of Contents

Using VCD Commands with SystemC.....	1241
Compressing Files with VCD Tasks.....	1242
VCD File from Source to Output.....	1242
VHDL Source Code.....	1242
VCD Simulator Commands.....	1243
VCD Output.....	1244
VCD to WLF.....	1245
Capturing Port Driver Data.....	1245
Driver States.....	1245
Driver Strength.....	1246
Identifier Code.....	1246
Resolving Values.....	1247

Chapter 33

Tcl and Macros (DO Files).....	1251
Tcl Features.....	1251
Tcl References.....	1251
Tcl Commands.....	1252
Tcl Command Syntax.....	1252
If Command Syntax.....	1255
set Command Syntax.....	1255
Command Substitution.....	1256
Command Separator.....	1257
Multiple-Line Commands.....	1257
Evaluation Order.....	1257
Tcl Relational Expression Evaluation.....	1257
Variable Substitution.....	1258
System Commands.....	1258
Simulator State Variables.....	1259
Referencing Simulator State Variables.....	1260
Special Considerations for the now Variable.....	1260
List Processing.....	1261
Reading Variable Values From the INI File.....	1261
Simulator Tcl Commands.....	1262
Simulator Tcl Time Commands.....	1262
Conversions.....	1263
Relations.....	1263
Arithmetic.....	1264
Tcl Examples.....	1264
Macros (DO Files).....	1266
Creating DO Files.....	1266
Using Parameters with DO Files.....	1267
Deleting a File from a .do Script.....	1267
Making Macro Parameters Optional.....	1268
Useful Commands for Handling Breakpoints and Errors.....	1269
Error Action in DO Files.....	1270
The Tcl Debugger.....	1271
Starting the Debugger.....	1271

How the debugger Works	1271
The Chooser	1271
The Debugger	1272
Breakpoints	1274
Configuration	1275
TclPro Debugger	1275
Appendix A	
modelsim.ini Variables	1277
Organization of the modelsim.ini File	1277
Making Changes to the modelsim.ini File	1278
Changing the modelsim.ini Read-Only Attribute	1278
The Runtime Options Dialog	1278
Editing modelsim.ini Variables	1282
Overriding the Default Initialization File	1282
Variables	1283
AcceptLowerCasePragmaOnly	1284
AddPragmaPrefix	1284
AmsStandard	1284
AssertFile	1285
AssertionActiveThreadMonitor	1285
AssertionActiveThreadMonitorLimit	1285
AssertionCover	1286
AssertionDebug	1286
AssertionEnable	1286
AssertionEnableVacuousPassActionBlock	1287
AssertionFailAction	1287
AssertionFailLocalVarLog	1287
AssertionFailLog	1288
AssertionLimit	1288
AssertionPassLog	1288
AssertionThreadLimit	1289
AssertionThreadLimitAction	1289
ATVStartTimeKeepCount	1289
AutoExclusionsDisable	1290
BindAtCompile	1290
BreakOnAssertion	1290
CheckPlusargs	1291
CheckpointCompressMode	1291
CheckSynthesis	1292
ClassDebug	1292
CodeCoverage	1292
CommandHistory	1293
CompilerTempDir	1293
ConcurrentFileLimit	1293
Coverage	1293
CoverAtLeast	1294
CoverCells	1294

Table of Contents

CoverClkOptBuiltin	1294
CoverCountAll	1295
CoverEnable	1295
CoverExcludeDefault	1295
CoverFEC	1296
CoverLimit	1296
CoverLog	1296
CoverOpt.	1297
CoverRespectHandL	1297
CoverReportCancelled	1298
CoverShortCircuit	1298
CoverSub.	1298
CoverThreadLimit	1299
CoverThreadLimitAction	1299
CoverUDP	1299
CoverWeight	1300
CppOptions	1300
CppPath	1300
CreateDirForFileAccess	1301
CvgZWNNoCollect	1301
DatasetSeparator	1301
DefaultForceKind	1302
DefaultRadix	1302
DefaultRestartOptions	1303
DelayFileOpen	1304
displaymsgmode	1304
DpiCppPath	1305
DpiOutOfTheBlue	1305
DumpportsCollapse	1306
EmbeddedPsl	1306
EnableSVCoverpointExprVariable	1306
EnableTypeOf	1307
EnumBaseInit	1307
error	1307
ErrorFile	1308
Explicit	1308
ExtendedToggleMode	1308
fatal	1309
FecEffort	1309
floatfixlib	1310
ForceSigNextIter	1310
ForceUnsignedIntegerToVHDLInteger	1310
FsmImplicitTrans	1311
FsmResetTrans	1311
FsmSingle	1311
FsmXAssign	1312
GenerateFormat	1312
GenerateLoopIterationMax	1313
GenerateRecursionDepthMax	1313

GenerousIdentifierParsing	1313
GlobalSharedObjectsList	1314
Hazard	1314
ieee	1314
IgnoreError	1314
IgnoreFailure	1315
IgnoreNote	1315
IgnorePragmaPrefix	1315
ignoreStandardRealVector	1316
IgnoreSVAError	1316
IgnoreSVAFatal	1317
IgnoreSVAInfo	1317
IgnoreSVAWarning	1317
IgnoreVitalErrors	1318
IgnoreWarning	1318
ImmediateContinuousAssign	1319
IncludeRecursionDepthMax	1319
InitOutCompositeParam	1319
IterationLimit	1320
LargeObjectSilent	1320
LargeObjectSize	1320
LibrarySearchPath	1321
License	1321
MaxReportRhsCrossProducts	1322
MaxReportRhsSVCrossProducts	1322
MaxSVCoverpointBinsDesign	1322
MaxSVCoverpointBinsInst	1322
MaxSVCrossBinsDesign	1323
MaxSVCrossBinsInst	1323
MessageFormat	1323
MessageFormatBreak	1324
MessageFormatBreakLine	1324
MessageFormatError	1325
MessageFormatFail	1325
MessageFormatFatal	1325
MessageFormatNote	1326
MessageFormatWarning	1326
MixedAnsiPorts	1326
modelsim_lib	1327
msgmode	1327
mtiAvm	1327
mtiOvm	1327
mtiPA	1328
mtiUPF	1328
mtiUvm	1328
MultiFileCompilationUnit	1329
MvcHome	1329
NoCaseStaticError	1329
NoDebug	1330

Table of Contents

NoDeferSubpgmCheck	1330
NoIndexCheck	1330
NoOthersStaticError	1331
NoRangeCheck	1331
note	1331
NoVital	1332
NoVitalCheck	1332
NumericStdNoWarnings	1332
OldVHDLConfigurationVisibility	1333
OldVhdlForGenNames	1333
OnFinish	1334
OnFinishPendingAssert	1334
Optimize_1164	1334
ParallelJobs	1335
PathSeparator	1335
PedanticErrors	1336
PliCompatDefault	1336
PreserveCase	1337
PrintSimStats	1338
PrintSVPackageLoadingAttribute	1338
Protect	1338
PsIOneAttempt	1339
PsIInfinityThreshold	1339
Quiet	1339
RequireConfigForAllDefaultBinding	1340
Resolution	1340
RunLength	1341
ScalarOpts	1341
SccomLogfile	1341
SccomVerbose	1342
ScEnableScSignalWriteCheck	1342
ScMainFinishOnQuit	1342
ScMainStackSize	1343
ScShowIeeeDeprecationWarnings	1343
ScTimeUnit	1343
ScvPhaseRelationName	1343
SeparateConfigLibrary	1344
Show_BadOptionWarning	1344
Show_Lint	1344
Show_PsIChecksWarnings	1345
Show_source	1345
Show_VitalChecksWarnings	1345
Show_Warning1	1345
Show_Warning2	1346
Show_Warning3	1346
Show_Warning4	1346
Show_Warning5	1346
ShowConstantImmediateAsserts	1347
ShowFunctions	1347

ShowUnassociatedScNameWarning	1347
ShowUndebuggableScTypeWarning	1348
ShutdownFile	1348
SignalSpyPathSeparator	1348
SimulateAssumeDirectives	1349
SimulateImmedAsserts	1349
SimulatePSL	1349
SimulateSVA	1350
SolveACTbeforeSpeculate	1350
SolveACTMaxOps	1350
SolveACTMaxTests	1351
SolveACTRetryCount	1351
SolveArrayResizeMax	1351
SolveEngine	1352
SolveFailDebug	1352
SolveFailDebugMaxSet	1352
SolveFailSeverity	1353
SolveFlags	1353
SolveGraphMaxEval	1353
SolveGraphMaxSize	1354
SolveIgnoreOverflow	1354
SolveRev	1354
SolveSpeculateDistFirst	1355
SolveSpeculateFirst	1355
SolveSpeculateLevel	1356
SolveSpeculateMaxCondWidth	1356
SolveSpeculateMaxIterations	1357
SparseMemThreshold	1357
Startup	1357
std	1358
std_developerskit	1358
StdArithNoWarnings	1358
suppress	1359
SuppressFileTypeReg	1359
Sv_Seed	1359
sv_std	1360
SVAPrintOnlyUserMessage	1360
SVCovergroupGetInstCoverageDefault	1360
SVCovergroupGoal	1361
SVCovergroupGoalDefault	1361
SVCovergroupMergeInstancesDefault	1362
SVCovergroupPerInstanceDefault	1363
SVCovergroupSampleInfo	1363
SVCovergroupStrobe	1364
SVCovergroupStrobeDefault	1364
SVCovergroupTypeGoal	1364
SVCovergroupTypeGoalDefault	1365
SVCovergroupZWNNoCollect	1365
SVCoverpointAutoBinMax	1366

Table of Contents

SVCoverpointExprVariablePrefix	1366
SVCoverpointWildCardBinValueSizeWarn	1367
SVCrossNumPrintMissing	1367
SVCrossNumPrintMissingDefault	1367
SVFileExtensions	1368
Svlog	1368
synopsys	1368
SyncCompilerFiles	1369
ToggleCountLimit	1369
ToggleFixedSizeArray	1369
ToggleMaxFixedSizeArray	1370
ToggleMaxIntValues	1370
ToggleMaxRealValues	1371
ToggleNoIntegers	1371
TogglePackedAsVec	1371
TogglePortsOnly	1372
ToggleVlogEnumBits	1372
ToggleVlogIntegers	1372
ToggleVlogReal	1373
ToggleWidthLimit	1373
TranscriptFile	1373
UCDBFilename	1374
UCDBTestStatusMessageFilter	1374
UnattemptedImmediateAssertions	1374
UnbufferedOutput	1375
UpCase	1375
UserTimeUnit	1375
UseScv	1376
UseSVCrossNumPrintMissing	1376
UVMControl	1376
verilog	1377
Veriuser	1377
VHDL93	1378
VhdlVariableLogging	1378
vital2000	1379
vlog95compat	1379
VoptFlow	1379
WarnConstantChange	1380
warning	1380
WaveSignalNameWidth	1380
WLFCacheSize	1381
WLFCollapseMode	1381
WLFCompress	1382
WLFDeleteOnQuit	1382
WLFFileLock	1383
WLFFilename	1383
WLFOptimize	1383
WLFSaveAllRegions	1384
WLFSimCacheSize	1384

WLFSizeLimit	1385
WLFTimeLimit	1385
WLFUseThreads	1386
Commonly Used modelsim.ini Variables	1386
Common Environment Variables	1386
Hierarchical Library Mapping	1387
Creating a Transcript File	1387
Using a Startup File	1388
Turning Off Assertion Messages	1388
Turning off Warnings from Arithmetic Packages	1388
Force Command Defaults	1389
Restart Command Defaults	1389
VHDL Standard	1389
Opening VHDL Files	1390
Appendix B	
Location Mapping	1391
Referencing Source Files with Location Maps	1391
Using Location Mapping	1391
Pathname Syntax	1392
How Location Mapping Works	1392
Mapping with TCL Variables	1392
Appendix C	
Error and Warning Messages	1393
Message System	1393
Message Format	1393
Getting More Information	1394
Changing Message Severity Level	1394
Suppressing Warning Messages	1394
Suppressing VCOM Warning Messages	1394
Suppressing VLOG Warning Messages	1395
Suppressing VOPT Warning Messages	1395
Suppressing VSIM Warning Messages	1396
Exit Codes	1396
Miscellaneous Messages	1398
sccom Error Messages	1402
Enforcing Strict 1076 Compliance	1403
Appendix D	
Verilog Interfaces to C	1407
Implementation Information	1407
GCC Compiler Support for use with C Interfaces	1409
Registering PLI Applications	1409
Registering VPI Applications	1411
Registering DPI Applications	1412
DPI Use Flow	1413
DPI and the vlog Command	1414

Table of Contents

When Your DPI Export Function is Not Getting Called	1415
Troubleshooting a Missing DPI Import Function.....	1415
Simplified Import of Library Functions	1416
Optimizing DPI Import Call Performance	1417
DPI Arguments of Parameterized Datatypes	1417
Making Verilog Function Calls from non-DPI C Models	1418
Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code	1418
Compiling and Linking C Applications for Interfaces	1419
For all UNIX Platforms	1419
Windows Platforms — C	1420
32-bit Linux Platform — C.....	1421
64-bit Linux Platform — C.....	1421
Compiling and Linking C++ Applications for Interfaces	1422
Windows Platforms — C++	1423
32-bit Linux Platform — C++	1424
64-bit Linux Platform — C++	1424
Specifying Application Files to Load	1424
PLI and VPI File Loading.....	1424
DPI File Loading.....	1425
Loading Shared Objects with Global Symbol Visibility	1426
PLI Example	1426
VPI Example	1426
DPI Example	1427
The PLI Callback reason Argument	1428
The sizetf Callback Function.....	1430
PLI Object Handles	1430
Third Party PLI Applications.....	1430
Support for VHDL Objects	1431
IEEE Std 1364 ACC Routines.....	1433
IEEE Std 1364 TF Routines.....	1435
SystemVerilog DPI Access Routines.....	1435
Verilog-XL Compatible Routines	1436
64-bit Support for PLI	1436
Using 64-bit ModelSim with 32-bit Applications	1436
PLI/VPI Tracing.....	1436
The Purpose of Tracing Files	1437
Invoking a Trace.....	1437
Checkpointing and Interface Code.....	1438
Checkpointing Code that Works with Heap Memory.....	1438
Debugging Interface Application Code.....	1438

Appendix E

Command and Keyboard Shortcuts	1441
Command Shortcuts.....	1441
Command History Shortcuts.....	1441
Main and Source Window Mouse and Keyboard Shortcuts	1442
List of Keyboard Shortcuts in GUI Windows.....	1445
List Window Keyboard Shortcuts	1446

Wave Window Mouse and Keyboard Shortcuts	1446
Appendix F	
Setting GUI Preferences.....	1449
Customizing the Simulator GUI Layout	1449
Layout Mode Loading Priority	1449
Configure Window Layouts Dialog Box	1450
Creating a Custom Layout Mode	1450
Changing Layout Mode Behavior.....	1450
Resetting a Layout Mode to its Default	1451
Deleting a Custom Layout Mode	1451
Configuring Default Windows for Restored Layouts.....	1451
Configuring the Column Layout.....	1452
Simulator GUI Preferences	1453
Setting Preference Variables from the GUI	1454
Setting Preference Variables from the Command Line	1456
Saving GUI Preferences	1456
The modelsim.tcl File	1457
Appendix G	
System Initialization.....	1459
Files Accessed During Startup.....	1459
Initialization Sequence.....	1459
Environment Variables	1462
Environment Variable Expansion.....	1462
Setting Environment Variables.....	1462
Creating Environment Variables in Windows	1468
Referencing Environment Variables.....	1469
Removing Temp Files (VSOUT)	1469
Appendix H	
Third-Party Model Support.....	1471
Synopsys SmartModels	1471
VHDL SmartModel Interface.....	1471
Verilog SmartModel Interface	1478
Synopsys Hardware Models	1478
VHDL Hardware Model Interface	1478
Creating Foreign Architectures with hm_entity	1479
Index	
Third-Party Information	
End-User License Agreement	

List of Examples

Example 2-1. Using the radix define Command	82
Example 2-2. Using radix define to Specify Color	83
Example 3-1. Encryption Envelope Contains Verilog IP Code to be Protected	301
Example 3-2. Encryption Envelope Contains `include Compiler Directives	302
Example 3-3. Results After Compiling with vlog +protect	307
Example 3-4. Using the Mentor Graphics Public Encryption Key in Verilog/SystemVerilog	328
Example 7-1. Memory Model Using VHDL87 and VHDL93 Architectures	410
Example 7-2. Conversions Package.	412
Example 7-3. Memory Model Using VHDL02 Architecture	414
Example 8-1. Invocation of the Verilog Compiler	423
Example 8-2. Incremental Compilation Example	426
Example 8-3. Sub-Modules with Common Names	429
Example 9-1. Simple SystemC-only sc_main().	492
Example 9-2. Generating SCV Extensions for a Structure	506
Example 9-3. Generating SCV Extensions for a Class without Friend (Private Data Not Generated).	506
Example 9-4. Generating SCV Extensions for a Class with Friend (Private Data Generated)	507
Example 9-5. Generating SCV Extensions for an Enumerated Type	508
Example 9-6. User-Defined Constraint	509
Example 9-7. Use of mti_set_typename	525
Example 9-8. Using the Custom Interface on Different Objects.	528
Example 9-9. Converting sc_main to a Module	532
Example 9-10. Using sc_main and Signal Assignments	533
Example 9-11. Using an SCV Transaction Database	534
Example 10-1. Binding with -cname and -mfcu Arguments.	556
Example 10-2. SystemC Instantiating Verilog - 1	601
Example 10-3. SystemC Instantiating Verilog - 2.	602
Example 10-4. Sample Foreign Module Declaration, with Constructor Arguments for Parameters 603	
Example 10-5. Passing Parameters as Constructor Arguments - 1	603
Example 10-6. SystemC Instantiating Verilog, Passing Integer Parameters as Template Arguments	605
Example 10-7. Passing Integer Parameters as Template Arguments and Non-integer Parameters as Constructor Arguments	606
Example 10-8. Verilog/SystemVerilog Instantiating SystemC, Parameter Information.	608
Example 10-9. SystemC Design Instantiating a VHDL Design Unit	612
Example 10-10. SystemC Instantiating VHDL, Generic Information.	613
Example 10-11. Passing Parameters as Constructor Arguments - 2	613
Example 10-12. SystemC Instantiating VHDL, Passing Integer Generics as Template Arguments	

614

Example 10-13. Passing Integer Generics as Template Arguments and Non-integer Generics as Constructor Arguments	615
Example 10-14. Global Import Function Registration	622
Example 10-15. SystemVerilog Global Import Declaration	622
Example 10-16. Registering a Global Function.	622
Example 10-17. Usage of scSetScopeByName and scGetScopeName.	625
Example 12-1. Transactions in List Window	654
Example 12-2. Verilog API Code Example.	669
Example 12-3. SCV Initialization and WLF Database Creation	670
Example 12-4. SCV API Code Example	675
Example 20-1. Branch Coverage	909
Example 20-2. Coverage Report for Branch	910
Example 20-3. FEC Coverage - Simple Expression	914
Example 20-4. FEC Coverage - Bimodal Expression	917
Example 20-5. UDP Condition Truth Table	921
Example 20-6. Vectors in UDP Condition Truth Table	921
Example 20-7. Expression UDP Truth Table	922
Example 20-8. Creating Coverage Exclusions with a .do File	937
Example 20-9. When Code Coverage Is Turned On	943
Example 20-10. Nesting and Code Coverage Types	944
Example 20-11. Excluding, Merging and Reporting on Several Runs	953
Example 20-12. Reporting Coverage Data from the Command Line	956
Example 21-1. Verilog Single-State Variable FSM	965
Example 21-2. VHDL Single-State Variable FSM	965
Example 21-3. Verilog Current-State Variable with a Single Next-State Variable FSM	966
Example 21-4. VHDL Current-State Variable and Single Next-State Variable FSM.	966
Example 22-1. Embedding Assertions in Your Code	1017
Example 22-2. Writing Assertions in an External File	1019
Example 22-3. Using PSL ended() in Verilog.	1023
Example 22-4. Using ended() in VHDL	1024
Example 22-5. Enable and Disable Assertion During Simulation.	1032
Example 23-1. With option.per_instance=1 or vsim -cvgperinstance.	1053
Example 23-2. Different Results with get_inst_coverage and get_coverage	1055
Example 23-3. Type-based Coverage	1057
Example 23-4. Bin Unions.	1058
Example 23-5. Sample Output From vcover report Command.	1067
Example 23-6. Functional Coverage in Code	1072
Example 24-1. The rand Variable	1077
Example 24-2. Generating New Random Values With randomize()	1077
Example 24-3. Using the solveflags attribute.	1080
Example 25-1. Dividing a UCDB by Module/DU.	1112
Example 25-2. Coverage Threshold Difference	1116
Example 25-3. Coverage Object Differences with Parameters	1116
Example 32-1. Verilog Counter.	1237

List of Examples

Example 32-2. VHDL Adder	1237
Example 32-3. Mixed-HDL Design.	1238
Example 32-4. Replacing Instances.	1239
Example 32-5. VCD Output from vcd dumpports.	1250
Example 33-1. Tcl while Loop	1264
Example 33-2. Tcl for Command	1264
Example 33-3. Tcl foreach Command.	1264
Example 33-4. Tcl break Command	1265
Example 33-5. Tcl continue Command	1265
Example 33-6. Access and Transfer System Information	1265
Example 33-7. Tcl Used to Specify Compiler Arguments	1266
Example 33-8. Tcl Used to Specify Compiler Arguments—Enhanced	1266
Example 33-9. Specifying Files to Compile With argc Macro	1268
Example 33-10. Specifying Compiler Arguments With Macro	1268
Example 33-11. Specifying Compiler Arguments With Macro—Enhanced.	1268
Example D-1. VPI Application Registration	1411

List of Figures

Figure 1-1. Operational Structure and Flow of ModelSim	58
Figure 2-1. Graphical User Interface	73
Figure 2-2. Find Mode	78
Figure 2-3. Filter Mode	78
Figure 2-4. Find Options Popup Menu	81
Figure 2-5. User-Defined Radix “States” in the Wave Window	83
Figure 2-6. User-Defined Radix “States” in the List Window	83
Figure 2-7. Setting the Global Signal Radix	85
Figure 2-8. Fixed Point Radix Dialog	85
Figure 2-9. Active Cursor Time	86
Figure 2-10. Enter Active Time Value	87
Figure 2-11. Schematic Keyboard Shortcuts	88
Figure 2-12. Keyboard Shortcuts Dialog Box	89
Figure 2-13. Add Keyboard Shortcut Dialog Box	90
Figure 2-14. Manage Bookmarks Dialog Box	93
Figure 2-15. Bookmark Options Dialog Box	94
Figure 2-16. Main Window of the GUI	96
Figure 2-17. Main Window — Menu Bar	97
Figure 2-18. Main Window — Toolbar Frame	97
Figure 2-19. Main Window — Toolbar	98
Figure 2-20. GUI Windows	99
Figure 2-21. GUI Tab Group	100
Figure 2-22. Wave Window Panes	101
Figure 2-23. Main Window Status Bar	101
Figure 2-24. Window Header Handle	103
Figure 2-25. Tab Handle	103
Figure 2-26. Window Undock Button	103
Figure 2-27. ATV Toolbar	116
Figure 2-28. Analysis Toolbar	116
Figure 2-29. Bookmarks Toolbar	117
Figure 2-30. Column Layout Toolbar	118
Figure 2-31. Compile Toolbar	119
Figure 2-32. Coverage Toolbar	120
Figure 2-33. FSM Toolbar	121
Figure 2-34. Help Toolbar	122
Figure 2-35. Layout Toolbar	122
Figure 2-36. Memory Toolbar	123
Figure 2-37. Mode Toolbar	123
Figure 2-38. Objectfilter Toolbar	124
Figure 2-39. Precision Toolbar	125

List of Figures

Figure 2-40. Process Toolbar	125
Figure 2-41. Profile Toolbar	126
Figure 2-42. Schematic Toolbar	127
Figure 2-43. Simulate Toolbar	128
Figure 2-44. Show Cause Dropdown Menu	130
Figure 2-45. Source Toolbar	131
Figure 2-46. Standard Toolbar	132
Figure 2-47. Add Selected to Window Dropdown Menu	133
Figure 2-48. Step Toolbar	134
Figure 2-49. Wave Toolbar	135
Figure 2-50. Wave Compare Toolbar	135
Figure 2-51. Wave Cursor Toolbar	136
Figure 2-52. Wave Edit Toolbar	137
Figure 2-53. Wave Expand Time Toolbar	138
Figure 2-54. Zoom Toolbar	139
Figure 2-55. Assertions Window	140
Figure 2-56. ATV Window	146
Figure 2-57. Call Stack Window	149
Figure 2-58. Capacity Window	151
Figure 2-59. Class Graph Window	153
Figure 2-60. Class Instances Window	155
Figure 2-61. Class Tree Window	157
Figure 2-62. Code Coverage Analysis	159
Figure 2-63. Missed Coverage in Code Coverage Analysis Windows	161
Figure 2-64. Coverage Details Window Showing Expression Truth Table	164
Figure 2-65. Coverage Details Window Showing Toggle Details	165
Figure 2-66. Coverage Details Window Showing FSM Details	166
Figure 2-67. Cover Directives Window	167
Figure 2-68. Covergroups Window	169
Figure 2-69. Dataflow Window	173
Figure 2-70. Dataflow Window and Panes	176
Figure 2-71. Files Window	177
Figure 2-72. FSM List Window	180
Figure 2-73. FSM Viewer Window	182
Figure 2-74. Combining Common Transition Conditions	185
Figure 2-75. Instance Coverage Window	189
Figure 2-76. Filter Instance List Dialog Box	190
Figure 2-77. Library Window	195
Figure 2-78. List Window	197
Figure 2-79. Locals Window	201
Figure 2-80. Change Selected Variable Dialog Box	203
Figure 2-81. Memory List Window	205
Figure 2-82. Memory Data Window	208
Figure 2-83. Split Screen View of Memory Contents	209
Figure 2-84. Message Viewer Window	212

Figure 2-85. Message Viewer Window — Tasks	213
Figure 2-86. Message Viewer Filter Dialog Box.....	217
Figure 2-87. Objects Window	218
Figure 2-88. Setting the Global Signal Radix from the Objects Window.....	219
Figure 2-89. Objects Window - Toggle Coverage.....	221
Figure 2-90. Processes Window	223
Figure 2-91. Column Heading Changes When States are Filtered	224
Figure 2-92. Next Active Process Displayed in Order Column.....	225
Figure 2-93. Sample Process Report in the Transcript Window	226
Figure 2-94. Profile Calltree Window	229
Figure 2-95. Profile Design Unit Window.....	229
Figure 2-96. Profile Ranked Window	230
Figure 2-97. Profile Structural Window	230
Figure 2-98. Profile Details Window.....	231
Figure 2-99. Schematic View Indicator.....	233
Figure 2-100. Schematic Window	234
Figure 2-101. Code Preview Window	235
Figure 2-102. Incremental Schematic Options Dialog	237
Figure 2-103. Active Time Label.....	238
Figure 2-104. Full Schematic Options Dialog.....	239
Figure 2-105. Source Window Showing Language Templates.....	243
Figure 2-106. Displaying Multiple Source Files	244
Figure 2-107. Setting Context from Source Files	245
Figure 2-108. Coverage in Source Window	248
Figure 2-109. Source Annotation Example	251
Figure 2-110. Popup Menu Choices for Textual Dataflow Information	252
Figure 2-111. Window Shows all Driving Processes	253
Figure 2-112. Source Readers Dialog Displays All Signal Readers	253
Figure 2-113. Language Templates	254
Figure 2-114. Create New Design Wizard.....	254
Figure 2-115. Inserting Module Statement from Verilog Language Template.....	255
Figure 2-116. Language Template Context Menus	256
Figure 2-117. Breakpoint in the Source Window	257
Figure 2-118. Modifying Existing Breakpoints.....	259
Figure 2-119. Source Code for <i>source.sv</i>	260
Figure 2-120. Source Window Description	263
Figure 2-121. Source Window with Find Toolbar.....	264
Figure 2-122. Preferences Dialog for Customizing Source Window	265
Figure 2-123. Structure Window	267
Figure 2-124. Find Mode Popup Displays Matches	268
Figure 2-125. Code Coverage Data in the Structure Window.....	274
Figure 2-126. Browser Tab	275
Figure 2-127. Changing the colorizeTranscript Preference Value	281
Figure 2-128. Transcript Window with Find Toolbar	282
Figure 2-129. Watch Window	284

List of Figures

Figure 2-130. Scrollable Hierarchical Display	285
Figure 2-131. Expanded Array	286
Figure 2-132. Grouping Objects in the Watch Window	287
Figure 2-133. Wave Window.	288
Figure 2-134. Pathnames Pane.	289
Figure 2-135. Setting the Global Signal Radix from the Wave Window	290
Figure 2-136. Values Pane.	290
Figure 2-137. Waveform Pane.	291
Figure 2-138. Analog Sidebar Toolbox	292
Figure 2-139. Cursor Pane.	293
Figure 2-140. Toolbox for Cursors and Timeline	293
Figure 2-141. Editing Grid and Timeline Properties	294
Figure 2-142. Cursor Properties Dialog.	295
Figure 2-143. Wave Window - Message Bar.	295
Figure 2-144. View Objects Window Dropdown Menu	296
Figure 3-1. Create an Encryption Envelope.	300
Figure 3-2. Verilog/SystemVerilog Encryption Usage Flow	310
Figure 3-3. Delivering IP Code with User-Defined Macros	312
Figure 3-4. Delivering IP with `protect Compiler Directives	322
Figure 5-1. Create Project Dialog	355
Figure 5-2. Project Window Detail	355
Figure 5-3. Add items to the Project Dialog	356
Figure 5-4. Create Project File Dialog.	357
Figure 5-5. Add file to Project Dialog.	357
Figure 5-6. Right-click Compile Menu in Project Window	358
Figure 5-7. Click Plus Sign to Show Design Hierarchy	358
Figure 5-8. Setting Compile Order	359
Figure 5-9. Grouping Files.	360
Figure 5-10. Start Simulation Dialog.	361
Figure 5-11. Structure Window with Projects.	361
Figure 5-12. Project Window Overview	362
Figure 5-13. Add Simulation Configuration Dialog	364
Figure 5-14. Simulation Configuration in the Project Window.	365
Figure 5-15. Add Folder Dialog.	366
Figure 5-16. Specifying a Project Folder.	366
Figure 5-17. Project Compiler Settings Dialog	367
Figure 5-18. Specifying File Properties.	368
Figure 5-19. Project Settings Dialog	369
Figure 6-1. Creating a New Library.	373
Figure 6-2. Design Unit Information in the Workspace	374
Figure 6-3. Edit Library Mapping Dialog	375
Figure 6-4. Import Library Wizard	380
Figure 7-1. VHDL Delta Delay Process	395
Figure 8-1. Fatal Signal Segmentation Violation (SIGSEGV)	447
Figure 8-2. Current Process Where Error Occurred	447

Figure 8-3. Blue Arrow Indicating Where Code Stopped Executing	447
Figure 8-4. Null Values in the Locals Window	448
Figure 9-1. SystemC Objects in GUI.	512
Figure 9-2. Breakpoint in SystemC Source	518
Figure 9-3. Setting the Allow lib step Function.	519
Figure 9-4. SystemC Objects and Processes	521
Figure 9-5. Aggregate Data Displayed in Wave Window	523
Figure 12-1. Transaction Anatomy in Wave Window.	644
Figure 12-2. Transaction Stream in Wave Window	646
Figure 12-3. Viewing Transactions and Attributes	648
Figure 12-4. Concurrent Parallel Transactions	649
Figure 12-5. Transaction in Wave Window - Viewing	649
Figure 12-6. Transaction Stream Properties	652
Figure 12-7. Changing Appearance of Attributes	653
Figure 12-8. Transactions in Objects Window	656
Figure 12-9. Recording Transactions.	658
Figure 13-1. Questa Verification IP Transaction at Different Levels of Abstraction	686
Figure 13-2. Arrays in Objects Window	690
Figure 13-3. Questa Verification IP Transactions in Wave Window	692
Figure 13-4. Concurrent Transactions Overlapping in Wave Window	694
Figure 13-5. Questa Verification IP Array (2X2) in Wave Window.	695
Figure 13-6. Color of Arrays	695
Figure 13-7. Questa Verification IP Objects in Objects Window	696
Figure 13-8. Questa Verification IP Objects in List Window	697
Figure 13-9. Viewing Questa Verification IP Relationships	698
Figure 13-10. Transaction Window - Data Tab.	700
Figure 13-11. Transaction Window - Relations Tab	701
Figure 14-1. Displaying Two Datasets in the Wave Window	704
Figure 14-2. Open Dataset Dialog Box	708
Figure 14-3. Structure Tabs	709
Figure 14-4. The Dataset Browser.	710
Figure 14-5. Dataset Snapshot Dialog	713
Figure 14-6. Virtual Objects Indicated by Orange Diamond.	715
Figure 15-1. Wave Window Object Pathnames Pane	720
Figure 15-2. Wave Window Object Values Pane	721
Figure 15-3. Wave Window Waveform Pane	721
Figure 15-4. Wave Window Cursor Pane	721
Figure 15-5. Wave Window Messages Bar	722
Figure 15-6. Tabular Format of the List Window	722
Figure 15-7. Grid and Timeline Properties	725
Figure 15-8. Cursor Properties Dialog Box	726
Figure 15-9. Find Previous and Next Transition Icons	726
Figure 15-10. Original Names of Wave Window Cursors	727
Figure 15-11. Sync All Active Cursors	727
Figure 15-12. Cursor Linking Menu	728

List of Figures

Figure 15-13. Configure Cursor Links Dialog.	728
Figure 15-14. Time Markers in the List Window	730
Figure 15-15. Waveform Pane with Collapsed Event and Delta Time	733
Figure 15-16. Waveform Pane with Expanded Time at a Specific Time	733
Figure 15-17. Waveform Pane with Event Not Logged	734
Figure 15-18. Waveform Pane with Expanded Time Over a Time Range	735
Figure 15-19. List Window After configure list -delta none Option is Used	739
Figure 15-20. List Window After configure list -delta collapse Option is Used.	739
Figure 15-21. List Window After write list -delta all Option is Used.	740
Figure 15-22. List Window After write list -event Option is Used	740
Figure 15-23. Bookmark Properties Dialog.	743
Figure 15-24. Wave Signal Search Dialog	744
Figure 15-25. Expression Builder Dialog Box	745
Figure 15-26. Selecting Signals for Expression Builder	746
Figure 15-27. Display Tab of the Wave Window Preferences Dialog Box.	748
Figure 15-28. Grid and Timeline Tab of Wave Window Preferences Dialog Box	750
Figure 15-29. Clock Cycles in Timeline of Wave Window	750
Figure 15-30. Wave Format Menu Selections.	751
Figure 15-31. Format Tab of Wave Properties Dialog	751
Figure 15-32. Changing Signal Radix	752
Figure 15-33. Global Signal Radix Dialog in Wave Window.	753
Figure 15-34. Separate Signals with Wave Window Dividers	754
Figure 15-35. Splitting Wave Window Panes	755
Figure 15-36. Wave Groups Denoted by Red Diamond	757
Figure 15-37. Modifying List Window Display Properties.	761
Figure 15-38. List Signal Properties Dialog	762
Figure 15-39. Changing the Radix in the List Window.	763
Figure 15-40. Save Format Dialog.	764
Figure 15-41. Waveform Save Between Cursors	766
Figure 15-42. Wave Filter Dialog	766
Figure 15-43. Wave Filter Dataset	767
Figure 15-44. Class Objects in the Wave Window	769
Figure 15-45. Class Waveforms	769
Figure 15-46. Class Information Popup.	770
Figure 15-47. Waveforms for Class Instances.	770
Figure 15-48. Signals Combined to Create Virtual Bus	771
Figure 15-49. Virtual Expression Builder	772
Figure 15-50. Line Triggering in the List Window	773
Figure 15-51. Setting Trigger Properties	774
Figure 15-52. Trigger Gating Using Expression Builder.	775
Figure 15-53. Modifying the Breakpoints Dialog	779
Figure 15-54. Signal Breakpoint Dialog	780
Figure 15-55. Breakpoints in the Source Window.	781
Figure 15-56. File Breakpoint Dialog Box	782
Figure 15-57. Waveform Comparison Wizard	784

Figure 15-58. Start Comparison Dialog.	785
Figure 15-59. Compare Tab in the Workspace Pane.	786
Figure 15-60. Structure Browser	787
Figure 15-61. Add Comparison by Region Dialog	787
Figure 15-62. Comparison Methods Tab	788
Figure 15-63. Adding a Clock for a Clocked Comparison	789
Figure 15-64. Waveform Comparison Options	790
Figure 15-65. Viewing Waveform Differences in the Wave Window	791
Figure 15-66. Waveform Differences in the List Window	793
Figure 15-67. Reloading and Redisplaying Compare Differences	794
Figure 16-1. Schematic Window	795
Figure 16-2. Schematic View Indicator.	798
Figure 16-3. Gray Dot Indicates Input in Process Sensitivity List	799
Figure 16-4. Left-Pointing Mouse Arrow Indicates Drivers	800
Figure 16-5. Double-Headed Arrow Indicates Inout with Drivers and Readers	801
Figure 16-6. Colors Help Identify Architectures, Modules, and Processes.	802
Figure 16-7. Show Incremental View Annotation.	803
Figure 16-8. Hover Mouse for Tooltip	803
Figure 16-9. Code Preview Window	804
Figure 16-10. Redundant Buffers and Inverters in the Incremental Schematic View.	805
Figure 16-11. Redundant Buffers and Inverters in the Full Schematic View	806
Figure 16-12. Highlight Selected Trace with Custom Color.	806
Figure 16-13. Folded Instances	807
Figure 16-14. Unfolded Instance Not Showing Contents	808
Figure 16-15. Unfolded Instance with All Contents Displayed.	808
Figure 16-16. Wave Viewer Displays Inputs and Outputs of Selected Process	809
Figure 16-17. Event Traceback Options	810
Figure 16-18. Active Time Label in Incremental View.	810
Figure 16-19. Enter Active Time Value	811
Figure 16-20. Signals for Selected Process in Embedded Wave Viewer	812
Figure 16-21. Active Driver Path Details for the <i>q</i> Signal	813
Figure 16-22. Click Schematic Window Button to View Path Details	813
Figure 16-23. Path to Root Cause	814
Figure 16-24. Unknown States Shown as Red Lines in Wave Window	815
Figure 16-25. Event Traceback Menu	816
Figure 16-26. Find Toolbar for Schematic Window	816
Figure 16-27. The Print Postscript Dialog.	821
Figure 16-28. The Schematic Page Setup Dialog	822
Figure 16-29. Configuring Incremental View Options	823
Figure 16-30. Configuring Full View Options	824
Figure 16-31. Display Options in Right-Click Menu	824
Figure 16-32. Keyboard Shortcut Table in the Schematic Window	827
Figure 17-1. The Dataflow Window	829
Figure 17-2. Dataflow Debugging Usage Flow.	830
Figure 17-3. Dot Indicates Input in Process Sensitivity Lis	833

List of Figures

Figure 17-4. Controlling Display of Redundant Buffers and Inverters	835
Figure 17-5. Green Highlighting Shows Your Path Through the Design	836
Figure 17-6. Highlight Selected Trace with Custom Color	837
Figure 17-7. Wave Viewer Displays Inputs and Outputs of Selected Process	838
Figure 17-8. Unknown States Shown as Red Lines in Wave Window	840
Figure 17-9. Dataflow: Point-to-Point Tracing	842
Figure 17-10. The Print Postscript Dialog	847
Figure 17-11. The Print Dialog	848
Figure 17-12. The Page Setup Dialog	848
Figure 17-13. Configuring Dataflow Options	849
Figure 18-1. Displaying Multiple Source Files	852
Figure 18-2. Language Templates	854
Figure 18-3. Create New Design Wizard.	855
Figure 18-4. Language Template Context Menus	856
Figure 18-5. Bookmark All Instances of a Search.	857
Figure 18-6. Setting Context from Source Files	858
Figure 18-7. Source Annotation Example	859
Figure 18-8. Time Indicator in Source Window	860
Figure 18-9. Enter an Event Time Value.	860
Figure 18-10. Popup Menu Choices for Textual Dataflow Information	862
Figure 18-11. Window Shows all Driving Processes	863
Figure 18-12. Source Readers Dialog Displays All Signal Readers	863
Figure 18-13. Coverage in Source Window	866
Figure 18-14. Breakpoint in the Source Window	868
Figure 18-15. Editing Existing Breakpoints	870
Figure 18-16. Source Code for <i>source.sv</i>	872
Figure 18-17. Preferences By - Window Tab	876
Figure 19-1. Event Traceback Toolbar Button Menu	880
Figure 19-2. Cause is Highlighted in Source Window	881
Figure 19-3. Active Driver Path Details Window	882
Figure 19-4. Active Cursor Show Time of Causal Process	882
Figure 19-5. Causal Process Highlighted in Structure Window	883
Figure 19-6. Causal Signal Highlighted in the Objects Window.	883
Figure 19-7. Time Indicator in Source Window	883
Figure 19-8. Enter an Event Time Value for Causality Tracing	884
Figure 19-9. Select Show Cause from Popup Menu	884
Figure 19-10. Selecting Show Driver from Show Cause Drop-Down Menu	886
Figure 19-11. Right-click Menu – Show Driver	886
Figure 19-12. Details of the Immediate Driving Process	886
Figure 19-13. Trace Event to Root Cause	888
Figure 19-14. Root Cause Highlighted in Source Window	888
Figure 19-15. Show X Cause.	889
Figure 19-16. Show All Possible Drivers Menu Selection	889
Figure 19-17. Possible Drivers Window	890
Figure 19-18. Selecting a Specific Time for a Trace.	890

Figure 19-19. Enter Value Dialog Box	891
Figure 19-20. Multiple Drivers	891
Figure 19-21. View Path Details Buttons	892
Figure 19-22. Causality Path Details in the Schematic Window.....	892
Figure 19-23. Time 810 Selected in Path Times Bar.....	893
Figure 19-24. Causality Path Details in the Wave Window	894
Figure 19-25. Select Preferences From Event Traceback Menu	896
Figure 19-26. Causality Trace Options	897
Figure 20-1. Enabling Code Coverage in the Start Simulation Dialog	903
Figure 20-2. Selecting Code Coverage Analysis Type	906
Figure 20-3. Focused Expression Report Sample	916
Figure 20-4. Toggle Coverage Menu.....	927
Figure 20-5. Toggle Coverage Data in the Objects Window.....	928
Figure 20-6. Sample Toggle Report.....	957
Figure 22-1. Assertion Matches.....	983
Figure 22-2. Configure Assertions Dialog.....	988
Figure 22-3. Assertion Enable Menu Selection.....	989
Figure 22-4. Selecting Profile On from Capacity Pane	989
Figure 22-5. Selecting Message Logging	991
Figure 22-6. Setting Immediate Assertion Break Severity	991
Figure 22-7. Enabling/Disabling Failure or Pass Logging	992
Figure 22-8. Setting Assertion Failure Limits	993
Figure 22-9. Set Assertion Actions	994
Figure 22-10. Configure Selected Cover Directives Dialog	995
Figure 22-11. Failure Counts in the Assertions Window	998
Figure 22-12. Assertion Failures Indicated by Red Triangles.....	998
Figure 22-13. Assertion Counts in the Assertions Window	999
Figure 22-14. Enable Assertion Coverage.....	999
Figure 22-15. Assertion Indicators When -assertdebug is Used	1000
Figure 22-16. Counts Columns in Assertions Window	1001
Figure 22-17. SystemVerilog Assertions in the Assertions Window	1002
Figure 22-18. Assertion Failures Appear in Red	1003
Figure 22-19. Hierarchy Display Mode.....	1003
Figure 22-20. PSL Cover Directives in the Cover Directives Window.....	1004
Figure 22-21. SystemVerilog Assert and Cover Directives in the Wave Window	1007
Figure 22-22. PSL Assert and Cover Directives in the Wave Window.....	1008
Figure 22-23. Antecedent Matches Indicated by Yellow Triangle	1009
Figure 22-24. Viewing Cover Directive Waveforms in Count Mode	1010
Figure 22-25. Assertions Report Dialog	1013
Figure 22-26. Usage Flow for PSL Assertions	1014
Figure 22-27. Usage Flow for SystemVerilog Assertions.....	1028
Figure 22-28. Enable Assertion Debug	1030
Figure 22-29. Enable ATV Menu Selection	1031
Figure 22-30. Assertion Debug Pane in Wave Window	1035
Figure 22-31. The ActiveCount Object in the Wave Window - SystemVerilog.....	1036

List of Figures

Figure 22-32. Selecting Assertion Thread Start Time	1037
Figure 22-33. Opening ATV from the Wave Window	1038
Figure 22-34. Opening ATV from the Message Viewer Window.	1039
Figure 22-35. ATV Panes - SystemVerilog.	1040
Figure 22-36. Failed Expressions Highlighted Red.	1041
Figure 22-37. Values of Local Variables.	1042
Figure 22-38. View Thread at 150 ns	1044
Figure 22-39. ATV Window Shows Where Boolean Sub-Expression Failed.	1044
Figure 22-40. ATV Window Displays All Clock Expressions	1045
Figure 22-41. Root Thread Analysis of a Directive Failure	1046
Figure 22-42. Root Thread Analysis	1047
Figure 22-43. ATV Mouse Hover Information	1047
Figure 22-44. ATV Mouse Hover Information - SystemVerilog	1048
Figure 22-45. Local Variables Annotation in Thread Viewer Pane	1048
Figure 23-1. SystemVerilog Functional Coverage Flow	1050
Figure 23-2. Functional Coverage Statistics in Covergroups Window	1060
Figure 23-3. Creating Functional Coverage Text Reports.	1062
Figure 23-4. Filter Setup Dialog	1064
Figure 23-5. Create Filter Dialog.	1065
Figure 23-6. Add-Modify Select Criteria Dialog.	1065
Figure 23-7. Copy and Rename Filter Dialogs	1066
Figure 25-1. Verification of a Design	1088
Figure 25-2. Aggregated Coverage Data in the Structure Window.	1094
Figure 25-3. Command Setup Dialog Box	1103
Figure 25-4. File Merge Dialog	1106
Figure 25-5. original.ucdb, dut.ucdb and tb.ucdb	1112
Figure 25-6. Test Data in Verification Browser Window	1121
Figure 25-7. Coverage Report Text Dialog	1124
Figure 25-8. Coverage HTML Report Dialog	1126
Figure 25-9. HTML Coverage Report	1127
Figure 25-10. Coverage Exclusions Report Dialog	1129
Figure 25-11. Filtering Displayed UCDB Data	1130
Figure 25-12. Filtering on User Attributes	1132
Figure 26-1. Specifying Path in C Debug setup Dialog.	1137
Figure 26-2. Setting Breakpoints in Source Code	1139
Figure 26-3. Right Click Pop-up Menu on Breakpoint	1139
Figure 26-4. Simulation Stopped at Breakpoint on PLI Task	1143
Figure 26-5. Stepping into Next File	1144
Figure 26-6. Function Pointer to Foreign Architecture	1145
Figure 26-7. Highlighted Line in Associated File	1146
Figure 26-8. Stop on quit Button in Dialog	1149
Figure 27-1. Status Bar: Profile Samples.	1154
Figure 27-2. Ranked Window	1156
Figure 27-3. Design Units Window	1157
Figure 27-4. Calltree Window	1158

Figure 27-5. Structural Window	1159
Figure 27-6. Expand and Collapse Selections in Popup Menu	1159
Figure 27-7. Profile Details Window: Function Usage	1160
Figure 27-8. Profile Details Window: Instance Usage	1160
Figure 27-9. Profile Details Window: Callers and Callees	1161
Figure 27-10. Accessing Source from Profile Views	1162
Figure 27-11. Profile Report Example.	1164
Figure 27-12. Profile Report Dialog Box	1165
Figure 27-13. Displaying Capacity Objects in the Wave Window	1169
Figure 29-1. JobSpy Job Manager	1200
Figure 29-2. Job Manager View Waveform	1201
Figure 30-1. Waveform Editor: Library Window	1206
Figure 30-2. Opening Waveform Editor from Structure or Objects Windows	1207
Figure 30-3. Create Pattern Wizard	1208
Figure 30-4. Wave Edit Toolbar	1209
Figure 30-5. Manipulating Waveforms with the Wave Edit Toolbar and Cursors	1211
Figure 30-6. Export Waveform Dialog	1213
Figure 30-7. Evcd Import Dialog.	1214
Figure 31-1. SDF Tab in Start Simulation Dialog	1218
Figure 33-1. TDebug Choose Dialog.	1272
Figure 33-2. Tcl Debugger for vsim	1273
Figure 33-3. Setting a Breakpoint in the Debugger	1274
Figure 33-4. Variables Dialog Box	1275
Figure A-1. Runtime Options Dialog: Defaults Tab	1279
Figure A-2. Runtime Options Dialog Box: Severity Tab	1280
Figure A-3. Runtime Options Dialog Box: WLF Files Tab	1281
Figure D-1. DPI Use Flow Diagram	1413
Figure E-1. Schematic Window Keyboard Shortcuts	1445
Figure F-1. Configure Column Layout Dialog	1452
Figure F-2. Edit Column Layout Dialog	1453
Figure F-3. Create Column Layout Dialog	1453
Figure F-4. Change Text Fonts for Selected Windows	1455
Figure F-5. Making Global Font Changes	1455

List of Tables

Table 1-1. Simulation Tasks	59
Table 1-2. Use Modes for ModelSim	64
Table 1-3. Possible Definitions of an Object, by Language	66
Table 1-4. Text Conventions	69
Table 1-5. Documentation List	69
Table 1-6. Deprecated Features	71
Table 1-7. Deprecated Commands	71
Table 1-8. Deprecated Command Arguments	71
Table 2-1. GUI Windows	73
Table 2-2. Design Object Icons	76
Table 2-3. Icon Shapes and Design Object Types	77
Table 2-4. Graphic Elements of Toolbar in Find Mode	79
Table 2-5. Graphic Elements of Toolbar in Filter Mode	80
Table 2-6. Information Displayed in Status Bar	101
Table 2-7. File Menu — Item Description	105
Table 2-8. Edit Menu — Item Description	107
Table 2-9. View Menu — Item Description	107
Table 2-10. Compile Menu — Item Description	108
Table 2-11. Simulate Menu — Item Description	108
Table 2-12. Add Menu — Item Description	110
Table 2-13. Tools Menu — Item Description	110
Table 2-14. Layout Menu — Item Description	112
Table 2-15. Bookmarks Menu — Item Description	112
Table 2-16. Window Menu — Item Description	112
Table 2-17. Help Menu — Item Description	114
Table 2-18. ATV Toolbar Buttons	116
Table 2-19. Analysis Toolbar Buttons	117
Table 2-20. Bookmarks Toolbar Buttons	117
Table 2-21. Change Column Toolbar Buttons	118
Table 2-22. Compile Toolbar Buttons	119
Table 2-23. Coverage Toolbar Buttons	120
Table 2-24. FSM Toolbar Buttons	121
Table 2-25. Help Toolbar Buttons	122
Table 2-26. Layout Toolbar Buttons	123
Table 2-27. Memory Toolbar Buttons	123
Table 2-28. Mode Toolbar Buttons	123
Table 2-29. Objectfilter Toolbar Buttons	124
Table 2-30. Precision Toolbar Buttons	125
Table 2-31. Process Toolbar Buttons	125
Table 2-32. Profile Toolbar Buttons	126

Table 2-33. Schematic Toolbar Buttons	127
Table 2-34. Simulate Toolbar Buttons	128
Table 2-35. Source Toolbar Buttons	131
Table 2-36. Standard Toolbar Buttons	132
Table 2-37. Step Toolbar Buttons	134
Table 2-38. Wave Toolbar Buttons	135
Table 2-39. Wave Compare Toolbar Buttons	136
Table 2-40. Wave Cursor Toolbar Buttons	136
Table 2-41. Wave Edit Toolbar Buttons	137
Table 2-42. Wave Expand Time Toolbar Buttons	138
Table 2-43. Zoom Toolbar Buttons	139
Table 2-44. Assertions Window Columns	141
Table 2-45. Assertions Window Popup Menu	144
Table 2-46. Graphic Symbols for Current Directive State	147
Table 2-47. Graphic Symbols for Clock, Thread, and Directive Status	147
Table 2-48. ATV Window Popup Menu	148
Table 2-49. Commands Related to the Call Stack Window	150
Table 2-50. Call Stack Window Columns	150
Table 2-51. Capacity Window Columns	151
Table 2-52. Class Graph Window Popup Menu	154
Table 2-53. Class Instances Window Popup Menu	156
Table 2-54. Class Tree Window Icons	157
Table 2-55. Class Tree Window Columns	158
Table 2-56. Class Tree Window Popup Menu	158
Table 2-57. Actions in Code Coverage Analysis Title Bar	159
Table 2-58. Cover Directives Window Columns	167
Table 2-59. Covergroups Window Columns	170
Table 2-60. Files Window Columns	177
Table 2-61. Files Window Popup Menu	178
Table 2-62. Files Menu	179
Table 2-63. FSM List Window Columns	180
Table 2-64. FSM List Window Popup Menu	181
Table 2-65. FSM List Menu	181
Table 2-66. FSM Viewer Window — Graphical Elements	186
Table 2-67. FSM View Window Popup Menu	187
Table 2-68. FSM View Menu	187
Table 2-69. Columns in the Instance Coverage Window	190
Table 2-70. Instance Coverage Popup Menu	194
Table 2-71. Library Window Columns	195
Table 2-72. Library Window Popup Menu	195
Table 2-73. List Window Popup Menu	200
Table 2-74. Locals Window Columns	202
Table 2-75. Locals Window Popup Menu	202
Table 2-76. Memory Identification	204
Table 2-77. Memory List Window Columns	206

List of Tables

Table 2-78. Memory List Popup Menu	207
Table 2-79. Memories Menu	207
Table 2-80. Memory Data Popup Menu — Address Pane	209
Table 2-81. Memory Data Popup Menu — Data Pane	209
Table 2-82. Memory Data Menu	210
Table 2-83. Message Viewer Tasks	213
Table 2-84. Message Viewer Window Columns	214
Table 2-85. Message Viewer Window Popup Menu	215
Table 2-86. Objects Window Popup Menu	220
Table 2-87. Toggle Coverage Columns in the Objects Window	222
Table 2-88. Processes Window Column Descriptions	226
Table 2-89. Profile Calltree Window Column Descriptions	231
Table 2-90. Incremental View Popup Menu	239
Table 2-91. Full View Popup Menu	241
Table 2-92. Source Window Code Coverage Indicators	249
Table 2-93. Columns in the Structure Window	270
Table 2-94. Verification Browser Icons	275
Table 2-95. Analog Sidebar Icons	292
Table 2-96. Icons and Actions	293
Table 3-1. Compile Options for the -nodebug Compiling	324
Table 8-1. Evaluation 1 of always Statements	442
Table 8-2. Evaluation 2 of always Statement	443
Table 8-3. IEEE Std 1364 System Tasks and Functions - 1	462
Table 8-4. IEEE Std 1364 System Tasks and Functions - 2	462
Table 8-5. IEEE Std 1364 System Tasks	463
Table 8-6. IEEE Std 1364 File I/O Tasks	464
Table 8-7. SystemVerilog System Tasks and Functions - 1	464
Table 8-8. SystemVerilog System Tasks and Functions - 2	464
Table 8-9. SystemVerilog System Tasks and Functions - 3	465
Table 8-10. SystemVerilog System Tasks and Functions - 4	465
Table 8-11. Simulator-Specific Verilog System Tasks and Functions	466
Table 9-1. Supported Platforms for SystemC	489
Table 9-2. Custom gcc Platform Requirements	490
Table 9-3. Generated Extensions for Each Object Type	505
Table 9-4. Time Unit and Simulator Resolution	513
Table 9-5. Viewable SystemC Objects	516
Table 9-6. Mixed-language Compares	517
Table 9-7. Simple Conversion: sc_main to Module	532
Table 9-8. Using sc_main and Signal Assignments	533
Table 9-9. Modifications Using SCV Transaction Database	534
Table 10-1. VHDL Types Mapped To SystemVerilog Port Vectors	552
Table 10-2. SystemVerilog-to-VHDL Data Type Mapping	562
Table 10-3. Verilog Parameter to VHDL Mapping	564
Table 10-4. Verilog States Mapped to std_logic and bit	565
Table 10-5. VHDL to SystemVerilog Data Type Mapping	567

Table 10-6. VHDL Generics to Verilog Mapping	567
Table 10-7. Mapping VHDL bit to Verilog States	568
Table 10-8. Mapping VHDL std_logic Type to Verilog States	568
Table 10-9. Mapping Table for Verilog-style Declarations	570
Table 10-10. Mapping Table for SystemVerilog-style Declarations	571
Table 10-11. Channel and Port Type Mapping	574
Table 10-12. Data Type Mapping – SystemC to Verilog or SystemVerilog	575
Table 10-13. Data Type Mapping – Verilog or SystemVerilog to SystemC	577
Table 10-14. Mapping Verilog Port Directions to SystemC	580
Table 10-15. Mapping Verilog States to SystemC States	580
Table 10-16. Mapping SystemC bool to Verilog States	581
Table 10-17. Mapping SystemC sc_bit to Verilog States	582
Table 10-18. Mapping SystemC sc_logic to Verilog States	582
Table 10-19. SystemC Port Type Mapping	582
Table 10-20. Mapping Between SystemC sc_signal and VHDL Types	583
Table 10-21. Mapping VHDL Port Directions to SystemC	587
Table 10-22. Mapping VHDL std_logic States to SystemC States	587
Table 10-23. Mapping SystemC bool to VHDL Boolean States	588
Table 10-24. Mapping SystemC sc_bit to VHDL bit	588
Table 10-25. Mapping SystemC sc_logic to VHDL std_logic	588
Table 10-26. Mapping Literals from VHDL to SystemVerilog	596
Table 10-27. Supported Types Inside VHDL Records	596
Table 10-28. Supported Types Inside SystemVerilog Structure	598
Table 10-29. SystemC Types as Represented in SystemVerilog	627
Table 11-1. Checkpoint and Restore Commands	633
Table 12-1. System Tasks and API for Recording Transactions	659
Table 13-1. Questa Verification IP Objects	688
Table 13-2. Questa Verification IP Colors and Causation	693
Table 14-1. WLF File Parameters	705
Table 14-2. Structure Tab Columns	709
Table 14-3. vsim Arguments for Collapsing Time and Delta Steps	714
Table 15-1. Actions for Cursors	724
Table 15-2. Actions for Time Markers	730
Table 15-3. Recording Delta and Event Time Information	731
Table 15-4. Menu Selections for Expanded Time Display Modes	736
Table 15-5. Actions for Bookmarks	742
Table 15-6. Actions for Dividers	754
Table 15-7. Triggering Options	774
Table 15-8. Mixed-Language Waveform Compares	783
Table 16-1. Icon and Menu Selections for Exploring Design Connectivity	801
Table 16-2. Code Preview Toolbar Buttons	804
Table 16-3. Schematic Window Links to Other Windows and Panes	820
Table 16-4. Keyboard Shortcuts	826
Table 17-1. Icon and Menu Selections for Exploring Design Connectivity	833
Table 17-2. Dataflow Window Links to Other Windows and Panes	846

List of Tables

Table 19-1. Setting Causality Traceback Report Destination	895
Table 19-2. Text Report Formatting	895
Table 20-1. Code Coverage in Windows	907
Table 20-2. AND Gate	916
Table 20-3. XOR Gate	917
Table 20-4. Condition UDP Truth Table for Line 180	921
Table 20-5. Condition UDP Truth Table for Line 38	922
Table 20-6. Expression UDP Truth Table for line 236	923
Table 21-1. Commands Used for FSM Coverage Collection	968
Table 21-2. Commands Used to Capture FSM Debug Information	972
Table 21-3. FSM Coverage Columns	974
Table 21-4. Additional FSM-Related Arguments	975
Table 21-5. Recognized FSM Note Parameters	976
Table 21-6. FSM Recognition Info Note Parameters	977
Table 22-1. Graphic Elements for Assertions and Cover Directives	1009
Table 23-1. Questa SIM and SystemVerilog IEEE 1800-2009 Options	1052
Table 23-2. Option Settings and Coverage Results	1055
Table 24-1. Attributes Usable with randomize()	1078
Table 25-1. Coverage Calculation for each Coverage Type	1092
Table 25-2. Coverage Modes	1095
Table 25-3. Predefined Fields in UCDB Test Attribute Record	1099
Table 26-1. Supported Platforms and gdb Versions	1136
Table 26-2. Simulation Stepping Options in C Debug	1139
Table 26-3. Command Reference for C Debug	1149
Table 27-1. How to Enable and View Capacity Analysis	1167
Table 28-1. Signal Spy Reference Comparison	1175
Table 29-1. SIMulation Commands You can Issue from JobSpy	1197
Table 30-1. Signal Attributes in Create Pattern Wizard	1208
Table 30-2. Waveform Editing Commands	1209
Table 30-3. Selecting Parts of the Waveform	1210
Table 30-4. Wave Editor Mouse/Keyboard Shortcuts	1212
Table 30-5. Formats for Saving Waveforms	1213
Table 30-6. Examples for Loading a Stimulus File	1213
Table 31-1. Matching SDF to VHDL Generics	1220
Table 31-2. Matching SDF IOPATH to Verilog	1224
Table 31-3. Matching SDF INTERCONNECT and PORT to Verilog	1224
Table 31-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog	1225
Table 31-5. Matching SDF DEVICE to Verilog	1225
Table 31-6. Matching SDF SETUP to Verilog	1225
Table 31-7. Matching SDF HOLD to Verilog	1225
Table 31-8. Matching SDF SETUPHOLD to Verilog	1226
Table 31-9. Matching SDF RECOVERY to Verilog	1226
Table 31-10. Matching SDF REMOVAL to Verilog	1226
Table 31-11. Matching SDF RECREM to Verilog	1226
Table 31-12. Matching SDF SKEW to Verilog	1226

Table 31-13. Matching SDF WIDTH to Verilog	1227
Table 31-14. Matching SDF PERIOD to Verilog	1227
Table 31-15. Matching SDF NOCHANGE to Verilog	1227
Table 31-16. RETAIN Delay Usage (default)	1228
Table 31-17. RETAIN Delay Usage (with +vlog_retain_same2same_on)	1228
Table 31-18. Matching Verilog Timing Checks to SDF SETUP	1229
Table 31-19. SDF Data May Be More Accurate Than Model	1229
Table 31-20. Matching Explicit Verilog Edge Transitions to Verilog	1229
Table 31-21. SDF Timing Check Conditions	1230
Table 31-22. SDF Path Delay Conditions	1230
Table 31-23. Disabling Timing Checks	1231
Table 32-1. VCD Commands and SystemTasks	1240
Table 32-2. VCD Dumpport Commands and System Tasks	1240
Table 32-3. VCD Commands and System Tasks for Multiple VCD Files	1241
Table 32-4. SystemC Types	1242
Table 32-5. Driver States	1245
Table 32-6. State When Direction is Unknown	1245
Table 32-7. Driver Strength	1246
Table 32-8. VCD Values When Force Command is Used	1247
Table 32-9. Values for file_format Argument	1249
Table 32-10. Sample Driver Data	1250
Table 33-1. Changes to ModelSim Commands	1252
Table 33-2. Tcl Backslash Sequences	1254
Table 33-3. Tcl List Commands	1261
Table 33-4. Simulator-Specific Tcl Commands	1262
Table 33-5. Tcl Time Conversion Commands	1263
Table 33-6. Tcl Time Relation Commands	1263
Table 33-7. Tcl Time Arithmetic Commands	1264
Table 33-8. Commands for Handling Breakpoints and Errors in Macros	1269
Table 33-9. Tcl Debug States	1274
Table A-1. Runtime Option Dialog: Defaults Tab Contents	1279
Table A-2. Runtime Option Dialog: Severity Tab Contents	1281
Table A-3. Runtime Option Dialog: WLF Files Tab Contents	1282
Table A-4. Commands for Overriding the Default Initialization File	1283
Table A-5. License Variable: License Options	1321
Table A-6. MessageFormat Variable: Accepted Values	1323
Table C-1. Severity Level Types	1393
Table C-2. Exit Codes	1396
Table D-1. VPI Compatibility Considerations	1408
Table D-2. vsim Arguments for DPI Application Using External Compilation Flows	1425
Table D-3. Supported VHDL Objects	1431
Table D-4. Supported ACC Routines	1433
Table D-5. Supported TF Routines	1435
Table D-6. Values for action Argument	1437
Table E-1. Command History Shortcuts	1441

List of Tables

Table E-2. Mouse Shortcuts 1442
Table E-3. Keyboard Shortcuts 1443
Table E-4. List Window Keyboard Shortcuts 1446
Table E-5. Wave Window Mouse Shortcuts 1446
Table E-6. Wave Window Keyboard Shortcuts 1447
Table F-1. Global Fonts 1455
Table G-1. Files Accessed During Startup 1459
Table G-2. Add Library Mappings to modelsim.ini File 1468

Chapter 1

Introduction

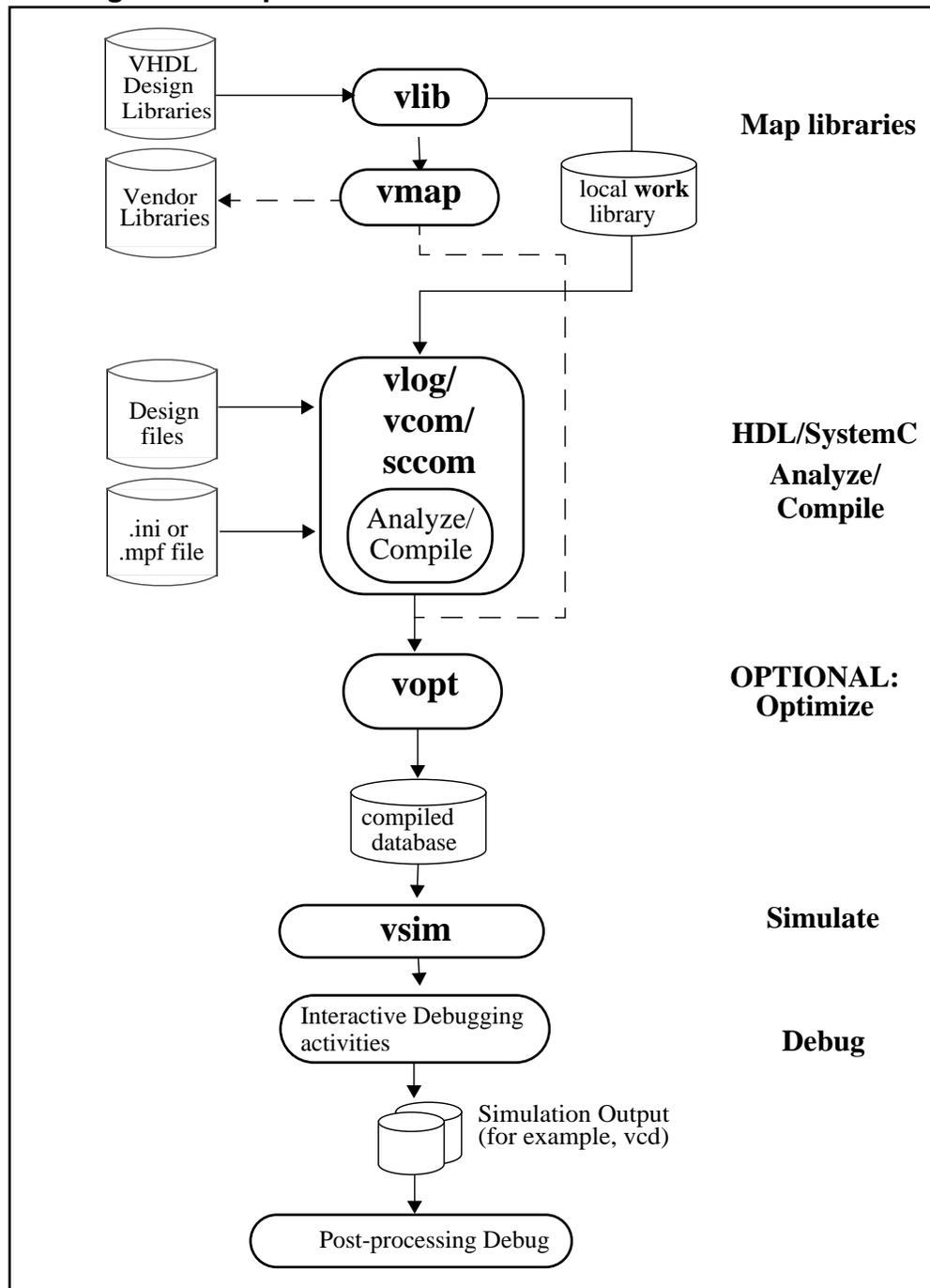
Documentation for ModelSim is intended for users of UNIX, Linux, and Microsoft Windows.

Not all versions of ModelSim are supported on all platforms. For more information on your platform or operating system, contact your Mentor Graphics sales representative.

Operational Structure and Flow

[Figure 1-1](#) illustrates the structure and general usage flow for verifying a design with ModelSim.

Figure 1-1. Operational Structure and Flow of ModelSim



Simulation Task Overview

The following table provides a reference for the tasks required for compiling, optimizing, loading, and simulating a design in ModelSim.

Table 1-1. Simulation Tasks

Task	Example Command Line Entry	GUI Menu Pull-down	GUI Icons
Step 1: Map libraries	vlib <library_name> vmap work <library_name>	1. File > New > Project 2. Enter library name 3. Add design files to project	N/A
Step 2: Compile the design	vlog file1.v file2.v ... (Verilog) vcom file1.vhd file2.vhd ... (VHDL) sccom <top> (SystemC) sccom -link <top>	Compile > Compile or Compile > Compile All	Compile or Compile All  
Step 3: Optimize the design (OPTIONAL)	Optimized when voptflow = 1 in modelsim.ini file (default setting for version 6.2 and later.	To disable optimizations: 1. Simulate > Start Simulation 2. Deselect Enable Optimization button To set optimization options: 1. Simulate > Design Optimization 2. Set desired optimizations	N/A
Step 4: Load the design into the simulator	vsim <top> or vsim <opt_name>	1. Simulate > Start Simulation 2. Click on top design module or optimized design unit name 3. Click OK This action loads the design for simulation.	Simulate 
Step 5: Run the simulation	run step	Simulate > Run	Run , or Run continue , or Run -all   

Table 1-1. Simulation Tasks

Task	Example Command Line Entry	GUI Menu Pull-down	GUI Icons
Step 6: Debug the design Note: Design optimization in step 3 limits debugging visibility	Common debugging commands: bp describe drivers examine force log show	N/A	N/A

Basic Steps for Simulation

This section describes the basic procedure for simulating your design using ModelSim.

Step 1 — Collect Files and Map Libraries

Files needed to run ModelSim on your design:

- design files (VHDL, Verilog, and/or SystemC), including stimulus for the design
- libraries, both working and resource
- *modelsim.ini* file (automatically created by the library mapping command)

For detailed information on the files accessed during system startup (including the *modelsim.ini* file), initialization sequences, and system environment variables, see the Appendix entitled “[System Initialization](#)”.

Providing Stimulus to the Design

You can provide stimulus to your design in several ways:

- Language-based test bench
- Tcl-based ModelSim interactive command, [force](#)

- VCD files / commands

See [Creating a VCD File](#) and [Using Extended VCD as Stimulus](#)

- Third-party test bench generation tools

What is a Library?

A library is a location on your file system where ModelSim stores data to be used for simulation. ModelSim uses one or more libraries to manage the creation of data before it is needed for use in simulation. A library also helps to streamline simulation invocation. Instead of compiling all design data each time you simulate, ModelSim uses binary pre-compiled data from its libraries. For example, if you make changes to a single Verilog module, ModelSim recompiles only that module, rather than all modules in the design.

Work and Resource Libraries

You can use design libraries in two ways:

- As a local working library that contains the compiled version of your design
- As a resource library

The contents of your working library will change as you update your design and recompile. A resource library is typically unchanging, and serves as a parts source for your design. Examples of resource libraries are shared information within your group, vendor libraries, packages, or previously compiled elements of your own working design. You can create your own resource libraries, or they may be supplied by another design team or a third party (for example, a silicon vendor).

For more information on resource libraries and working libraries, refer to [Working Library Versus Resource Libraries](#), [Managing Library Contents](#), [Working with Design Libraries](#), and [Specifying Resource Libraries](#).

Creating the Logical Library (vlib)

Before you can compile your source files, you must create a library in which to store the compilation results. You can create the logical library using the GUI, by choosing **File > New > Library** from the main menu (see [Creating a Library](#)), or you can use the `vlib` command. For example, the following command:

```
vlib work
```

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

Mapping the Logical Work to the Physical Work Directory (vmap)

VHDL uses logical library names that can be mapped to ModelSim library directories. If libraries are not mapped properly, and you invoke your simulation, necessary components will not be loaded and simulation will fail. Similarly, compilation can also depend on proper library mapping.

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

You can use the GUI ([Library Mappings with the GUI](#)), a command ([Library Mapping from the Command Line](#)), or a project ([Getting Started with Projects](#)) to assign a logical name to a design library.

The format for command line entry is:

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

This command sets the mapping between a logical library name and a directory.

Use braces ({}) for cases where the path contains multiple items that need to be escaped, such as spaces in the pathname or backslash characters. For example:

```
vmap celllib {$LIB_INSTALL_PATH/Documents And Settings/All/celllib}
```

Step 2 — Compile the Design

To compile a design, run one of the following ModelSim commands, depending on the language used to create the design:

- vlog — Verilog
- vcom — VHDL
- sccom — SystemC

Compiling Verilog (vlog)

The **vlog** command compiles Verilog modules in your design. You can compile Verilog files in any order, since they are not order dependent. See [Verilog Compilation](#) for details.

Compiling VHDL (vcom)

The **vcom** command compiles VHDL design units. You must compile VHDL files in the order necessitate to any design requirements. Projects may assist you in determining the compile order: for more information, see [Auto-Generating Compile Order](#). See [Compilation and Simulation of VHDL](#) for details on VHDL compilation.

Compiling SystemC (sccom)

The **sccom** command compiles SystemC design units. Use this command only if you have SystemC components in your design. See [Compiling SystemC Files](#) for details.

Step 3 — Load the Design for Simulation

Running the vsim Command on the Top Level of the Design

After you have compiled your design, it is ready for simulation. You can then run the `vsim` command using the names of any top-level modules (many designs contain only one top-level module). For example, if your top-level modules are named “testbench” and “globals,” then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references.

You can optionally optimize the design with `vopt`. For more information on optimization, see [Optimizing Designs with vopt](#).

Using Standard Delay Format Files

You can incorporate actual delay values to the simulation by applying standard delay format (SDF) back-annotation files to the design. For more information on how SDF is used in the design, see [Specifying SDF Files for Simulation](#).

Step 4 — Simulate the Design

Once you have successfully loaded the design, simulation time is set to zero, and you must enter a `run` command to begin simulation. For more information, see [Verilog and SystemVerilog Simulation](#), [SystemC Simulation](#), and [VHDL Simulation](#).

The basic commands you use to run simulation are:

- `add wave`
- `bp`
- `force`
- `run`
- `step`

Step 5 — Debug the Design

The ModelSim GUI provides numerous commands, operations, and windows useful in debugging your design. In addition, you can also use the command line to run the following basic simulation commands for debugging:

- [describe](#)
- [drivers](#)
- [examine](#)
- [force](#)
- [log](#)
- [checkpoint](#)
- [restore](#)
- [show](#)

Modes of Operation

Many users run ModelSim interactively with the graphical user interface (GUI)—using the mouse to perform actions from the main menu or in dialog boxes. However, there are really three modes of ModelSim operation, as described in [Table 1-2](#).

Table 1-2. Use Modes for ModelSim

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

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

A command is available to help batch users access commands not available for use in batch mode. Refer to the [batch_mode](#) command in the ModelSim Reference Manual for more details.

Command Line Mode

In command line mode ModelSim executes any startup command specified by the [Startup](#) variable in the *modelsim.ini* file. If [vsim](#) 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 after the design loads (see the example below). The **transcript on** command writes all of the commands you invoke to the transcript file.

For example, the following series of commands results in a transcript file that can be used for command input if *top* is re-simulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages... then execute:

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

Rename a transcript file that you intend to use as a DO file—if you do not rename it, ModelSim will overwrite it the next time you run **vsim**. 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 pound sign (#).

Refer to [Creating a Transcript File](#) for more information about creating, locating, and saving a transcript file.

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

Basic Command Line Editing and Navigation

While in command line mode you can use basic command line editing and navigation techniques similar to other command line environments, such as:

- History navigation — use the up and down arrows to select commands you have already used.
- Command line editing — use the left and right arrows to edit your current command line.
- Filename completion — use the Tab key to expand filenames.

Batch Mode

Batch mode is an operational mode that provides neither an interactive command line nor interactive windows. In a Windows environment, you run **vsim** from a Windows command prompt and standard input and output are redirected to and from files.

In a UNIX environment, you can invoke **vsim** in batch mode by redirecting standard input using the “here-document” technique.

Here is an example of the “here-document” technique:

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

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

```
vsim counter <yourfile >outfile
```

where “yourfile” represents a script containing various ModelSim commands, and the angle brackets (<>) are redirection indicators.

You can use the CTRL-C keyboard interrupt to terminate batch simulation in UNIX and Windows environments.

Definition of an Object

Because ModelSim supports a variety of design languages (SystemC, Verilog, VHDL, SystemVerilog, and PSL), the word “object” is used to refer to any valid design element in those languages, whenever a specific language reference is not needed. [Table 1-3](#) summarizes the language constructs that an object can refer to.

Table 1-3. Possible Definitions of an Object, by Language

Design Language	An object can be
VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, alias, variable
Verilog	function, module instantiation, named fork, named begin, net, task, register, variable
SystemVerilog	In addition to those listed above for Verilog: class, package, program, interface, array, directive, property, sequence
SystemC	module, channel, port, variable, aggregate

Table 1-3. Possible Definitions of an Object, by Language

Design Language	An object can be
PSL	property, sequence, directive, endpoint

Graphic Interface Overview

While your operating system interface provides the window-management frame, ModelSim controls all internal window features including menus, buttons, and scroll bars. Because the graphical interface is based on Tcl/TK, you also have the capability to build your own simulation environment. Preference variables and configuration commands (see [modelsim.ini Variables](#) for details) give you control over the use and placement of windows, menus, menu options, and buttons. See [Tcl and Macros \(DO Files\)](#) for more information on Tcl.

Standards Supported

Standards documents are sometimes informally referred to as the Language Reference Manual (LRM). This standards listed here are the complete name of each manual. Elsewhere in this manual the individual standards are referenced using the IEEE Std number.

The following standards are supported for the ModelSim products:

- VHDL —
 - IEEE Std 1076-2008, *IEEE Standard VHDL Language Reference Manual*.
 ModelSim supports a subset of the VHDL 2008 standard features. For detailed standard support information see the vhd12008 technote available at `<install_dir>/docs/technotes/vhd12008.note`, or from the GUI menu pull-down **Help** > **Technotes** > **vhd12008**.

 Potential migration issues and mixing use of VHDL 2008 with older VHDL code are addressed in the vhd12008migration technote.
 - IEEE Std 1164-1993, *Standard Multivalued Logic System for VHDL Model Interoperability*
 - IEEE Std 1076.2-1996, *Standard VHDL Mathematical Packages*

Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specifications.
- Verilog/SystemVerilog —
 - IEEE Std 1364-2005, *IEEE Standard for Verilog Hardware Description Language*
 - IEEE Std 1800-2009, *IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language*

Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim users.

- SDF and VITAL —
 - SDF – IEEE Std 1497-2001, *IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process*
 - VITAL 2000 – IEEE Std 1076.4-2000, *IEEE Standard for VITAL ASIC Modeling Specification*
- SystemC —
 - IEEE Std 1666-2005, SystemC Language Reference Manual
- Unified Power Format (UPF) —
 - (UPF 1.0) The Accellera Unified Power Format (UPF) Standard Version 1.0 – February 22, 2007
 - (UPF 2.0) IEEE Std 1801-2009 – Standard for Design and Verification of Low Power Integrated Circuits – March 27, 2009

ModelSim supports most of UPF 1.0 features and some of UPF 2.0 features. Support details are summarized in the [Power Aware User's Manual](#).

- PSL —
 - IEEE Std 1850-2005, *IEEE Standard for Property Specific Language (PSL)*. For exceptions, see the [Verification with Assertions and Cover Directives](#) chapter.

Assumptions

Using the ModelSim product and its documentation is based on the following assumptions:

- You are familiar with how to use your operating system and its graphical interface.
- You have a working knowledge of the design languages. Although ModelSim is an excellent application to use while learning HDL concepts and practices, this document is not written to support that goal.
- You have worked through the appropriate lessons in the ModelSim Tutorial and are familiar with the basic functionality of ModelSim. You can find the ModelSim Tutorial by choosing Help from the main menu.

Text Conventions

Table 1-4 lists the text conventions used in this manual.

Table 1-4. Text Conventions

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

Installation Directory Pathnames

When referring to installation paths, this manual uses “<installdir>” as a generic representation of the installation directory for all versions of ModelSim. The actual installation directory on your system may contain version information.

Where to Find ModelSim Documentation

Table 1-5. Documentation List

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

Table 1-5. Documentation List

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

Mentor Graphics Support

Mentor Graphics product support includes software enhancements, technical support, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service. For details, refer to the following location on the Worldwide Web:

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

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

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

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

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

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

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

Deprecated Features, Commands, and Variables

This section provides tables of features, commands, command arguments, and *modelsim.ini* variables that have been superseded by new versions. Although you can still use superseded features, commands, arguments, or variables, Mentor Graphics deprecates their usage—you should use the corresponding new version whenever possible or convenient.

The following tables indicate the in which the item was superseded and a link to the new item that replaces it, where applicable.

Table 1-6. Deprecated Features

Feature	Version	New Feature / Information
Support for Solaris SPARC and Solaris x86 operating systems	10.1	None. Solaris is no longer supported.

Table 1-7. Deprecated Commands

Command	Version	New Command / Information

Table 1-8. Deprecated Command Arguments

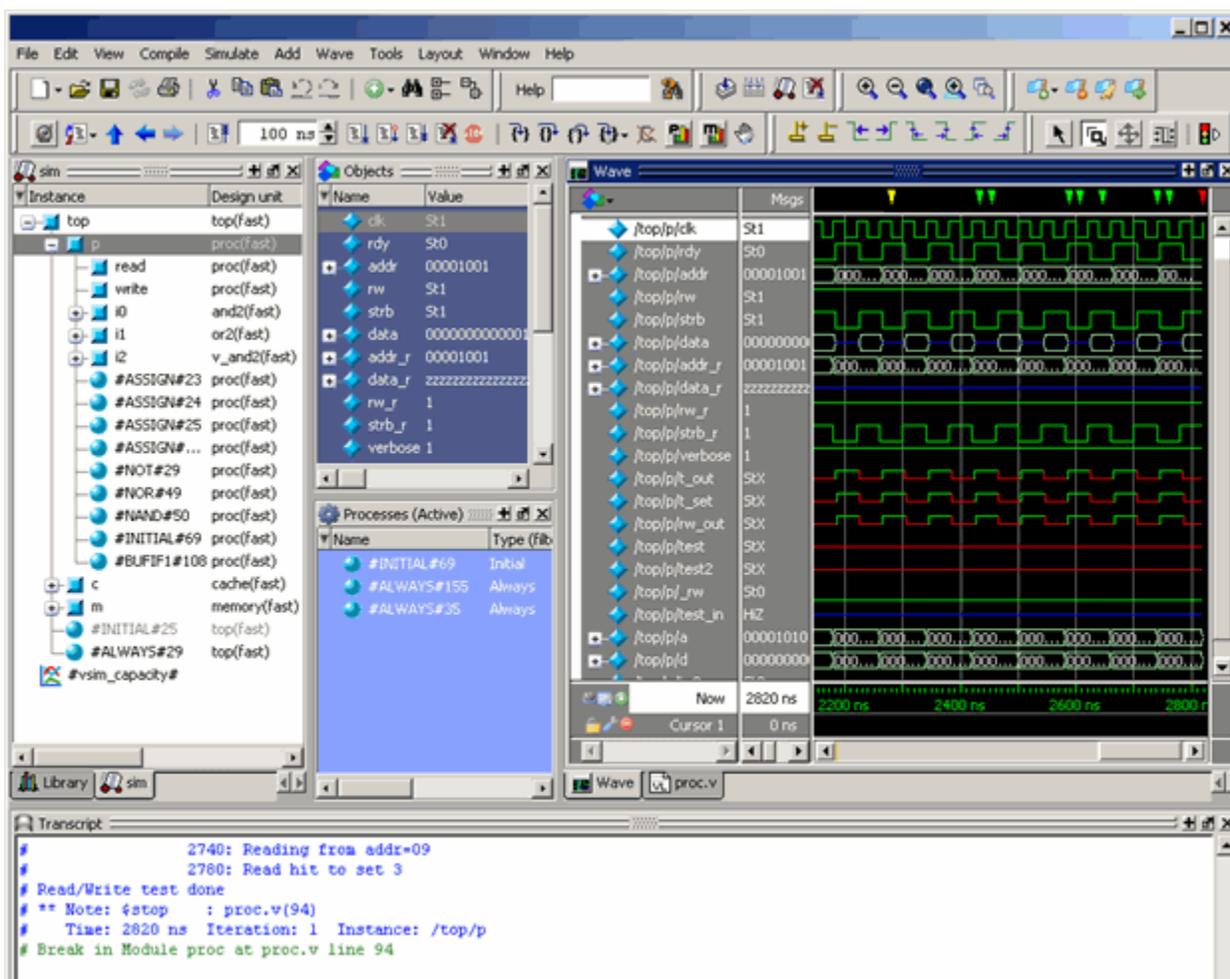
Argument	Version	New Argument / Information
vopt -bbox	10.1	vopt -pdu, name change only.
vopt -save_bbox_hier_refs	10.1	vopt -save_pdu_hier_refs, name change only.
vsim -ignore_bbox	10.1	vsim -ignore_pdu, name change only.
vcom -synthprefix	10.1	vcom -addpragmaprefix
vlog -synthprefix	10.1	vlog -addpragmaprefix

Chapter 2

Graphical User Interface

The ModelSim graphical user interface (GUI) provides access to numerous debugging tools and windows that enable you to analyze different parts of your design. All windows initially display within the ModelSim Main window.

Figure 2-1. Graphical User Interface



The following table summarizes all of the available windows.

Table 2-1. GUI Windows

Window name	Description	More details
Main	central GUI access point	Main Window

Table 2-1. GUI Windows (cont.)

Window name	Description	More details
Assertion Thread Viewer	displays a graphical, time-based view of your SystemVerilog and PSL assertions	ATV Window
Assertions	manages SystemVerilog and PSL assertions	Assertions Window
Call Stack	displays the current call stack, allowing you to debug your design by analyzing the depth of function calls	Call Stack Window
Capacity	displays capacity data (memory usage) about SystemVerilog constructs	Capacity Window
Class Graph	displays interactive relationships of SystemVerilog classes in graphical form	Class Graph Window
Class Instances	displays class instances	Class Instances Window
Class Tree	displays interactive relationships of SystemVerilog classes in tabular form.	Class Tree Window
Code Coverage Analysis	displays missing code coverage, details, and code coverage exclusions	Code Coverage Analysis Window
Cover Directives	manages SystemVerilog and PSL cover directives	Cover Directives Window
Covergroups	manages SystemVerilog covergroups	Covergroups Window
Coverage Details	contains details about coverage metrics based on selections in other coverage windows	Coverage Details Window
Dataflow	displays "physical" connectivity and lets you trace events (causality)	Dataflow Window
Files	displays the source files and their locations for the loaded simulation	Files Window
FSM List	lists all recognized FSMs in the design	FSM List Window
FSM View	graphical representation of a recognized FSM	FSM Viewer Window
Instance Coverage	displays coverage statistics for each instance in a flat, non-hierarchical view	Instance Coverage Window
Library	lists design libraries and compiled design units	Library Window
List	shows waveform data in a tabular format	List Window

Table 2-1. GUI Windows (cont.)

Window name	Description	More details
Locals	displays data objects that are immediately visible at the current execution point of the selected process	Locals Window
Memory	windows that show memories and their contents	Memory List Window Memory Data Window
Message Viewer	allows easy access, organization, and analysis of Note, Warning, Errors or other messages written to transcript during simulation	Message Viewer Window
Objects	displays all declared data objects in the current scope	Objects Window
OVM-Aware Debug	windows to assist in debugging OVM testbenches	OVM-Aware Debug Windows
Process	displays all processes that are scheduled to run during the current simulation cycle	Processes Window
Profile	windows that display performance and memory profiling data	Profiling Windows
Project	provides access to information about Projects	Projects
Schematic	displays information about the design in schematic format	Schematic Window
Source	a text editor for viewing and editing files, such as Verilog, VHDL, SystemC, and DO files	Source Window
Structure (sim)	displays hierarchical view of active simulation. Name of window is either “sim” or “<dataset_name>”	Structure Window
Transaction View	displays the details of Questa Verification IP transaction instances. Not available for any other type of transactions	Transaction View Window
Transcript	keeps a running history of commands and messages and provides a command-line interface	Transcript Window

Table 2-1. GUI Windows (cont.)

Window name	Description	More details
Verification Management	displays information about UCDB tests for the purpose of managing the verification process	Verification Management Browser Window Verification Results Analysis Window Verification Test Analysis Window Verification Tracker Window Verification Trender Window
Watch	displays signal or variable values at the current simulation time	Watch Window
Wave	displays waveforms	Wave Window

The windows are customizable in that you can position and size them as you see fit, and ModelSim will remember your settings upon subsequent invocations. You can restore ModelSim windows and panes to their original settings by selecting **Layout > Reset** in the menu bar.

You can copy the title text in a window or pane header by selecting it and right-clicking to display a popup menu. This is useful for copying the file name of a source file for use elsewhere (see [Figure 2-73](#) for an example of this in an FSM Viewer window).

Design Object Icons and Their Meaning

The color and shape of icons convey information about the language and type of a design object. [Table 2-2](#) shows the icon colors and the languages they indicate.

Table 2-2. Design Object Icons

Icon color	Design Language
light blue	Verilog or SystemVerilog
dark blue	VHDL
green	SystemC
magenta	PSL
orange	virtual object

Here is a list of icon shapes and the design object types they indicate:

Table 2-3. Icon Shapes and Design Object Types

Icon shape	Example	Design Object Type
Square		any scope (VHDL block, Verilog named block, SC module, class, interface, task, function, and so forth.)
Square and red asterix		abstract scope (VHDL block, Verilog named block, SC module, class, interface, task, function, and so forth.)
Circle		process
Diamond		valued object (signals, nets, registers, SystemC channel, and so forth.)
Diamond and yellow pulse on red dot		an editable waveform created with the waveform editor
Diamond and red asterix		valued object (abstract)
Diamond and green arrow		indicates mode (In, Inout, Out) of an object port
Triangle		assertions (SV, PSL)
Triangle		caution sign on comparison object
Chevron		cover directives (SV, PSL)
Star		transaction; The color of the star for each transaction depends on the language of the region in which the transaction stream occurs: dark blue for VHDL, light blue for Verilog and SystemVerilog, green for SystemC.

Setting Fonts

You may need to adjust font settings to accommodate the aspect ratios of wide screen and double screen displays or to handle launching ModelSim from an X-session. Refer to [Making Global Font Changes](#) for more information.

Font Scaling

To change font scaling, select the Transcript window, then **Transcript > Adjust Font Scaling**. You will need a ruler to complete the instructions in the lower right corner of the dialog. When you have entered the pixel and inches information, click OK to close the dialog. Then, restart ModelSim to see the change. This is a one time setting; you should not need to set it again unless you change display resolution or the hardware (monitor or video card). The font scaling applies to Windows and UNIX operating systems. On UNIX systems, the font scaling is stored based on the \$DISPLAY environment variable.

Using the Find and Filter Functions

Finding and/or filtering capabilities are available for most windows. The Find mode toolbar is shown in [Figure 2-2](#). The filtering function is denoted by a “Contains” field ([Figure 2-3](#)).

Figure 2-2. Find Mode



Figure 2-3. Filter Mode



Windows that support both Find ([Figure 2-2](#)) and Filter modes ([Figure 2-3](#)) allow you to toggle between the two modes by doing any one of the following:

- Use the **Ctrl+M** hotkey.
- Click the “Find” or “Contains” words in the toolbar at the bottom of the window.
- Select the mode from the Find Options popup menu (see [Using the Find Options Popup Menu](#)).

The last selected mode is remembered between sessions.

A “Find” toolbar will appear along the bottom edge of the active window when you do either of the following:

- Select **Edit > Find** in the menu bar.
- Click the **Find** icon in the [Standard Toolbar](#). 

All of the above actions are toggles - repeat the action and the Find toolbar will close.

The Find or Filter entry fields prefill as you type, based on the context of the current window selection. The find or filter action begins as you type.

There is a simple history mechanism that saves find or filter strings for later use. The keyboard shortcuts to use this feature are:

- **Ctrl+P** — retrieve previous search string
- **Ctrl+N** — retrieve next search string

Other hotkey actions include:

- **Esc** — closes the Find toolbar
- **Enter** (Windows) or **Return** (UNIX or Linux) — initiates a “Find Next” action
- **Ctrl+T** — search while typing (default is on)

The entry field turns red if no matches are found.

The graphic elements associated with the Find toolbar are shown in [Table 2-4](#).

Table 2-4. Graphic Elements of Toolbar in Find Mode

Graphic Element	Action
 Find	opens the find toolbar in the active window
 Close	closes the find toolbar
 Find entry field	allows entry of find parameters
 Find Options	opens the Find Options popup menu at the bottom of the active window. The contents of the menu changes for each window.
 Clear Entry Field	clears the entry field
 Execute Search	initiates the search
 Toggle Search Direction	toggles search direction upward or downward through the active window
 Find All Matches; Bookmark All Matches (for Source window only)	highlights every occurrence of the find item; for the Source window only, places a blue flag (bookmark) at every occurrence of the find item
 Search For	Click and hold the button to open a drop down menu with the following options: <ul style="list-style-type: none"> • Instance • Design Unit • Design Unit Type

Table 2-4. Graphic Elements of Toolbar in Find Mode (cont.)

Graphic Element	Action
 Match Case	search must match the case of the text entered in the Find field
 Exact (whole word)	searches for whole words that match those entered in the Find field
 Regular Expression	searches for a regular expression
 Wrap Search	searches from cursor to bottom of window then continues search from top of the window

Using the Filter Mode

By entering a string in the “Contains” text entry box you can filter the view of the selected window down to the specific information you are looking for.

Table 2-5. Graphic Elements of Toolbar in Filter Mode

Button	Name	Shortcuts	Description
	Filter Regular Expression	None	A drop down menu that allows you to set the wildcard mode. A text entry box for your filter string.
	Clear Filter	None	Clears the text entry box and removes the filter from the active window.

Wildcard Usage

There are three wildcard modes:

- **glob-style** — Allows you to use the following special wildcard characters:
 - * — matches any sequence of characters in the string
 - ? — matches any single character in the string
 - [<chars>] — matches any character in the set <chars>.
 - \

For more information refer to the Tcl documentation:

Help > Tcl Man Pages
Tcl Commands > string > string match

- **regular-expression** — allows you to use wildcard characters based on Tcl regular expressions. For more information refer to the Tcl documentation:

Help > Tcl Man Pages
Tcl Commands > re_syntax

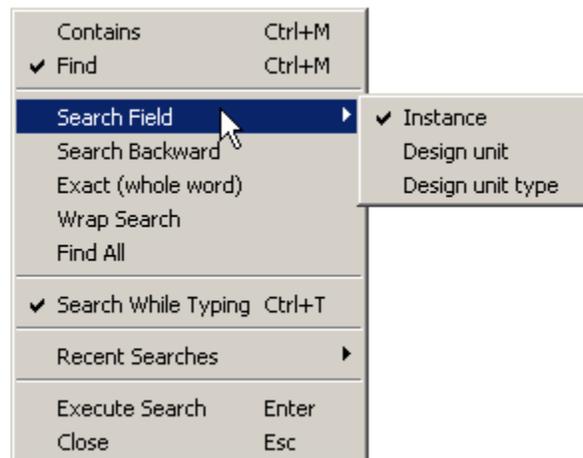
- **exact** — indicates that no characters have special meaning, thus disabling wildcard features.

The string entry field of the Contains toolbar item is case-insensitive, If you need to search for case-sensitive strings use “regular-expression” and prepend the string with (?c)

Using the Find Options Popup Menu

When you click the Find Options icon  in the Find entry field it will open a Find Options popup menu (Figure 2-4).

Figure 2-4. Find Options Popup Menu



The Find Options menu displays the options available to you as well as hot keys for initiating the actions without the menu.

User-Defined Radices

A user definable radix is used to map bit patterns to a set of enumeration labels. After defining a new radix, the radix will be available for use in the List, Watch, and Wave windows or with the [examine](#) command.

There are four commands used to manage user defined radices:

- [radix define](#)

- [radix names](#)
- [radix list](#)
- [radix delete](#)

Using the radix define Command

The [radix define](#) command is used to create or modify a radix. It must include a radix name and a definition body, which consists of a list of number pattern, label pairs. Optionally, it may include the `-color` argument for setting the radix color (see [Example 2-2](#)).

```
{
    <numeric-value> <enum-label>,
    <numeric-value> <enum-label>
    -default <radix>
}
```

A `<numeric-value>` is any legitimate HDL integer numeric literal. To be more specific:

```
<base>#<base-integer># --- <base> is 2, 8, 10, or 16
<base>"bit-value" --- <base> is B, O, or X
<integer>
<size>'<base><number> --- <size> is an integer, <base> is b, d, o, or h.
```

Check the Verilog and VHDL LRMs for exact definitions of these numeric literals.

The comma (,) in the definition body is optional. The `<enum-label>` is any arbitrary string. It should be quoted (""), especially if it contains spaces.

The `-default` entry is optional. If present, it defines the radix to use if a match is not found for a given value. The `-default` entry can appear anywhere in the list, it does not have to be at the end.

[Example 2-1](#) shows the **radix define** command used to create a radix called “States,” which will display state values in the List, Watch, and Wave windows instead of numeric values.

Example 2-1. Using the radix define Command

```
radix define States {
    11'b000000000001 "IDLE",
    11'b000000000010 "CTRL",
    11'b000000000100 "WT_WD_1",
    11'b000000001000 "WT_WD_2",
    11'b000000010000 "WT_BLK_1",
    11'b000000100000 "WT_BLK_2",
    11'b000001000000 "WT_BLK_3",
    11'b000010000000 "WT_BLK_4",
    11'b000100000000 "WT_BLK_5",
    11'b010000000000 "RD_WD_1",
    11'b100000000000 "RD_WD_2",
    -default hex
}
```

Figure 2-5 shows an FSM signal called `/test-sm/sm_seq0/sm_0/state` in the Wave window with a binary radix and with the user-defined “States” radix (as defined in Example 2-1).

Figure 2-5. User-Defined Radix “States” in the Wave Window

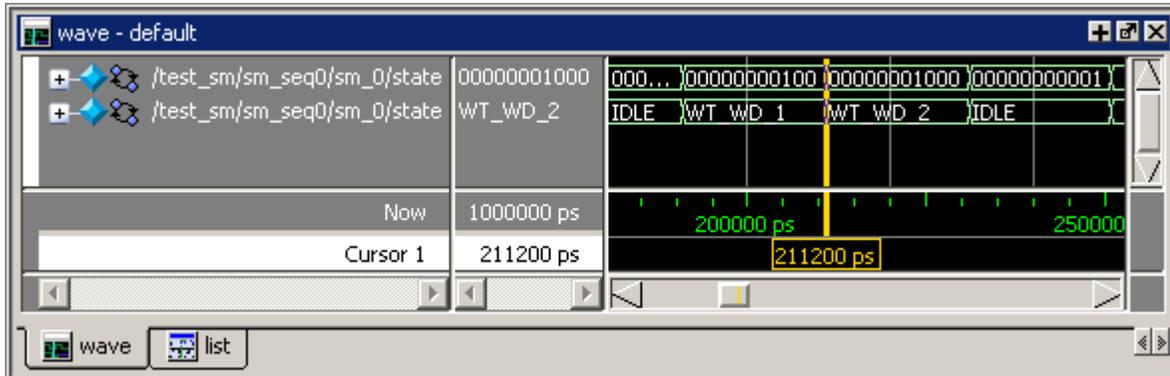
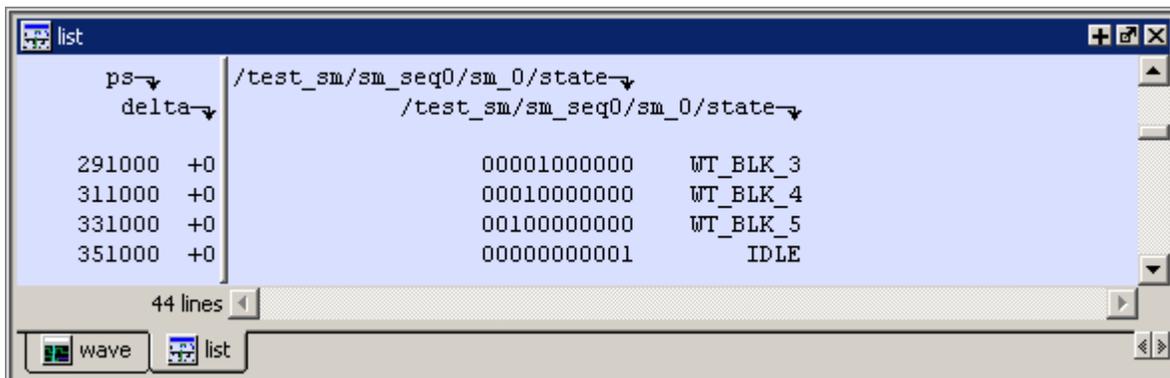


Figure 2-6 shows an FSM signal called `/test-sm/sm_seq0/sm_0/state` in the List window with a binary radix and with the user-defined “States” radix (as defined in Example 2-1)

Figure 2-6. User-Defined Radix “States” in the List Window



Using radix define to Specify Radix Color

The following example illustrates how to use the `radix define` command to specify the radix color:

Example 2-2. Using radix define to Specify Color

```
radix define States {
  11'b000000000001 "IDLE" -color yellow,
  11'b000000000010 "CTRL" -color #ffee00,
  11'b000000000100 "WT_WD_1" -color orange,
  11'b000000001000 "WT_WD_2" -color orange,
  11'b000000010000 "WT_BLK_1",
  11'b000000100000 "WT_BLK_2",
  11'b000001000000 "WT_BLK_3",
  11'b000010000000 "WT_BLK_4",
```

```
11'b001000000000 "WT_BLK_5",  
11'b010000000000 "RD_WD_1" -color green,  
11'b100000000000 "RD_WD_2" -color green,  
-default hex  
-defaultcolor white  
}
```

If a pattern/label pair does not specify a color, the normal wave window colors will be used. If the value of the waveform does not match any pattern, then the -default radix and -defaultcolor will be used.

To specify a range of values, wildcards may be specified for bits or characters of the value. The wildcard character is '?', similar to the iteration character in a Verilog UDP, for example:

```
radix define {  
    6'b01??00 "Write" -color orange,  
    6'b10??00 "Read" -color green  
}
```

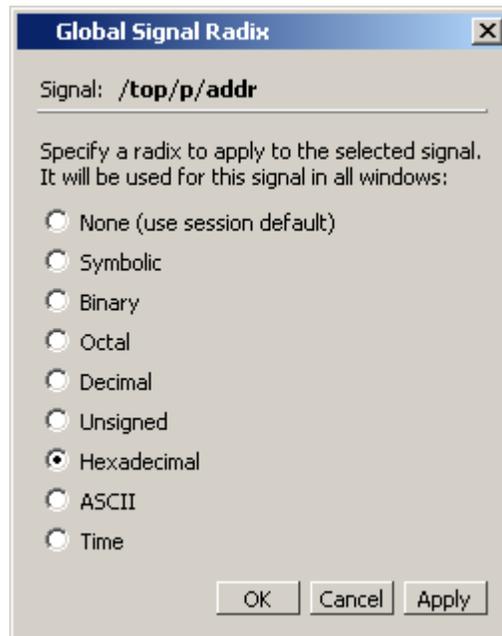
In this example, the first pattern will match "010000", "010100", "011000", and "011100". In case of overlaps, the first matching pattern is used, going from top to bottom.

Setting Global Signal Radix

The Global Signal Radix feature allows you to set the radix for a selected signal or signals in the active window and in other windows where the signal appears. The Global Signal Radix can be set from the Locals, Objects, Schematic, or Wave windows as follows:

- Select a signal or group of signals.
- Right-click the selected signal(s) and click **Global Signal Radix** from the popup menu (in the Wave window, select **Radix > Global Signal Radix**).

This opens the Global Signal Radix dialog box ([Figure 2-7](#)), where you may select a radix. This sets the radix for the selected signal(s) in the active window and every other window where the signal appears.

Figure 2-7. Setting the Global Signal Radix

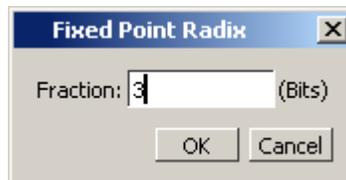
Setting a Fixed Point Radix

Fixed point types are used in VHDL and SystemC to represent non-integer numbers without using a floating point format. ModelSim automatically recognizes VHDL sfixed and ufixed types as well as SystemC SC_FIXED and SC_UFIXED types and displays them correctly with a fixed point format.

In addition, a general purpose fixed point radix feature is available for displaying any vector, regardless of type, in a fixed point format in the Wave window. You simply have to specify how many bits to use as fraction bits from the whole vector.

With the Wave window active:

1. Select (LMB) a signal or signals in the Pathnames pane of the Wave window.
2. Right-click the selected signal(s) and select **Radix > Fixed Point** from the popup menu. This opens the Fixed Point Radix dialog.

Figure 2-8. Fixed Point Radix Dialog

3. Type the number of bits you want to appear as the fraction and click OK.

Saving and Reloading Formats and Content

You can use the [write format](#) restart command to create a single *.do* file that will recreate all debug windows and breakpoints (see [Saving and Restoring Breakpoints](#)) when invoked with the [do](#) command in subsequent simulation runs. The syntax is:

```
write format restart <filename>
```

If the [ShutdownFile](#) *modelsim.ini* variable is set to this *.do* filename, it will call the **write format restart** command upon exit.

Active Time Label

The Active Time Label displays the current time of the active cursor or the Now (end of simulation) time in the Dataflow, Schematic, Source, and FSM windows. This is the time used to control state values displayed or annotated in the window.

Figure 2-9. Active Cursor Time



When you run a simulation and it comes to an end, the Active Time Label displays the Now time - which is the end-of-simulation time. When you select a cursor in the Wave window, or in the Wave viewer of the Schematic or Dataflow window, the Active Time Label automatically changes to display the time of the current active cursor.

The Active Time label includes a minimize/maximize button that allows you to hide or display the label.

When a signal or net is selected, you can jump to the previous or next transition of that signal, with respect to the active time, by clicking the Find Previous/Next Transition buttons.

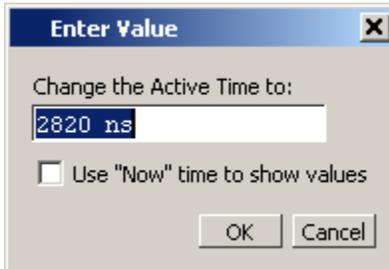
To change the display from showing the Active Time to showing the Now time, or vice versa, do the following:

- Make the Source, Dataflow, Schematic, or FSM window the active window by clicking on it.
- Open the dedicated menu for the selected window (i.e., if the Schematic window is active, open the Schematic menu in the menu bar).

- Select either “Examine Now” or “Examine Current Cursor.”

You can also change the Active Time by simply clicking on the Active Time Label to open the Enter Value dialog box (Figure 2-10), where you can change the value.

Figure 2-10. Enter Active Time Value

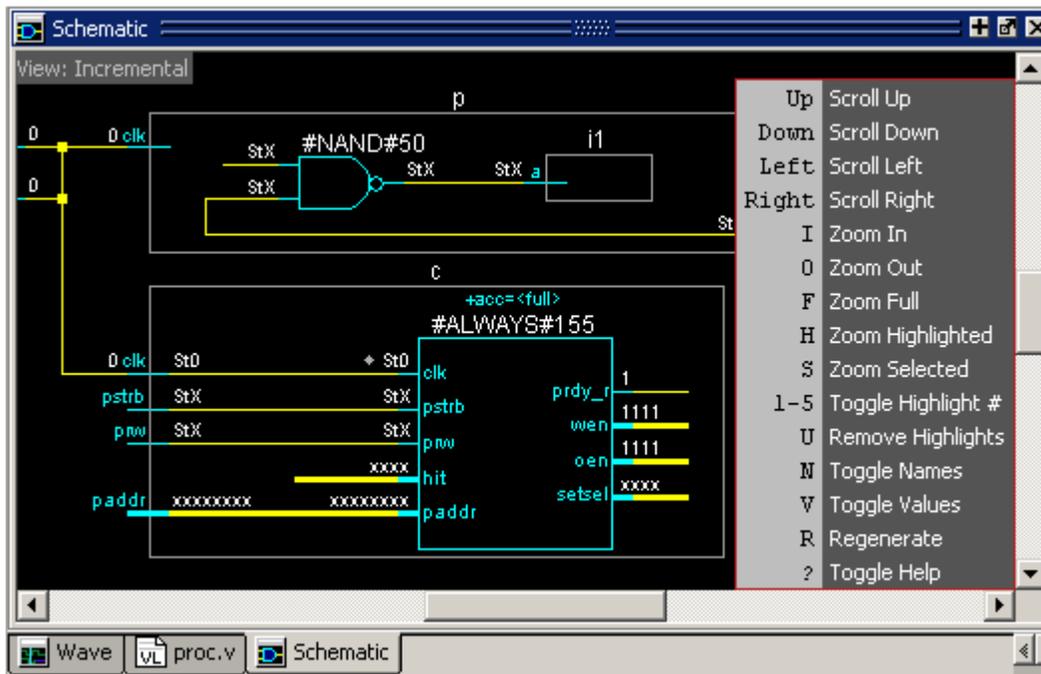


To enable the Active Time Label in the Dataflow window, select **Dataflow > Dataflow Preferences > Options** and check **Active Time label** in the **Show** field of the **Dataflow Options** dialog box.

Window Specific Keyboard Shortcuts

You can open a dynamic list of common, and user defined keyboard shortcuts, for many windows by entering Ctrl-?. You can find a complete list of the keyboard shortcuts for a window by opening the Keyboard Shortcuts dialog box and unselecting the "Show Custom Shortcuts Only" checkbox. Refer to [User Defined Keyboard Shortcuts](#) for more information.

Figure 2-11. Schematic Keyboard Shortcuts



The following windows have keyboard shortcuts assigned:

- Dataflow Window
- Library Window
- Objects Window
- Schematic Window
- Source Window
- Structure Window
- Transcript Window
- Wave Window

For more information about Window specific keyboard shortcuts as well as Global keyboard shortcuts, refer to the [Command and Keyboard Shortcuts](#) Appendix.

User Defined Keyboard Shortcuts

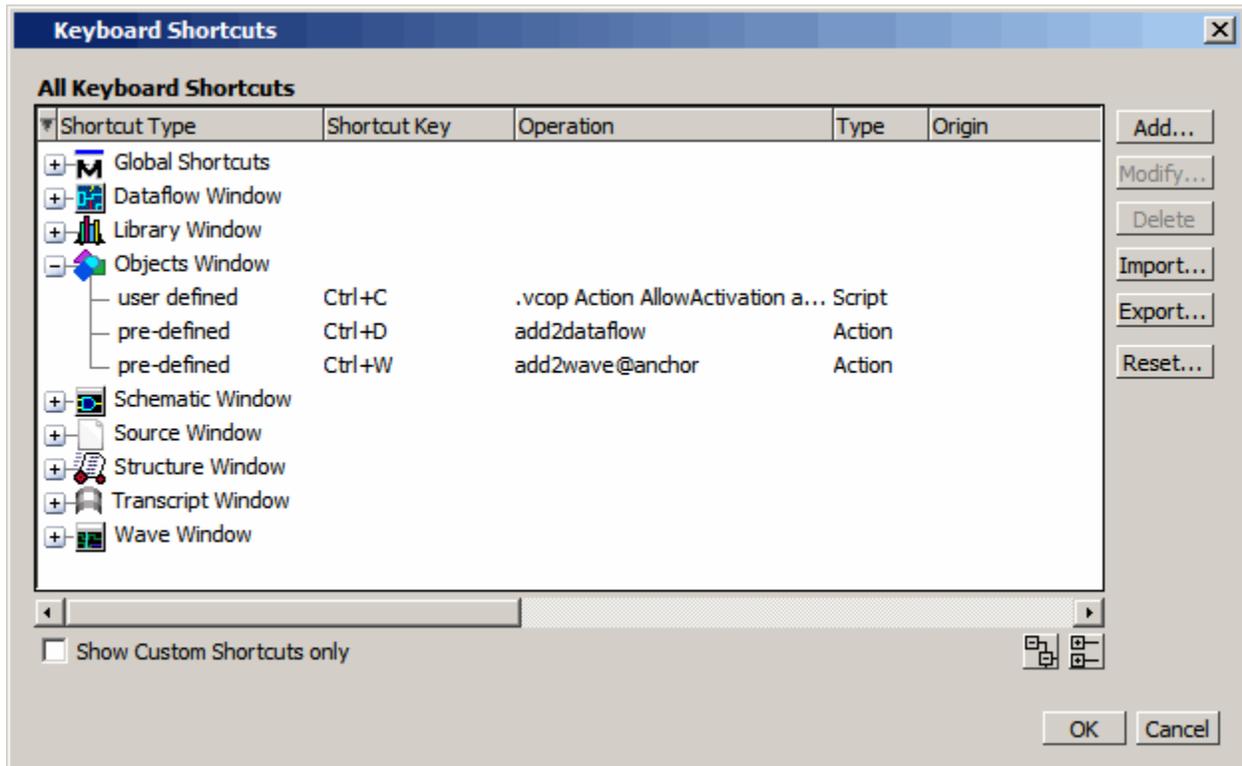
In addition to the predefined keyboard shortcuts you can create your own shortcuts or modify predefined keyboard shortcuts with the Keyboard Shortcuts dialog box (Figure 2-12). Shortcuts can be either window specific (available only when the window is active) or global (available from anywhere in the tool). You can create a keyboard shortcut for any window in ModelSim.

Once a shortcut is defined, it will be available in all subsequent invocations of the tool. The dynamic nature of the architecture makes the keyboard shortcuts available to any tool that is based upon the ModelSim GUI, such as ADMS, 0-In, MVC, and Codelink.

The Keyboard Shortcuts Dialog Box

The Keyboard Shortcuts dialog box lists all existing keyboard shortcuts. The dialog distinguishes between shortcuts that are user defined and shortcuts that come predefined in the tool (Figure 2-12).

Figure 2-12. Keyboard Shortcuts Dialog Box

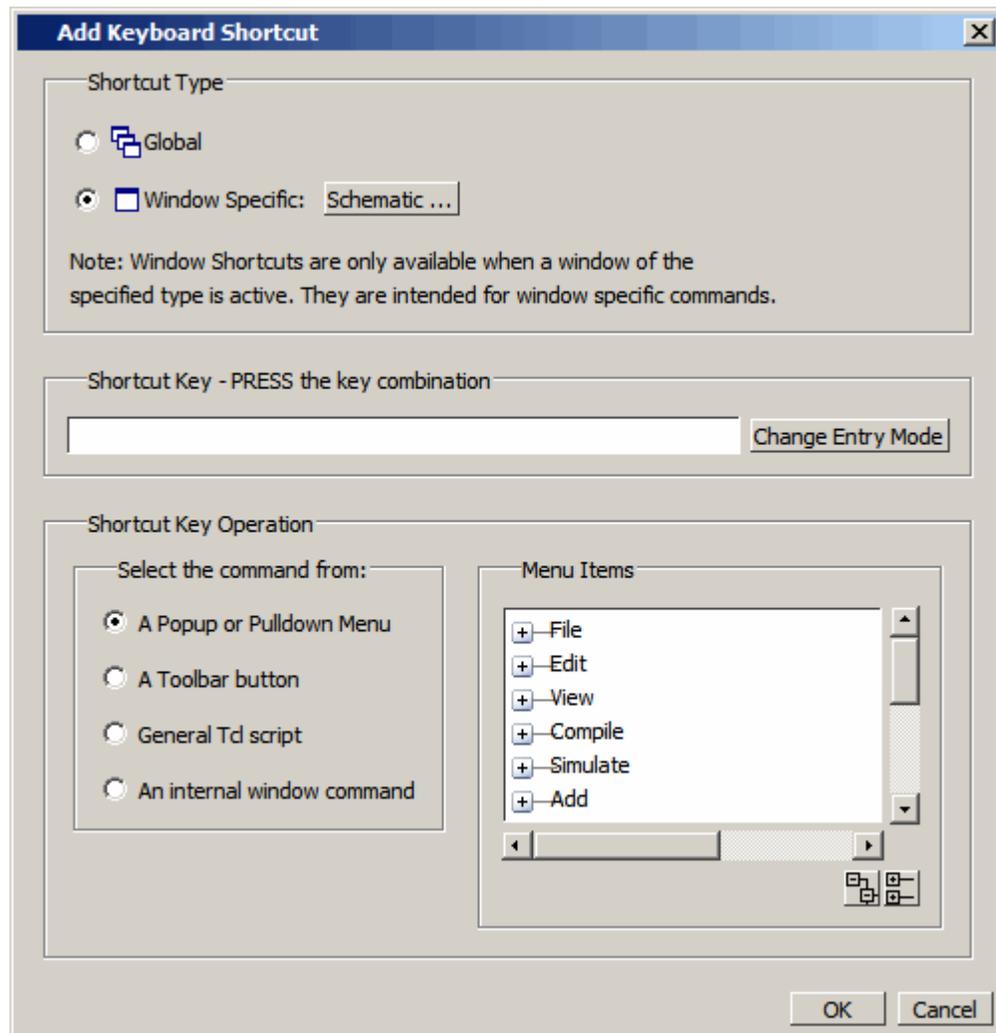


How Do I Create A Keyboard Shortcut?

You can create either a Global or a window specific shortcut.

1. If you are creating a window specific shortcut, the window must have been opened sometime during the simulation run.
2. Open the **Add Keyboard Shortcut** dialog box by selecting **Window > Keyboard Shortcuts**.
3. Click the **Add** button to open the Add Keyboard Shortcut dialog box.

Figure 2-13. Add Keyboard Shortcut Dialog Box



4. Select the Shortcut Type, either Global or Window. If you are creating a window specific shortcut, click the window button to open the **Select Window Type** dialog box. The dialog box displays every window that has been opened during the current simulation. If you do not see the window you are looking for, close both dialog boxes, open the window you want by entering **view** <window> on the command line, or by selecting the window from the **View** menu. Choosing Global or a specific window changes the options available in the **Shortcut Key Operation** field and the dynamically populated field to the right.
5. Enter the key combination in the **Shortcut Key** field. Or select the **Change Entry Mode** button to enter a key combination.
6. Choose the type of operation the shortcut will execute.

- A Popup or Pulldown Menu — Opens the **Menu Items** dialog with a hierarchical list of all popup and pulldown menu items available either globally or for the window specified in step 4.
- A Toolbar button — Opens the Toolbar Buttons dialog with a hierarchical list of all toolbar button actions available either globally or for the window specified in step 4.
- General Tcl script — Selecting this option opens the Tcl Script field to the right. You can enter any Tcl script or command line sequence.
- An Internal window command — This choice is available only for window specific commands. Refer to step 4. Opens the Window Action dialog on the right with a list of all window specific commands.

Bookmarks

You can create bookmarks that allow you to return to a specific view or place in your design for some of the windows. The bookmarks you make can be saved and automatically restored. Some of the windows that allow bookmarking include the Structure, Files, Objects, Wave, and Objects windows.

Working with Bookmarks

The Bookmarks toolbar and the Bookmarks menu give you access to the following bookmarking features:

- Add Bookmarks

Bookmarks are added to an active window by selecting **Bookmarks > Add Bookmark** or by clicking the **Add Bookmark** button. You will be prompted to automatically save and restore your bookmarks when you set the first bookmark. You can change the automatic save and restore settings in the [Bookmark Options Dialog Box](#).

- Add Custom

Selecting **Add Custom** opens the **New Bookmark** dialog box with the context field(s) populated and a field for specifying an alias for the bookmark. Click and hold the **Add Bookmark** button to access this feature from the **Bookmarks** toolbar.

Note



Aliases are mapped to the window in which a bookmark is set. You can use the same alias for different bookmarks as long as each alias is assigned to a bookmark set in a different window.

- Deleting Bookmarks

You can choose to delete the bookmarks from the currently active window or from all windows.

- **Manage Bookmarks**

Opens the **Manage Bookmarks** dialog box. Refer to [Managing Your Bookmarks](#) for more information.

- **Load Bookmarks**

Loads the bookmarks saved in the *bookmarks.do* file. You can choose whether to load bookmarks for the currently active window or all the bookmarks saved in the *bookmarks.do* file. Bookmarks are automatically loaded from the saved *bookmarks.do* file when you start a new simulation session.

Note



You must reload bookmarks for a window if you close then reopen that window during the current session.

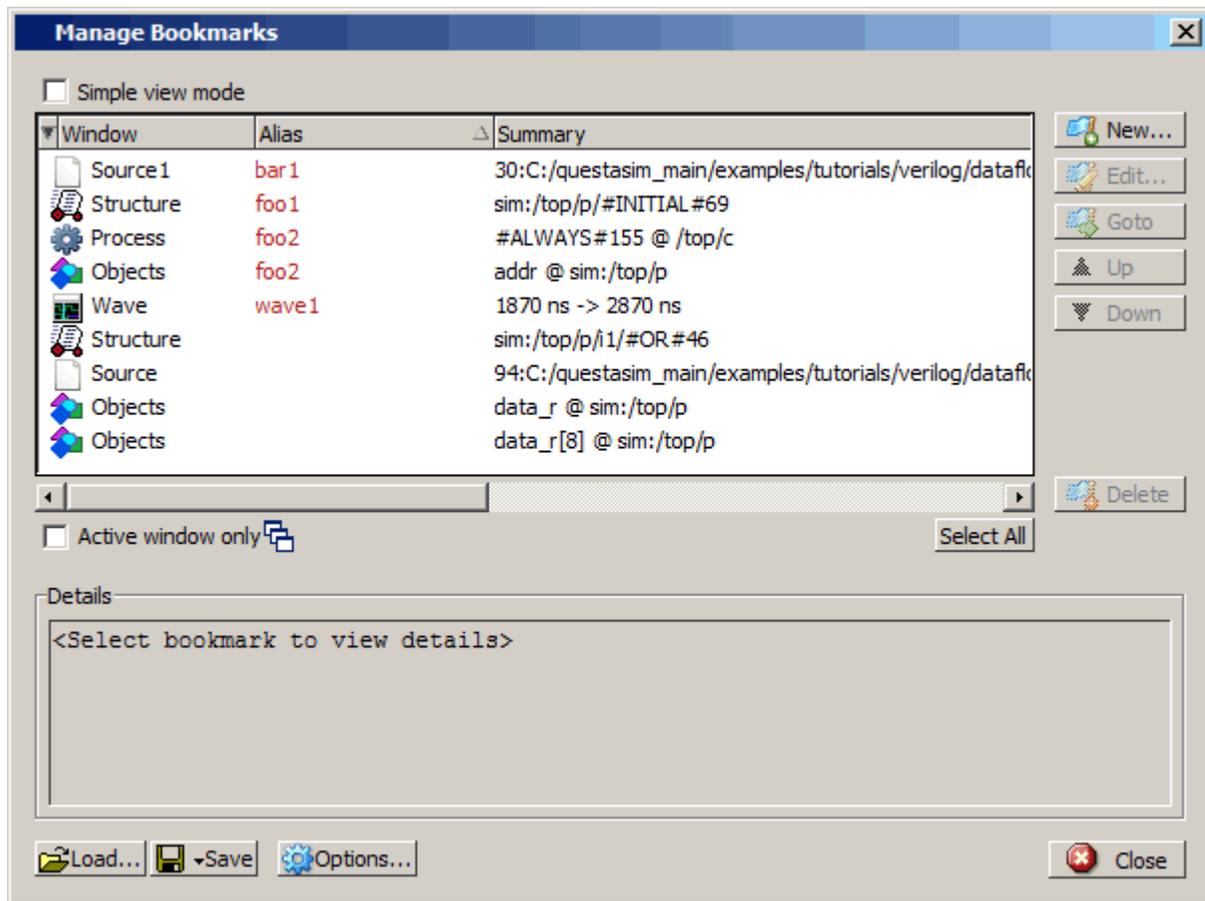
- **Jump to Bookmark**

Shows the available bookmarks in the currently active window followed by a drop down list of bookmarks for each window. You can set the maximum number of bookmarks listed in the [Bookmark Options Dialog Box](#).

Managing Your Bookmarks

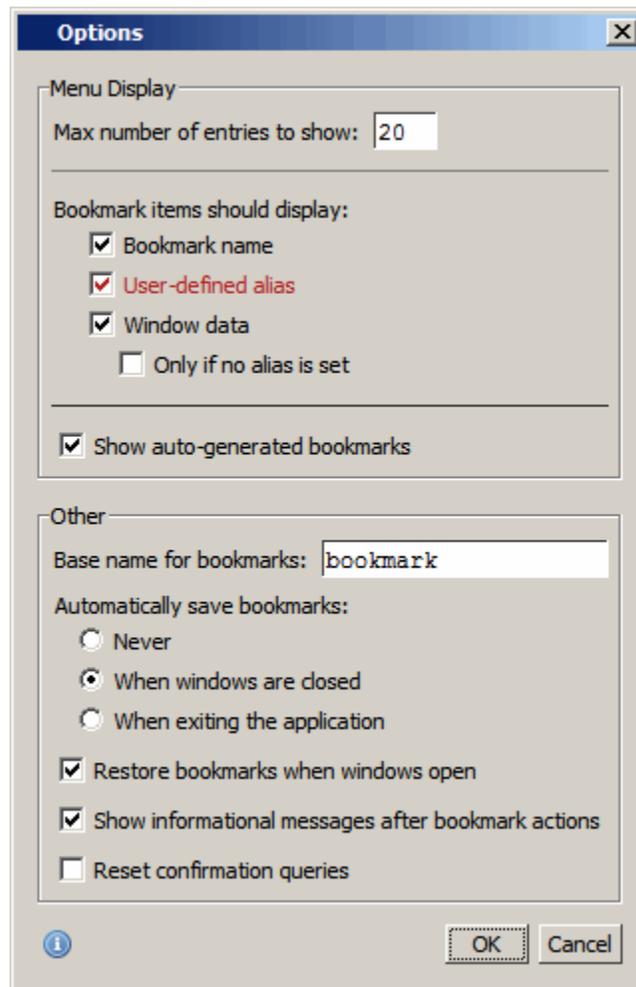
You can open the Manage Bookmarks dialog box with the **Manage Bookmarks** toolbar button or by selecting **Bookmarks > Manage Bookmarks**. The dialog box can be kept open during your simulation ([Figure 2-14](#)).

Figure 2-14. Manage Bookmarks Dialog Box



- **Simple view mode** changes the buttons from name and icon mode to icon only mode.
- Checking **Active window only** changes the display to show the bookmarks in the currently active window. Selecting a different window in the tool changes the display to the bookmarks set in that window.
- Selecting **New** opens the **New Bookmark** dialog box. The fields in the dialog automatically load the settings of the view in the currently active window. You can choose to name the bookmark with an alias to provide a more meaningful description. Aliases are displayed in the Alias column in the Manage Bookmarks dialog box.
- Selecting **Options** opens the **Bookmark Options** dialog box ([Figure 2-15](#)).

Figure 2-15. Bookmark Options Dialog Box



The **Menu Display** section allows you to:

- Set the number of bookmarks displayed in the Bookmarks menu or the Jump to Bookmark button menu.
- Select the types of information displayed for each bookmark.

The **Other** section allows you to:

- Specify a different base name for bookmarks.
- Choose whether you want to automatically save bookmarks and when they are saved.
- Automatically restore the bookmarks when windows are first loaded in the current session.
- **Show informational message after bookmark actions** sends bookmark actions to the transcript. For example:

Bookmark(s) were restored for window "Source"

Main Window

The primary access point in the ModelSim GUI is called the Main window. It provides convenient access to design libraries and objects, source files, debugging commands, simulation status messages, and so forth. When you load a design, or bring up debugging tools, ModelSim opens windows appropriate for your debugging environment.

Elements of the Main Window

The following sections outline the GUI terminology used in this manual.

[Menu Bar](#)

[Toolbar Frame](#)

[Toolbar](#)

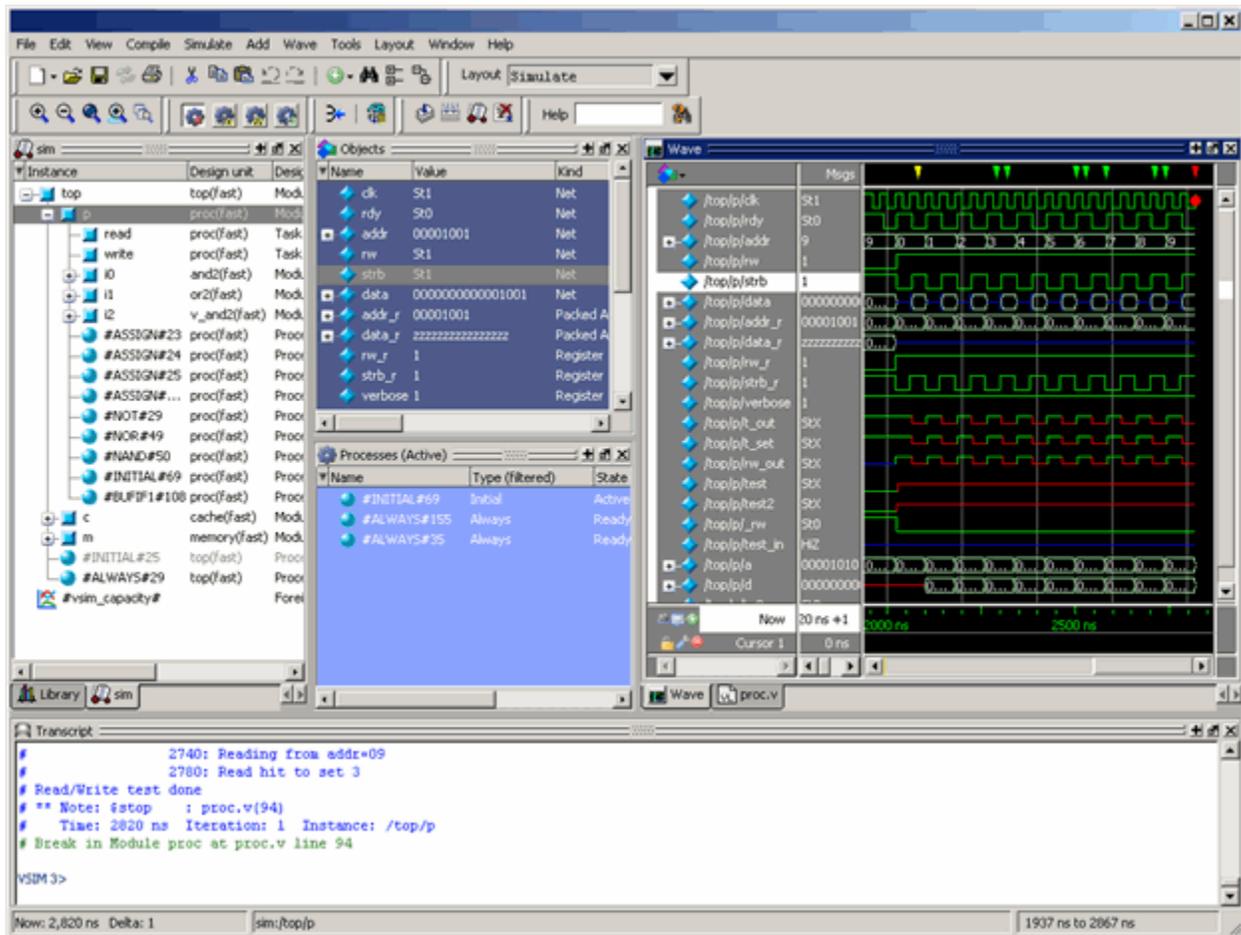
[Window](#)

[Tab Group](#)

[Pane](#)

The Main window is the primary access point in the GUI. [Figure 2-16](#) shows an example of the Main window during a simulation run.

Figure 2-16. Main Window of the GUI



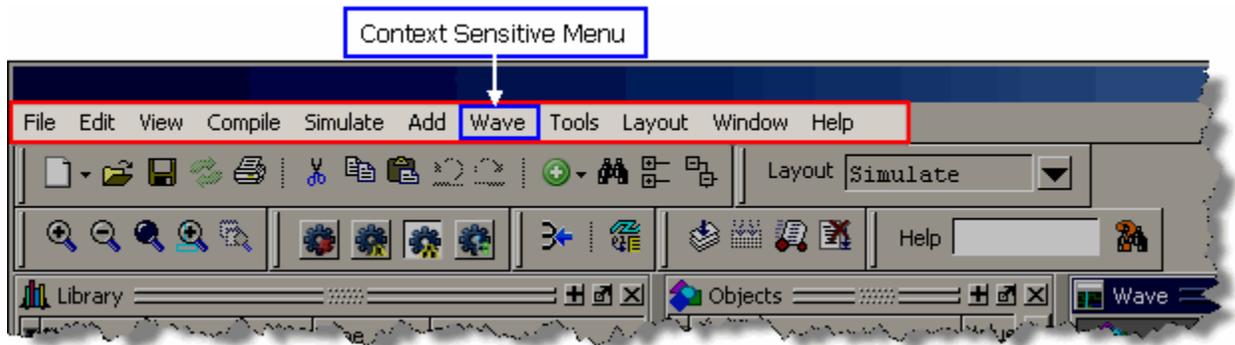
The Main window contains a menu bar, toolbar frame, windows, tab groups, and a status bar, which are described in the following sections.

Menu Bar

The menu bar provides access to many tasks available for your workflow. [Figure 2-17](#) shows the selection in the menu bar that changes based on whichever window is currently active.

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

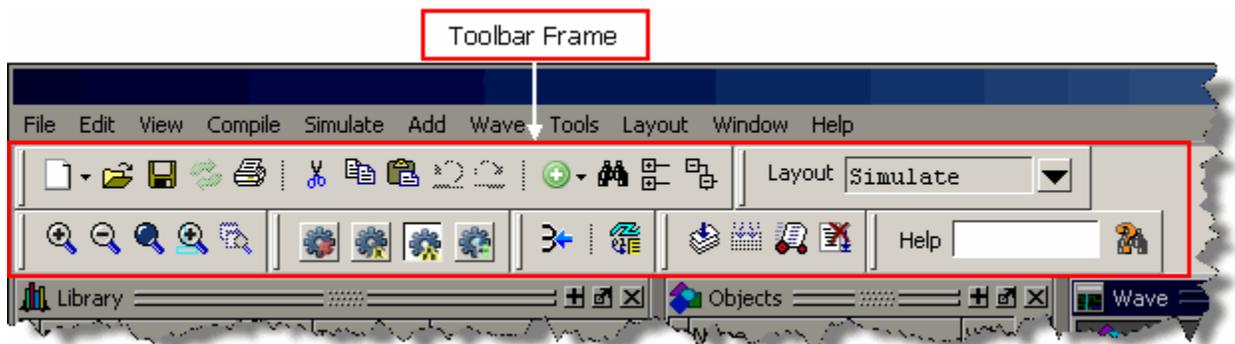
Figure 2-17. Main Window — Menu Bar



Toolbar Frame

The toolbar frame contains several toolbars that provide quick access to various commands and functions.

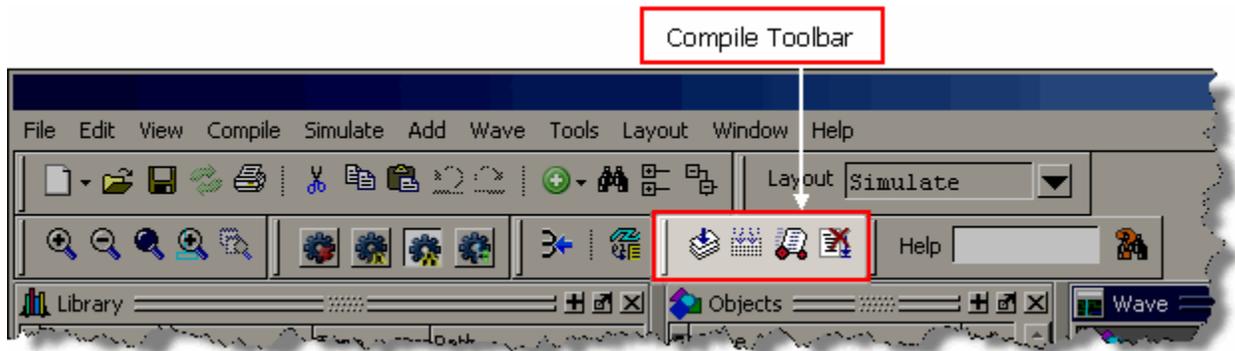
Figure 2-18. Main Window — Toolbar Frame



Toolbar

A toolbar is a collection of GUI elements in the toolbar frame and grouped by similarity of task. There are many toolbars available within the GUI, refer to the section “[Main Window Toolbar](#)” for more information about each toolbar. [Figure 2-19](#) highlights the Compile toolbar in the toolbar frame.

Figure 2-19. Main Window — Toolbar

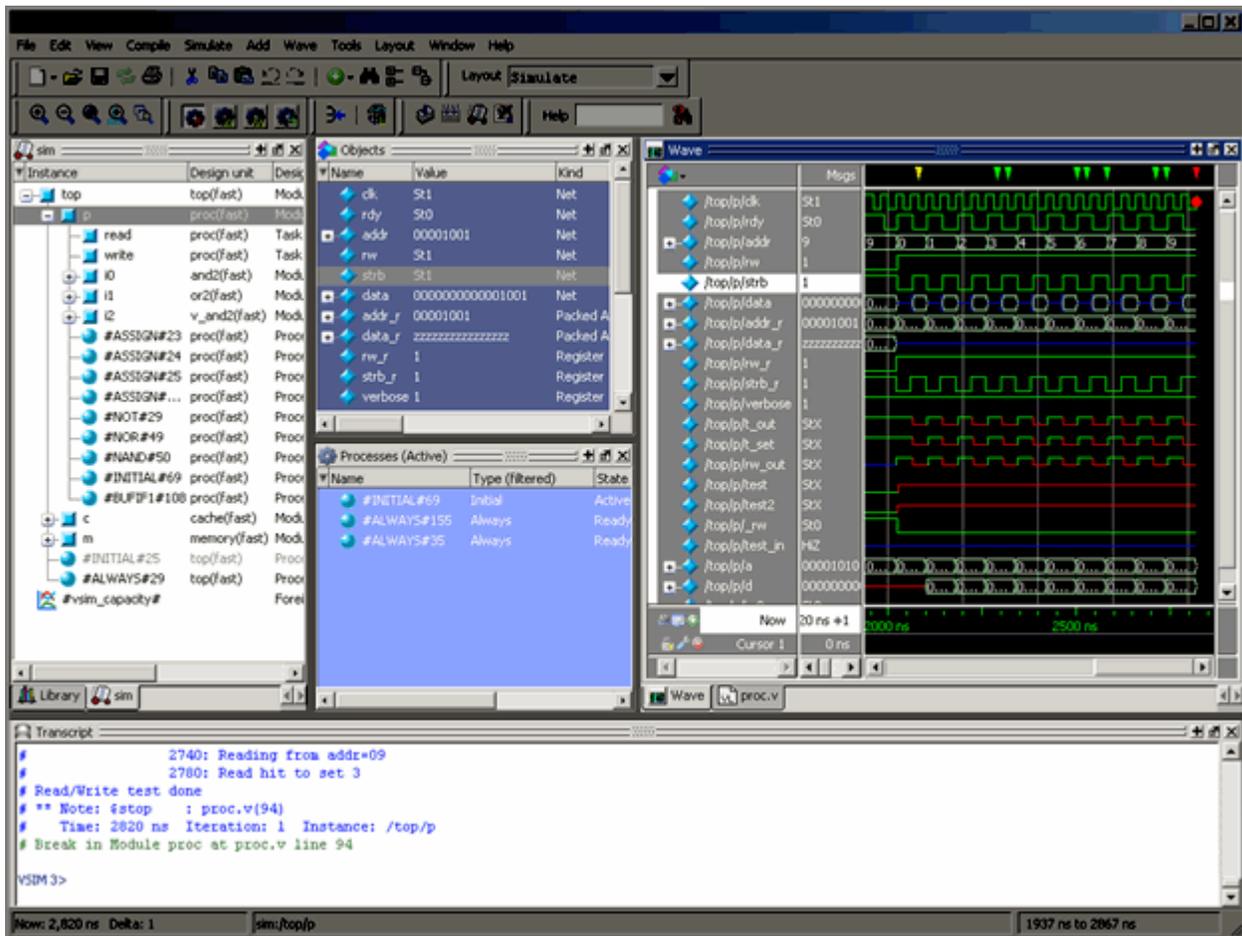


Window

ModelSim can display over 40 different windows you can use with your workflow. This manual refers to all of these objects as windows, even though you can rearrange them such that they appear as a single window with tabs identifying each window.

[Figure 2-20](#) shows an example of a layout with five windows visible; the Structure, Objects, Processes, Wave and Transcript windows.

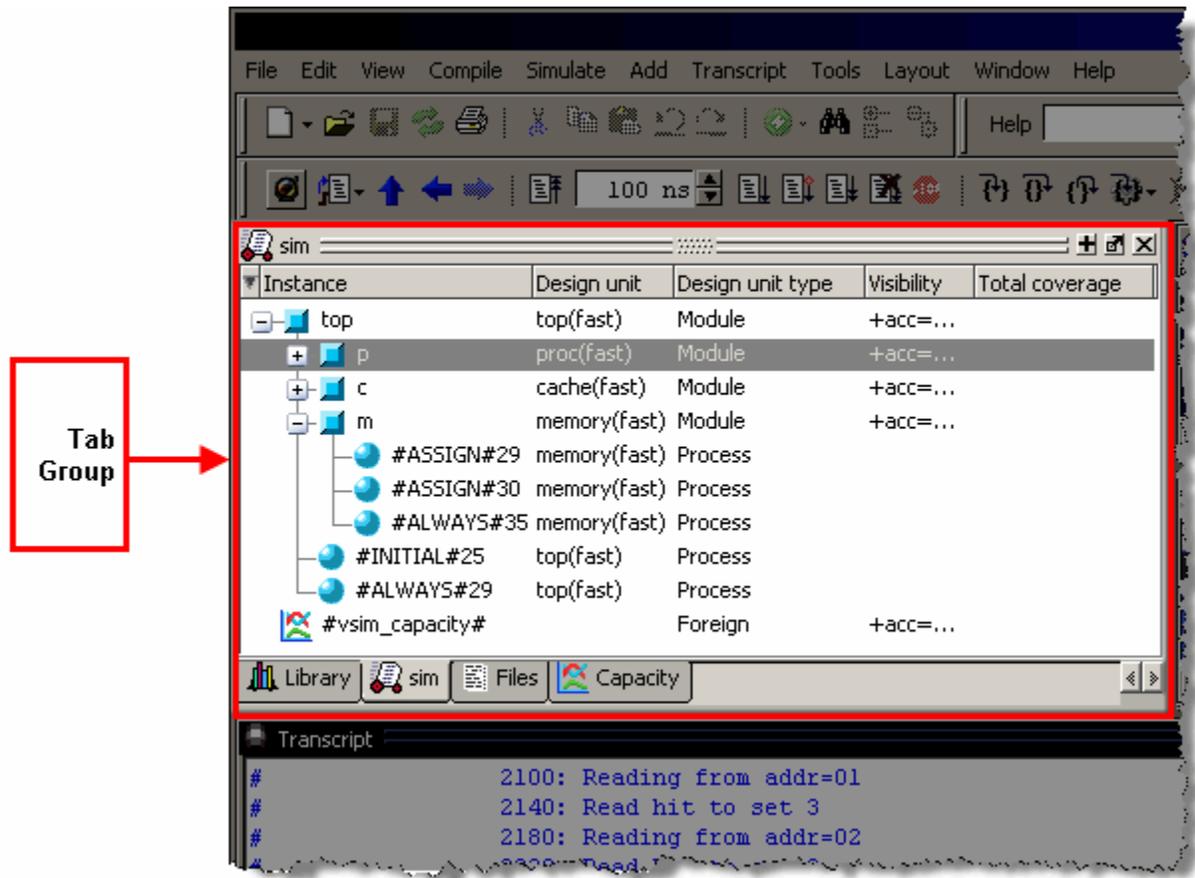
Figure 2-20. GUI Windows



Tab Group

You can group any number of windows into a single space called a tab group, allowing you to show and hide windows by selecting their tabs. [Figure 2-21](#) shows a tab group of the Library, Files, Capacity and Structure windows, with the Structure (sim) window visible.

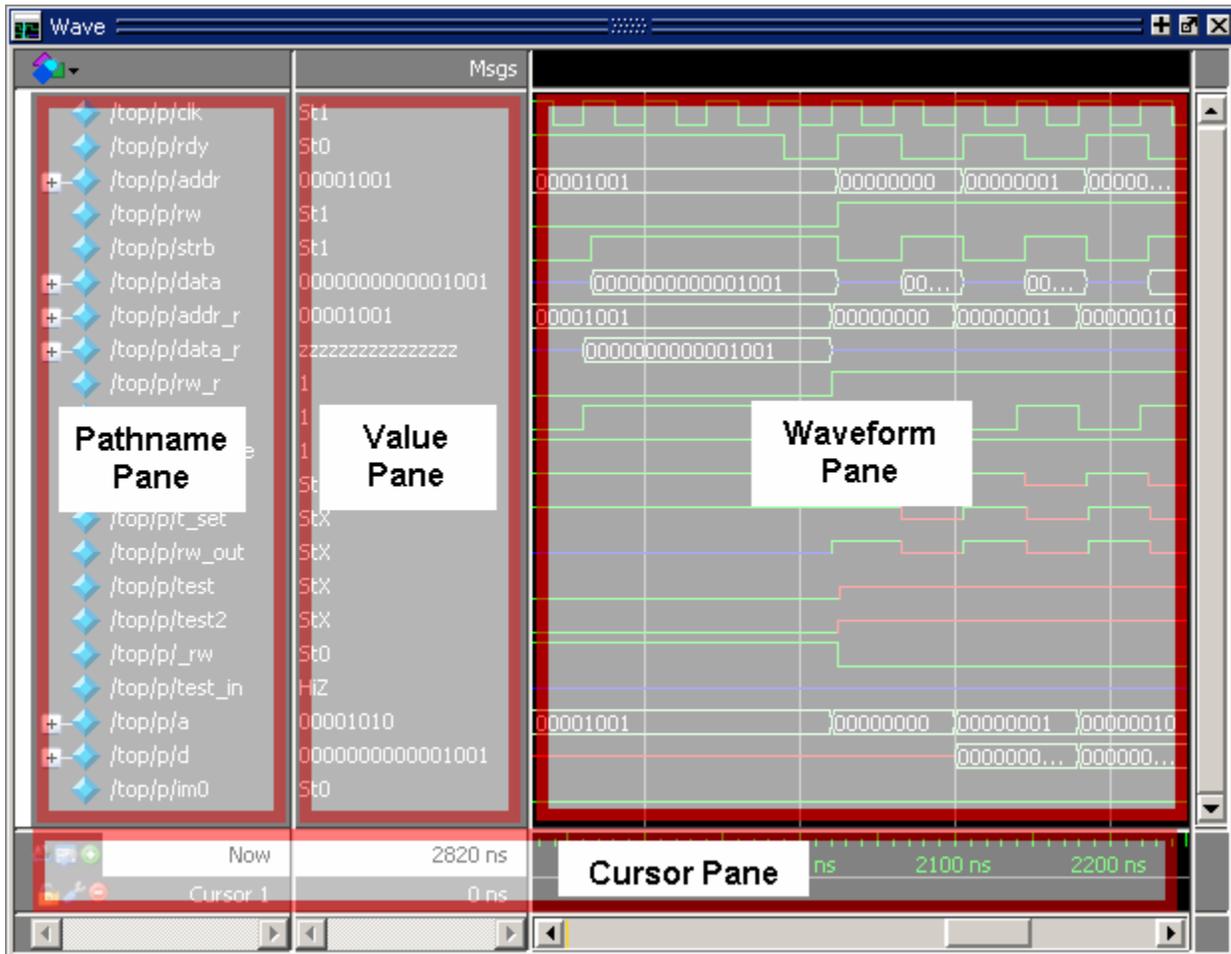
Figure 2-21. GUI Tab Group



Pane

Some windows contain panes, which are separate areas of a window display containing distinct information within that window. One way to tell if a window has panes is whether you receive different popup menus (right-click menu) in different areas. Windows that have panes include the Wave, Source, and List windows. Figure 2-22 shows the Wave window with its three panes.

Figure 2-22. Wave Window Panes



Main Window Status Bar

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

Figure 2-23. Main Window Status Bar



Table 2-6. Information Displayed in Status Bar

Field	Description
Project	name of the current project
Now	the current simulation time

Table 2-6. Information Displayed in Status Bar (cont.)

Field	Description
Delta	the current simulation iteration number
Profile Samples	the number of profile samples collected during the current simulation
Memory	the total memory used during the current simulation
environment	name of the current context (object selected in the active Structure window)
line/column	line and column numbers of the cursor in the active Source window
Total Coverage	the aggregated coverage, as a percent, of the top level object in the design
coverage mode	recursive or non-recursive

Selecting the Active Window

When the title bar of a window is highlighted - solid blue - it is the active window. All menu selections will correspond to this active window. You can change the active window in the following ways.

- (default) Click anywhere in a window or on its title bar.
- Move the mouse pointer into the window.

To turn on this feature, select **Window > FocusFollowsMouse**. Default time delay for activating a window after the mouse cursor has entered the window is 300ms. You can change the time delay with the PrefMain(FFMDelay) preference variable.

Rearranging the Main Window

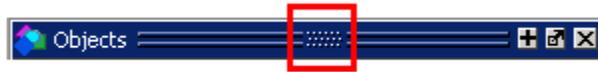
You can alter the layout of the Main window using any of the following methods.

- [Moving a Window or Tab Group](#)
- [Moving a Tab out of a Tab Group](#)
- [Undocking a Window from the Main Window](#)

When you exit ModelSim, the current layout is saved for a given design so that it appears the same the next time you invoke the tool.

Moving a Window or Tab Group

1. Click on the header handle in the title bar of the window or tab group.

Figure 2-24. Window Header Handle

2. Drag, without releasing the mouse button, the window or tab group to a different area of the Main window

Wherever you move your mouse you will see a dark blue outline that previews where the window will be placed.

If the preview outline is a rectangle centered within a window, it indicates that you will convert the window or tab group into new tabs within the highlighted window.

3. Release the mouse button to complete the move.

Moving a Tab out of a Tab Group

1. Click on the tab handle that you want to move.

Figure 2-25. Tab Handle

2. Drag, without releasing the mouse button, the tab to a different area of the Main window

Wherever you move your mouse you will see a dark blue outline that previews where the tab will be placed.

If the preview outline is a rectangle centered within a window, it indicates that you will move the tab into the highlighted window.

3. Release the mouse button to complete the move.

Undocking a Window from the Main Window

- Follow the steps in [Moving a Window or Tab Group](#), but drag the window outside of the Main window, or
- Click on the Dock/Undock button for the window.

Figure 2-26. Window Undock Button

Navigating in the Main Window

The Main window can contain of a number of windows that display various types of information about your design, simulation, or debugging session.

Main Window Menu Bar

The main window menu bar is dynamic based on which window is selected, resulting in some menu items changing name or becoming unavailable (greyed out). This section describes the menu items at the highest-possible level.

File Menu

Table 2-7. File Menu — Item Description

Menu Item	Description
New	<ul style="list-style-type: none"> • Folder — create a new folder in the current directory • Source — create a new VHDL, Verilog or other source file • Project — create a new project • Library — create a new library and mapping •
Open	Open a file of any type.
Load	Load and run a macro file (.do or .tcl)
Close	Close an opened file
Import	<ul style="list-style-type: none"> • Library — import FPGA libraries • EVCD — import an extended VCD file previously created with the ModelSim Waveform Editor. This item is enabled only when a Wave window is active • Memory Data — initialize a memory by reloading a previously saved memory file. • Test Plan — import a verification management test plan. Only applies to the Verification Management Browser window. • Column Layout — apply a previously saved column layout to the active window •
Export	<ul style="list-style-type: none"> • Waveform — export a created waveform • Tabular list — writes List window data to a file in tabular format • Event list — writes List window data to a file as a series of transitions that occurred during simulation • TSSI list — writes List window data to a file in TSSI format • Image — saves an image of the active window • Memory Data — saves data from the selected memory in the Memory List window or an active Memory Data window to a text file • Column Layout — saves a column layout from the active window •
Save Save as	These menu items change based on the active window.
Report	Produce a textual report based on the active window
Change Directory	Opens a browser for you to change your current directory. Not available during a simulation, or if you have a dataset open.

Table 2-7. File Menu — Item Description (cont.)

Menu Item	Description
Use Source	Specifies an alternative file to use for the current source file. This mapping only exists for the current simulation. This option is only available from the Structure window.
Source Directory	Control which directories are searched for source files.
Datasets	Manage datasets for the current session.
Environment	Set up how different windows should be updated, by dataset, process, and/or context. This is only available when the Structure, Locals, Processes, and Objects windows are active.
Page Setup Print Print Postscript	Manage the printing of information from the selected window.
Recent Directories	Display a list of recently opened working directories
Recent Projects	Display a list of recently opened projects
Close Window	Close the active window
Quit	Quit the application

Edit Menu

Table 2-8. Edit Menu — Item Description

Menu Item	Description
Undo Redo	Alter your previous edit in a Source window.
Cut Copy Paste	Use or remove selected text.
Delete	Remove an object from the Wave and List windows
Clear	Clear the Transcript window
Select All Unselect All	Change the selection of items in a window
Expand	Expand or collapse hierarchy information
Goto	Goto a specific line number in the Source window
Find	Open the find toolbar. Refer to the section “ Using the Find and Filter Functions ” for more information
Replace	Find and replace text in a Source window.
Signal Search	Search the Wave or List windows for a specified value, or the next transition for the selected object
Find in Files	search for text in saved files
Previous Coverage Miss Next Coverage Miss	Find the previous or next line with missed coverage in the active Source window

View Menu

Table 2-9. View Menu — Item Description

Menu Item	Description
<i>window name</i>	Displays the selected window
New Window	Open additional instances of the Wave, List, Schematic or Dataflow windows
Sort	Change the sort order of the Wave window
Filter	Filters information from the Objects and Structure windows.
Justify	Change the alignment of data in the selected window.
Properties	Displays file property information from the Files or Source windows.

Compile Menu

Table 2-10. Compile Menu — Item Description

Menu Item	Description
Compile	Compile source files
Compile Options	Set various compile options.
SystemC Link	Collect the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library
Compile All	Compile all files in the open project. Disabled if you don't have a project open
Compile Selected	Compile the files selected in the project tab. Disabled if you don't have a project open
Compile Order	Set the compile order of the files in the open project. Disabled if you don't have a project open
Compile Report	report on the compilation history of the selected file(s) in the project. Disabled if you don't have a project open
Compile Summary	report on the compilation history of all files in the project. Disabled if you don't have a project open

Simulate menu

Table 2-11. Simulate Menu — Item Description

Menu item	Description
Design Optimization	Open the Design Optimization dialog to configure simulation optimizations
Start Simulation	Load the selected design unit
Runtime Options	Set various simulation runtime options

Table 2-11. Simulate Menu — Item Description (cont.)

Menu item	Description
Run	<ul style="list-style-type: none"> • Run <default> — run simulation for one default run length; change the run length with Simulate > Runtime Options, or use the Run Length text box on the toolbar • Run -All — run simulation until you stop it • Continue — continue the simulation • Run -Next — run to the next event time • Step — single-step the simulator • 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 various objects (logged signals, breakpoints, etc.)
Break	Stop the current simulation run
End Simulation	Quit the current simulation run

Add Menu

Table 2-12. Add Menu — Item Description

Menu Item	Description
To Wave	Add information to the Wave window
To List	Add information to the List window
To Log	Add information to the Log file
To Schematic	Add information to the Schematic window
To Dataflow	Add information to the Dataflow window
Window Pane	Add an additional pane to the Wave window. You can remove this pane by selecting Wave > Delete Window Pane .

Tools Menu

Table 2-13. Tools Menu — Item Description

Menu Item	Description
Waveform Compare	Access tasks for waveform comparison. Refer to the section “ Waveform Compare ” for more information.
Code Coverage	Access tasks for code coverage. Refer to the chapter “ Code Coverage ” for more information.
Functional Coverage	Access menus for configuring cover directives and filtering functional coverage items. See “ Configuring Covergroups, Coverpoints, and Crosses ” and “ Filtering Functional Coverage Data ” for more information.
Toggle Coverage	Add toggle coverage tracking. Refer to the section “ Toggle Coverage ” for more information.
Coverage Save	Save the coverage metrics to a UCDB file.
Coverage Report	Save the coverage metrics to report file.
Profile	Access tasks for the memory and performance profilers. Refer to the chapter “ Profiling Performance and Memory Use ” for more information.
Breakpoints	Manage breakpoints
Trace	Perform signal trace actions.
Dataset Snapshot	Enable periodic saving of simulation data to a .wlf file.
C Debug	Access tasks for the C Debug interface. Refer to the chapter “ C Debug ” for more information.

Table 2-13. Tools Menu — Item Description (cont.)

Menu Item	Description
JobSpy	Access tasks for the JobSpy utility. Refer to the chapter “ Monitoring Simulations with JobSpy ” for more information.
Tcl	Execute or debug a Tcl macro.
Wildcard Filter	Refer to the section “ Using the WildcardFilter Preference Variable ” for more information
Edit Preferences	Set GUI preference variables. Refer to the section “ Simulator GUI Preferences ” for more information.

Layout Menu

Table 2-14. Layout Menu — Item Description

Menu Item	Description
Reset	Reset the GUI to the default appearance for the selected layout.
Save Layout As	Save your reorganized view to a custom layout. Refer to the section “ Customizing the Simulator GUI Layout ” for more information.
Configure	Configure the layout-specific behavior of the GUI. Refer to the section “ Configure Window Layouts Dialog Box ” for more information.
Delete	Delete a customized layout. You can not delete any of the five standard layouts.
<i>layout name</i>	Select a standard or customized layout.

Bookmarks Menu

Table 2-15. Bookmarks Menu — Item Description

Menu Item	Description
Add	Clicking this button bookmarks the current view of the Wave window.
Add Custom	Opens the New Bookmark dialog box.
Manage	Opens the Manage Bookmarks dialog box.
Delete All	<ul style="list-style-type: none">• Active Window Only• All Windows.
Reload from File	<ul style="list-style-type: none">• Active Window Only• All Windows.

Window Menu

Table 2-16. Window Menu — Item Description

Menu Item	Description
Cascade Tile Horizontally Tile Vertically	Arrange all undocked windows. These options do not impact any docked windows.
Icon Children Icon All Deicon All	Minimize (Icon) or Maximize (Deicon) undocked windows. These options do not impact any docked windows.

Table 2-16. Window Menu — Item Description (cont.)

Menu Item	Description
Show Toolbar	Toggle the appearance of the Toolbar frame of the Main window
Show Window Headers	Toggle the appearance of the window headers. Note that you will be unable to rearrange windows if you do not show the window headers.
FocusFollowsMouse	Mouse pointer makes window active when pointer hovers in the window briefly. Refer to Navigating in the Main Window for more information.
Keyboard Shortcuts	Opens the Keyboard Shortcuts dialog box. Refer to User Defined Keyboard Shortcuts for more information.
Customize Toolbar	Add a button to the toolbar frame.
Toolbars	Toggle the appearance of available toolbars. Similar behavior to right-clicking in the toolbar frame.
<i>window name</i>	Make the selected window active.
Windows	Display the Windows dialog box, which allows you to activate, close or undock the selected window(s).

Help Menu

Table 2-17. Help Menu — Item Description

Menu Item	Description
About	Display ModelSim application information.
Release Notes	Display the current Release Notes in the ModelSim Notepad editor. You can find past release notes in the <code><install_dir>/docs/rlsnotes/</code> directory.
Welcome Window	Display the Important Information splash screen. By default this window is displayed on startup. You can disable the automatic display by toggling the Don't show this dialog again radio button.
Command Completion	Toggles the command completion dropdown box in the transcript window. When you start typing a command at the Transcript prompt, a dropdown box appears which lists the available commands matching what has been typed so far. You may use the Up and Down arrow keys or the mouse to select the desired command. When a unique command has been entered, the command usage is presented in the drop down box.
Register File Types	Associate files types (such as <code>.v</code> , <code>.vhd</code> , <code>.do</code>) with the product. These associations are typically made upon install, but this option allows you to update your system in case changes have been made since installation.
ModelSim Documentation - InfoHub	Open the HTML-based portal for all PDF and HTML documentation.
ModelSim Documentation - PDF Bookcase	Open the PDF-based portal for the most commonly used PDF documents.
Tcl Help	Open the Tcl command reference (man pages) in Windows help format.
Tcl Syntax	Open the Tcl syntax documentation in your web browser.
Tcl Man pages	Open the Tcl/Tk manual in your web browser.
Technotes	Open a technical note in the ModelSim Notepad editor.

Main Window Toolbar

The Main window contains a toolbar frame that displays context-specific toolbars. The following sections describe the toolbars and their associated buttons.

- [ATV Toolbar](#)
- [Analysis Toolbar](#)
- [Bookmarks Toolbar](#)
- [Column Layout Toolbar](#)
- [Compile Toolbar](#)
- [Coverage Toolbar](#)
- [FSM Toolbar](#)
- [Help Toolbar](#)
- [Layout Toolbar](#)
- [Memory Toolbar](#)
- [Mode Toolbar](#)
- [Objectfilter Toolbar](#)
- [Process Toolbar](#)
- [Profile Toolbar](#)
- [Schematic Toolbar](#)
- [Simulate Toolbar](#)
- [Source Toolbar](#)
- [Standard Toolbar](#)
- [Step Toolbar](#)
- [Wave Compare Toolbar](#)
- [Wave Cursor Toolbar](#)
- [Wave Edit Toolbar](#)
- [Wave Expand Time Toolbar](#)
- [Wave Toolbar](#)
- [Zoom Toolbar](#)

ATV Toolbar

The ATV toolbar allows you to control aspects of the Assertion Thread Viewer.

Figure 2-27. ATV Toolbar



Table 2-18. ATV Toolbar Buttons

Button	Name	Shortcuts	Description
	View Grid	Menu: ATV > View Grid	creates horizontal grid lines from the assertion expression through the Thread Viewer pane
	Ascending Expressions	Menu: ATV > Ascending Expressions	changes the display of expressions in the Expressions pane from descending order (default) to ascending order
	Annotate Local Vars	Menu: ATV > Annotate Local Vars	displays annotation of local variables in the Thread Viewer pane
	Show Local Vars	Menu: ATV > Show Local Vars	displays the Local Variables pane at the bottom of the viewer
	Show Design Objects	Menu: ATV > Show Design Objects	displays the Design Objects pane at the bottom of the ATVviewer

Analysis Toolbar

The Analysis (coverage) toolbar allows you to control aspects of the Code Coverage Analysis window.

Figure 2-28. Analysis Toolbar

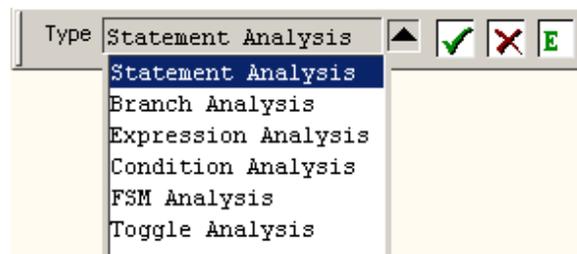


Table 2-19. Analysis Toolbar Buttons

Button	Name	Shortcuts	Description
	Type	Command: view canalysis	A dropdown box that allows you to specify the type of code coverage to view in the Code Coverage Analysis Window .
	Covered	Menu: N/A	All covered (hit) items are displayed.
	Missed	Menu: N/A	All missed items (not executed) are displayed.
	Excluded	Menu: N/A	Restores the precision to the default value (2).

Bookmarks Toolbar

The Bookmark toolbar allows you to manage your bookmarks of the Wave window

Figure 2-29. Bookmarks Toolbar



Table 2-20. Bookmarks Toolbar Buttons

Button	Name	Shortcuts	Description
	Add Bookmark	Command Wave window only: bookmark add wave Menu Wave window only: Add > To Wave > Bookmark	Clicking this button bookmarks the current view of the active window. Click and hold the button to open a drop down menu with the following options: <ul style="list-style-type: none"> • Add Current View • Add Custom ... • Set Default Action

Table 2-20. Bookmarks Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Delete All Bookmarks	CommandWave window only: bookmark delete wave -all	Removes all bookmarks, after prompting for your confirmation. Click and hold the button to open a drop down menu with the following options: <ul style="list-style-type: none"> • Active Window • All Windows
	Manage Bookmarks	None	Displays the Manage Bookmarks dialog box .
	Reload from File	None	Reloads bookmarks from the bookmarks.do file. <ul style="list-style-type: none"> • Set Default Action
	Jump to Bookmark	CommandWave window only: bookmark goto wave <name>	Displays bookmarks grouped by window. Select the bookmark you want to display.

Column Layout Toolbar

The Column Layout toolbar allows you to specify the column layout for the active window.

Figure 2-30. Column Layout Toolbar

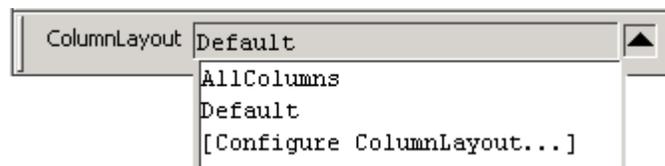
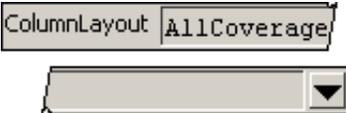


Table 2-21. Change Column Toolbar Buttons

Button	Name	Shortcuts	Description
	Column Layout	Menu: Verification Browser > Configure Column Layout	A dropdown box that allows you to specify the column layout for the active window.

The Column Layout dropdown menu allows you to select pre-defined column layouts for the active window. For example, the layouts you can select for the [Verification Management Browser Window](#) include:

- AllColumns — displays all available columns.
- Default — displays only columns that are displayed by default.
- CodeCoverageRanked — displays all columns related to ranked code coverage results.
- FunctionalCoverageRanked - displays all columns related to ranked functional coverage results.
- AllCoverage — displays all columns related to coverage statistics.
- TestData — displays all columns containing data about the test, including information about how and when the coverage data was generated.
- Testplan — displays all columns containing data about the testplan, including Testplan Section / Coverage Link, Type, Goal, % of Goal, Weight.
- AllCoverageRanked — displays all columns related to ranking results.
- FunctionalCoverage — displays all columns related to functional coverage statistics.
- CodeCoverage — displays all columns related to code coverage statistics.
- [Configure ColumnLayout . . .] — opens the Configure Column Layout dialog, which allows you to create, edit, remove, copy, or rename a column layout. See [Configuring the Column Layout](#).

Compile Toolbar

The Compile toolbar provides access to compile and simulation actions.

Figure 2-31. Compile Toolbar



Table 2-22. Compile Toolbar Buttons

Button	Name	Shortcuts	Description
	Compile	Command: <code>vcom</code> or <code>vlog</code> Menu: Compile > Compile	Opens the Compile Source Files dialog box.
	Compile All	Command: <code>vcom</code> or <code>vlog</code> Menu: Compile > Compile all	Compiles all files in the open project.
	Simulate	Command: <code>vsim</code> Menu: Simulate > Start Simulation	Opens the Start Simulation dialog box.

Table 2-22. Compile Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Break	Menu: Simulate > Break Hotkey: Break	Stop a compilation, elaboration, or the current simulation run.

Coverage Toolbar

The Coverage toolbar provides tools for filtering code coverage data in the Structure and Instance Coverage windows.

Figure 2-32. Coverage Toolbar

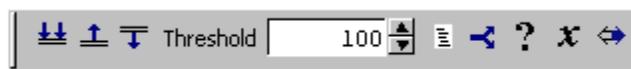


Table 2-23. Coverage Toolbar Buttons

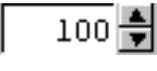
Button	Name	Shortcuts	Description
	Enable Filtering	None	Enables display filtering of coverage statistics in the Structure and Instance Coverage windows.
	Threshold Above	None	Displays all coverage statistics above the Filter Threshold for selected columns.
	Threshold Below	None	Displays all coverage statistics below the Filter Threshold for selected columns
	Filter Threshold	None	Specifies the display coverage percentage for the selected coverage columns
	Statement	None	Applies the display filter to all Statement coverage columns in the Structure and Instance Coverage windows.
	Branch	None	Applies the display filter to all Branch coverage columns in the Structure and Instance Coverage windows.

Table 2-23. Coverage Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Condition	None	Applies the display filter to all Condition coverage columns in the Structure and Instance Coverage windows.
	Expression	None	Applies the display filter to all Expression coverage columns in the Structure and Instance Coverage windows.
	Toggle	None	Applies the display filter to all Toggle coverage columns in the Structure and Instance Coverage windows.

FSM Toolbar

The FSM toolbar provides access to tools that control the information displayed in the FSM Viewer window.

Figure 2-33. FSM Toolbar

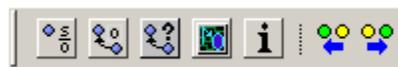


Table 2-24. FSM Toolbar Buttons

Button	Name	Shortcuts	Description
	Show State Counts	Menu: FSM View > Show State Counts	(only available when simulating with -coverage) Displays the coverage count over each state.
	Show Transition Counts	Menu: FSM View > Show Transition Counts	(only available when simulating with -coverage) Displays the coverage count for each transition.
	Show Transition Conditions	Menu: FSM View > Show Transition Conditions	Displays the conditions of each transition.
	Track Wave Cursor	Menu: FSM View > Track Wave Cursor	The FSM Viewer tracks your current cursor location.

Table 2-24. FSM Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Enable Info Mode Popups	Menu: FSM View > Enable Info Mode Popups	Displays information when you mouse over each state or transition
	Previous State	None	Steps to the previous state in the FSM Viewer window.
	Next State	None	Steps to the next state in the FSM Viewer window.

Help Toolbar

The Help toolbar provides a way for you to search the HTML documentation for a specified string. The HTML documentation will be displayed in a web browser.

Figure 2-34. Help Toolbar



Table 2-25. Help Toolbar Buttons

Button	Name	Shortcuts	Description
	Search Documentation	None	A text entry box for your search string.
	Search Documentation	Hotkey: Enter	Activates the search for the term you entered into the text entry box.

Layout Toolbar

The Layout toolbar allows you to select a predefined or user-defined layout of the graphical user interface. Refer to the section “[Customizing the Simulator GUI Layout](#)” for more information.

Figure 2-35. Layout Toolbar

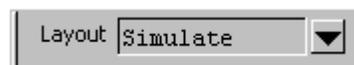


Table 2-26. Layout Toolbar Buttons

Button	Name	Shortcuts	Description
	Change Layout	Menu: Layout > <layoutName>	A dropdown box that allows you to select a GUI layout. <ul style="list-style-type: none"> • NoDesign • Simulate • Coverage • VMgmt

Memory Toolbar

The Memory toolbar provides access to common functions.

Figure 2-36. Memory Toolbar



Table 2-27. Memory Toolbar Buttons

Button	Name	Shortcuts	Description
	Split Screen	Menu: Memory > Split Screen	Splits the memory window.
	Goto Address		Highlights the first element of the specified address.

Mode Toolbar

The Mode toolbar provides access to tools for controlling the mode of mouse navigation.

Figure 2-37. Mode Toolbar



Table 2-28. Mode Toolbar Buttons

Button	Name	Shortcuts	Description
	Select Mode	Menu: Dataflow or Schematic > Mouse Mode > Select Mode	Set the left mouse button to select mode and middle mouse button to zoom mode.

Table 2-28. Mode Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Zoom Mode	Menu: Dataflow or Schematic > Mouse Mode > Zoom Mode	Set left mouse button to zoom mode and middle mouse button to pan mode.
	Pan Mode	Menu: Dataflow or Schematic > Mouse Mode > Pan Mode	Set left mouse button to pan mode and middle mouse button to zoom mode.
	Edit Mode	Menu: Wave or Dataflow > Mouse Mode > Edit Mode	Set mouse to Edit Mode, where you drag the left mouse button to select a range and drag the middle mouse button to zoom.
	Stop Drawing	None	Halt any drawing currently happening in the window.

Objectfilter Toolbar

The Objectfilter toolbar provides filtering of design objects appearing in the Objects window.

Figure 2-38. Objectfilter Toolbar



Table 2-29. Objectfilter Toolbar Buttons

Button	Name	Shortcuts	Description
	View Inputs Only	None	Changes the view of the Objects Window to show inputs.
	View Outputs Only	None	Changes the view of the Objects Window to show outputs.
	View Inouts Only	None	Changes the view of the Objects Window to show inouts.
	Vies Internal Signals	None	Changes the view of the Objects Window to show Internal Signals.
	Reset All Filters	None	Clears the filtering of Objects Window entries and displays all objects.

Table 2-29. Objectfilter Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Change Filter	None	Opens the Filter Objects dialog box.

Precision Toolbar

The Precision toolbar allows you to control the precision of the data in the Verification Management windows (Browser, Tracker, Results Analysis).

Figure 2-39. Precision Toolbar

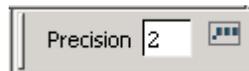


Table 2-30. Precision Toolbar Buttons

Button	Name	Shortcuts	Description
	Set Precision for VMgmt	Menu: Verification Browser > Set Precision	A text entry box that allows you to control the precision of the data in the Verification Browser window.
	Restore Default Precision		Restores the precision to the default value (2).

Process Toolbar

The Process toolbar contains three toggle buttons (only one can be active at any time) that controls the view of the Process window.

Figure 2-40. Process Toolbar



Table 2-31. Process Toolbar Buttons

Button	Name	Shortcuts	Description
	View Active Processes	Menu: Process > Active	Changes the view of the Processes Window to only show active processes.

Table 2-31. Process Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	View Processes in Region	Menu: Process > In Region	Changes the view of the Processes window to only show processes in the active region.
	View Processes for the Design	Menu: Process > Design	Changes the view of the Processes window to show processes in the design.
	View Process hierarchy	Menu: Process > Hierarchy	Changes the view of the Processes window to show process hierarchy.

Profile Toolbar

The Profile toolbar provides access to tools related to the profiling windows (Ranked, Calltree, Design Unit, and Structural).

Figure 2-41. Profile Toolbar



Table 2-32. Profile Toolbar Buttons

Button	Name	Shortcuts	Description
	Collapse Sections	Menu: Tools > Profile > Collapse Sections	Toggle the reporting for collapsed processes and functions.
	Profile Cutoff	None	Display performance and memory profile data equal to or greater than set percentage.
	Refresh Profile Data	None	Refresh profile performance and memory data after changing profile cutoff.
	Save Profile Results	Menu: Tools > Profile > Profile Report	Save profile data to output file (prompts for file name).

Schematic Toolbar

The Schematic toolbar provides access to tools for manipulating highlights and signals in the Dataflow and Schematic windows.

Figure 2-42. Schematic Toolbar



Table 2-33. Schematic Toolbar Buttons

Button	Name	Shortcuts	Description
	Trace Input Net to Event	Menu: Tools > Trace > Trace next event	Move the next event cursor to the next input event driving the selected output.
	Trace Set	Menu: Tools > Trace > Trace event set	Jump to the source of the selected input event.
	Trace Reset	Menu: Tools > Trace > Trace event reset	Return the next event cursor to the selected output.
	Trace Net to Driver of X	Menu: Tools > Trace > TraceX	Step back to the last driver of an unknown value.
	Expand Net to all Drivers	None	Display driver(s) of the selected signal, net, or register.
	Expand Net to all Drivers and Readers	None	Display driver(s) and reader(s) of the selected signal, net, or register.
	Expand Net to all Readers	None	Display reader(s) of the selected signal, net, or register.
	Remove All Highlights	Menu: Dataflow > Remove Highlight or Schematic > Edit > Remove Highlight	<p>Clear the red highlighting identifying the path you've traversed through the design.</p> <p>Click and hold the button to open a drop down menu with the following options:</p> <ul style="list-style-type: none"> • Remove All Highlights • Remove Selected Highlights • Set Default Action

Table 2-33. Schematic Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Delete Content	Menu: Dataflow > Delete or Schematic > Edit > Delete Schematic > Edit > Delete All	Delete the selected signal. Click and hold the button to open a drop down menu with the following options: <ul style="list-style-type: none"> • Delete Selected • Delete All • Set Default Action
	Regenerate	Menu: Dataflow > Regenerate or Schematic > Edit > Regenerate	Clear and redraw the display using an optimal layout.
	Enable 1-Click Mode		Enables single-click sprout expansion. Default is double-click to sprout.
	Show Wave	Menu: Schematic > Show Wave	Display the embedded Wave Viewer pane.

Simulate Toolbar

The Simulate toolbar provides various tools for controlling your active simulation.

Figure 2-43. Simulate Toolbar

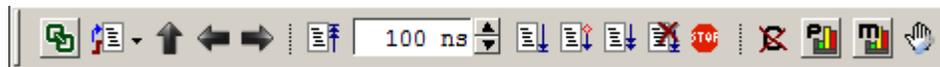


Table 2-34. Simulate Toolbar Buttons

Button	Name	Shortcuts	Description
	Source Hyperlinking	None	Toggles display of hyperlinks in design source files.
	Show Cause	Command: find drivers -active	Traces a selected wave signal from the current time back to the first sequential element. See Show Cause Button for more information. <ul style="list-style-type: none"> • Set Default Action

Table 2-34. Simulate Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Environment Up	Command: env .. Menu: File > Environment	Changes your environment up one level of hierarchy.
	Environment Back	Command: env -back Menu: File > Environment	Change your environment to its previous location.
	Environment Forward	Command: env -forward Menu: File > Environment	Change your environment forward to a previously selected environment.
	Restart	Command: restart Menu: Simulate > Run > Restart	Reload the design elements and reset the simulation time to zero, with the option of maintaining various settings and objects.
	Run Length	Command: run Menu: Simulate > Runtime Options	Specify the run length for the current simulation.
	Run	Command: run Menu: Simulate > Run > Run <i>default_run_length</i>	Run the current simulation for the specified run length.
	Continue Run	Command: run -continue Menu: Simulate > Run > Continue	Continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event.
	Run All	Command: run -all Menu: Simulate > Run > Run -All	Run the current simulation forever, or until it hits a breakpoint or specified break event.
	Break	Menu: Simulate > Break Hotkey: Break	Immediate stop of a compilation, elaboration, or simulation run. Similar to hitting a breakpoint if the simulator is in the middle of a process.
	Stop -sync	None	Stop simulation the next time time/delta is advanced.
	C Interrupt	Command: cdbg interrupt Menu: Tools > C Debug > C Interrupt	Reactivate the C debugger when stopped in HDL code.

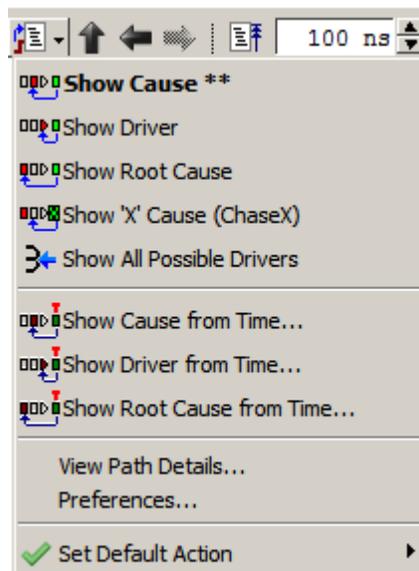
Table 2-34. Simulate Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Performance Profiling	Menu: Tools > Profile > Performance	Enable collection of statistical performance data.
	Memory Profiling	Menu: Tools > Profile > Memory	Enable collection of memory usage data.
	Edit Breakpoints	Menu: Tools > Breakpoint	Enable breakpoint editing, loading, and saving.

Show Cause Button

Clicking this button initiates a trace on a signal event back to the first sequential driving process. When you click-and-hold the button you can access additional options via a dropdown menu, as shown in [Figure 2-44](#).

Figure 2-44. Show Cause Dropdown Menu



- Show Cause — Refer to [“Tracing to the First Sequential Process”](#).
- Show Driver — Refer to [“Tracing to the Immediate Driving Process”](#).
- Show Root Cause — Refer to [“Tracing to the Root Cause”](#).
- Show ‘X’ Cause (ChaseX) — Refer to [“Tracing to the Root Cause of an ‘X’”](#)
- Show All Possible Drivers — Refer to [“Finding All Possible Drivers”](#).
- Show Cause from Time — Refer to [“Tracing from a Specific Time”](#).
- Show Driver from Time — Refer to [“Tracing from a Specific Time”](#).

- Show Root Cause from Time — Refer to “[Tracing from a Specific Time](#)”.
- View Path Details — Refer to “[Viewing Causality Path Details](#)”.
- Preferences — Refer to “[Setting Causality Traceback Preferences](#)”.
- Set Default Action — Selecting one of the items from the dropdown menu sets that item as the default action when you click the Event Traceback button. The title of the selection is shown in bold type in the Event Traceback dropdown menu and two asterisks (**) are placed after the title to indicate the current default action. For example, **Show Cause** is the default action in [Figure 2-44](#).

For more information, refer to “[Using Causality Traceback](#)”.

Source Toolbar

The Source toolbar allows you to perform several activities on Source windows.

Figure 2-45. Source Toolbar



Table 2-35. Source Toolbar Buttons

Button	Name	Shortcuts	Description
	Previous Zero Hits	None	Jump to previous line with zero coverage.
	Next Zero Hits	None	Jump to next line with zero coverage.
	Show Language Templates	Menu: Source > Show Language Templates	Display language templates in the left hand side of every open source file.
	Source Annotation	Menu: Source > Show Annotation	Allows Debugging with Source Annotation in every open source file.
	Clear Bookmarks	Menu: Source > Clear Bookmarks	Removes any bookmarks in the active source file.

Standard Toolbar

The Standard toolbar contains common buttons that apply to most windows.

Figure 2-46. Standard Toolbar



Table 2-36. Standard Toolbar Buttons

Button	Name	Shortcuts	Description
	New File	Menu: File > New > Source	<p>Opens a new Source text file. The icon changes to reflect the default file type set with the Set Default Action menu pick from the dropdown menu.</p> <p>Click and hold the button to open a dropdown menu with the following options:</p> <ul style="list-style-type: none"> • VHDL • Verilog • SystemC • SystemVerilog • Do • Other • Set Default Action
	Open	Menu: File > Open	Opens the Open File dialog
	Save	Menu: File > Save	<p>Saves the contents of the active window or</p> <p>Saves the current wave window display and signal preferences to a macro file (DO file).</p>
	Reload	<p>Command: Dataset Restart</p> <p>Menu: File > Datasets</p>	Reload the current dataset.
	Print	Menu: File > Print	Opens the Print dialog box.
	Cut	<p>Menu: Edit > Cut</p> <p>Hotkey: Ctrl+x</p>	
	Copy	<p>Menu: Edit > Copy</p> <p>Hotkey: Ctrl+c</p>	

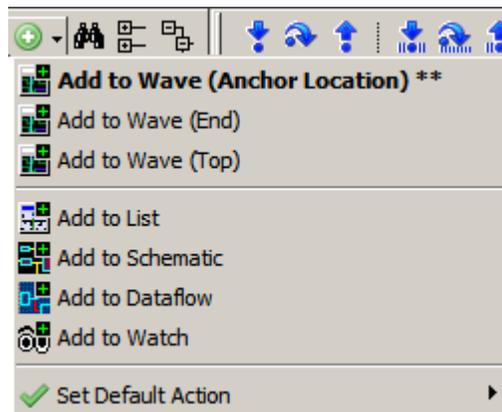
Table 2-36. Standard Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Paste	Menu: Edit > Paste Hotkey: Ctrl+v	
	Undo	Menu: Edit > Undo Hotkey: Ctrl+z	
	Redo	Menu: Edit > Redo Hotkey: Ctrl+y	
	Add Selected to Window	Menu: Add > to Wave	Clicking adds selected objects to the Wave window. Refer to “ Add Selected to Window Button ” for more information about the dropdown menu selections. <ul style="list-style-type: none"> • Set Default Action
	Find	Menu: Edit > Find Hotkey: Ctrl+f (Windows) or Ctrl+s (UNIX)	Opens the Find dialog box.
	Collapse All	Menu: Edit > Expand > Collapse All	

Add Selected to Window Button

This button is available when you have selected an object in any of the following windows: Dataflow, List, Locals, Memory, Objects, Process, Schematic, Structure, Watch, and Wave windows. Using a single click, the objects are added to the Wave window. However, if you click-and-hold the button you can access additional options via a dropdown menu, as shown in [Figure 2-47](#).

Figure 2-47. Add Selected to Window Dropdown Menu



- Add to Wave (Anchor Location) — Adds selected signals above the currently selected signal, or the first signal if no selection has been made, in the [Pathname Pane](#).
- Add to Wave (End) — Adds selected signals after the last signal in the [Wave Window](#).
- Add to Wave (Top) — Adds selected signals above the first signal in the [Wave window](#).
- Add to List — Adds selected objects to the [List Window](#).
- Add to Dataflow — Adds selected objects to the [Dataflow Window](#).
- Add to Schematic — Adds selected objects to the [Schematic Window](#).
- Add to Watch — Adds selected objects to the [Watch Window](#).
- Set Default Action — Selecting one of the items from the dropdown menu sets that item as the default action when you click the **Add Selected to Window** button. The title of the selection is shown in bold type in the **Add Selected to Window** dropdown menu and two asterisks (**) are placed after the title to indicate the current default action. For example, **Add to Wave (Anchor Location)** is the default action in [Figure 2-47](#).

Step Toolbar

The Step toolbar allows you to step through your source code.

Figure 2-48. Step Toolbar



Table 2-37. Step Toolbar Buttons

Button	Name	Shortcuts	Description
	Step Into	Command: step Menu: Simulate > Run > Step	Step the current simulation to the next statement.
	Step Over	Command: step -over Menu: Simulate > Run > Step -Over	Execute HDL statements, treating them as simple statements instead of entered and traced line by line.
	Step Out	Command: step -out	Step the current simulation out of the current function or procedure.

Table 2-37. Step Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Step Into Current	Command: <code>step -inst <env></code>	Step the current simulation into an instance, process, or thread. <ul style="list-style-type: none"> • Into current • Over current • Out current
	Step Over Current	Command: <code>step - over -inst <env></code>	Step the simulation over the current instance, process, or thread.
	Step Out Current	Command: <code>step -out -inst <env></code>	Step the simulation out of the current instance, process, or thread.

Wave Toolbar

The Wave toolbar allows you to perform specific actions in the Wave window.

Figure 2-49. Wave Toolbar



Table 2-38. Wave Toolbar Buttons

Button	Name	Shortcuts	Description
	Show Drivers	None	Display driver(s) of the selected signal, net, or register in the Dataflow, Schematic, or Wave window.
	Export Waveform	Menu: File > Export > Waveform	Export a created waveform.

Wave Compare Toolbar

The Wave Compare toolbar allows you to quickly find differences in a waveform comparison.

Figure 2-50. Wave Compare Toolbar



Table 2-39. Wave Compare Toolbar Buttons

Button	Name	Shortcuts	Description
	Find First Difference	None	Find the first difference in a waveform comparison
	Find Previous Annotated Difference	None	Find the previous annotated difference in a waveform comparison
	Find Previous Difference	None	Find the previous difference in a waveform comparison
	Find Next Difference	None	Find the next difference in a waveform comparison
	Find Next Annotated Difference	None	Find the next annotated difference in a waveform comparison
	Find Last Difference	None	Find the last difference in a waveform comparison

Wave Cursor Toolbar

The Wave Cursor toolbar provides various tools for manipulating cursors in the Wave window.

Figure 2-51. Wave Cursor Toolbar



Table 2-40. Wave Cursor Toolbar Buttons

Button	Name	Shortcuts	Description
	Insert Cursor	None	Adds a new cursor to the active Wave window.
	Delete Cursor	Menu: Wave > Delete Cursor	Deletes the active cursor.

Table 2-40. Wave Cursor Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Find Previous Transition	Menu: Edit > Signal Search Hotkey: Shift + Tab	Moves the active cursor to the previous signal value change for the selected signal.
	Find Next Transition	Menu: Edit > Signal Search Hotkey: Tab	Moves the active cursor to the next signal value change for the selected signal.
	Find Previous Falling Edge	Menu: Edit > Signal Search	Moves the active cursor to the previous falling edge for the selected signal.
	Find Next Falling Edge	Menu: Edit > Signal Search	Moves the active cursor to the next falling edge for the selected signal.
	Find Previous Rising Edge	Menu: Edit > Signal Search	Moves the active cursor to the previous rising edge for the selected signal.
	Find Next Rising Edge	Menu: Edit > Signal Search	Moves the active cursor to the next rising edge for the selected signal.

Wave Edit Toolbar

The Wave Edit toolbar provides easy access to tools for modifying an editable wave.

Figure 2-52. Wave Edit Toolbar



Table 2-41. Wave Edit Toolbar Buttons

Button	Name	Shortcuts	Description
	Insert Pulse	Menu: Wave > Wave Editor > Insert Pulse Command: wave edit insert_pulse	Insert a transition at the selected time.
	Delete Edge	Menu: Wave > Wave Editor > Delete Edge Command: wave edit delete	Delete the selected transition.

Table 2-41. Wave Edit Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Invert	Menu: Wave > Wave Editor > Invert Command: wave edit invert	Invert the selected section of the waveform.
	Mirror	Menu: Wave > Wave Editor > Mirror Command: wave edit mirror	Mirror the selected section of the waveform.
	Change Value	Menu: Wave > Wave Editor > Value Command: wave edit change_value	Change the value of the selected section of the waveform.
	Stretch Edge	Menu: Wave > Wave Editor > Stretch Edge Command: wave edit stretch	Move the selected edge by increasing/decreasing waveform duration.
	Move Edge	Menu: Wave > Wave Editor > Move Edge Command: wave edit move	Move the selected edge without increasing/decreasing waveform duration.
	Extend All Waves	Menu: Wave > Wave Editor > Extend All Waves Command: wave edit extend	Increase the duration of all editable waves.

Wave Expand Time Toolbar

The Wave Expand Time toolbar provides access to enabling and controlling wave expansion features.

Figure 2-53. Wave Expand Time Toolbar



Table 2-42. Wave Expand Time Toolbar Buttons

Button	Name	Shortcuts	Description
	Expanded Time Off	Menu: Wave > Expanded Time > Off	turns off the expanded time display (default mode)
	Expanded Time Deltas Mode	Menu: Wave > Expanded Time > Deltas Mode	displays delta time steps

Table 2-42. Wave Expand Time Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Expanded Time Events Mode	Menu: Wave > Expanded Time > Events Mode	displays event time steps
	Expand All Time	Menu: Wave > Expanded Time > Expand All	expands simulation time over the entire simulation time range, from 0 to current time
	Expand Time at Active Cursor	Menu: Wave > Expanded Time > Expand Cursor	expands simulation time at the simulation time of the active cursor
	Collapse All Time	Menu: Wave > Expanded Time > Collapse All	collapses simulation time over entire simulation time range
	Collapse Time at Active Cursor	Menu: Wave > Expanded Time > Collapse Cursor	collapses simulation time at the simulation time of the active cursor

Zoom Toolbar

The Zoom toolbar allows you to change the view of the Wave window.

Figure 2-54. Zoom Toolbar



Table 2-43. Zoom Toolbar Buttons

Button	Name	Shortcuts	Description
	Zoom In	Menu: Wave > Zoom > Zoom In Hotkey: i, I, or +	Zooms in by a factor of 2x
	Zoom Out	Menu: Wave > Zoom > Zoom Out Hotkey: o, O, or -	Zooms out by a factor of 2x
	Zoom Full	Menu: Wave > Zoom > Zoom Full Hotkey: f or F	Zooms to show the full length of the simulation.

Table 2-43. Zoom Toolbar Buttons (cont.)

Button	Name	Shortcuts	Description
	Zoom in on Active Cursor	Menu: Wave > Zoom > Zoom Cursor Hotkey: c or C	Zooms in by a factor of 2x, centered on the active cursor
	Zoom Other Window		Changes the view in additional instances of the Wave window to match the view of the active Wave window.

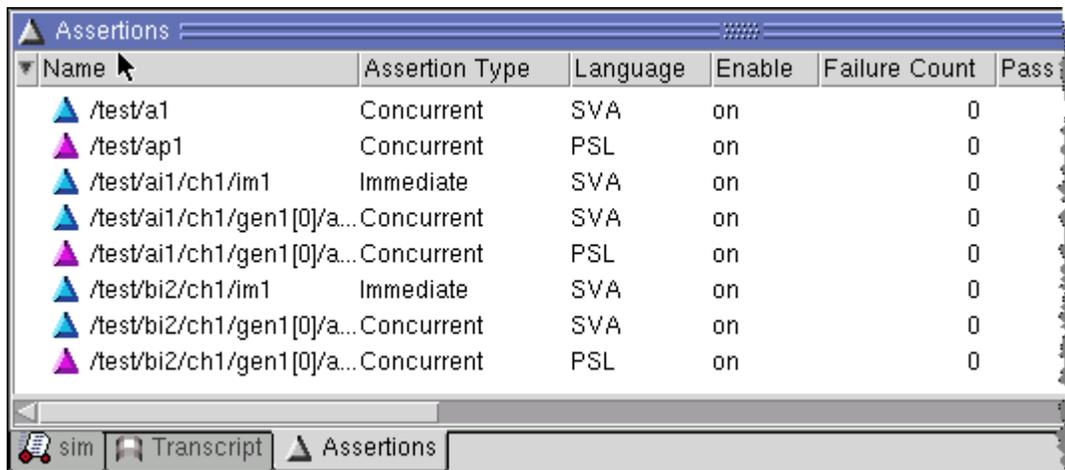
Assertions Window

Use this window to view all embedded and external assertions that were successfully compiled and simulated during the current session.

Accessing

- Menu item: **View > Coverage > Assertions**
- Command: view assertions

Figure 2-55. Assertions Window



Assertions Window Tasks

Refer to the section [“Viewing Assertions in the Assertions Window”](#) for more information.

GUI Elements of the Assertions Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-44. Assertions Window Columns

Column Title	Description
Active Count	The number of active assertion attempts at the current time. Only enabled if the simulator is invoked with the “-assertdebug” option or the <i>.ini</i> file variable <code>AutoExclusionsDisable</code> is set to 1.
Assertion Expression	Displays the actual assertion expression.
Assertion Type	Indicates the assertion type: Immediate or Concurrent.
Attempt Count	The number of times the assertion has been attempted. This is the number of assertion clocks for clocked assertions, or the number of passes and fails for unclocked assertions. Only enabled if the simulator is invoked with either the “-assertcover” or the “-assertdebug” option for vsim, or if either the AssertionCover or AssertionDebug <i>.ini</i> file variable is set to 1.
ATV	Indicates the status of assertion thread viewing: on or off . Only enabled if the simulator is invoked with the “-assertdebug” option or the <i>.ini</i> file variable <code>AssertionDebug</code> is set to 1.
Cumulative Threads	The cumulative thread count for the assertion. This count is designed to highlight those directives that are starting too many attempts. For example, given the assertion: <pre>assert property ((@posedge clk) a => b);</pre> If ‘a’ is true throughout the simulation, then the above assertion will start a brand new attempt at every clock. An attempt, once started, will only be alive until the next clock. So this assertion will not appear abnormally high in the Memory and Peak Memory columns, but it will have a high count in the Cumulative Threads column.
Design Unit	Identifies the design unit to which the assertion is bound.
Design Unit Type	Identifies the HDL type of the design unit.

Table 2-44. Assertions Window Columns (cont.)

Column Title	Description
Disable Count	<p>The number of assertion attempts that have been disabled due to either an <i>abort</i> expression becoming true (PSL) or a <i>disable if</i> expression becoming true (SVA). Only enabled if the simulator is invoked with either the “-assertcover” or the “-assertdebug” option for vsim, or if either the AssertionCover or AssertionDebug .ini file variable is set to 1.</p>
Enable	<p>Identifies whether failure checking is active for the assertion.</p>
EOS Note	<p>Identifies when assertion directives are active at the end of simulation (EOS): on or off. Refer to the assertion active command. If the assert directive is strong, the EOS Note column will report both the "active at end of simulation" note along with a strong error message.</p>
Failure Count	<p>Total number of times the assertion has failed in the current simulation.</p>
Failure Limit	<p>The number of times the simulator will respond to a failure event on an assertion.</p>
Failure Log	<p>enabled — failure messages will be logged to the transcript. disabled — failure messages will not be logged to the transcript.</p>
Formal	<p>Indicates formal analysis has been performed. Data appears in this column only during post-process analysis (Coverage View mode). Displayed values include:</p> <ul style="list-style-type: none"> • blank — no formal analysis has been performed • assumption — indicates that property was used as an assumption in the formal analysis • conflict — indicates conflict when inconsistent data merged in formal analysis • failure — indicates formal analysis has determined that property can fail • inconclusive — indicates formal analysis has not proved or falsified the property. See Proof Radius column for amount of formal analysis performed. • proof — indicates formal analysis has proven that property cannot fail for all legal stimulus • vacuous — indicates formal analysis has proven that the antecedent of the property cannot be reached; property needs to be examined closer

Table 2-44. Assertions Window Columns (cont.)

Column Title	Description
FPSA Actions	<p>Displays a matrix of information relating the current action for each possible state: Failure, Pass, Start, and Antecedent. The field will always show four letters that indicate the current action:</p> <ul style="list-style-type: none"> • C - Continue • B - Break • E - Exit • S - Subroutine Call <p>where the order of the letters relates to the state:</p> <ul style="list-style-type: none"> • F - Failure • P - Pass • S - Start • A - Antecedent <p>For example, if you see CCSB, you can derive the following:</p> <ul style="list-style-type: none"> • Failure state - Continue action • Pass state - Continue action • Start state - Subroutine Call action • Antecedent state - Break action
Language	Identifies the HDL used to write the assertion.
Memory	Tracks the current memory used by the assertion.
Name	Identifies the assert directive you specified in the assertion code.
Pass Count	The total number of times the assertions has passed in the current simulation. Only enabled if the simulator is invoked with either the “-assertcover” or the “-assertdebug” option for vsim, or if either the AssertionCover or AssertionDebug .ini file variable is set to 1.
Pass Log	enabled — pass messages will be logged to the transcript. disabled — pass messages will not be logged to the transcript. Only enabled if the simulator is invoked with the “-assertdebug” option or the .ini file variable AssertionDebug is set to 1.
Peak Active Count	The peak simultaneously active thread count that has occurred up to the current time. Peak Active Count will also be shown in reports created with the vcover report command. Only enabled if the simulator is invoked with the “-assertdebug” option or the .ini file variable AssertionDebug is set to 1.
Peak Memory	The peak memory used by the assertion.

Table 2-44. Assertions Window Columns (cont.)

Column Title	Description
Peak Memory Time	Indicates the simulation run time at which the peak memory usage occurred.
Proof Radius	Indicates that the property has been verified to a depth of x number of cycles in the formal analysis. Shown as positive integer. Data appears in this column only during post-process analysis (Coverage View mode).
Vacuous Count	The number of assertion attempts that have succeeded vacuously, that is, if the left hand side of an implication is false on a clock edge. Only enabled if the simulator is invoked with either the “-assertcover” or the “-assertdebug” option for vsim, or if either the AssertionCover or AssertionDebug .ini file variable is set to 1.

Popup Menu

Right-click in the window to display the popup menu and select one of the following options:

Table 2-45. Assertions Window Popup Menu

Popup Menu Item	Description
Add ...	Add information about the selected assertions to the specified window.
View Source	Opens a source file for the selected assertion.
Enable ATV	Enables an assertion for use with the ATV Window.
View ATV	Opens an ATV window for the selected assertion.
Report	Generates a report about the selected assertion.
Configure	Allows you to configure the simulators behavior for the selected assertion.
Enable	Enables checking of the assertion for failures during the simulation
Failure Log	Logs failure messages (PSL only) to the transcript.
Pass Log	Logs pass messages (PSL only) to the transcript. Only enabled if the simulator is invoked with the "-assertdebug" option or the .ini file variable AssertionDebug is set to 1.
... Action	Allows you to take specified actions when the selected assertions meet certain conditions.

Table 2-45. Assertions Window Popup Menu (cont.)

Popup Menu Item	Description
Test Analysis	When a UCDB file is selected, allows you to: Find Test Hits, Find Test Misses.
XML Import Hint	Produces information about the selected assertion for you to use when generating an XML file for use with UCDB.
Expand	Expand or collapse the hierarchy.
Display Options	Allows you to control the appearance of information in the window.

ATV Window

Use this window to view the progress of assertion threads (PSL and SystemVerilog assertions) over time.

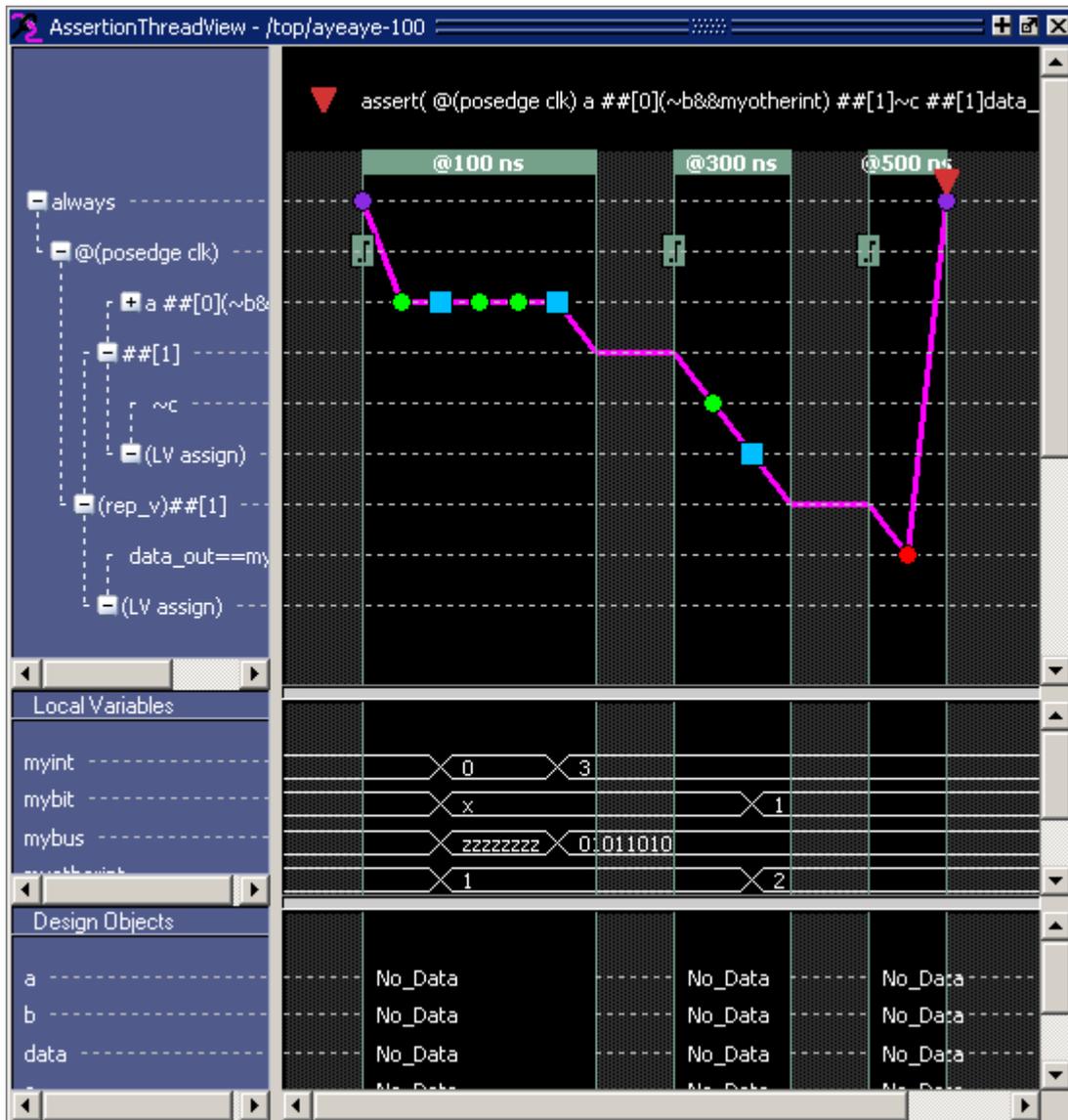
Accessing

Access the window using either of the following:

- Menu item: Assertions > Add ATV, when selecting an assertion in the [Assertions Window](#).
- Command: `add atv`

Refer to the section “[Viewing Assertion Threads in the ATV Window](#)” for detailed instructions.

Figure 2-56. ATV Window



ATV Window Tasks

Refer to the section “[Actions in the ATV Window](#)” for more information.

GUI Elements of the ATV Window

This section describes GUI elements specific to this window.

Pane Descriptions

The ATV window contains four panes:

- **Expression Pane** — shows a hierarchical representation of the assertion.
- **Thread Viewer Pane** — shows the progress of the assertion threads over time for a given thread instance and starting time.
- **Local Variables Pane** — shows any values related to local variables in the assertion.
- **Design Objects Pane** — shows values of the design objects in the expression at that time.

ATV Window Graphic Symbols

The symbols in [Table 2-46](#) indicate the current state of the assert or cover directive.

Table 2-46. Graphic Symbols for Current Directive State

Graphic Symbol	Status Information
	State: Start
	State: Active
	State: Antecedent
	State: Passed
	State: Failed

[Table 2-47](#) shows the status information that is displayed when you hover the mouse over graphic symbols used to indicate clock, thread, and directive status.

Table 2-47. Graphic Symbols for Clock, Thread, and Directive Status

Graphic Symbol	Status Information
	Directive Passed
	Directive Failed
	Root thread started/completed
	New evaluation thread forked
	Boolean passed
	Boolean failed
	clock time & clock name
	Local variable and value
	Thread failed but is redundant since other threads are still running

Table 2-47. Graphic Symbols for Clock, Thread, and Directive Status (cont.)

Graphic Symbol	Status Information
	Thread killed because directive was aborted or disabled
	Thread killed because other thread caused unilateral pass/fail
	Thread passed but directive waiting on other running threads

Popup Menu

Right-click in the window to display the popup menu and select one of the following options:

Table 2-48. ATV Window Popup Menu

Popup Menu Item	Description
View Source	Open the source file and highlight the assertion.
View Grid	Toggles the appearance of grid lines
Ascending Expressions	Toggles the display of the assertion into ascending or descending mode.
Annotate Local Vars	Displays local variable information in the Thread Viewer pane.
Show Local Vars	Toggles the display of the Local Variables pane
Show Design Objects	Toggles the display of the Design Objects pane
View Full Object Names	Toggles the display of the object names in the Design Objects pane
Add ...	Adds the assertion to the selected window
Zoom ... ¹	Controls the zoom level of the Thread Viewer pane

1. These menu items are only available when right-clicking on the right-hand side of the window

Call Stack Window

The Call Stack window displays the current call stack when:

- you single step the simulation.
- the simulation has encountered a breakpoint.
- you select any process in either the Structure or Processes windows.

When debugging your design you can use the call stack data to analyze the depth of function calls that led up to the current point of the simulation, which include:

- Verilog functions and tasks
- VHDL functions and procedures
- SystemC methods and threads
- C/C++ functions

The Call Stack window also supports [C Debug](#) mode.

Accessing

View > Call Stack

Figure 2-57. Call Stack Window



#	In	Line	File	Address
0	Module bot	30	C:/QuestaTestcases/callstackView/callstack.sv	7e83a722
1	Function f3	25	C:/QuestaTestcases/callstackView/callstack.sv	7e83a41f
2	Function f2	20	C:/QuestaTestcases/callstackView/callstack.sv	7e83a18f
3	Function f1	15	C:/QuestaTestcases/callstackView/callstack.sv	7e839efd
4	Module top	35	C:/QuestaTestcases/callstackView/callstack.sv	7e83a9b5

Call Stack Window Tasks

This window allows you to perform the following actions:

- Double-click on the line of any function call:
 - Displays the local variables at that level in the [Locals Window](#).
 - Displays the corresponding source code in the [Source Window](#).

Related Commands of the Call Stack Window

Table 2-49. Commands Related to the Call Stack Window

Command Name	Description
stack down	this command moves down the call stack.
stack frame	this command selects the specified call frame.
stack level	this command reports the current call frame number.
stack tb	this command is an alias for the tb command.
stack up	this command moves up the call stack.

GUI Elements of the Call Stack Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-50. Call Stack Window Columns

Column Title	Description
#	indicates the depth of the function call, with the most recent at the top.
In	indicates the function. If you see “unknown” in this column, you have most likely optimized the design such that the information is not available during the simulation.
Line	indicates the line number containing the function call.
File	indicates the location of the file containing the function call.
Address	indicates the address of the execution in a foreign subprogram, such as C.

Capacity Window

Use this window to display memory capacity data about your simulation. Refer to the section “[Capacity Analysis](#)” for more information.

Accessing

Access the window using either of the following:

- Menu item: **View > Capacity**

- Command: view capacity

Figure 2-58. Capacity Window

Type/Object	Count	Current Memory	Peak Memory	Peak Time
Classes	88	11.9K	11.9K	0ns
QDAS	1491	22.7K	24.9K	0ns
Assertions	8	0B	144B	5ns
Verilog Memories	132	1.8K	1.8K	0ns
Module instances	2	136B		
Function instances	2532	253.2K		
Task instances	229	21.1K		
Blocks	576	36.9K		
System task instances	945	178.4K		
Verilog Ports	16	944B		
Continuous assignment	3	396B		
Initial	73	9.3K		
Always	16	1.8K		
Nets	26	2.3K		
Parameters	1617	58.2K		
Registers	1726	75.9K		

GUI Elements of the Capacity Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-51. Capacity Window Columns

Column Title	Description
Type/Object	Refer to the section “ Type/Object Listing ”
Count	Quantity of design objects analyzed
Current Memory	Current amount of memory allocated, in bytes
Peak Memory	Peak amount of memory allocated, in bytes
Peak Time	The time, in ns, at which the peak memory was reached

Type/Object Listing

When your design contains Verilog design units you will see the following entries in the Type/Object column

- Always

- Always blocks
- Assertions — When using fine-grained analysis (vsim -capacity) this entry expands and provides information for each assertion.
- Blocks
- Classes — When using fine-grained analysis (vsim -capacity) this entry expands and provides information for each class.
- Continuous assignment
- Covergroups
- Function instances
- Initial
- Initial blocks
- Module instances
- Nets
- Parameters
- QDAs — When using fine-grained analysis (vsim -capacity) this entry expands and provides information for Queues, Dynamic Arrays, and Associative Arrays.
- Registers
- Solver
- System task instances
- Task instances
- Verilog Memories
- Verilog Ports

When your design contains VHDL design units you will see the following entries in the Type/Object column

- Instances
- Ports
- Signals
- Processes

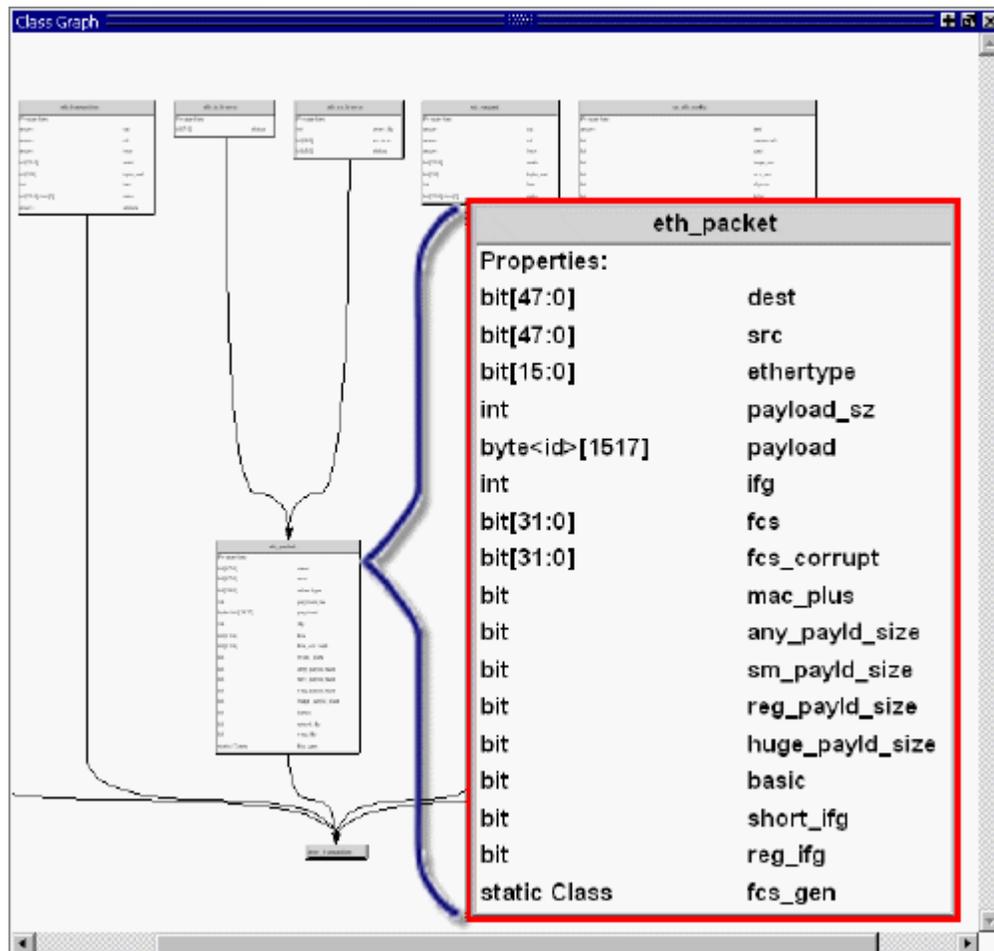
Class Graph Window

The Class Graph window provides a graphical view of your SystemVerilog classes, including any extensions of other classes and related methods and properties.

Accessing

- Menu item: **View > Class Browser > Class Graph**
- Command: view classgraph

Figure 2-59. Class Graph Window



Class Graph Window Tasks

This section describes tasks for using the Cover Directives window.

Navigating in the Class Graph Window

You can change the view of the Class Graph window with your mouse or the arrow keys on your keyboard.

- Left click-drag — allows you to move the contents around in the window.
- Middle Mouse scroll — zooms in and out.
- Middle mouse button strokes:
 - Upper left — zoom full
 - Upper right — zoom out. The length of the stroke changes the zoom factor.
 - Lower right — zoom area.
- Arrow Keys — scrolls the window in the specified direction.
 - Unmodified — scrolls by a small amount.
 - Ctrl+<arrow key> — scrolls by a larger amount
 - Shift+<arrow key> — shifts the view to the edge of the display

GUI Elements of the Class Graph Window

This section describes the GUI elements specific to the Class Graph window.

Popup Menu Items

Table 2-52. Class Graph Window Popup Menu

Popup Menu Item	Description
Filter	Controls the display of methods and properties from the class boxes.
Zoom Full	
View Entire Design	Reloads the view to show the class hierarchy of the complete design.
Print to Postscript	
Organize by Base/Extended Class	reorganizes the window so that the base or extended (default) classes are at the top of the hierarchy.

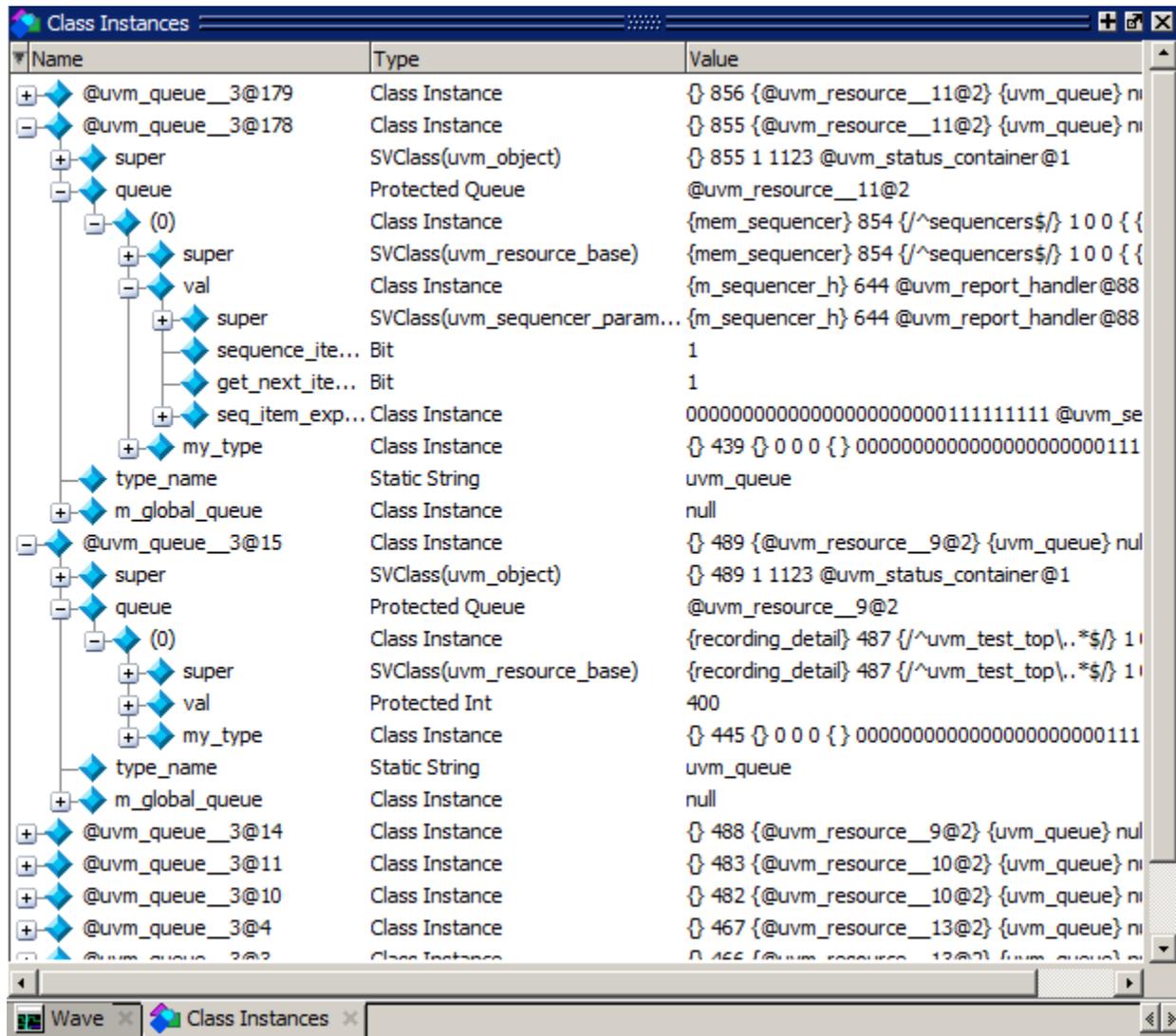
Class Instances Window

The Class Instances window shows the list of class instances and their values for a particular selected class type. You may add the class items directly to the wave or list windows, log them, or get other information about them.

Accessing

- Menu item: **View > Class Browser > Class Instances**
- Command: view classinstances

Figure 2-60. Class Instances Window



Viewing Class Instances

The Class Instances window is dynamically populated by selecting SVClasses in the Structure (sim) window. All currently active instances of the selected class are displayed in the Class Instances window. Class instances that have not yet come into existence or have been destroyed are not displayed.

Once you have chosen the class type you want to observe, you can fix that instance in the window while you debug by selecting **File > Environment > Fix to Current Context**.

Class Naming Format

Class instance names are formatted as follows: @<class_type>@<nnn> where @<class_type>@ is the name of the class type and <n> is the reference identifier for a particular instance of the class type. For example, @uvm_queue_3@14 is the 14th instance of the class uvm_queue_3.

GUI Elements of the Class Instances Window

This section describes the GUI elements specific to the Class Instances window.

Popup Menu Items

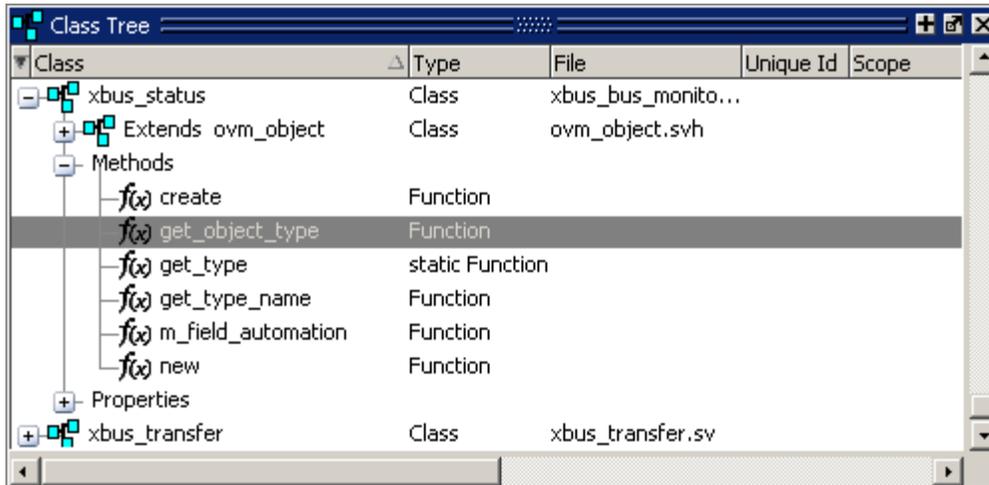
Table 2-53. Class Instances Window Popup Menu

Popup Menu Item	Description
View Declaration	Highlights the line of code where the type of the instance is declared, opening the source file if necessary.
Add Wave	Adds the selected class instance to the Wave window.
Add to	Allows you to log the selected class instance, or add it to the Wave or List windows.

Class Tree Window

The Class Tree window provides a hierarchical view of your SystemVerilog classes, including any extensions of other classes, related methods and properties, as well as any covergroups.

Figure 2-61. Class Tree Window



Accessing

- Select **View > Class Browser > Class Tree**
- Use the command:
view classtree

GUI Elements of the Class Tree Window

This section describes the GUI elements specific to the Class Tree window.

Icons

Table 2-54. Class Tree Window Icons

Icon	Description
	Class
	Parameterized Class
	Function
	Task
	Variable
	Virtual Interface
	Covergroup

Table 2-54. Class Tree Window Icons (cont.)

Icon	Description
	Structure

Column Descriptions

Table 2-55. Class Tree Window Columns

Column	Description
Class	The name of the item
Type	The type of item
File	The source location of the item
Unique Id	The internal name of the parameterized class (only available with parameterized classes)
Scope	The scope of the covergroup (only available with covergroups)

Popup Menu Items

Table 2-56. Class Tree Window Popup Menu

Popup Menu Item	Description
View Declaration	Highlights the line of code where the item is declared, opening the source file if necessary.
View as Graph	Displays the class and any dependent classes in the Class Graph window. (only available for classes)
Filter	allows you to filter out methods and or properties
Organize by Base/Extended Class	reorganizes the window so that the base or extended (default) classes are at the top of the hierarchy.

Code Coverage Analysis Window

Use this window to view covered (executed), uncovered (missed), and/or excluded statements, branches, conditions, expressions, FSM states and transitions, as well as signals that have and have not toggled.

The Code Coverage Analysis window replaces all functionality previously found in the Missing <coverage type> and Current Exclusions windows.

Prerequisites

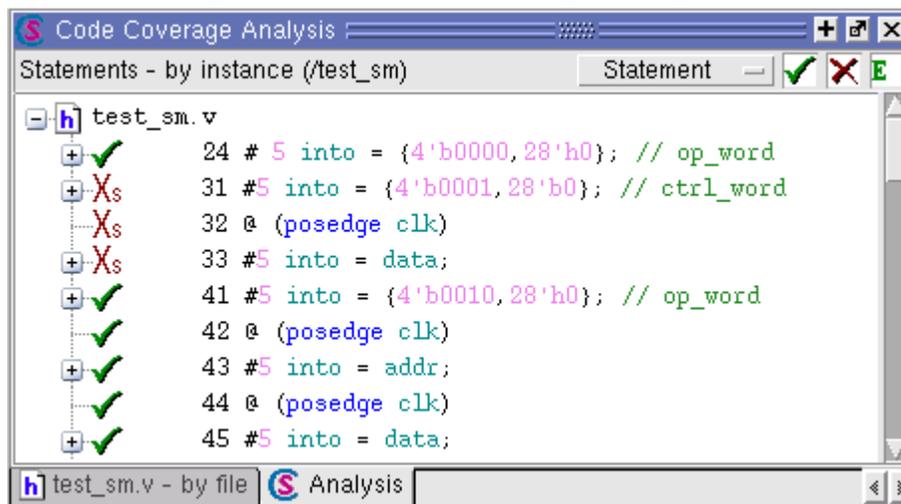
This window is specific to the collection of coverage metrics, therefore you must have run your simulation with coverage collection enabled. Refer to the “Code Coverage” chapter for more information.

Accessing

Access the window using either of the following:

- Menu item: **View > Coverage > Code Coverage Analysis**
Then, select the desired analysis type from the Code Coverage Analysis window’s title bar, detailed in [Table 2-57](#).
- Command: view canalysis

Figure 2-62. Code Coverage Analysis



The selection of icons on the right side of the banner of the Code Coverage Analysis window function as described below. By default, the icons are selected (active).

Table 2-57. Actions in Code Coverage Analysis Title Bar

Icon	Action
	<p>Analysis Type button: Specifies the type of coverage analysis currently selected in the sub-window inside the Code Coverage Analysis window. Six selections are available when you click on the type, as shown in the image to the left.</p>

Table 2-57. Actions in Code Coverage Analysis Title Bar

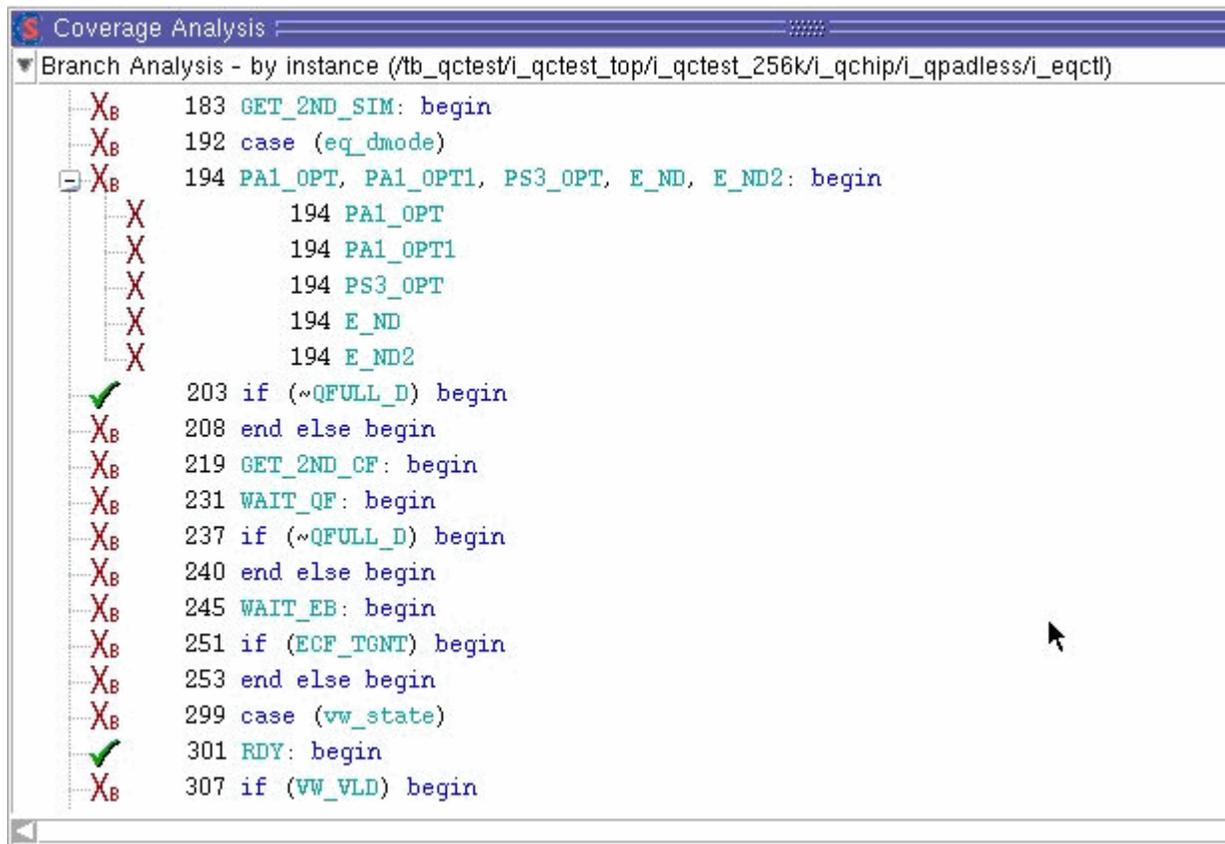
Icon	Action
	Covered items button: When selected (default, as shown in image), all covered (hit) items are displayed in the window. When not selected, all items are filtered from view.
	Missed items button: When selected (default), all missed items (not executed) are displayed in the window. When not selected, all missed items are filtered from view.
	Excluded items button: When selected (default), all excluded items are displayed in window. When not selected, excluded items are filtered from view.

Viewing Code Coverage Data and Current Exclusions

To view executed or missed statements, branches, conditions, expressions, or FSMs, as well as items excluded from coverage, do the following:

1. Select a file in the Files window, or an instance or design unit in the Structure window whose coverage you wish to analyze.
2. With the Code Coverage Analysis window active, select the type of coverage to view (Branch Analysis, Condition analysis, etc.) from the pulldown menu in the Analysis toolbar ([Figure 2-62](#)).

Figure 2-63. Missed Coverage in Code Coverage Analysis Windows



Each coverage type window includes a column for the line number and a column for statement, branch, condition, expression, or toggle on that line. An icon indicates whether the object was executed (green check mark), not executed (red X), or excluded (green E). See [Table 2-92](#) for a complete list of icons.

In the banner for all Coverage Analysis window types, the following information appears:

- name of the window
- whether the coverage is **by file** or **by instance** (depending on whether a file was selected in the **Files** tab or an instance or du from the **sim** tab)
- scope of the coverage item (in parentheses) being displayed
- Analysis Type button, Covered Items button, Missed Coverage buttons, Excluded Items button (see [Table 2-57](#))

You can change the scope displayed (for all Code Coverage Analysis windows) by selecting a new scope in the Structure or Files windows.

When you select (left-click) any item in the Statement, Branch, Condition, Expression, FSM or Toggle Analysis windows, the [Coverage Details Window](#) populates with related details

(coverage statistic details, truth tables, exclusions and so on) about that object. In the case of a multi-line statement, branch, condition or expression, select the object on the last line of the item.

The Branch Analysis window includes a column for branch code (conditional "if/then/else" and "case" statements). "X_T" indicates that the true condition of the branch was *not* executed. "X_F" indicates that the false condition of the branch was *not* executed. Fractional numbers indicate how many case statement labels were executed.

When you right-click any item in the window, a menu appears with options to control adding or removing coverage exclusions.

Note



Multi-line objects are rooted in the last line, and exclusions must be applied on that line # in order to take effect.

See "[Coverage Details Window](#)" for a description of the type of detailed information viewed in the Details window for each coverage type.\

See "[Source Window](#)" for a description of adding comments with exclusions,

Coverage Details Window

Use this window to view detailed results about coverage metrics from your simulation.

Prerequisites

This window is specific to the collection of coverage metrics, therefore you must have run your simulation with coverage collection enabled. Refer to the chapter "[Code Coverage](#)" for more information.

Accessing

Access the window using either of the following:

- Menu item: **View > Coverage > Details**
- Command: view details

You can populate this window by selecting an item in one of the panes of the [Code Coverage Analysis Window](#): either Statement, Branch, Expression, Condition, FSM or Toggle.

Coverage Details of Statement Coverage

The Coverage Details window displays the following information about [Statement Coverage](#) metrics:

- Instance — the dataset name followed by the hierarchical location of the statement. Only appears when you are analyzing coverage metrics by instance.
- File — the name of the file containing the statement.
- Line — the line number of the statement. In the case of a multi-line statement, this is the last line of the statement.
- Statement Coverage for — name of the statement itself.
- Hits — the number of times the statement was hit during the simulation.

If a line number contains multiple statements, the coverage details window contains the metrics for each statement.

Coverage Details of Branch Coverage

The Coverage Details window displays the following information about [Branch Coverage](#) metrics:

- Instance — the dataset name followed by the hierarchical location of the branch. Only appears when you are analyzing coverage metrics by instance.
- File — the name of the file containing the statement.
- Line — the line number of the statement. In the case of a multi-line branch statement, this is the last line of the statement.
- Branch Coverage for — the statement itself.
 - Active — the number of times the branch has been executed.
 - True Hits — the number of times the branch resolved to True.

Coverage Details of Condition and Expression Coverage

The Coverage Details window displays the following information about [Condition and Expression Coverage](#) metrics:

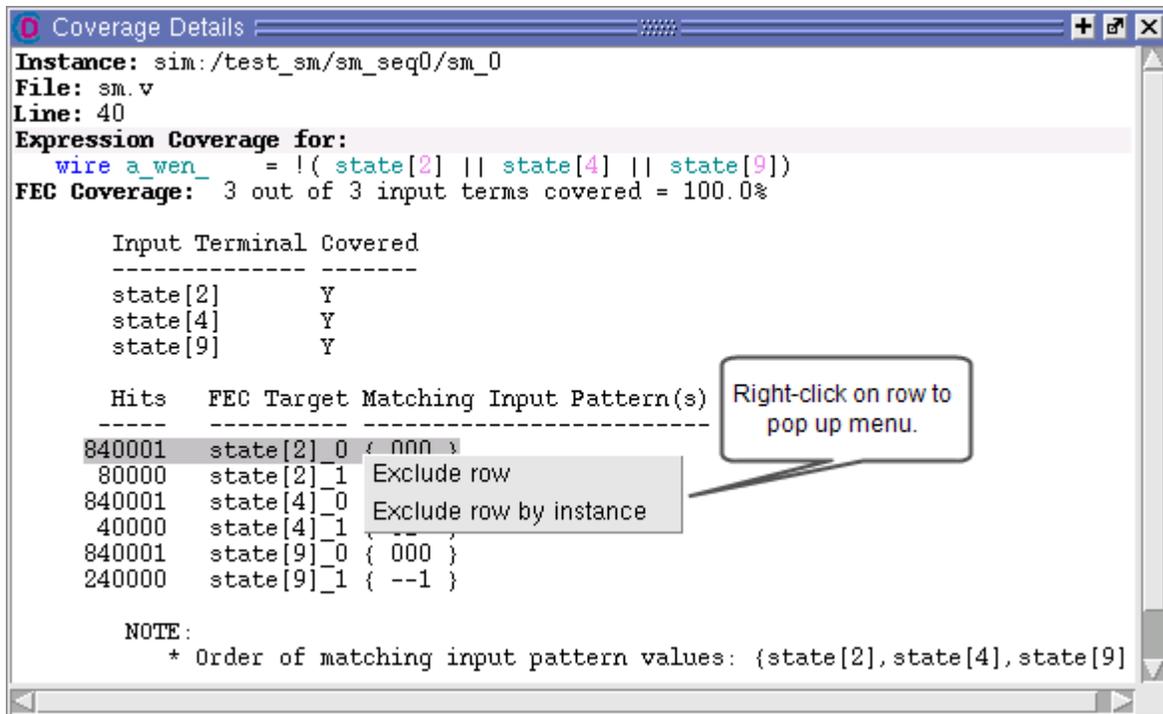
- Instance — the dataset name followed by the hierarchical location of the condition. Only appears when you are analyzing coverage metrics by instance.
- File — the filename containing the condition.
- Line — the line number of the filename containing the condition or expression. In the case of a multi-line condition statement, this is the last line of the statement.
- Condition/Expression Coverage for — the syntax of the condition.
- FEC Coverage — a tabular representation of the focused expression coverage metrics to satisfy the condition. Refer to the section “[FEC Coverage Detailed Examples](#)” for more information about FEC condition/expression coverage. You can exclude rows or rows

by instance through a popup menu accessible by right-clicking on a row in the table (see [Figure 2-64](#)).

- UDP Coverage — not included in the Details window, unless -coverudp was specified with vcom/vlog/vopt.

Refer to the section “[UDP Coverage Details and Examples](#)” for more information about UDP condition/expression coverage.

Figure 2-64. Coverage Details Window Showing Expression Truth Table



Coverage Details of Toggle Coverage

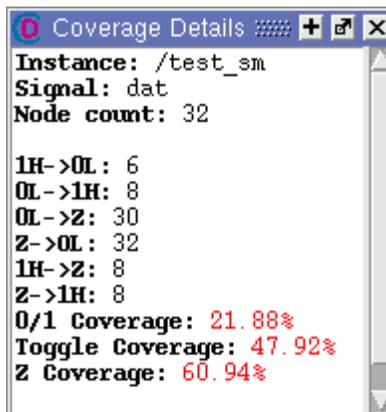
The Coverage Details window displays the following information about [Toggle Coverage](#) metrics:

- Instance — the dataset name followed by the hierarchical location of the signal. Only appears when you are analyzing coverage metrics by instance.
- Signal — the name of the signal (*data[6]*) or (*data*).
- Node Count — The size of the signal.
- Toggle List — the list of toggles analyzed during simulation. This list will differ depending on whether you specified extended toggle coverage. Refer to the section “[Standard and Extended Toggle Coverage](#)” for more information.
 - Toggle coverage shows toggle metrics between 0 and 1

- Extended toggle coverage shows toggle metrics between 0, 1 and Z.
- Toggle Coverage — The percentage of nodes that were covered.
- 0/1 Coverage — The percentage of standard toggles that were covered.
- Full Coverage — The percentage of extended toggles that were covered.
- Z Coverage — The percentage of toggles involving Z that were covered.

Toggle details are displayed as follows:

Figure 2-65. Coverage Details Window Showing Toggle Details



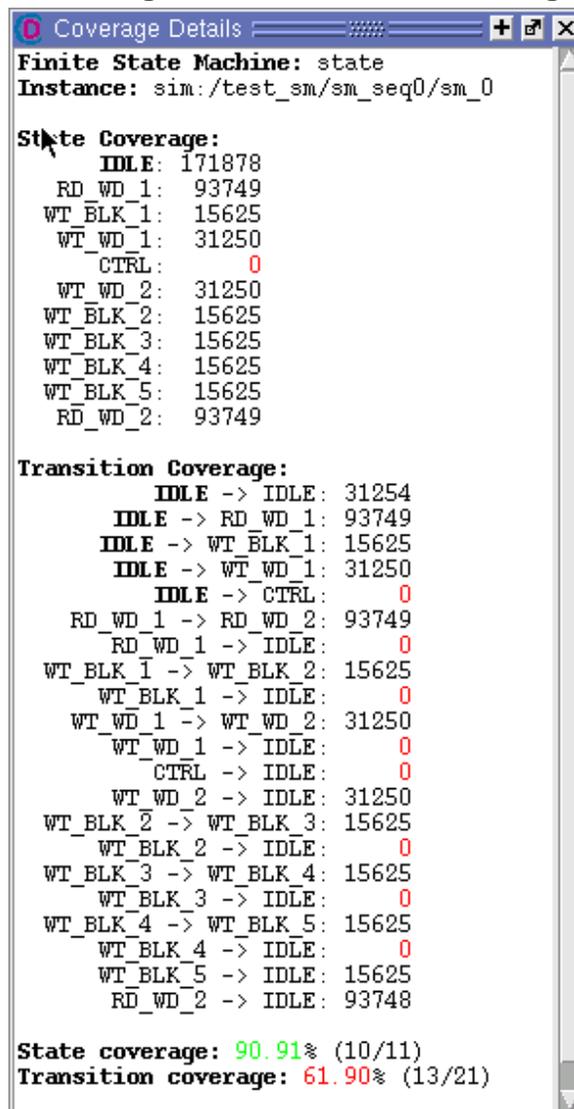
Coverage Details of FSM Coverage

The Coverage Details window displays the following information about [Finite State Machine Coverage](#) metrics:

- Finite State Machine — the name of the finite state machine
- Instance — the dataset name followed by the hierarchical location of the FSM. Only appears when you are analyzing coverage metrics by instance.
- State Coverage — a list of all the states, followed by the number of hits.
- Transition Coverage — a list of all the transitions, followed by the number hits.
- State Coverage — the coverage percentage for the states.
- Transition Coverage — the coverage percentage for the transitions.

FSM details are displayed as shown in [Figure 2-66](#):

Figure 2-66. Coverage Details Window Showing FSM Details



```
Coverage Details
Finite State Machine: state
Instance: sim:/test_sm/sm_seq0/sm_0

State Coverage:
  IDLE: 171878
  RD_WD_1: 93749
  WT_BLK_1: 15625
  WT_WD_1: 31250
  CTRL: 0
  WT_WD_2: 31250
  WT_BLK_2: 15625
  WT_BLK_3: 15625
  WT_BLK_4: 15625
  WT_BLK_5: 15625
  RD_WD_2: 93749

Transition Coverage:
  IDLE -> IDLE: 31254
  IDLE -> RD_WD_1: 93749
  IDLE -> WT_BLK_1: 15625
  IDLE -> WT_WD_1: 31250
  IDLE -> CTRL: 0
  RD_WD_1 -> RD_WD_2: 93749
  RD_WD_1 -> IDLE: 0
  WT_BLK_1 -> WT_BLK_2: 15625
  WT_BLK_1 -> IDLE: 0
  WT_WD_1 -> WT_WD_2: 31250
  WT_WD_1 -> IDLE: 0
  CTRL -> IDLE: 0
  WT_WD_2 -> IDLE: 31250
  WT_BLK_2 -> WT_BLK_3: 15625
  WT_BLK_2 -> IDLE: 0
  WT_BLK_3 -> WT_BLK_4: 15625
  WT_BLK_3 -> IDLE: 0
  WT_BLK_4 -> WT_BLK_5: 15625
  WT_BLK_4 -> IDLE: 0
  WT_BLK_5 -> IDLE: 15625
  RD_WD_2 -> IDLE: 93748

State coverage: 90.91% (10/11)
Transition coverage: 61.90% (13/21)
```

Cover Directives Window

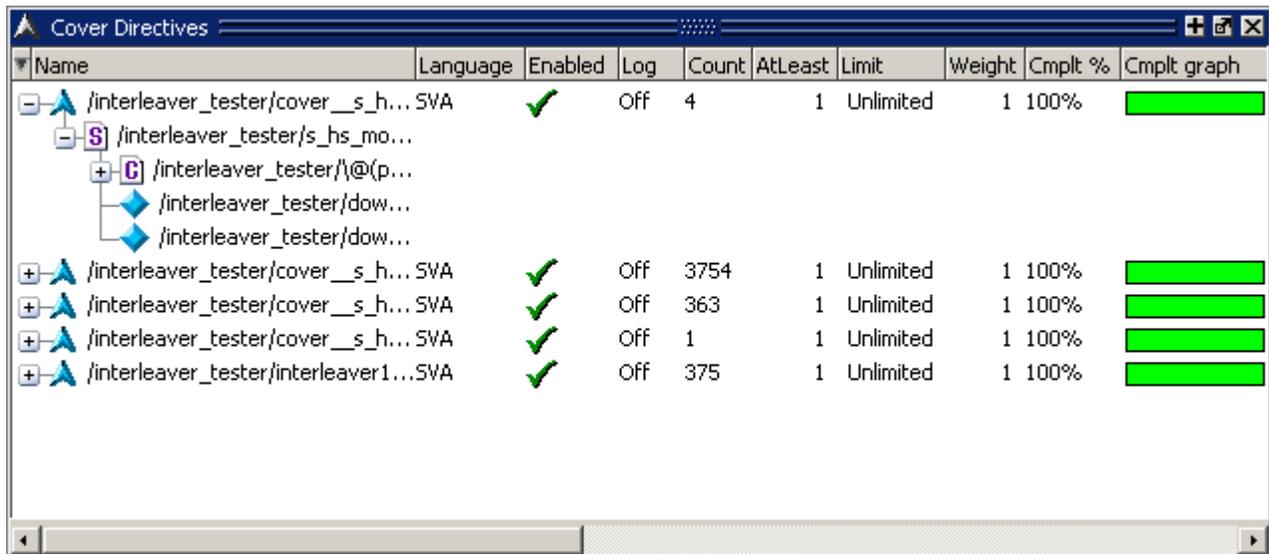
The Cover Directives window displays a list of SystemVerilog and PSL, embedded and external, cover directives that were successfully compiled and simulated during the current simulation.

Accessing

Access the window using either of the following:

- Menu item: **View > Coverage > Cover Directives**
- Command: `view coverdirectives`

Figure 2-67. Cover Directives Window



GUI Elements of the Cover Directives Window

Column Descriptions

Table 2-58. Cover Directives Window Columns

Column	Description
AtLeast	number of times a directive has to fire to be considered 100% covered.
Cmplt %	coverage percentage for a directive. The percentage is the lesser of 100% or Count divided by AtLeast.
Cmplt graph	graphical bar chart of the completion percentage. Directives with 100% coverage are displayed in green.
Count	number of times a directive has "fired" during the current simulation.
Design Unit	design unit to which the directive is bound.
Design Unit Type	HDL type of the design unit. Not displayed by default.
Enabled	displays a green checkmark when a directive is enabled or a red X when a directive is disabled.
Included	indicates whether the directive is included in aggregate statistics and reports.
Language	identifies the HDL used to write the assertion.

Table 2-58. Cover Directives Window Columns (cont.)

Column	Description
Limit	number of times the directive will execute before being disabled by the simulator. Default is Unlimited.
Log	indicates whether data for the directive is currently being added to the functional coverage database.
Memory	tracks the current memory used by the cover directive.
Name	lists directive names and design units. Also, any signals referenced in a directive are included in the hierarchy. Refer to the section “ Using Assert Directive Names ” for more information.
Peak Memory	tracks the peak memory used by the cover directive.
Peak Memory Time	indicates the simulation run time at which the peak memory usage occurred.
Type	shows the cover directive type (Immediate or Concurrent). Not displayed by default.
Weight	shows the weighting factor that has been applied to the directive.

Cover Directives Window Tasks

This section describes tasks for using the Cover Directives window.

Changing the Cover Directives Window Display Options

You can set the window to display cover directives in a **Recursive Mode** or in a **Show All Contexts** mode.

- The **Recursive Mode** displays all cover directives at and below the selected hierarchy instance, the selection being taken from a Structure window. (that is, the **sim** tab). Otherwise only items actually in that particular scope are shown.
- The **Show All Contexts** selection displays all instances in the design. It does not follow the current context selection in a structure pane. The Show All Context display mode implies the recursive display mode as well, so the **Recursive Mode** selection is automatically grayed out.

You can choose between these two display modes by right-clicking in the Cover Directives window and selecting the option from the **Display Options** sub-menu.

Covergroups Window

The Covergroups window displays SystemVerilog covergroups, coverpoints, crosses and bins in the current region (which is selected via the Structure window). Refer to the section “[Viewing Functional Coverage Statistics in the GUI](#)” for more information.

Accessing

Access the window using either of the following:

- Menu item: **View > Coverage > Covergroups**
- Command: view covergroups

Figure 2-68. Covergroups Window

Name	Coverage	Goal	% of Goal	Status	M
[-] /top/dut/fifo					
[+] TYPE ram_cvg	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	1
[-] /top/dut					
[-] TYPE sm_transitions_cvg	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	1
[+] CVP sm_transitions_cvg::int_state	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
[-] TYPE sm_cvg	76.9%	100	76.9%	<div style="width: 76.9%; height: 10px; background-color: red;"></div>	1
[+] CVP sm_cvg::int_state	92.3%	100	92.3%	<div style="width: 92.3%; height: 10px; background-color: green;"></div>	
[-] CVP sm_cvg::in_hs	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
[B] bin valid	17543	500	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
[+] CVP sm_cvg::out_hs	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
[+] CROSS sm_cvg::in_hsXint_state	46.1%	100	46.1%	<div style="width: 46.1%; height: 10px; background-color: red;"></div>	
[+] CROSS sm_cvg::out_hsXint_state	46.1%	100	46.1%	<div style="width: 46.1%; height: 10px; background-color: red;"></div>	
[+] /interleaver_svc_pkg/interleaver_cover/interleaver...					

GUI Elements of the Covergroups Window

Column Descriptions

Table 2-59. Covergroups Window Columns

Column	Description
Auto_bin_max	the maximum number of automatically created bins when no bins are explicitly defined for coverpoints.
Class Type	lists the parameterized class type name of the corresponding classes in the window.
Comment	displays comments that appear with the instance of a covergroup, or a coverpoint or cross of the covergroup instance.
Coverage	weighted average of the coverage of the constituent coverpoints and crosses.
Cover Testname	In post processing mode (Coverage View) it displays the name of the test which covered the bin. During live simulation, the '<current_test>' is displayed.
Cover Time	displays the time when the bin was covered.
Cross_num_print_missing	the number of missing (not covered) cross product bins that must be saved to the coverage database and printed in the coverage report.
Detect_overlap	shows when a warning has been issued for an overlap between the range list (or transition list) of two bins of a coverpoint.
Get_inst_coverage	the value of the get_inst_coverage covergroup option. <ul style="list-style-type: none"> • When true, it enables the tracking of per instance coverage with the get_inst_coverage built-in method. • When false, the value returned by get_inst_coverage shall equal the value returned by get_coverage.
Goal	the desired coverage total as an integer percent.
% of Goal	the percentage of the coverage goal that has been reached.
% over Goal	the percentage over the coverage goal that has been reached
Merge_instances	the value of the merge_instances covergroup type option. <ul style="list-style-type: none"> • When true, cumulative (or type) coverage is computed by merging instances together as the union of coverage of all instances. • When false, type coverage is computed as the weighted average of instances.

Table 2-59. Covergroups Window Columns

Column	Description
Missing Bins	the number of covergroup bins missing coverage
Name	the name of the covergroup and its components.
Peak Transient Memory	the peak transient memory used by the covergroup.
Peak Transient Memory Time	the simulation run time at which the peak transient memory usage occurred.
Persistent Memory	the persistent memory used by the covergroup.
Status	graphical representation of the Coverage column.
Strobe	the value of the strobe coverage group type option. If set to 1, all samples happen at the end of the time slot, like the \$strobe system task.
Transient Memory	the transient memory used by the covergroup.
Weight	the weighting of a covergroup instance for computing the overall instance coverage simulation. For coverpoints or crosses, it shows the weighting of a coverpoint or cross for computing the instance coverage of the enclosing covergroup.
Weighted Missing Bins	the number of weighted bins missing coverage

Covergroups Window Tasks

This section describes tasks for using the Covergroups window.

Changing the Covergroups Window Display Options

Note



Covergroups are created dynamically during simulation. This means they will not display in the GUI until you run the simulation.

You can set the window to display covergroups in a **Recursive Mode** or in a **Show All Contexts** mode.

- The **Recursive Mode** displays all covergroups at and below the selected hierarchy instance, the selection being taken from a Structure window. (that is, the **sim** tab). Otherwise only items actually in that particular scope are shown.
- The **Show All Contexts** selection displays all instances in the design. It does not follow the current context selection in the Structure window. The Show All Context display mode implies the recursive display mode as well, so the **Recursive Mode** selection is automatically grayed out.

You can choose between these two display modes by right-clicking in the Covergroups window and selecting the option from the **Display Options** sub-menu.

Dataflow Window

Use this window to explore the "physical" connectivity of your design. You can also use it to trace events that propagate through the design; and to identify the cause of unexpected outputs.

The Dataflow window displays:

- processes
- signals, nets, and registers
- interconnects

The window has built-in mappings for all Verilog primitive gates (that is, AND, OR, PMOS, NMOS, and so forth.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See [Symbol Mapping](#) for details.

Note



You cannot view SystemC objects in the Dataflow window.

Note



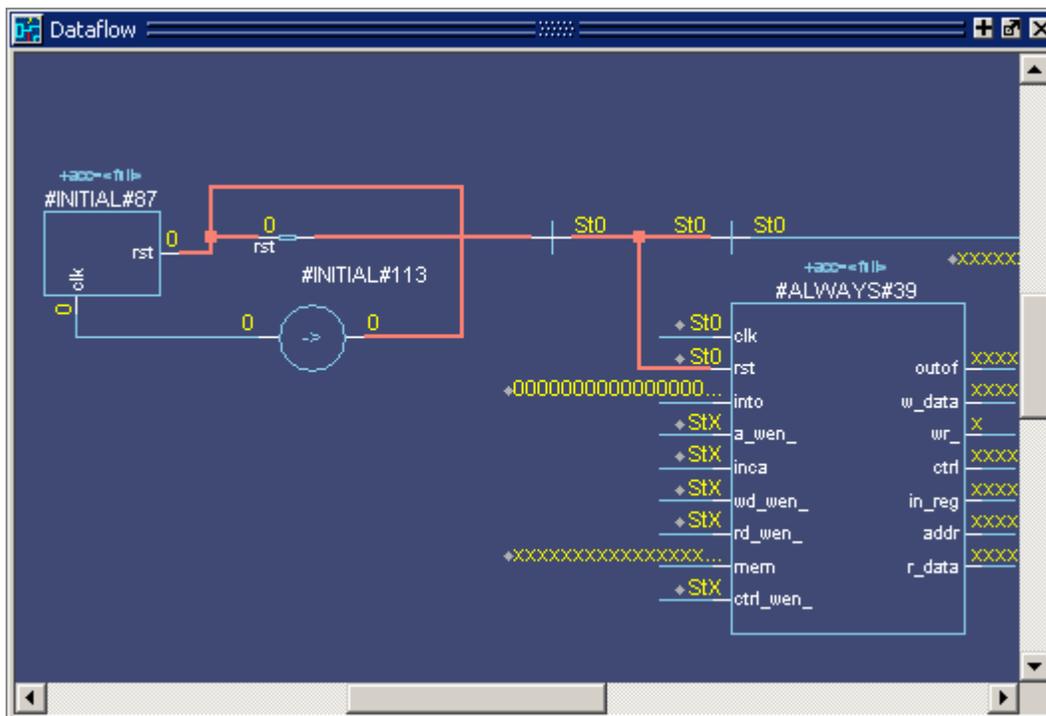
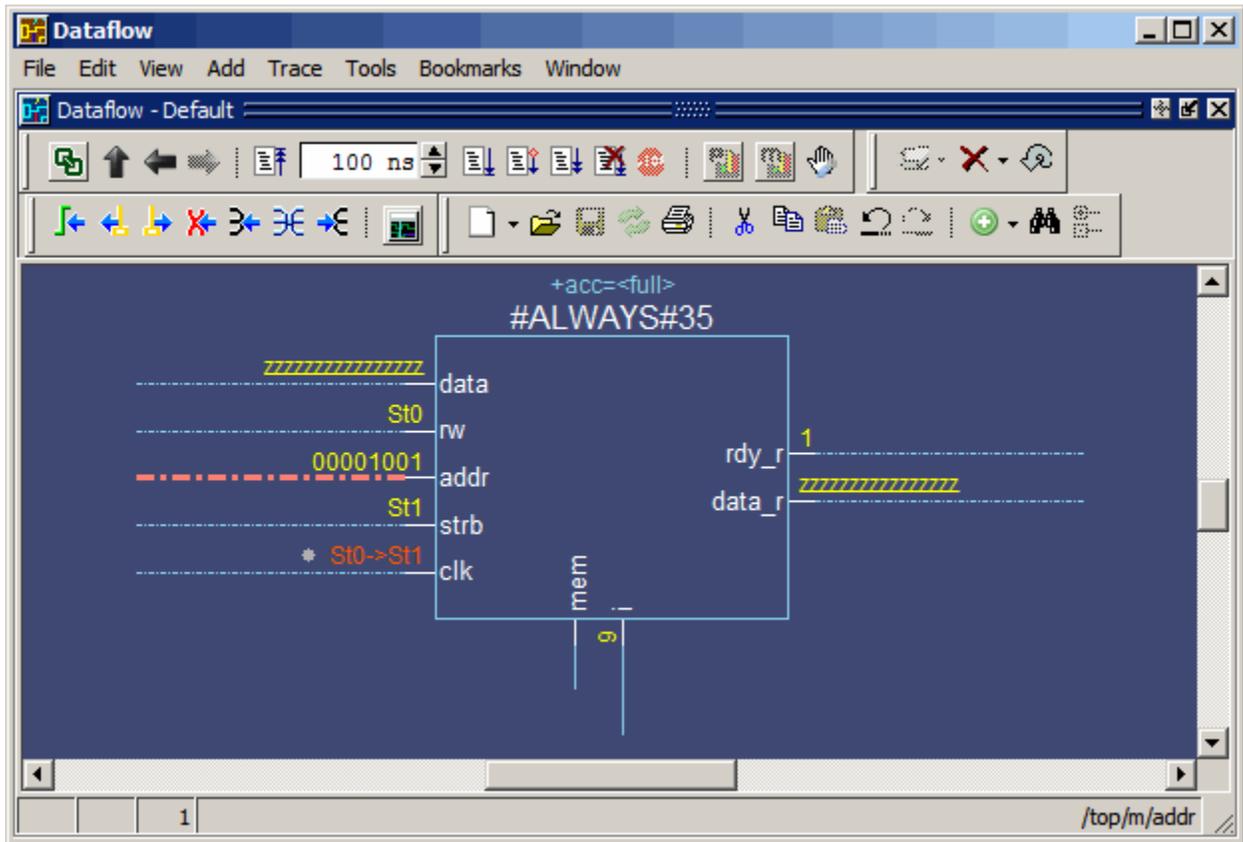
ModelSim versions operating without a dataflow license feature have limited Dataflow functionality. Without the license feature, the window will show only one process and its attached signals or one signal and its attached processes.

Accessing

Access the window using either of the following:

- Menu item: **View > Dataflow**
- Command: view dataflow

Figure 2-69. Dataflow Window



Dataflow Window Tasks

This section describes tasks for using the Dataflow window.

You can interact with the Dataflow in one of three different Mouse modes, which you can change through the DataFlow menu or the [Zoom Toolbar](#):

- **Select Mode** — your left mouse button is used for selecting objects and your middle mouse button is used for zooming the window. This is the default mode.
- **Zoom Mode** — your left mouse button is used for zooming the window and your middle mouse button is used for panning the window.
- **Pan Mode** — your left mouse button is used for panning the window and your middle mouse button is used for zooming the window.

Selecting Objects in the Dataflow Window

When you select an object, or objects, it will be highlighted an orange color.

- **Select a single object** — Single click.
- **Select multiple objects** — Shift-click on all objects you want to select or click and drag around all objects in a defined area. Only available in Select Mode.

Zooming the View of the Dataflow Window

Several zoom controls are available for changing the view of the Dataflow window, including mouse strokes, toolbar icons and a mouse scroll wheel.

- **Zoom Full** — Fills the Dataflow window with all visible data.
 - **Mouse stroke** — Up/Left. Middle mouse button in Select and Pan mode, Left mouse button in Zoom mode.
 - **Menu** — DataFlow > Zoom Full
 - **Zoom Toolbar** — Zoom Full
- **Zoom Out**
 - **Mouse stroke** — Up/Right. Middle mouse button in Select and Pan mode, Left mouse button in Zoom mode.
 - **Menu** — DataFlow > Zoom Out
 - **Zoom Toolbar** — Zoom Out
 - **Mouse Scroll** — Push forward on the scroll wheel.
- **Zoom In**

- Menu — DataFlow > Zoom In
- Zoom Toolbar — Zoom In
- Mouse Scroll — Pull back on the scroll wheel.
- Zoom Area — Fills the Dataflow window with the data within the bounding box.
 - Mouse stroke — Down/Right
- Zoom Selected — Fills the Dataflow window so that all selected objects are visible.
 - Mouse stroke — Down/Left

Panning the View of the Dataflow Window

You can pan the view of the Dataflow window with the mouse or keyboard.

- Pan with the Mouse — In Zoom mode, pan with the middle mouse button. In Pan mode, pan with the left mouse button. In Select mode, pan with the Ctrl key and the middle mouse button.
- Pan with the Keyboard — Use the arrow keys to pan the view. Shift+<arrow key> pans to the far edge of the view. Ctrl+<arrow key> pans by a moderate amount.

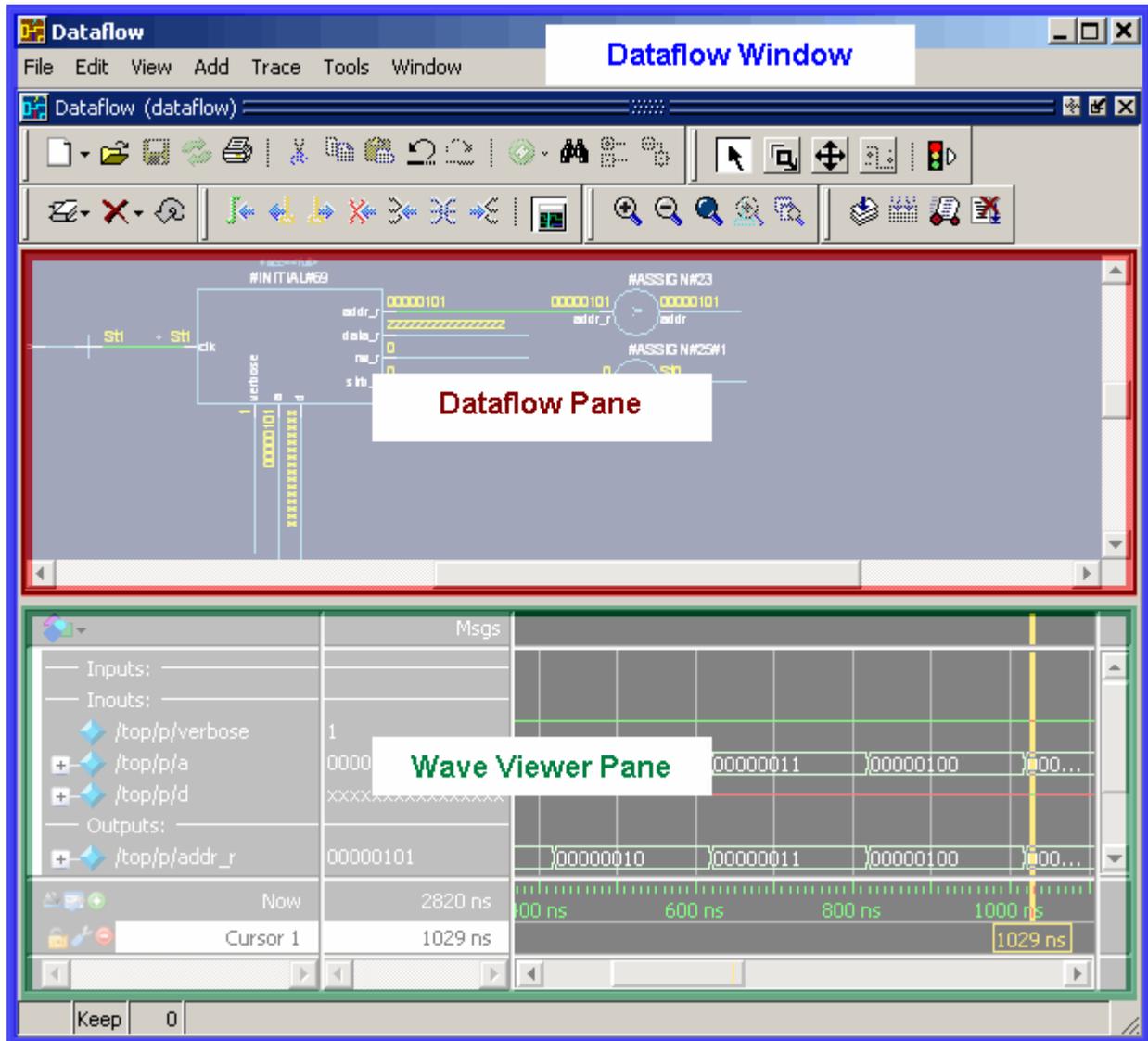
Displaying the Wave Viewer Pane

You can embed a miniature wave viewer in the Dataflow window ([Figure 2-70](#)).

1. Select the **DataFlow > Show Wave** menu item.
2. Select a process in the Dataflow pane to populate the Wave pane with signal information.

Refer to the section “[Exploring Designs with the Embedded Wave Viewer](#)” for more information.

Figure 2-70. Dataflow Window and Panes



Files Window

Use this window to display the source files and their locations for the loaded simulation.

Prerequisites

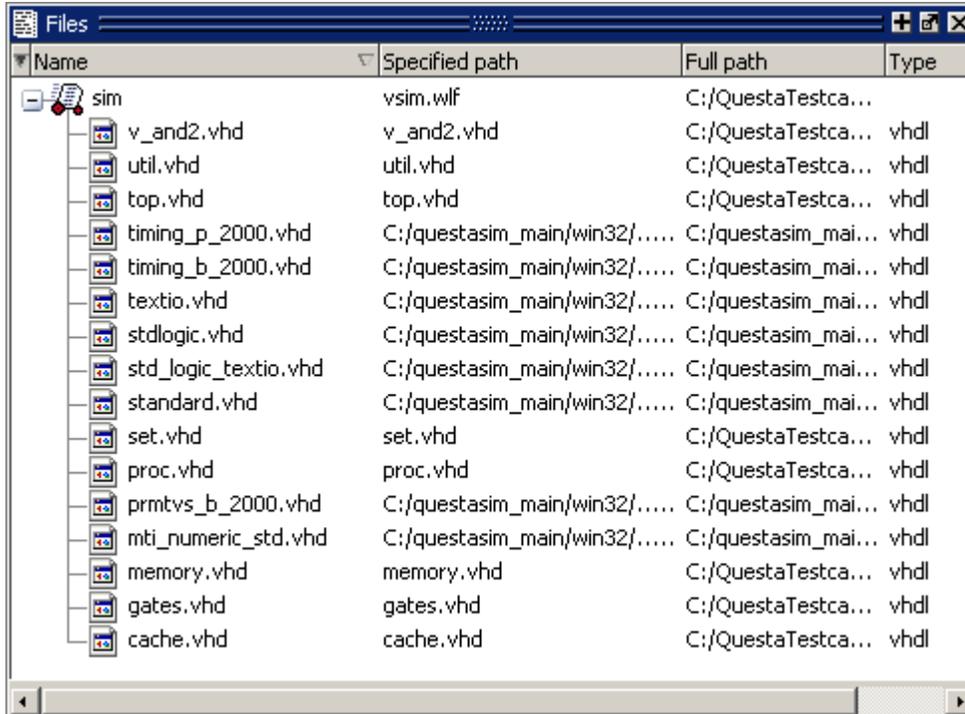
You must have executed the vsim command before this window will contain any information about your simulation environment.

Accessing

Access the window using either of the following:

- Menu item: **View > Files**
- Command: view files

Figure 2-71. Files Window



GUI Elements of the Files Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-60. Files Window Columns

Column Title	Description
Name	The name of the file
Specified Path	The location of the file as specified in the design files.
Full Path	The full-path location of the design files.
Type	The file type.
Branch <i>info</i>	A series of columns reporting branch coverage

Table 2-60. Files Window Columns (cont.)

Column Title	Description
Condition <i>info</i>	A series of columns reporting condition coverage
Expression <i>info</i>	A series of columns reporting expression coverage
FEC condition <i>info</i>	A series of columns reporting condition coverage based on Focused Expression Coverage
FEC expression <i>info</i>	A series of columns reporting expression coverage based on Focused Expression Coverage
States <i>info</i>	A series of columns reporting finite state machine coverage
Statement <i>info</i>	A series of columns reporting statement coverage
Toggles <i>info</i>	A series of columns reporting toggle coverage
Transition <i>info</i>	A series of columns reporting transition coverage

Refer to [Table 2-93 “Columns in the Structure Window”](#) for detailed information about the coverage metric columns.

Popup Menu

Right-click anywhere in the window to display the popup menu and select one of the following options:

Table 2-61. Files Window Popup Menu

Menu Item	Description
View Source	Opens the selected file in a Source window
Open in external editor	Opens the selected file in an external editor. Only available if you have set the Editor preference: <ul style="list-style-type: none"> • set PrefMain(Editor) {<path_to_executable>} • Tools > Edit Preferences; by Name tab, Main group.

Table 2-61. Files Window Popup Menu (cont.)

Menu Item	Description
Code Coverage >	<p>These menu items are only available if you ran the simulation with the -coverage switch.</p> <ul style="list-style-type: none"> • Code Coverage Reports — Opens the Coverage Text Report dialog box, allowing you to create a coverage report for the selected file. • Exclude Selected File — Executes the coverage exclude command for the selected file(s). • Clear Code Coverage Data — Clears all code coverage information collected during simulation
Properties	Displays the File Properties dialog box, containing information about the selected file.

Files Menu

This menu becomes available in the Main menu when the Files window is active.

Table 2-62. Files Menu

Files Menu Item	Description
View Source	Opens the selected file in a Source window
Open in external editor	<p>Opens the selected file in an external editor. Only available if you have set the Editor preference:</p> <ul style="list-style-type: none"> • set PrefMain(Editor) {<path_to_executable>} • Tools > Edit Preferences; by Name tab, Main group.
Save Files	Saves a text file containing a sorted list of unique files, one per line. The default name is <i>summary.txt</i> .

FSM List Window

Use this window to view a list of finite state machines in your design.

Prerequisites

This window is populated when you specify any of the following switches during compilation (vcom/vlog) or optimization (vopt).

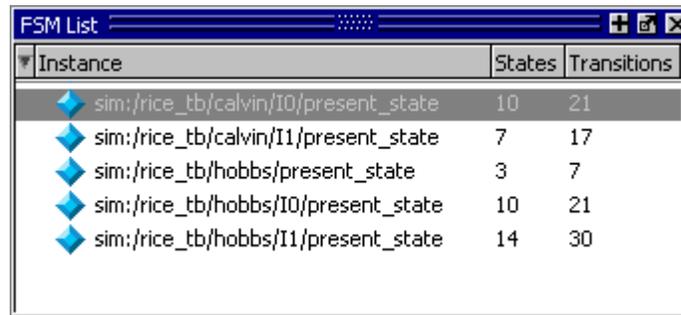
- +cover or +cover=f
- +acc or +acc=f

Accessing

Access the window using either of the following:

- Menu item: **View > FSM List**
- Command: view fsmlist

Figure 2-72. FSM List Window



The screenshot shows a window titled "FSM List" with a table containing five rows of data. Each row starts with a blue diamond icon followed by a path name, then the number of states, and finally the number of transitions.

Instance	States	Transitions
sim:/rice_tb/calvin/I0/present_state	10	21
sim:/rice_tb/calvin/I1/present_state	7	17
sim:/rice_tb/hobbs/present_state	3	7
sim:/rice_tb/hobbs/I0/present_state	10	21
sim:/rice_tb/hobbs/I1/present_state	14	30

GUI Elements of the FSM List Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-63. FSM List Window Columns

Column Title	Description
Instance	Lists the FSM instances. You can reduce the number of path elements in this column by selecting the FSM List > Options menu item and altering the Number of Path Elements selection box.
States	The number of states in the FSM.
Transitions	The number of transitions in the FSM.

Popup Menu

Right-click on one of the FSMs in the window to display the popup menu and select one of the following options:

Table 2-64. FSM List Window Popup Menu

Popup Menu Item	Description
View FSM	Opens the FSM in the FSM Viewer window.
View Declaration	Opens the source file for the FSM instance.
Set Context	Changes the context to the FSM instance.
Add to <window>	Adds FSM information to the specified window.
Properties	Displays the FSM Properties dialog box containing detailed information about the FSM.

FSM List Menu

This menu becomes available in the Main menu when the FSM List window is active.

Table 2-65. FSM List Menu

Popup Menu Item	Description
View FSM	Opens the FSM in the FSM Viewer window.
View Declaration	Opens the source file for the FSM instance.
Add to <window>	Adds FSM information to the specified window.
Options	Displays the FSM Display Options dialog box, which allows you to control: <ul style="list-style-type: none"> • how FSM information is added to the Wave Window. • how much information is shown in the Instance Column

FSM Viewer Window

Use this window to graphically analyze finite state machines in your design.

Prerequisites

- Analyze FSMs and their coverage data — you must specify +cover, or explicitly +cover=f, during compilation and -coverage on the vsim command line to fully analyze FSMs with coverage data.

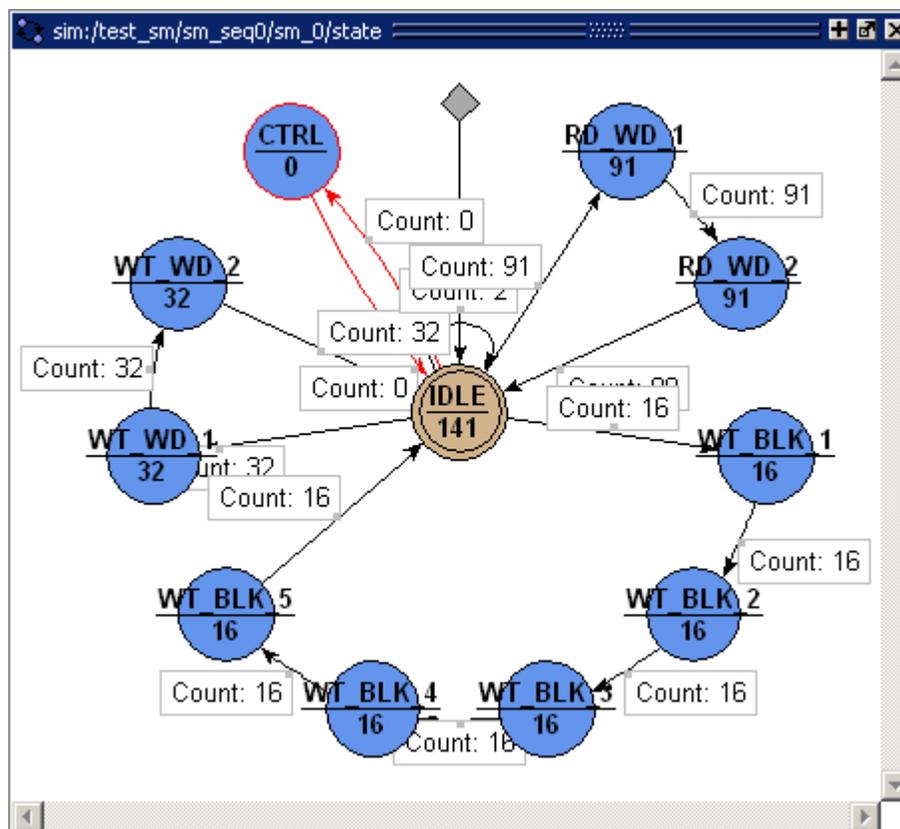
- Analyze FSMs without coverage data — you must specify +acc, or explicitly +acc=f, during compilation (vcom/vlog) or optimization (vopt) to analyze FSMs with the FSM Viewer window.

Accessing

Access the window:

- From the FSM List window, double-click on the FSM you want to analyze.
- From the Objects, Locals, Wave, or Code Coverage Analyze's FSM Analysis windows, click on the FSM button  for the FSM you want to analyze.

Figure 2-73. FSM Viewer Window



FSM Viewer Window Tasks

This section describes tasks for using the FSM Viewer window.

Using the Mouse in the FSM Viewer

These mouse operations are defined for the FSM Viewer:

- The mouse wheel performs zoom & center operations on the diagram.
 - Mouse wheel up — zoom out.
 - Mouse wheel down — zoom in.

Whether zooming in or out, the view will re-center towards the mouse location.

- Left mouse button — click and drag to move the view of the FSM.
- Middle mouse button — click and drag to perform the following stroke actions:
 - Up and left — Zoom Full.
 - Up and right — Zoom Out. The amount is determined by the distance dragged.
 - Down and right — Zoom In on the area of the bounding box.

Using the Keyboard in the FSM Viewer

These keyboard operations are defined for the FSM Viewer:

- Arrow Keys — scrolls the window in the specified direction.
 - Unmodified — scrolls by a small amount.
 - Ctrl+<arrow key> — scrolls by a larger amount.
 - Shift+<arrow key> — shifts the view to the edge of the display.

Exporting the FSM Viewer Window as an Image

Save the FSM view as an image for use in other applications.

1. Select the FSM Viewer window.
2. Export to one of the following formats:
 - Postscript — **File > Print Postscript**
 - Bitmap (*.bmp*) — **File > Export > Image**
 - JPEG (*.jpg*)
 - PNG (*.png*)
 - GIF (*.gif*)

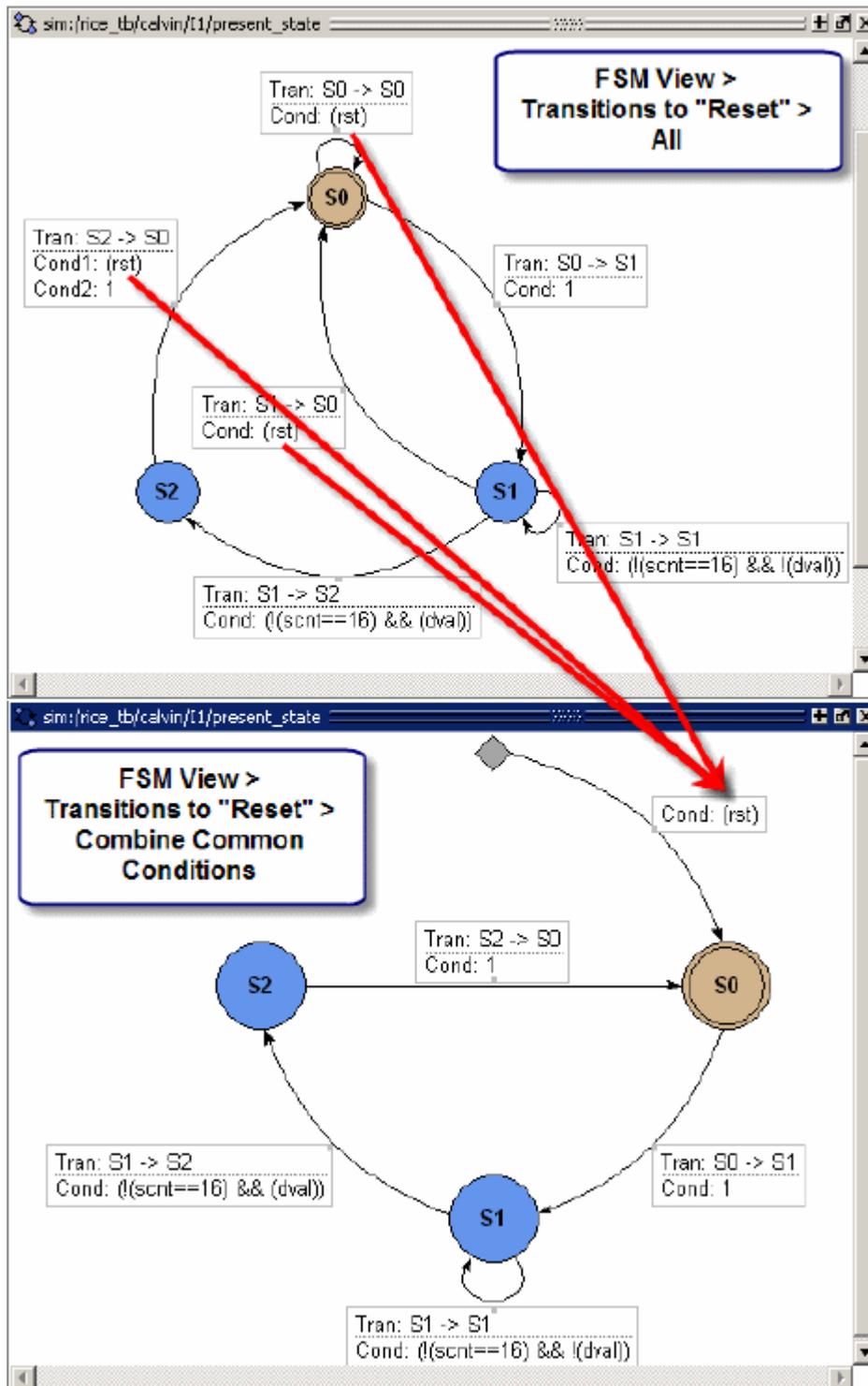
Combining Common Transitions to Reset

By default, the FSM Viewer window combines transitions to reset that are based upon common conditions. This reduces the amount of information drawn in the window and eases your FSM debugging tasks.

Figure 2-74 shows two versions of the same FSM. The top image shows all of the transitions and the bottom image combines the common conditions (rst) into a single transition, as referenced by the gray diamond placeholder.

You control the level of detail for transitions with the **FSM View > Transitions to “reset”** menu items.

Figure 2-74. Combining Common Transition Conditions



GUI Elements of the FSM Viewer Window

This section describes GUI elements specific to this Window.

Table 2-66. FSM Viewer Window — Graphical Elements

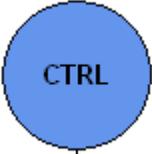
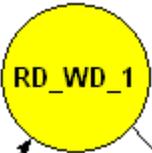
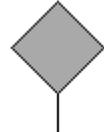
Graphical Element	Description	Definition
	Blue state bubble	Default appearance for non-reset states.
	Green state bubble	Indicates the FSMs current state, or the state of the wave cursor location (when tracking the wave cursor).
	Yellow state bubble	Indicates the FSMs previous state, or the state of the wave cursor location (when tracking the wave cursor).
	Tan state bubble with double outline.	Indicates a reset state.
	Gray diamond	Indicates there are several transitions to reset with the same expression. This is a placeholder to reduce the number of objects drawn in the window. You can view all common expressions by selecting: FSM View > Transitions to “reset” > Show All
	Transition box	Contains information about the transition, <ul style="list-style-type: none"> • Cond: specifies the transition condition¹ • Count: specifies the coverage count
	Black transition line.	Indicates a transition.

Table 2-66. FSM Viewer Window — Graphical Elements

Graphical Element	Description	Definition
	Red transition line.	Indicates a transition that has zero (0) coverage.

1. The condition format is based on the GUI_expression_format [Operators](#).

Popup Menu

Right-click in the window to display the popup menu and select one of the following options:

Table 2-67. FSM View Window Popup Menu

Popup Menu Item	Description
Transition	Only available when right-clicking on a transition. <ul style="list-style-type: none"> • Goto Source — Opens the source file containing the state machine and highlights the transition code. • View Full Text — Opens the View Transition dialog box, which contains the full text of the condition.
View Declaration	Opens the source file and bookmarks the file line containing the declaration of the state machine
Zoom Full	Displays the FSM completely within the window.
Set Context	Executes the env command to change the context to that of the state machine.
Add to ...	Adds information about the state machine to the specific window.
Properties	Displays the FSM Properties dialog box containing detailed information about the FSM.

FSM View Menu

This menu becomes available in the Main menu when the FSM View window is active.

Table 2-68. FSM View Menu

FSM View Menu Item	Description
Show State Counts	Displays the coverage counts for each state in the state bubble.
Show Transition Counts	Displays the coverage counts for each transition.

Table 2-68. FSM View Menu

FSM View Menu Item	Description
Show Transition Conditions	Displays the condition for each transition. The condition format is based on the GUI_expression_format Operators .
Enable Info Mode Popups	Displays popup information when you hover over a state or transition.
Track Wave Cursor	Displays current and previous state information based on the cursor location in the Wave window.
Transitions to “Reset”	Controls the display of transitions to a reset state: <ul style="list-style-type: none">• Show All• Show None — will also add a “hide all” note to the lower-right hand corner.• Hide Asynchronous Only• Combine Common Transitions — (default) creates a single transition for any transitions to reset that use the same condition. The transition is shown from a gray diamond that acts as a placeholder.
Options	Displays the FSM Display Options dialog box, which allows you to control: <ul style="list-style-type: none">• how FSM information is added to the Wave Window.• how much information is shown in the Instance Column

Instance Coverage Window

Use this window to analyze coverage statistics for each instance in a flat, non-hierarchical view. You can sort data columns to be more meaningful, and not be confused by hierarchy. This window contains the same code coverage statistics columns as in the Files and Structure windows.

Prerequisites

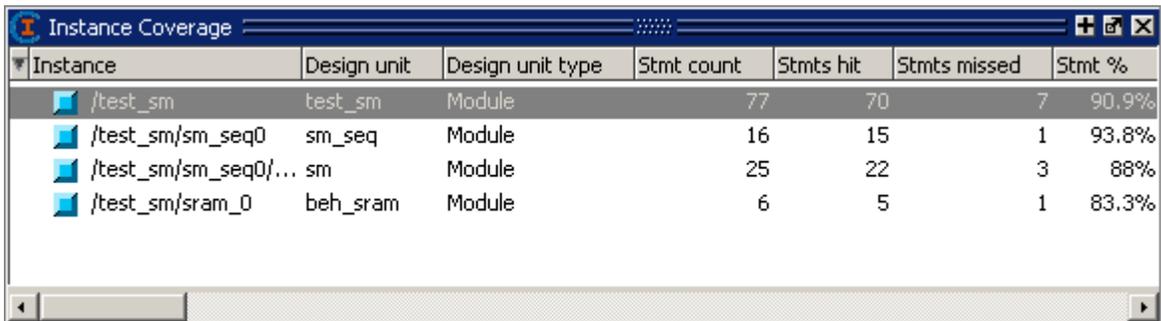
This window is specific to the collection of coverage metrics, therefore you must have run your simulation with coverage collection enabled. Refer to the chapter “[Code Coverage](#)” for more information.

Accessing

Access the window using either of the following:

- Menu item: **View > Coverage > Instance Coverage**

- Command: view instance

Figure 2-75. Instance Coverage Window

Instance	Design unit	Design unit type	Stmt count	Stmts hit	Stmts missed	Stmt %
/test_sm	test_sm	Module	77	70	7	90.9%
/test_sm/sm_seq0	sm_seq	Module	16	15	1	93.8%
/test_sm/sm_seq0/... sm	sm	Module	25	22	3	88%
/test_sm/sram_0	beh_sram	Module	6	5	1	83.3%

Instance Coverage Window Tasks

This section describes tasks for using the Instance Coverage window.

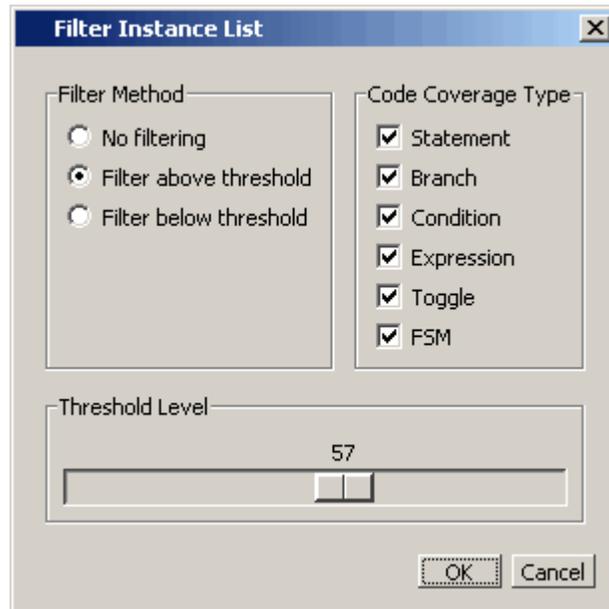
Setting a Coverage Threshold

You can specify a percentage above or below which you don't want to see coverage statistics. For example, you might set a threshold of 85% such that only objects with coverage below that percentage are displayed. Anything above that percentage is filtered.

Procedure

1. Right-click any object in the Instance Coverage window.
2. Select **Set filter**. The "Filter instance list" dialog appears as in [Figure 2-76](#).

Figure 2-76. Filter Instance List Dialog Box



GUI Elements of the Instance Coverage Window

This section describes GUI elements specific to this Window.

Column Descriptions

The table below lists columns in the Instance Coverage window with a description of their contents (Table 2-69).

Table 2-69. Columns in the Instance Coverage Window

Column name	Description
Assertion %	the number of hits from the total number of assertions, as a percentage
Assertion graph	a bar chart displaying the Assertion %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Assertion hits	Assertion hits shows different counts based on whether the -assertdebug is used: <ul style="list-style-type: none"> • with -assertdebug argument to vsim command: number of assertions whose pass count is greater than 0, and fail count is equal to 0. • without -assertdebug: number of assertions whose fail count is equal to 0.
Assertion misses	the number of assertions whose fail counts are greater than 0
Branch %	the current ratio of Branch hits to Branch count

Table 2-69. Columns in the Instance Coverage Window

Column name	Description
Branch count	Files window — the number of executable branches in each file Structure window — the number of executable branches in each level and all levels under that level
Branch graph	a bar chart displaying the Branch %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Branches hit	the number of executable branches that have been executed in the current simulation
Branches missed	the number of executable branches that were not executed in the current simulation
Cover %	the number of hits from the total number of cover directives, as a percentage
Cover graph	a bar chart displaying the Cover directive %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Cover hits	the number of cover directives whose count values are greater than or equal to the at_least value.
Cover misses	the number of cover directives whose count values are less than the at_least value
Covergroup %	the number of hits from the total number of covergroups, as a percentage
Covergroup bins	the number of covergroup bins
Covergroup graph	a bar chart displaying the Covergroup %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green
Covergroup hit bins	the number of hit covergroup bins
Covergroup missed bins	the number of missed covergroup bins
Covergroup weighted missed bins	the number of weighted covergroup bins that were missed
Design Unit	the name of the design unit
Design Unit Type	the type of design unit
FEC Condition %	the current ratio of FEC Condition hits to FEC Condition rows

Table 2-69. Columns in the Instance Coverage Window

Column name	Description
FEC Condition graph	a bar chart displaying the FEC Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
FEC Condition rows	the number of FEC conditions in each instance
FEC Conditions hit	the number of times the FEC conditions in an instance that have been executed
FEC Conditions missed	the number of FEC conditions in an instance that were not executed
FEC Expression %	the current ratio of FEC Expression hits to FEC Expression rows
FEC Expression graph	a bar chart displaying the FEC Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
FEC Expression rows	the number of executable expressions in each instance
FEC Expressions hit	the number of times expressions in an instance have been executed
FEC Expressions missed	the number of executable expressions in an instance that were not executed
State %	the current ratio of State hits to State rows
State graph	a bar chart displaying the State %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
States	the number of states encountered in each instance
States hit	the number of times the states were hit
States missed	the number of states in an instance that were not hit
Stmt %	the current ratio of Stmt hits to Stmt count
Stmt graph	a bar chart displaying the Stmt %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Stmt count	the number of executable statements in each level and all levels under that level
Stmts hit	the number of executable statements that were executed in each level and all levels under that level

Table 2-69. Columns in the Instance Coverage Window

Column name	Description
Stmts missed	the number of executable statements that were not executed in each level and all levels under that level
Toggle %	the current ratio of Toggle hits to Toggle nodes
Toggle graph	a bar chart displaying the Toggle %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Toggle nodes	the number of points in each instance where the logic will transition from one state to another
Toggles hit	the number of nodes in each instance that have transitioned at least once
Toggles missed	the number of nodes in each instance that have not transitioned at least once
Total Coverage	The weighted average of all the coverage types (functional coverage and code coverage) is recursive. Deselect Code Coverage > Enable Recursive Coverage Sums to view results for the local instance. See “Calculation of Total Coverage” for coverage statistics details.
Transition %	the current ratio of Transition hits to Transition rows
Transition graph	a bar chart displaying the State %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Transitions	the number of transitions encountered in each instance
Transitions hit	the number of times the transitions were hit
Transitions missed	the number of transitions in an instance that were not hit
UDP Condition %	the current ratio of UDP Condition hits to UDP Condition rows
UDP Condition graph	a bar chart displaying the Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
UDP Condition rows	the number of UDP conditions in each instance
UDP Conditions hit	the number of times the UDP conditions in an instance that have been executed
UDP Conditions missed	the number of conditions in an instance that were not executed
UDP Expression %	the current ratio of UDP Expression hits to UDP Expression rows

Table 2-69. Columns in the Instance Coverage Window

Column name	Description
UDP Expression graph	a bar chart displaying the UDP Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
UDP Expression rows	the number of executable UDP expressions in each level and all levels subsumed under that level
UDP Expressions hit	the number of times UDP expressions in a level, and each level under that level, have been executed
UDP Expressions missed	the number of executable UDP expressions in a level, and all levels under that level, that were not executed

Popup Menu

Right-click anywhere in the window to display the popup menu and select one of the following options:

Table 2-70. Instance Coverage Popup Menu

Popup Menu Item	Description
Code coverage reports	Displays the Coverage Text Report dialog box, which allows you to create reports based on your code coverage metrics.
Set Filter	Displays the Filter Instance List Dialog Box
Clear code coverage data	clears all of the code coverage data from the GUI.
Test Analysis	Executes coverage analyze on the selected instance and prints information to the Test Analysis window. <ul style="list-style-type: none"> • Find Least Coverage • Find Most Coverage • Find Zero Coverage • Find Non Zero Coverage • Summary
XML Import Hint	Displays the XML Import Hint dialog box with information about the data for you to cut and paste.

Library Window

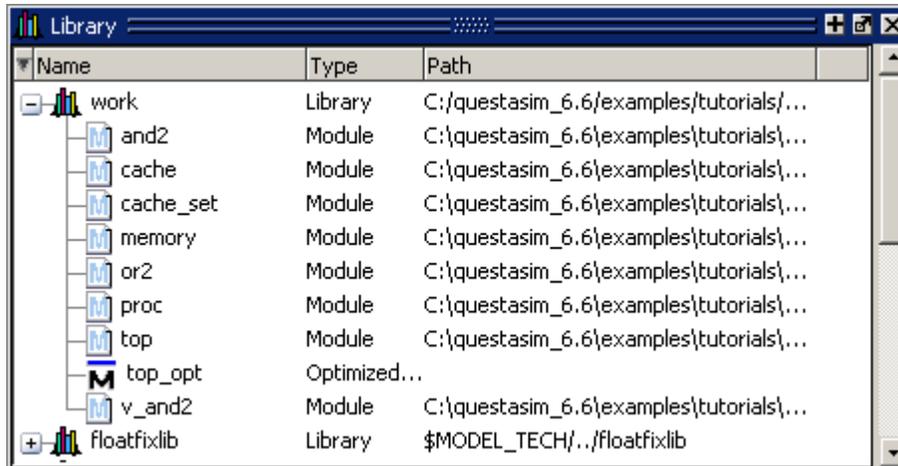
Use this window to view design libraries and compiled design units.

Accessing

Access the window using either of the following:

- Menu item: **View > Library**
- Command: view library

Figure 2-77. Library Window



GUI Elements of the Library Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-71. Library Window Columns

Column Title	Description
Name	Name of the library or design unit
Path	Full pathname to the file
Type	Type of file

Popup Menu

Right-click anywhere in the window to display the popup menu and select one of the following options:

Table 2-72. Library Window Popup Menu

Popup Menu Item	Description
Simulate	Loads a simulation of the selected design unit, implicitly calling vopt with full visibility (-voptargs=+acc)

Table 2-72. Library Window Popup Menu

Popup Menu Item	Description
Simulate without Optimization	Loads a simulation of the selected design unit, where it does not use optimization (-novopt)
Simulate with full Optimization	Loads a simulation of the selected design unit, implicitly calling vopt with no visibility
Simulate with Coverage	Loads a simulation of the selected design unity, enabling coverage (-coverage)
Edit	Opens the selected file in your editor window.
Refresh	Reloads the contents of the window
Recompile	Compiles the selected file.
Optimize	Runs vopt on the selected file.
Update	
Create Wave	Runs the wave create command for any ports in the selected design unit.
Delete	Removes a design unit from the library or runs the vdel command on a selected library.
Copy	Copies the directory location of libraries or the library location of design units within the library.
New	Allows you to create a new library with the Create a New Library dialog box.
Properties	Displays information about the selected library or design unit.

List Window

Use this window to display a textual representation of waveforms, which you can configure to show events and delta events for the signals or objects you have added to the window.

You can view the following object types in the List window:

- VHDL — signals, aliases, process variables, and shared variables
- Verilog — nets, registers, and variables
- SystemC — primitive channels, ports, and transactions
- Comparisons — comparison objects; see [Waveform Compare](#) for more information
- Virtuals — virtual signals and functions
- SystemVerilog and PSL assertions — assert directives

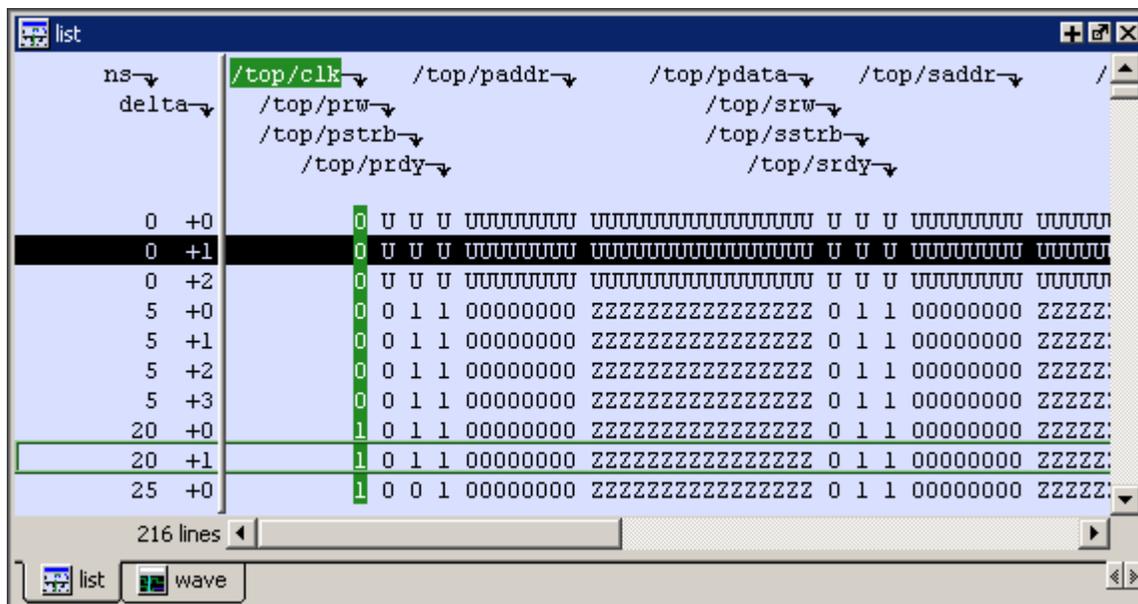
- SystemVerilog and PSL cover directives — cover directives
- Questa Verification IP objects (see [Questa Verification IP Objects in the GUI](#)).
- SystemVerilog — transactions

Accessing

Access the window using either of the following:

- Menu item: **View > List**
- Command: view list

Figure 2-78. List Window



List Window Tasks

This section describes tasks for using the List window.

Adding Data to the List Window

You can add objects to the List window in any of the following ways:

- right-clicking on signals and objects in the Objects window or the Structure window and selecting Add > to List.
- using the [add list](#) command.
- using the “[Add Selected to Window Button](#)“.

Selecting Multiple Signals

To create a larger group of signals and assign a new name to this group, do the following:

1. Select a group of signals
 - Shift-click on signal columns to select a range of signals.
 - Control-click on signal columns to select a group of specific signals.
2. Select **List > Combine Signals**
3. Complete the Combine Selected Signals dialog box
 - Name — Specify the name you want to appear as the name of the new signal.
 - Order of Indexes — Specify the order of the new signal as ascending or descending.
 - Remove selected signals after combining — Specify whether the grouped signals should remain in the List window.

This process creates virtual signals. For more information, refer to the section [Virtual Signals](#).

Other List Window Tasks

- **List > List Preferences** — Allows you to specify the preferences of the List window.
- **File > Export > Tabular List** — Exports the information in the List window to a file in tabular format. Equivalent to the command:

```
write list <filename>
```

- **File > Export > Event List** — Exports the information in the List window to a file in print-on-change format. Equivalent to the command:

```
write list -event <filename>
```

- **File > Export > TSSI List** — Exports the information in the List window to a file in TSSI. Equivalent to the command:

```
write tssi -event <filename>
```

- **Edit > Signal Search** — Allows you to search the List window for activity on the selected signal.

GUI Elements of the List Window

This section describes the GUI elements specific to the List window.

Window Panes

The List 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.

- The left pane shows the time and any deltas that exist for a given time.
- The right pane contains the data for the signals and objects you have added for each time shown in the left pane. The top portion of the window contains the names of the signals. The bottom portion shows the signal values for the related time.

Note

The display of time values in the left column is limited to 10 characters. Any time value of more than 10 characters is replaced with the following:

```
too narrow
```

Markers

The markers in the List window are analogous to cursors in the Wave window. You can add, delete and move markers in the List window similarly to the Wave window. You will notice two different types of markers:

- Active Marker — The most recently selected marker shows as a black highlight.
- Non-active Marker — Any markers you have added that are not active are shown with a green border.

You can manipulate the markers in the following ways:

- Setting a marker — When you click in the right-hand portion of the List window, you will highlight a given time (black horizontal highlight) and a given signal or object (green vertical highlight).
- Moving the active marker — List window markers behave the same as Wave window cursors. There is one active marker which is where you click along with inactive markers generated by the Add Marker command. Markers move based on where you click. The closest marker (either active or inactive) will become the active marker, and the others remain inactive.
- Adding a marker — You can add an additional marker to the List window by right-clicking at a location in the right-hand side and selecting Add Marker.
- Deleting a marker — You can delete a marker by right-clicking in the List window and selecting Delete Marker. The marker closest to where you clicked is the marker that will be deleted.

Popup Menu

Right-click in the right-hand pane to display the popup menu and select one of the following options:

Table 2-73. List Window Popup Menu

Popup Menu Item	Description
Examine	Displays the value of the signal over which you used the right mouse button, at the time selected with the Active Marker
Annotate Diff	Allows you to annotate a waveform comparison difference with additional information. For more information refer to the compare annotate command. Available only during a Waveform Comparison.
Ignore Diff	Flags the waveform compare difference as “ignored”. For more information refer to the compare annotate command. Available only during a Waveform Comparison
Add Marker	Adds a marker at the location of the Active Marker
Delete Marker	Deletes the closest marker to your mouse location

The following menu items are available when the List window is active:

Locals Window

Use this window to display data objects declared in the current, or local, scope of the active process. These data objects are immediately visible from the statement that will be executed next, which is denoted by a blue arrow in a Source window. The contents of the window change from one statement to the next.

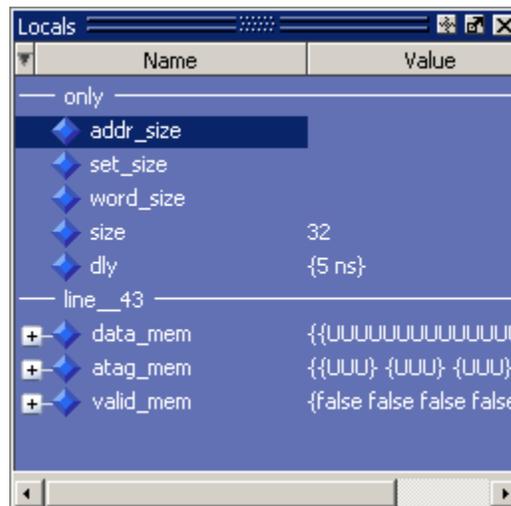
When encountering a C breakpoint, the Locals window displays automatic local variables and their value in current C/C++ function scope.

Accessing

Access the window using either of the following:

- Menu item: **View > locals**
- Command: `view locals`

Figure 2-79. Locals Window



Locals Window Tasks

This section describes tasks for using the Locals window.

Viewing Data in the Locals Window

You cannot actively place information in the Locals window, it is updated as you go through your simulation. However, there are several ways you can trigger the Locals window to be updated.

- Run your simulation while debugging.
- Select a Process from the [Processes Window](#).
- Select a Verilog function or task or VHDL function or procedure from the [Call Stack Window](#).

GUI Elements of the Locals Window

This section describes the GUI elements specific to the Locals Window.

Column Descriptions

Table 2-74. Locals Window Columns

Column	Description
Name	lists the names of the immediately visible data objects. This column also includes design object icons for the objects, refer to the section “ Design Object Icons and Their Meaning ” for more information.
Value	lists the current value(s) associated with each name.
State Count	Not shown by default. This column, State Hits, and State % are all specific to coverage analysis
State Hits	Not shown by default.
State %	Not shown by default.

Popup Menu

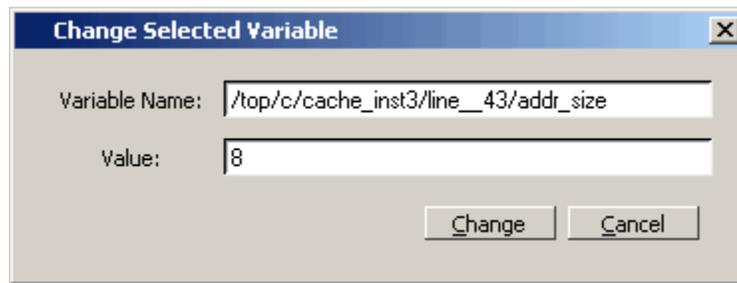
Right-click anywhere in the Locals window to open a popup menu.

Table 2-75. Locals Window Popup Menu

Popup Menu Item	Description
View Declaration	Displays, in the Source window, the declaration of the object.
Add	Adds the selected object(s) to the specified window (Wave, List, Log, Dataflow, Schematic).
Copy	Copies selected item to clipboard
Find	Opens the Find toolbar at the bottom of the window
Expand/Collapse	Expands or collapses data in the window
Global Signal Radix	Sets radix for selected signal(s) in all windows
Change	Displays the Change Selected Variable Dialog Box , which allows you to alter the value of the object.

Change Selected Variable Dialog Box

This dialog box allows you to change the value of the object you selected. When you click Change, the tool executes the [change](#) command on the object.

Figure 2-80. Change Selected Variable Dialog Box

The Change Selected Variable dialog is prepopulated with the following information about the object you had selected in the Locals window:

- Variable Name — contains the complete name of the object.
- Value — contains the current value of the object.

When you change the value of the object, 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.

Memory List Window

Use this window to view a list of all memories in your design.

Single dimensional arrays of integers are interpreted as 2D memory arrays. In these cases, the word width listed in the Memory window is equal to the integer size, and the depth is the size of the array itself.

Memories with three or more dimensions display with a plus sign '+' next to their names in the Memory window. Click the '+' to show the array indices under that level. When you finally expand down to the 2D level, you can double-click on the index, and the data for the selected 2D slice of the memory will appear in a memory contents window.

Prerequisites

The simulator identifies certain kinds of arrays in various scopes as memories. Memory identification depends on the array element kind as well as the overall array kind (that is, associative array, unpacked array, and so forth.).

Table 2-76. Memory Identification

	VHDL	Verilog/SystemVerilog	SystemC
Element Kind¹	<ul style="list-style-type: none"> • enum² • bit_vector • floating point type • std_logic_vector • std_ulogic_vector • integer type 	any integral type (that is, integer_type): <ul style="list-style-type: none"> • shortint • int • longint • byte • bit (2 state) • logic • reg • integer • time (4 state) • packed_struct/ packed_union (2 state) • packed_struct/ packed_union (4 state) • packed_array (single-Dim, multi-D, 2 state and 4 state) • enum • string 	<ul style="list-style-type: none"> • unsigned char • unsigned short • unsigned int • unsigned long • unsigned long long • char • short • int • float • double • enum • sc_bigint • sc_biguint • sc_int • sc_uint • sc_signed • sc_unsigned
Scope: Recognizable in	<ul style="list-style-type: none"> • architecture • process • record 	<ul style="list-style-type: none"> • module • interface • package • compilation unit • struct • static variables within a <ul style="list-style-type: none"> • task • function • named block • class 	<ul style="list-style-type: none"> • sc_module
Array Kind	<ul style="list-style-type: none"> • single-dimensional • multi-dimensional 	<ul style="list-style-type: none"> • any combination of unpacked, dynamic and associative arrays³ • real/shortreal • float 	<ul style="list-style-type: none"> • single-dimensional • multi-dimensional

1. The element can be "bit" or "std_ulogic" if the array has dimensionality ≥ 2 .

2. These enumerated types must have at least one enumeration literal that is not a character literal. The listed width is the number of entries in the enumerated type definition and the depth is the size of the array itself.

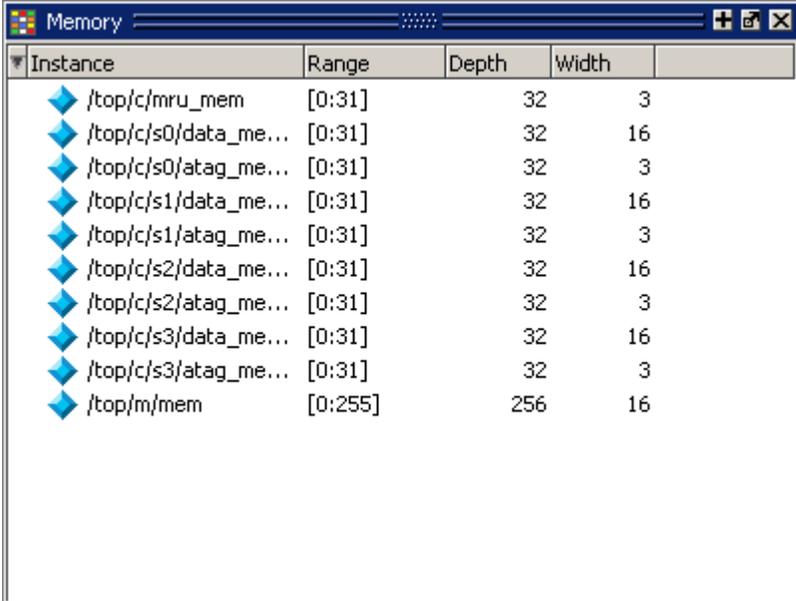
3. Any combination of unpacked, dynamic, and associative arrays is considered a memory, provided the leaf level of the data structure is a string or an integral type.

Accessing

Access the window using either of the following:

- Menu item: **View > Memory List**
- Command: view memory list

Figure 2-81. Memory List Window



Instance	Range	Depth	Width
◆ /top/c/mru_mem	[0:31]	32	3
◆ /top/c/s0/data_me...	[0:31]	32	16
◆ /top/c/s0/atag_me...	[0:31]	32	3
◆ /top/c/s1/data_me...	[0:31]	32	16
◆ /top/c/s1/atag_me...	[0:31]	32	3
◆ /top/c/s2/data_me...	[0:31]	32	16
◆ /top/c/s2/atag_me...	[0:31]	32	3
◆ /top/c/s3/data_me...	[0:31]	32	16
◆ /top/c/s3/atag_me...	[0:31]	32	3
◆ /top/m/mem	[0:255]	256	16

Memory List Window Tasks

This section describes tasks for using the Memory List window.

Viewing Packed Arrays

By default, packed dimensions are treated as single vectors in the Memory List window. To expand packed dimensions of packed arrays, select **Memories > Expand Packed Memories**.

To change the permanent default, edit the PrefMemory(ExpandPackedMem) variable. This variable affects only packed arrays. If the variable is set to 1, the packed arrays are treated as unpacked arrays and are expanded along the packed dimensions such that they appear as a linearized bit vector. Refer to the section “[Simulator GUI Preferences](#)” for details on setting preference variables.

Viewing Memory Contents

When you double-click an instance on the Memory List window, ModelSim automatically displays a Memory Data window, where the name used on the tab is taken from the name of the instance, as seen in the Memory window. You can also enter the command **add mem** <instance> at the **vsim** command prompt.

Viewing Multiple Memory Instances

You can view multiple memory instances simultaneously. A Memory Data window appears for each instance you double-click in the Memory List window. When you open more than one window for the same memory, the name of the tab receives an numerical identifier after the name, such as “(2)”.

Saving Memory Formats in a DO File

You can save all open memory instances and their formats (for example, address radix, data radix, and so forth) by creating a DO file.

1. Select the Memory List window
2. Select **File > Save Format**
displays the Save Memory Format dialog box
3. Enter the file name in the “Save memory format” dialog box

By default it is named *mem.do*. The file will contain all open memory instances and their formats.

To load it at a later time, select **File > Load**.

GUI Elements of the Memory List Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-77. Memory List Window Columns

Column Title	Description
Instance	Hierarchical name of the memory
Range	Memory range
Depth	Memory depth
Width	Word width

Popup Menu

Right-click anywhere in the window to display the popup menu and select one of the following options:

Table 2-78. Memory List Popup Menu

Popup Menu Item	Description
View Contents	Opens a Memory Data window for the selected memory.
Memory Declaration	Opens a Source window to the file and line number where the memory is declared.
Compare Contents	Allows you to compare the selected memory against another memory in the design or an external file.
Import Data Patterns	Allows you to import data patterns into the selected memory through the Import Memory dialog box.
Export Data Patterns	Allows you to export data patterns from the selected memory through the Export Memory dialog box.

Memory List Menu

This menu becomes available in the Main menu when the Memory List window is active.

Table 2-79. Memories Menu

Popup Menu Item	Description
View Contents	Refer to items in the Memory List Popup Menu
Memory Declaration	
Compare Contents	
Import Data Patterns	
Export Data Patterns	
Expand Packed Memories	Toggle the expansion of packed memories.
Identify Memories Within Cells	Toggle the identification of memories within Verilog cells.
Show VHDL String as Memory	Toggle the identification of VHDL strings as memories.

Memory Data Window

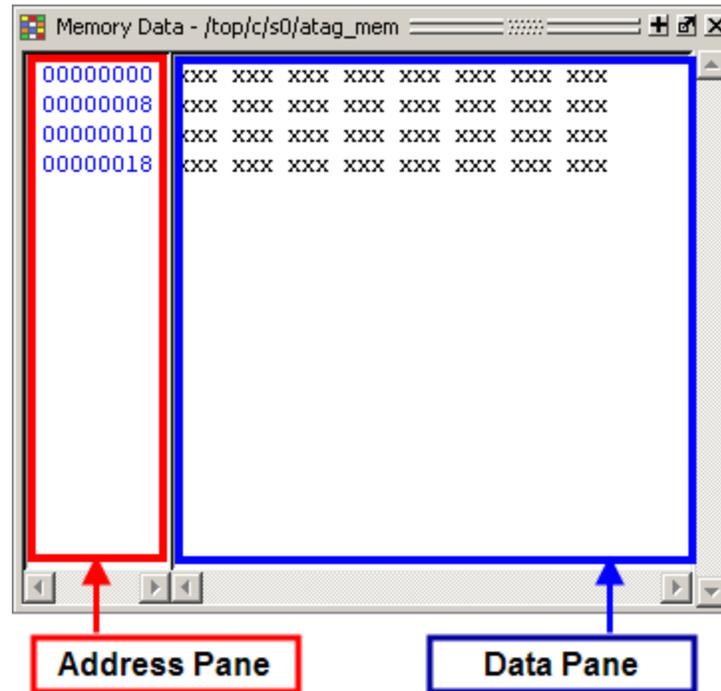
Use this window to view the contents of a memory.

Accessing

Access the window by:

- Double-clicking on a memory in the Memory List window.

Figure 2-82. Memory Data Window



Memory Data Window Tasks

This section describes tasks for using the Memory Data window.

Direct Address Navigation

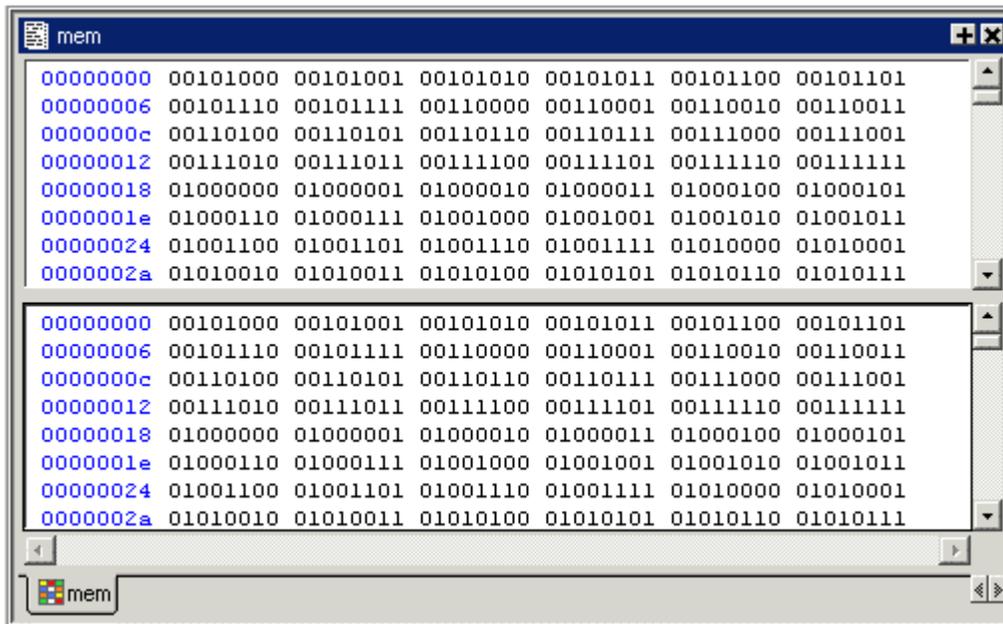
You can navigate to any address location directly by editing the address in the address column. Double-click on any address, type in the desired address, and hit **Enter**. The address display scrolls to the specified location.

Splitting the Memory Contents Window

To split a memory contents window into two screens displaying the contents of a single memory instance select **Memory Data > Split Screen**.

This allows you to view different address locations within the same memory instance simultaneously.

Figure 2-83. Split Screen View of Memory Contents



GUI Elements of the Memory Data Window

This section describes GUI elements specific to this Window.

Popup Menu

Right-click in the window to display the popup menu and select one of the following options:

Table 2-80. Memory Data Popup Menu — Address Pane

Popup Menu Item	Description
Goto	Allows you to go to a specific address
Split Screen	Splits the Memory Data window to allow you to view different parts of the memory simultaneously.
Properties	Allows you to set various properties for the Memory Data window.
Close Instance	Closes the active Memory Data window.
Close All	Closes all Memory Data windows.

Table 2-81. Memory Data Popup Menu — Data Pane

Popup Menu Item	Description
Edit	Allows you to edit the value of the selected data.
Change	Allows you to change data within the memory through the use of the Change Memory dialog box.

Table 2-81. Memory Data Popup Menu — Data Pane

Popup Menu Item	Description
Import Data Patterns	Allows you to import data patterns into the selected memory through the Import Memory dialog box.
Export Data Patterns	Allows you to export data patterns from the selected memory through the Export Memory dialog box.
Split Screen	Refer to items in the Memory Data Popup Menu — Address Pane
Properties	
Close Instance	
Close All	

Memory Data Menu

This menu becomes available in the Main menu when the Memory Data window is active.

Table 2-82. Memory Data Menu

Popup Menu Item	Description
Memory Declaration	Opens a Source window to the file and line number where the memory is declared.
Compare Contents	Allows you to compare the selected memory against another memory in the design or an external file.
Import Data Patterns	Refer to items in the Memory Data Popup Menu — Data Pane
Export Data Patterns	
Expand Packed Memories	Toggle the expansion of packed memories.
Identify Memories Within Cells	Toggle the identification of memories within Verilog cells.
Show VHDL String as Memory	Toggle the identification of VHDL strings as memories.
Split Screen	Refer to items in the Memory Data Popup Menu — Address Pane

Message Viewer Window

Use this window to easily access, organize, and analyze any Note, Warning, Error or other elaboration and runtime messages written to the transcript during the simulation run.

Prerequisites

By default, the tool writes transcribed messages during elaboration and runtime only to the transcript. To write messages to the WLF file (thus the Message Viewer window), use the `-displaymsgmode` and `-msgmode` options to change the default behavior. By writing messages to the WLF file, the Message Viewer window is able to organize the messages for your analysis during the current simulation as well as during post simulation.

You can control what messages are available in the transcript, WLF file, or both with the following switches:

- `displaymsgmode` messages — User generated messages resulting from calls to Verilog Display System Tasks and PLI/FLI print function calls. By default, these messages are written only to the transcript, which means you cannot access them through the Message Viewer window. In many cases, these user generated messages are intended to be output as a group of transcribed messages, thus the default of transcript only. The Message Viewer treats each message individually, therefore you could lose the context of these grouped messages by modifying the view or sort order of the Message Viewer.

To change this default behavior you can use the `-displaymsgmode` argument to `vsim`. The syntax is:

```
vsim -displaymsgmode {both | tran | wlf}
```

You can also use the `displaymsgmode` variable in the `modelsim.ini` file.

The message transcribing methods that are controlled by `-displaymsgmode` include:

- Verilog Display System Tasks — `$write`, `$display`, `$monitor`, and `$strobe`. The following also apply if they are sent to `STDOUT`: `$fwrite`, `$fdisplay`, `$fmonitor`, and `$fstrobe`.
- FLI Print Function Calls — `mti_PrintFormatted` and `mti_PrintMessage`.
- PLI Print Function Calls — `io_printf` and `vpi_printf`.
- `msgmode` messages — All elaboration and runtime messages not part of the `displaymsgmode` messages. By default, these messages are written only to the transcript. To change this default behavior you can use the `-msgmode` argument to `vsim`. The syntax is:

```
vsim -msgmode {both | tran | wlf}
```

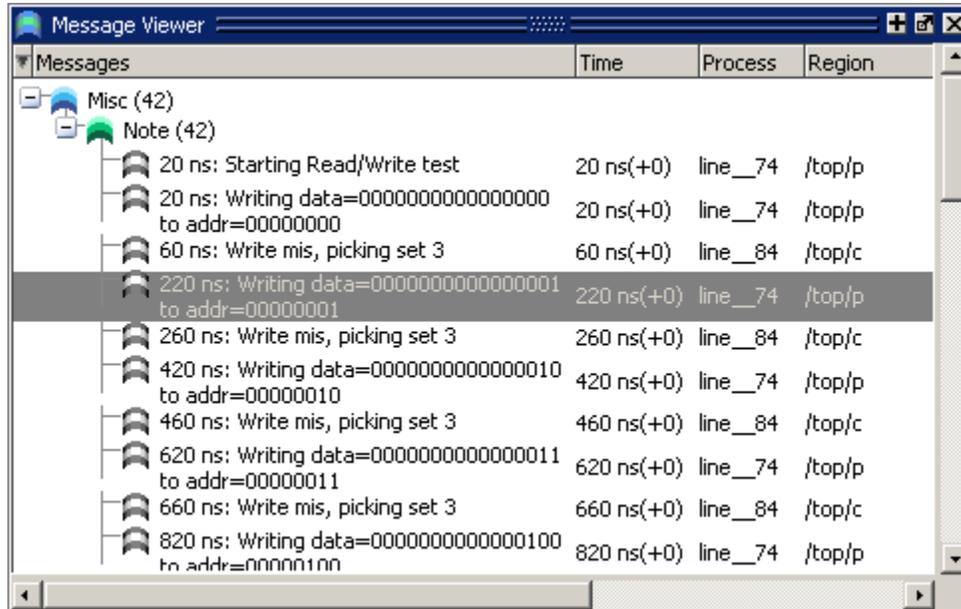
To write messages to the WLF file and transcript, which provides access to the messages through the Message Viewer window. You can also use the `msgmode` variable in the `modelsim.ini` file.

Accessing

Access the window using either of the following:

- Menu item: **View > Message Viewer**
- Command: view msgviewer

Figure 2-84. Message Viewer Window



Message Viewer Window Tasks

Figure 2-85 and Table 2-83 provide an overview of the Message Viewer and several tasks you can perform.

Figure 2-85. Message Viewer Window — Tasks

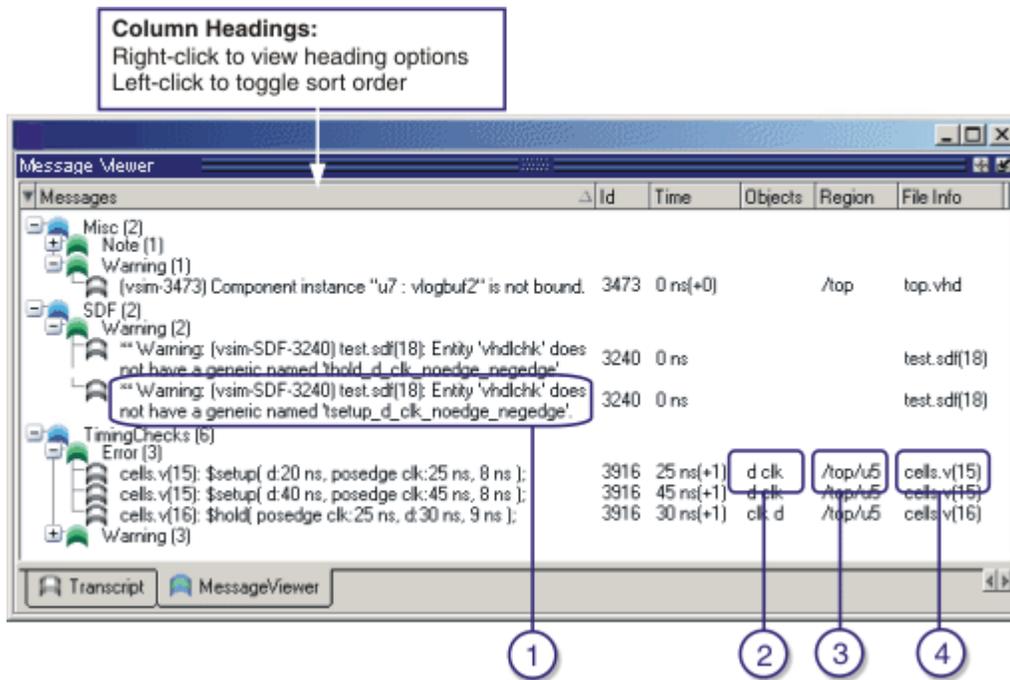


Table 2-83. Message Viewer Tasks

Icon	Task	Action
1	Display a detailed description of the message.	right click the message text then select View Verbose Message .
2	Open the source file and add a bookmark to the location of the object(s).	double click the object name(s).
3	Change the focus of the Structure and Objects windows.	double click the hierarchical reference.
4	Open the source file and set a marker at the line number.	double click the file name.

GUI Elements of the Message Viewer Window

This section describes the GUI elements specific to this window.

Column Descriptions

Table 2-84. Message Viewer Window Columns

Column	Description
Assertion Expression	
Assertion Name	
Assertion Start Time	
Category	Keyword for the various categories of messages: <ul style="list-style-type: none"> • DISPLAY • FLI • PA • PLI • SDF • TCHK • VCD • VITAL • WLF • MISC • <user-defined>
Effective Time	
File Info	Filename related to the cause of the message, and in some cases the line number in parentheses.
Id	Message number
Messages	Organized tree-structure of the sorted messages, as well as, when expanded, the text of the messages.
Objects	Object(s) related to the message, if any.
Process	
Region	Hierarchical region related to the message, if any.
Severity	Message severity, such as Warning, Note or Error.
Time	Time of simulation when the message was issued.
Timing Check Kind	Information about timing checks
Verbosity	Verbosity information from \$messagelog system tasks.

Popup Menu

Right-click anywhere in the window to open a popup menu that contains the following

selections:

Table 2-85. Message Viewer Window Popup Menu

Popup Menu Item	Description
View Source	Opens a Source window for the file, and in some cases takes you to the associated line number.
View Verbose Message	Displays the Verbose Message dialog box containing further details about the selected message.
Object Declaration	Opens and highlights the object declaration related to the selected message.
Goto Wave	Opens the Wave window and places the cursor at the simulation time for the selected message.
Display Reset	Resets the display of the window.
Display Options	Displays the Message Viewer Display Options dialog box, which allows you to further control which messages appear in the window.
Filter	Displays the Message Viewer Filter Dialog Box , which allows you to create specialized rules for filtering the Message Viewer.
Clear Filter	Restores the Message Viewer to an unfiltered view by issuing the messages clearfilter command.
Expand/Collapse Selected/All	Manipulates the expansion of the Messages column.

Related GUI Features

- The [Messages Bar](#) in the Wave window provides indicators as to when a message occurred.

Message Viewer Display Options Dialog Box

This dialog box allows you to control display options for the message viewer tab of the transcript window.

- **Hierarchy Selection** — This field allows you to control the appearance of message hierarchy, if any.
 - **Display with Hierarchy** — enables or disables a hierarchical view of messages.
 - **First by, Then by** — specifies the organization order of the hierarchy, if enabled.
- **Time Range** — Allows you to filter which messages appear according to simulation time. The default is to display messages for the complete simulation time.

- **Displayed Objects** — Allows you to filter which messages appear according to the values in the Objects column. The default is to display all messages, regardless of the values in the Objects column. The Objects in the list text entry box allows you to specify filter strings, where each string must be on a new line.

Message Viewer Filter Dialog Box

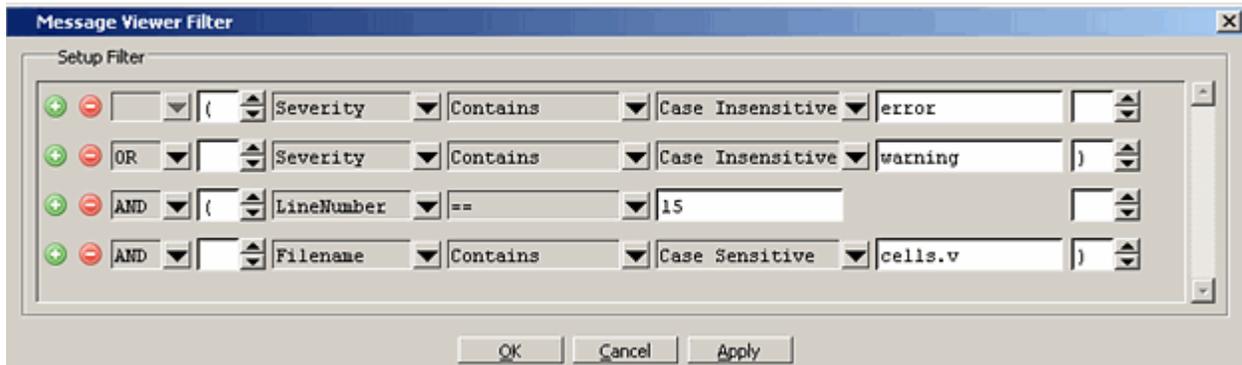
This dialog box allows you to create filter rules that specify which messages should be shown in the message viewer. It contains a series of dropdown and text entry boxes for creating the filter rules and supports the addition of additional rule (rows) to create logical groupings.

From left to right, each filter rule is made up of the following:

- **Add and Remove buttons** — either add a rule filter row below the current row or remove that rule filter row.
- **Logic field** — specifies a logical argument for combining adjacent rules. Your choices are: AND, OR, NAND, and NOR. This field is greyed out for the first rule filter row.
- **Open Parenthesis field** — controls rule groupings by specifying, if necessary, any open parentheses. The up and down arrows increase or decrease the number of parentheses in the field.
- **Column field** — specifies that your filter value applies to a specific column of the Message Viewer.
- **Inclusion field** — specifies whether the Column field should or should not contain a given value.
 - For text-based filter values your choices are: Contains, Doesn't Contain, or Exact.
 - For numeric- and time-based filter values your choices are: ==, !=, <, <=, >, and >=.
- **Case Sensitivity field** — specifies whether your filter rule should treat your filter value as Case Sensitive or Case Insensitive. This field only applies to text-based filter values.
- **Filter Value field** — specifies the filter value associated with your filter rule.
- **Time Unit field** — specifies the time unit. Your choices are: fs, ps, ns, us, ms. This field only applies to the Time selection from the Column field.
- **Closed Parenthesis field** — controls rule groupings by specifying, if necessary, any closed parentheses. The up and down arrows increase or decrease the number of parentheses in the field.

Figure 2-86 shows an example where you want to show all messages, either errors or warnings, that reference the 15th line of the file *cells.v*.

Figure 2-86. Message Viewer Filter Dialog Box



When you select OK or Apply, the Message Viewer is updated to contain only those messages that meet the criteria defined in the Message Viewer Filter dialog box.

Also, when selecting OK or Apply, the Transcript window will contain an echo of the messages setfilter command, where the argument is a Tcl definition of the filter. You can then cut/paste this command for reuse at another time.

Objects Window

Use this window to view the names and current values of declared data objects in the current region, as selected in the Structure window. Data objects include:

- signals
- nets
- registers
- constants and variables not declared in a process
- generics
- parameters
- transactions
- SystemC member data variables
- Questa Verification IP objects. Refer to the section [“Questa Verification IP Objects in the GUI”](#) for more information.

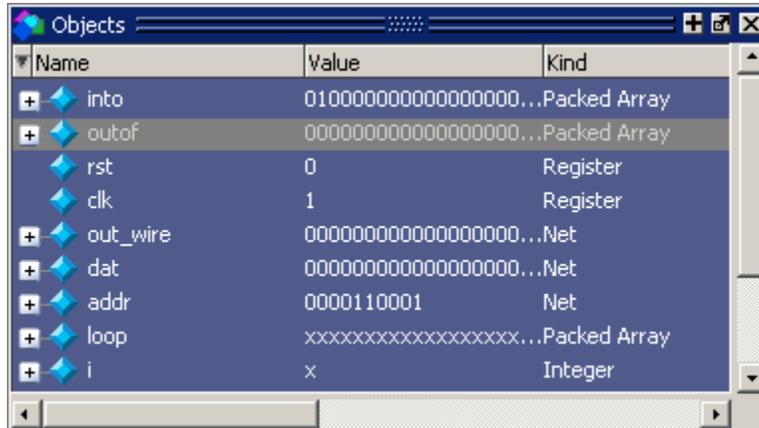
Accessing

Access the window using either of the following:

- Menu item: **View > Objects**

- Command: view objects
- Wave window: [View Objects Window Button](#)

Figure 2-87. Objects Window



Objects Window Tasks

This section describes tasks for using the Objects window.

Interacting with Other Windows

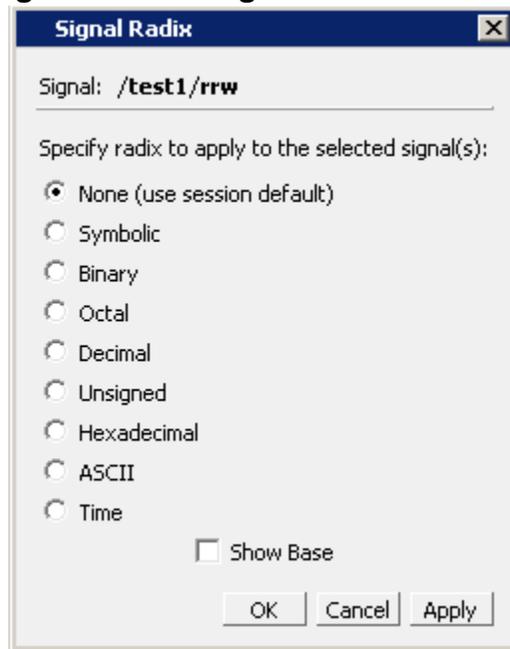
1. Click an entry in the window to highlight that object in the Dataflow, Schematic, and Wave windows.
2. Double-click an entry to highlights that object in a Source window

Setting Signal Radix

You can set the signal radix for a selected signal or signals in the Objects window as follows:

1. Click (LMB) a signal to select it or use Ctrl-Click Shift-Click to select a group of signals.
2. Select **Objects > Radix** from the menu bar; or right-click the selected signal(s) and select **Radix** from the popup menu.

This opens the Signal Radix dialog box ([Figure 2-88](#)), where you may select a radix. This sets the radix for the selected signal(s) in the Objects window and every other window where the signal appears.

Figure 2-88. Setting the Global Signal Radix from the Objects Window

Finding Contents of the Objects Window

You can filter the contents of the Objects window by either the Name or Value columns.

1. Ctrl-F to display the Find box at the bottom of the window.
2. Click the “Search For” button and select the column to filter on.
3. Enter a string in the Find text box
4. Enter

Refer to the section [“Using the Find and Filter Functions”](#) for more information.

Filtering Contents of the Objects Window

You can filter the contents of the Objects window by the Name column.

1. Ctrl-F to display the Find box at the bottom of the window.
2. Ctrl-M to change to “Contains” mode.
3. Enter a string in the Contains text box

The filtering will occur as you begin typing. You can disable this feature with ctrl-T.

Filters are stored relative to the region selected in the Structure window. If you re-select a region that had a filter applied, that filter is restored. This allows you to apply different filters to different regions.

Refer to the section “[Using the Find and Filter Functions](#)” for more information.

Filtering by Signal Type

The **View > Filter** menu selection allows you to specify which signal types to display in the Objects window. Multiple options can be selected. Select Change Filter to open the Filter Objects dialog, where you can select port modes and object types to be displayed.

Popup Menu

Right-click anywhere in the window to display the popup menu and select one of the following options:

Table 2-86. Objects Window Popup Menu

Popup Menu Item	Description
View Declaration	Opens a Source window to the declaration of the object
View Memory Contents	
Add Wave	Adds the selected object(s) to the Wave window
Add Dataflow	Adds the selected object(s) to a Dataflow window
Add to	Add the selected object(s) to any one of the following: Wave window, List window, Log file, Schematic window, Dataflow window. You may choose to add only the Selected Signals, all Signals in Region, all Signals in Design.
Event Traceback	Enables Causality Traceback <ul style="list-style-type: none"> • Show Cause • Show Driver • Show Root Cause • Show 'X' Cause (ChaseX)
Copy	Copies information about the object to the clipboard
Find	Opens the Find box
Insert Breakpoint	Adds a breakpoint for the selected object
Toggle Coverage	Control toggle coverage of the selected object(s). Submenus allow the following options: <ul style="list-style-type: none"> • Add - add to toggle coverage • Extended - include as extended toggle coverage • Enable - enable toggle coverage • Disable - disable toggle coverage • Reset - reset toggle coverage

Table 2-86. Objects Window Popup Menu

Popup Menu Item	Description
Modify	Modify the selected object(s) by selecting one of the following from the submenu: <ul style="list-style-type: none"> • Force - opens Force Selected Signal dialog • Remove Force - remove effect of force command • Change Value - change value of selected • Apply Clock - opens Define Clock dialog • Apply Wave - opens Create Pattern Wizard
Radix	Opens Signal Radix dialog, allowing you to set the radix of selected signal(s) in all windows
Show	Shows list of port types and object kinds that are displayed. Includes a Change Filter selection that opens the Filter Objects dialog, which allows you to filter the display.

Viewing Toggle Coverage in the Objects Window

Toggle coverage data can be displayed in the Objects window in multiple columns, as shown in [Figure 2-89](#). Right-click the column title bar and select Show All Columns to make sure all Toggle coverage columns are displayed. There is a column for each transition type.

Figure 2-89. Objects Window - Toggle Coverage

Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% Z
into	0100000000...	Reg	Internal	119628	119629	0	0	0	0	32	11	34.38%	34.38%	11.46%	0%
outof	0000000000...	Reg	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
rst	0	Reg	Internal	2	1	0	0	0	0	1	1	100%	100%	33.33%	0%
clk	1	Reg	Internal	83222	83223	0	0	0	0	1	1	100%	100%	33.33%	0%
out_wire	0000000000...	Net	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
dat	0000000000...	Net	Internal	23401	28607629308634542	119620	114418			32	6	18.75%	21.88%	47.92%	60.94%
addr	0000100000	Net	Internal	26006	26007	0	0	0	0	10	4	40%	40%	13.33%	0%
loop	xxxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	S10	Net	Internal	15602	15601	0	0	0	0	1	1	100%	100%	33.33%	0%
wr_	S11	Net	Internal	7803	7803	0	0	0	0	1	1	100%	100%	33.33%	0%

GUI Elements of the Objects Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-87. Toggle Coverage Columns in the Objects Window

Column name	Description
Name	the name of each object in the current region
Value	the current value of each object
Kind	the object type
Mode	the object mode (internal, in, out, and so forth.)
1H -> 0L	the number of times each object has transitioned from a 1 or a High state to a 0 or a Low state
0L -> 1H	the number of times each object has transitioned from a 0 or a Low state to 1 or a High state
0L -> Z	the number of times each object has transitioned from a 0 or a Low state to a high impedance (Z) state
Z -> 0L	the number of times each object has transitioned from a high impedance state to a 0 or a Low state
1H -> Z	the number of times each object has transitioned from a 1 or a High state to a high impedance state
Z -> 1H	the number of times each object has transitioned from a high impedance state to 1 or a High state
State Count	the number of values a state machine variable can have
State Hits	the number of state machine variable values that have been hit
State %	the current ration of State Hits to State Count
# Nodes	the number of scalar bits in each object
# Toggled	<p>the number of nodes that have transitioned at least once. A signal is considered toggled if and only if:</p> <ul style="list-style-type: none"> • it has 0- >1 and 1->0 transitions and NO Z transitions, or • if there are ANY Z transitions, it must have ALL four of the Z transitions. <p>Otherwise, the counts are place in % 01 or % Z columns. For more specifics on what is considered “toggled”, see “Understanding Toggle Counts”</p>
% Toggled	the current ratio of the # Toggled to the # Nodes for each object
% 01	the percentage of 1H -> 0L and 0L -> 1H transitions that have occurred (transitions in the first two columns)
% Full	the percentage of all transitions that have occurred (all six columns)

Table 2-87. Toggle Coverage Columns in the Objects Window

Column name	Description
% Z	the percentage of 0L -> Z , Z -> 0L , 1H -> Z , and Z -> 1H transitions that have occurred (last four columns)

Processes Window

Use this window to view a list of HDL and SystemC processes in one of four viewing modes:

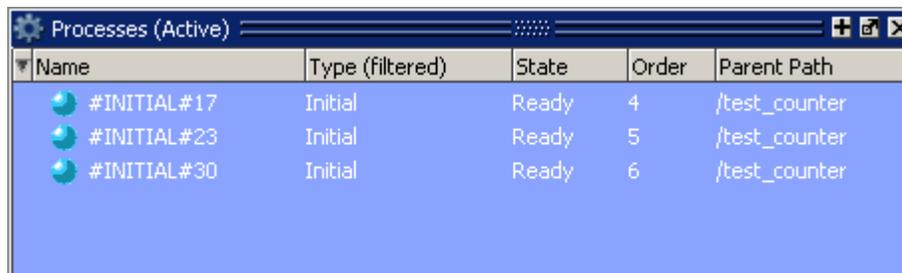
- Active (default) — active processes in your simulation.
- In Region — process in the selected region.
- Design — intended for primary navigation of ESL (Electronic System Level) designs where processes are a foremost consideration.
- Hierarchy — a tree view of any SystemVerilog nested fork-joins.

In addition, the data in this window will change as you run your simulation and processes change states or become inactive.

Accessing

Access the window using either of the following:

- Menu item: **View > Process**
- Command: view process

Figure 2-90. Processes Window


Name	Type (filtered)	State	Order	Parent Path
#INITIAL#17	Initial	Ready	4	/test_counter
#INITIAL#23	Initial	Ready	5	/test_counter
#INITIAL#30	Initial	Ready	6	/test_counter

Processes Window Tasks

This section describes tasks for using the Processes window.

Changing Your Viewing Mode

You can change the display to show all the processes in a region or in the entire design by doing any one of the following:

- Select **Process > In Region, Design, Active, or Hierarchy**.
- Use the [Process Toolbar](#)
- Right-click in the Process window and select **In Region, Design, Active, or Hierarchy**.

The view mode you select is persistent and is “remembered” when you exit the simulation. The next time you bring up the tool, this window will initialize in the last view mode used.

Filtering Processes

You can control which processes are visible in the Processes window as follows:

1. Right-click in the Processes window and select Display Options.
2. In the Process Display Options dialog box select the Type or States you want to include or exclude from the window.
3. OK

When you filter the window according to specific process states, the heading of the State column changes to “State (filtered)” as shown in [Figure 2-91](#).

Figure 2-91. Column Heading Changes When States are Filtered



Name	Type (filtered)	State (filtered)	Order	Parent Path
#ALWAYS#35	Always	Ready	3	/top/m
#ALWAYS#155	Always	Ready	2(Next Active)	/top/c

The default “No Implicit & Primitive” selection causes the Process window to display all process types except implicit and primitive types. When you filter the display according to specific process types, the heading of the Type column becomes “Type (filtered)”.

Once you select the options, data will update as the simulation runs and processes change their states. When the In Region view mode is selected, data will update according to the region selected in the Structure window.

Viewing the Full Path of the Process

By default, all processes are displayed without the full hierarchical context (path). You can display processes with the full path by selecting **Process > Show Full Path**

Viewing Processes in Post-Processing Mode

This window also shows data in the post-processing (WLF view or Coverage view) mode. You will need to log processes in the simulation mode to be able to view them in post-processing mode.

In the post-processing mode, the default selection values will be same as the default values in the live simulation mode.

Things to remember about the post-processing mode:

- There are no active processes, so the Active view mode selection will not show anything.
- All processes will have same ‘Done’ state in the post-processing mode.
- There is no order information, so the Order column will show ‘-’ for all processes.

Setting a Ready Process as the Next Active Process

You can select any “Ready” process and set it to be the next Active process executed by the simulator, ahead of any other queued processes. To do this, simply right-click any “Ready” process and select **Set Next Active** from the popup context menu.

When you set a process as the next active process, you will see “(Next Active)” in the Order column of that process (Figure 2-92).

Figure 2-92. Next Active Process Displayed in Order Column

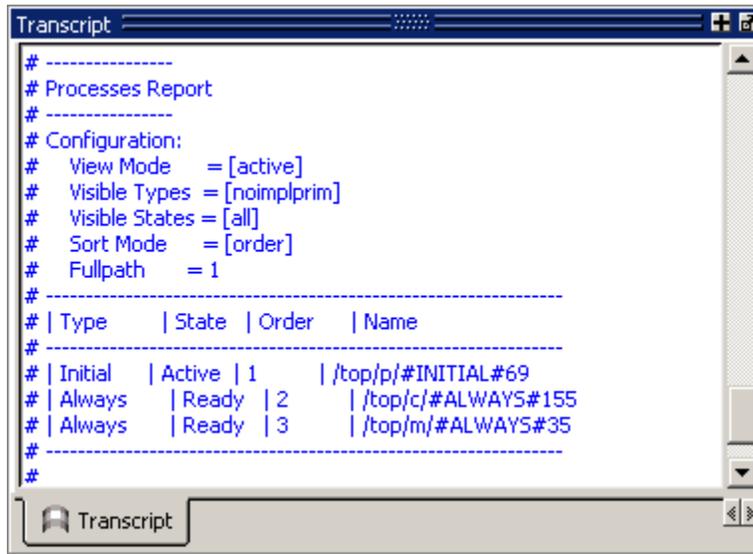


Name	Type (filtered)	State	Order	Parent Path
#ALWAYS#35	Always	Ready	3	/top/m
#ALWAYS#155	Always	Ready	2(Next Active)	/top/c
#INITIAL#69	Initial	Active	1	/top/p
#ASSIGN#25	Assign	Wait	-	/top/p
#ASSIGN#24	Assign	Wait	-	/top/p
#ASSIGN#23	Assign	Wait	-	/top/p
#ASSIGN#21	Assign	Wait	-	/top/c/s3

Creating Textual Process Report

You can create a textual report of all processes by using the [process report](#) command.

Figure 2-93. Sample Process Report in the Transcript Window



GUI Elements of the Processes Window

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-88. Processes Window Column Descriptions

Column Title	Description
Name	Name of the process.
Order	Execution order of all processes in the Active and Ready states. Refer to the section “ Process Order Description ” for more information.
Parent Path	Hierarchical parent pathname of the process
State	Process state. Refer to the section “ Process State Definitions ” for more information.
Type	Process type, according to the language. Refer to the section “ Process Type Definitions ” for more information.

Process State Definitions

- **Idle** — Indicates an inactive SystemC Method, or a process that has never been active. The Idle state will occur only for SC processes or methods. It will never occur for HDL processes.

- **Wait** — Indicates the process is waiting for a wake up trigger (change in VHDL signal, Verilog net, SystemC signal, or a time period).
- **Ready** — Indicates the process is scheduled to be executed in current simulation phase (or in active simulation queue) of current delta cycle.
- **Active** – Indicates the process is currently active and being executed.
- **Queued** — Indicates the process is scheduled to be executed in current delta cycle, but not in current simulation phase (or in active simulation queue).
- **Done** — Indicates the process has been terminated, and will never restart during current simulation run.

Processes in the Idle and Wait states are distinguished as follows. Idle processes (except for ScMethods) have never been executed before in the simulation, and therefore have never been suspended. Idle processes will become Active, Ready, or Queued when a trigger occurs. A process in the Wait state has been executed before but has been suspended, and is now waiting for a trigger.

SystemC methods can have one of the four states: Active, Ready, Idle or Queued. When ScMethods are not being executed (Active), or scheduled (Ready or Queued), they are inactive (Idle). ScMethods execute in 0 time, whenever they get triggered. They are never suspended or terminated.

Process Type Definitions

The **Type** column displays the process type according to the language used. It includes the following types:

- Always
- Assign
- Final
- Fork-Join (dynamic process like fork-join, sc_spawn, and so forth.)
- Initial
- Implicit (internal processes created by simulator like Implicit wires, and so forth.)
- Primitive (UDP, Gates, and so forth.)
- ScMethod
- ScThread (SC Thread and SC CThread processes)
- VHDL Process

Process Order Description

The **Order** column displays the execution order of all processes in the Active and Ready states in the active kernel queue. Processes that are not in the Active or Ready states do not yet have any order, in which case the column displays a dash (-). The Process window updates the execution order automatically as simulation proceeds.

Profiling Windows

Use these five windows to view performance or memory profiling information about your simulation.

- Calltree — Displays information in a hierarchical form that indicates the call order dependencies of functions or routines.
- Design Units — Displays information aggregated for the different design units.
- Ranked — Displays information for each function or instance.
- Structural — Displays information aggregated for different instances.
- Profile Details — Displays detailed profiling information based on selections in the other Profile Windows

Prerequisites

You must have enabled performance or memory profiling. Refer to the chapter “[Profiling Performance and Memory Use](#)” for more information.

Accessing

Access the Profile Calltree window using either of the following:

- Menu item: **View > Profiling > Call Tree Profile**
- Command: view calltree

Figure 2-94. Profile Calltree Window

Name	Under(row)	In(row)	Under(%)	In(%)
proc.v:79	2	0	50.0%	0.0%
_vl_attach_process	2	2	50.0%	50.0%
proc.v:88	1	0	25.0%	0.0%
proc.v:39	1	0	25.0%	0.0%
_vl_systf_calltf	1	1	25.0%	25.0%
proc.v:94	1	0	25.0%	0.0%
_vl_systf_calltf	1	0	25.0%	0.0%
Tcl_DoOneEvent	1	0	25.0%	0.0%
Tcl_WaitForEvent	1	1	25.0%	25.0%
<NoCallStack>	0	0	0.0%	0.0%

Access the Profile Design Unit window using either of the following:

- Menu item: **View > Profiling > Design Unit Profile**
- Command: view du

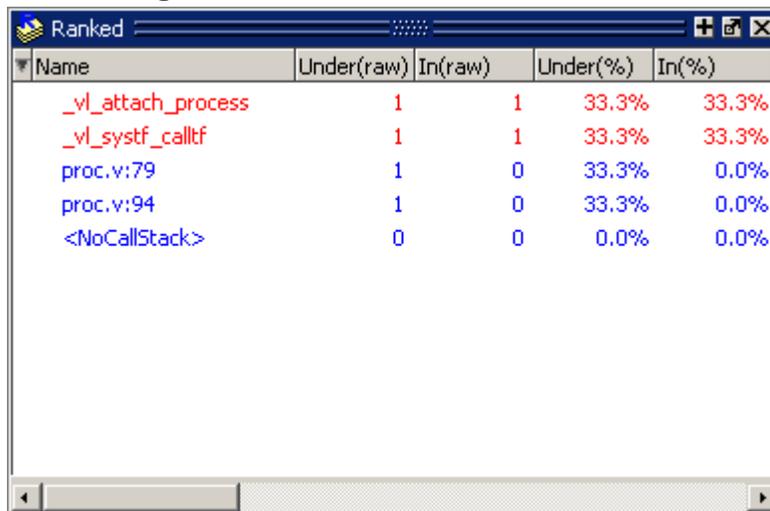
Figure 2-95. Profile Design Unit Window

Name	Count	Under(row)	In(row)	Under(%)
proc	1	4	4	100.0%
proc.v:79		2	0	50.0%
_vl_attach_proc...		2	2	50.0%
proc.v:88		1	0	25.0%
+ proc.v:39		1	0	25.0%
proc.v:94		1	0	25.0%
+ _vl_systf_calltf		1	0	25.0%
+ <NoContext>	1	0	0	0.0%

Access the Profile Ranked window using either of the following:

- Menu item: **View > Profiling > Ranked Profile**
- Command: view ranked

Figure 2-96. Profile Ranked Window

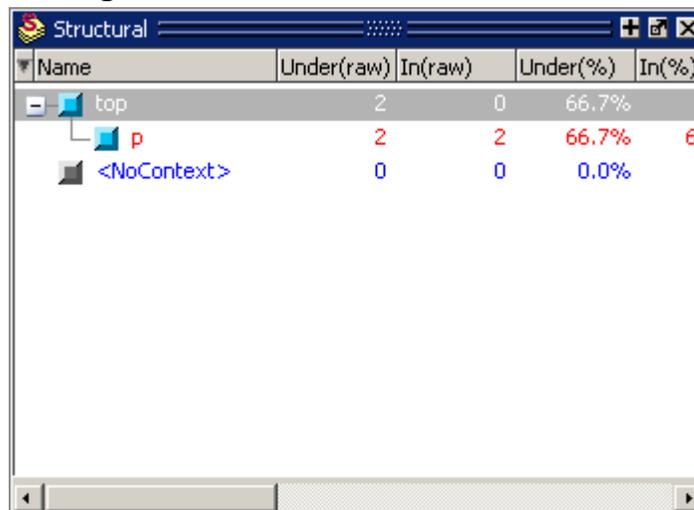


Name	Under(row)	In(row)	Under(%)	In(%)
_vl_attach_process	1	1	33.3%	33.3%
_vl_systf_calltf	1	1	33.3%	33.3%
proc.v:79	1	0	33.3%	0.0%
proc.v:94	1	0	33.3%	0.0%
<NoCallStack>	0	0	0.0%	0.0%

Access the Profile Structural window using either of the following:

- Menu item: **View > Profiling > Structural Profile**
- Command: view structural

Figure 2-97. Profile Structural Window

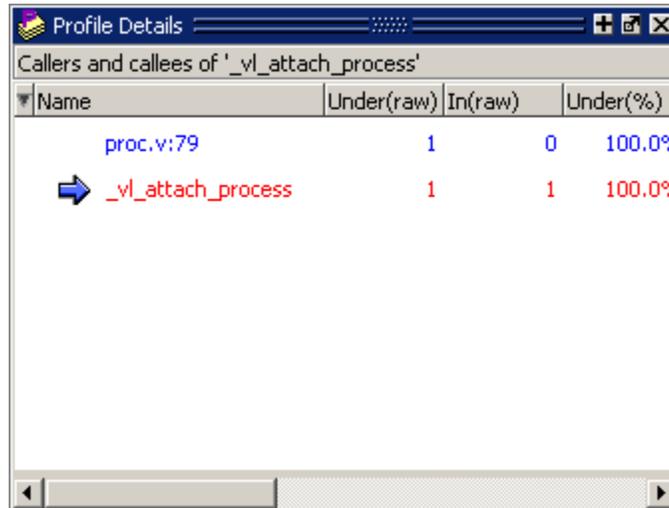


Name	Under(row)	In(row)	Under(%)	In(%)
top	2	0	66.7%	0.0%
p	2	2	66.7%	66.7%
<NoContext>	0	0	0.0%	0.0%

Access the Profile Structural window using either of the following:

- Menu item: **View > Profiling > Profile Details**
- Command: view profiledetails

Figure 2-98. Profile Details Window



GUI Elements of the Profile Windows

This section describes GUI elements specific to this Window.

Column Descriptions

Table 2-89. Profile Calltree Window Column Descriptions

Column Title	Description
%Parent	lists the ratio, as a percentage, of the samples collected during the execution of a function or instance to the samples collected in the parent function or instance. (Not available in the Profile Ranked window.)
Count	(Only available in the Profile Design Unit window.)
In%	lists the ratio (as a percentage) of the total samples collected during a function or instance.
In(raw)	lists the raw number of Profiler samples collected during a function or instance.
MemIn	lists the amount of memory allocated to a function or instance.
MemIn(%)	lists the ratio (as a percentage) of the amount of memory allocated to a function or instance to the total memory available.
MemUnder	lists the amount of memory allocated to a function, including all support routines under that function; or, the amount of memory allocated to an instance, including all instances beneath it in the structural hierarchy.

Table 2-89. Profile Calltree Window Column Descriptions

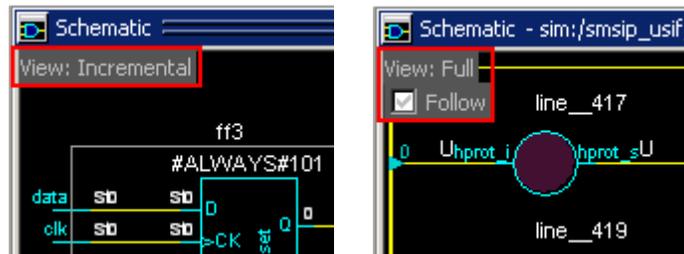
Column Title	Description
MemUnder(%)	lists the ratio (as a percentage) of the amount of memory allocated to a function and all of its support routines to the total memory available; or, the ratio of the amount of memory allocated to an instance, including all instances beneath it in the structural hierarchy, to the total memory available.
Name	lists the parts of the design for which profiling information was captured.
sum(MemIn(%))	lists the ratio of the cumulative memory allocated. (Only available in the Profile Ranked and Profile Design Unit windows.)
sum(MemIn)	lists the cumulative memory allocated. (Only available in the Profile Ranked and Profile Design Unit windows.)
Under (raw)	lists the raw number of Profiler samples collected during the execution of a function, including all support routines under that function; or, the number of samples collected for an instance, including all instances beneath it in the structural hierarchy.
Under(%)	lists the ratio (as a percentage) of the samples collected during the execution of a function and all support routines under that function to the total number of samples collected; or, the ratio of the samples collected during an instance, including all instances beneath it in the structural hierarchy, to the total number of samples collected.

Schematic Window

The Schematic window provides two views of the design — a Full View, which is a structural overview of the design; and an Incremental View, which uses Click-and-Sprout actions to incrementally add to the selected net's fanout. The Incremental view displays the logical gate equivalent of the RTL portion of the design, making it easier to understand the intent of the design. (For additional information, please refer to the [Schematic Window](#) chapter.)

A “View:” indicator is displayed in the top left corner of the window ([Figure 2-99](#)). You can toggle back and forth between views by simply clicking this “View:” indicator.

Figure 2-99. Schematic View Indicator



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

The Full View is a more static view that can be dynamically linked to your selections in other windows with the “Follow” selection.

Prerequisites

To create the necessary data to display the schematic you must:

- use the `+acc` switch with the `vopt` command to provide accessibility into the design for debugging; and use the `-debug` switch with the `vopt` command to collect combinatorial and sequential logic data into the work library
- use the `-debugdb` switch with the `vsim` command to create the debug database
- `log` the results

For example, if you have a Verilog design with a top level module named `top.v` you would do the following:

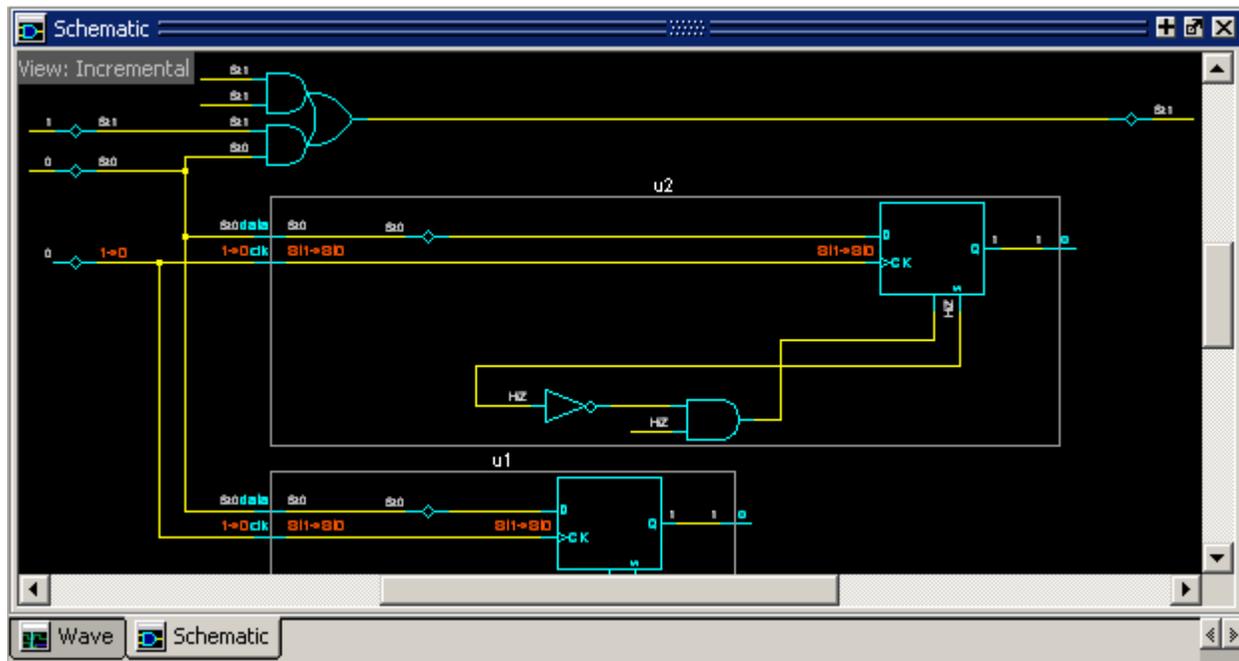
```
vlib work
vlog *.v
vopt +acc top -o top_opt -debugdb
vsim -debugdb top_opt
log -r /*
```

Accessing

Access the window using either of the following:

- Menu item: **View > Schematic** or **Add > To Schematic**
- Command: `view schematic` or `add schematic <design_unit_name>`

Figure 2-100. Schematic Window



Schematic Window Tasks

This section describes tasks for using the Schematic window.

Adding Objects to the Incremental View

You can use any of the following methods to add objects to the Incremental View of the Schematic window:

- Drag and drop objects from other windows. Both nets and instances may be dragged and dropped. Dragging an instance will result in the addition of all nets connected to that instance. When an object is dragged and dropped into the Incremental view, the [add schematic](#) command will be reflected in the Transcript window.
- Use the **Add > To Schematic** menu options:
 - **Selected Signals** — Display selected signals
 - **Signals in Region** — Display all signals from the current region.
 - **Signals in Design** — Clear the window and display all signals from the entire design.
- Select the object(s) you want placed in the Schematic Window, then click-and-hold the [Add Selected to Window Button](#) in the [Standard Toolbar](#) and select **Add to Schematic**. 
- Use the [add schematic](#) command.

Navigating in the Schematic Window

You can use the mouse to navigate and select items within the Schematic window. The following descriptions are based on the Select mouse mode (as set by the **Schematic > Mouse Mode** menu).

- Strokes with the **middle mouse button**:
 - Up — Move up in the hierarchy (does nothing when you are already at the top level)
 - Down — Move down in the hierarchy to the selected instance or the instance from which you began the stroke.
 - Up/Left — Zoom full.
 - Down/Left — Zoom in on any selected items.
 - Up/Right — Zoom out. The factor of the zoom changes depending on the length of the stroke.
 - Down/Right — Zoom area. The box indicates your desired zoom view.
- Zoom in and out with the **mouse scroll wheel**.

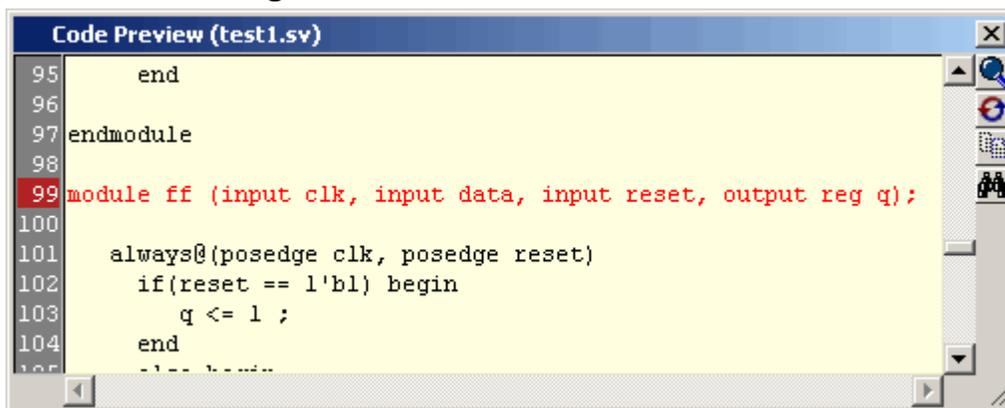
The view will zoom and center on your mouse location.
- Selecting objects with the **left mouse button**:

Your selection is also reflected in the Structure, Objects, and Wave windows.

Single click — highlights selected objects.

Double click — opens a Code Preview window with the source code of the selected item highlighted.

Figure 2-101. Code Preview Window



Click and drag — selects all objects within the bounding box.

Shift click — highlights multiple selected objects.

The alternate mouse modes affect the above mouse controls as follows:

- **Schematic > Mouse Mode > Zoom Mode**
 - Left mouse button — Single click selects items, click and drag performs zoom actions.
 - Middle mouse button — click and drag pans the schematic view.
- **Schematic > Mouse Mode > Pan Mode**
 - Left mouse button — Single click selects items, click and drag pans the schematic view.
 - Middle mouse button — click and drag performs zoom actions.

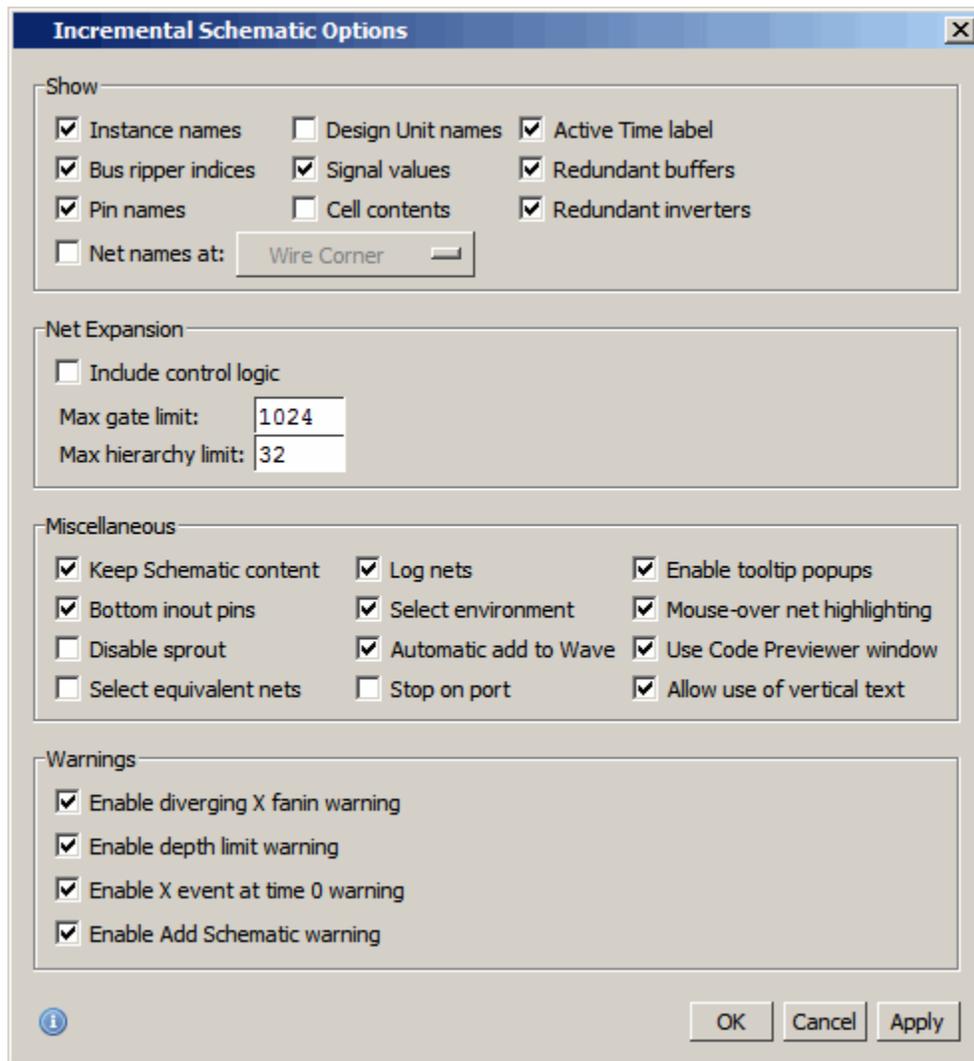
Controlling the Data Displayed in the Schematic Window

The schematic window provides additional control over the amount and type of information displayed, depending on whether you are using the Incremental View or the Full View. Display options are available by selecting **Schematic > Preferences** from the Main menu when the Schematic window is active.

Incremental View Display Options

When the Incremental View is active the **Schematic > Preferences** menu selection opens the Incremental Schematic Options dialog ([Figure 2-102](#)).

Figure 2-102. Incremental Schematic Options Dialog



The **Show** section of the dialog provides the following options when the boxes are checked:

- Instance names — Show instance names for architectures, modules, processes, etc.
- Bus ripper indices — Show ripper indices for busses.
- Pin names — Show pin names in the architectures, modules, processes, etc.
- Design Unit names — Show design unit names for architectures, modules, processes, etc.
- Signal values — Show signal values annotated onto the nets. Signal values are based on the “active time”— which is set by the active cursor in the Wave window or the active time Label in the Source or Schematic windows.

- Cell contents — Show the internals of a library cell (^ celldefine or VITAL).

Note



Changing this Hierarchy option causes the Schematic window to be erased.

- Active Time Label — Displays the current Active Time or the Now (end of simulation) time. This is the time used to control state values annotated in the window. (For details, see [Active Time Label](#).)

Figure 2-103. Active Time Label



- Redundant buffers — Display redundant buffers.
- Redundant inverters — Displays redundant inverters.
- Net Names — Show all net names:
 - Wire corner — Displays net names at or near wire corners.
 - Window Edge— Displays net names only if net extends past the edge of the window.

The **Net Expansion** section of the dialog controls whether control logic is followed when doing an Expand To Fanin/Fanout:

- Max gate limit — Specifies the maximum number of gates the fanin/fanout should go through before stopping. The default value is 1024.
- Max hierarchy limit — specifies the maximum number of hierarchy levels the fanin/fanout should go through before stopping. The default value is 32.

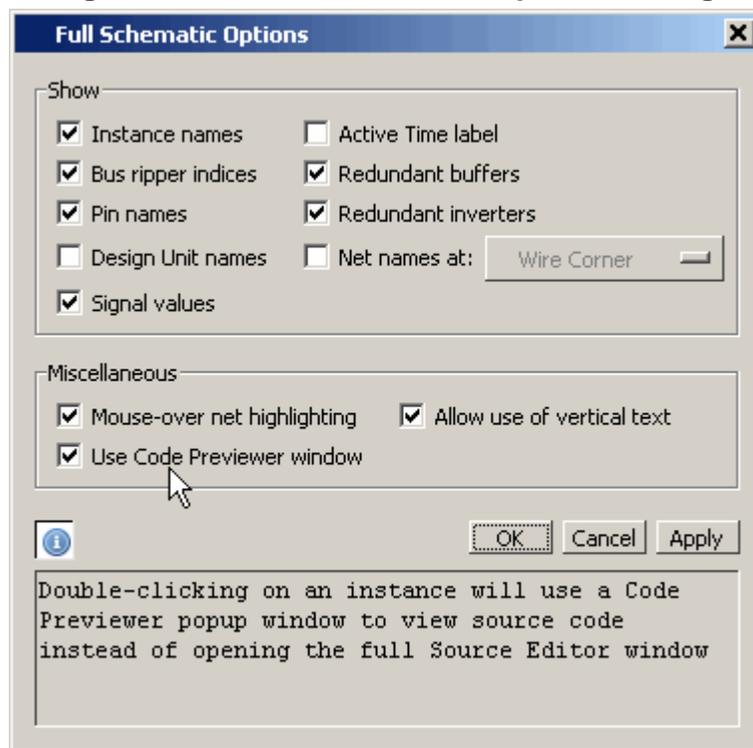
Click the Information button at the bottom left corner to get option descriptions when you mouse over the option.



Full View Display Options

When the Incremental View is active the **Schematic > Preferences** menu selection opens the Full Schematic Options dialog. [Figure 2-104](#) show the Options dialog with the information window open at the bottom.

Figure 2-104. Full Schematic Options Dialog



GUI Elements of the Schematic Window

This section describes GUI elements specific to this Window.

Popup Menu for Incremental View

Right-click anywhere in the Incremental View to display the popup menu and select one of the following options:

Table 2-90. Incremental View Popup Menu

Popup Menu Item	Selection	Description
View Selection	Declaration	Opens a Source window to the line of code where the object is declared.
	Instantiation	Applies only to modules. Opens a Source window to the line of code where the module is instantiated.
	FSM Viewer	Displays the selected item in the FSM Viewer.
	Wave Pane	Displays selected signals in embedded Wave viewer
	This Window	Erase contents of current Incremental view and redraw only selected object
	New Window	Redraw selected object(s) in Incremental View of new Schematic window

Table 2-90. Incremental View Popup Menu

Popup Menu Item	Selection	Description
Zoom	Zoom In Zoom Out Zoom Full Zoom Selected Zoom Highlight	Zoom in 2x Zoom out 2x Zoom so all objects fits into display, Zoom into the selected object Zoom into the highlighted object.
Fold/Unfold		Hides or shows the contents of selected instance. Displays folded instance as a solid blue box with dashed gray border.
Expand Net To	Drivers Readers Drivers & Readers Design Inputs Hierarchy Inputs	Displays all drivers of the selected signal Displays all readers of the selected signal Displays all drivers & readers of the selected signal
Event Traceback	Show Cause Show Driver Show Root Cause	Traces to the first sequential process that drives the selected signal at the current time Traces to the immediate driving process of the selected signal at the current time Traces to the root cause of the selected signal at the current time
Highlight	Add Remove Remove All	Highlights any selected objects with color you select Removes highlight color from selected objects Removes all highlight colors from display
Edit	Undo Redo Cut Copy Paste Delete Select Highlighted Select All Unselect All Regenerate Layout	Undo previous action Redo undone action Cut selected Copy selected Paste selected Delete selected Make highlighted objects the selected objects Select all objects in display Unselect all objects in display Regenerate display to improve layout of complex design
Find		Opens the Search Bar (at bottom of window) in the Find mode to make searching for objects easier, especially with large designs.

Table 2-90. Incremental View Popup Menu

Popup Menu Item	Selection	Description
Show	Unconnected Pins	Shows or hides unconnected pins in folded instances
	Instance Names	Display instance names when checked
	Net Names	Display net names when checked
	Net Rip Index	Display bus ripper indices when checked
	Pin Names	Display pin names when checked
	Design Unit Names	Display design unit names when checked
	Signal Values	Display signal values when checked
	Show All	Display all of the above
Hide All	Hide all of the above	

Popup Menu for Full View

Right-click anywhere in the Full View to display the popup menu and select one of the following options:

Table 2-91. Full View Popup Menu

Popup Menu Item	Selection	Description
View Selection	Declaration	Opens a Source window to the line of code where the object is declared.
	Instantiation	Applies only to modules. Opens a Source window to the line of code where the module is instantiated.
Zoom	Zoom In	Zoom in 2x
	Zoom Out	Zoom out 2x
	Zoom Full	Zoom so all objects fits into display,
	Zoom Selected	Zoom into the selected object
	Zoom Highlight	Zoom into the highlighted object.
Add	Add all signals to Wave	Add all signals in Full view to Wave window
	Add to Wave	Add selected to Wave window.
	Add to Schematic	Current Window — Erase contents of current Full view and redraw only selected object New Window — Redraw current selection in Incremental View of new Schematic window
	Add to Dataflow	Add selected to Dataflow window
	Add to List	Add selected to List window
	Add to Watch	Add selected to Watch window
	Log	Log all signals and objects in view
Highlight	Add	Highlights any selected objects with color you select
	Remove	Removes highlight color from selected objects
	Remove All	Removes all highlight colors from display

Table 2-91. Full View Popup Menu

Popup Menu Item	Selection	Description
Edit	Undo Redo Copy Select Highlighted Select All Unselect All Regenerate Layout	Undo previous action Redo undone action Copy selected Make highlighted objects the selected objects Select all objects in display Unselect all objects in display Regenerate display to improve layout of complex design
Find		Opens the Search Bar (at bottom of window) in the Find mode to make searching for objects easier, especially with large designs.
Show	Instance Names Net Names Net Rip Index Pin Names Design Unit Names Signal Values Show All Hide All	Display instance names when checked Display net names when checked Display bus ripper indices when checked Display pin names when checked Display design unit names when checked Display signal values when checked Display all of the above Hide all of the above

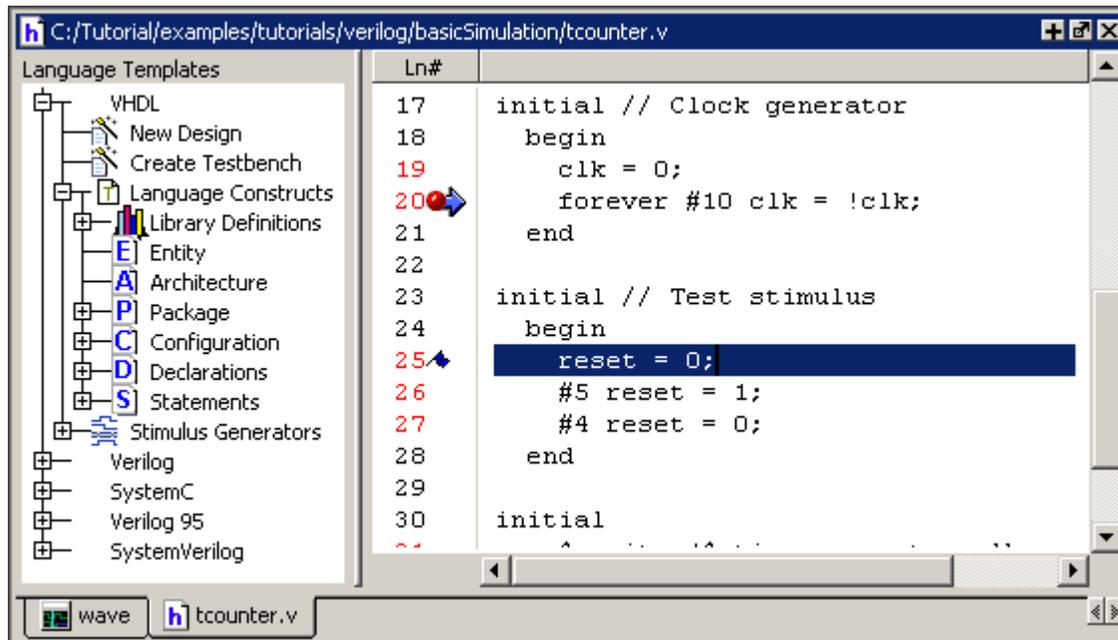
Source Window

The Source window allows you to view and edit source files as well as set breakpoints, step through design files, and view code coverage statistics.

By default, the Source window displays your source code with line numbers. You may also see the following graphic elements:

- Red line numbers — denote executable lines, where you can set a breakpoint
- Blue arrow — denotes the currently active line or a process that you have selected in the [Processes Window](#)
- Red ball in line number column — denotes file-line breakpoints; gray ball denotes breakpoints that are currently disabled
- Blue flag in line number column — denotes line bookmarks
- Language Templates pane — displays templates for writing code in VHDL, Verilog, SystemC, Verilog 95, and SystemVerilog ([Figure 2-105](#)). See [Using Language Templates](#).

Figure 2-105. Source Window Showing Language Templates



- Underlined text — denotes a hypertext link that jumps to a linked location, either in the same file or to another Source window file. Display is toggled on and off by the Source Navigation button.
- Active Time Label — Displays the current Active Time or the Now (end of simulation) time. This is the time used to control state values annotated in the window. (For details, see [Active Time Label](#).)

Also, you will see various code coverage indicator icons (see “[Coverage Data in the Source Window](#)” for details):

- Green check mark — denotes statements, branches (true), or expressions in a particular line that have been covered.
- Red X with no subscripts denotes that multiple kinds of coverage on the line are not covered.
- Red X with subscripts — denotes a statement, branch (false or true), condition or expression was not covered.
- Green E with no subscripts— denotes a line of code to which active coverage exclusions have been applied. Every item on line is excluded; none are hit.
- Green E with subscripts — denotes a line of codes with various degrees of exclusion.

Opening Source Files

You can open source files using the **File > Open** command or by clicking the **Open** icon. Alternatively, you can open source files by double-clicking objects in other windows. For example, if you double-click an item in the Objects window or in the structure tab (**sim** tab), the underlying source file for the object will open in the Source window and scroll to the line where the object is defined.

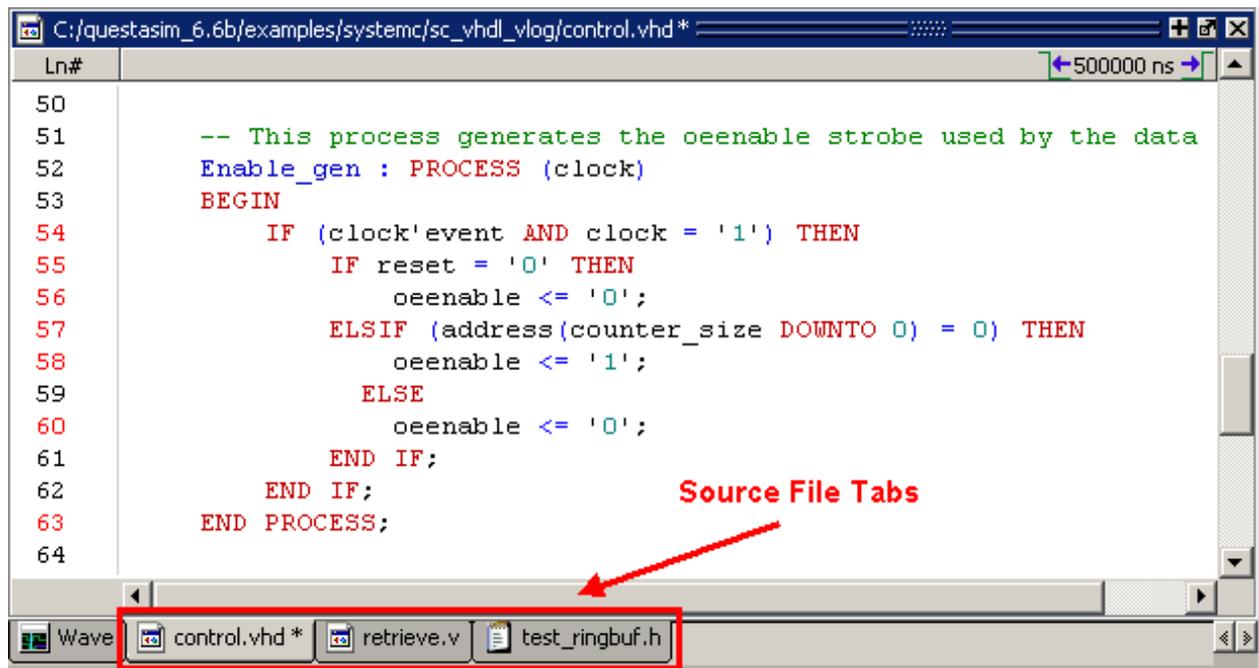
From the command line you can use the [edit](#) command.

By default, files you open from within the design (such as when you double-click an object in the Objects window) open in Read Only mode. To make the file editable, right-click in the Source window and select (uncheck) Read Only. To change this default behavior, set the PrefSource(ReadOnly) variable to 0. See [Simulator GUI Preferences](#) for details on setting preference variables.

Displaying Multiple Source Files

By default each file you open or create is marked by a window tab, as shown in the graphic below.

Figure 2-106. Displaying Multiple Source Files



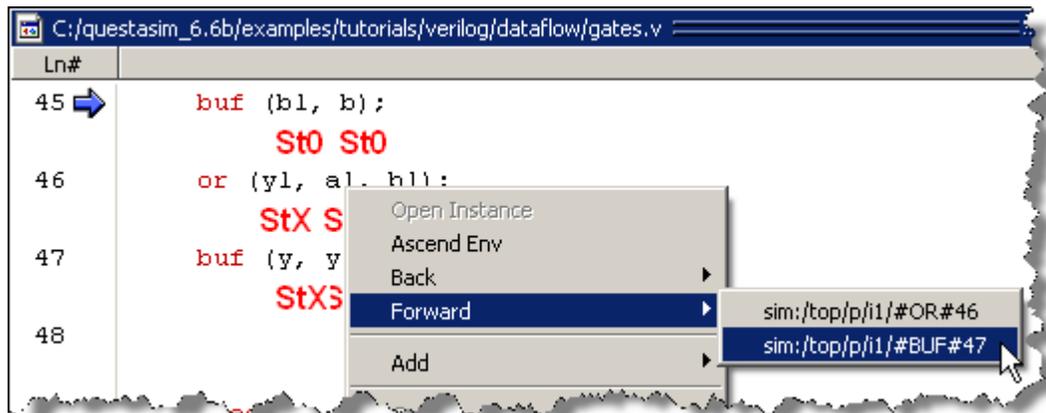
Dragging and Dropping Objects into the Wave and List Windows

ModelSim allows you to drag and drop objects from the Source window to the Wave and List windows. Double-click an object to highlight it, then drag the object to the Wave or List window. To place a group of objects into the Wave and List windows, drag and drop any section of highlighted code. When an object is dragged and dropped into the Wave window, the [add wave](#) command will be reflected in the Transcript window.

Setting your Context by Navigating Source Files

When debugging your design from within the GUI, you can change your context while analyzing your source files. [Figure 2-107](#) shows the pop-up menu the tool displays after you select then right-click an instance name in a source file.

Figure 2-107. Setting Context from Source Files



This functionality allows you to easily navigate your design for debugging purposes by remembering where you have been, similar to the functionality in most web browsers. The navigation options in the pop-up menu function as follows:

- **Open Instance** — changes your context to the instance you have selected within the source file. This is not available if you have not placed your cursor in, or highlighted the name of, an instance within your source file.

If any ambiguities exists, most likely due to generate statements, this option opens a dialog box allowing you to choose from all available instances.

- **Ascend Env** — changes your context to the file and line number in the parent region where the current region is instantiated. This is not available if you are at the top-level of your design.
- **Forward/Back** — allows you to change to previously selected contexts. This is not available if you have not changed your context.

The Open Instance option is essentially executing an **environment** command to change your context, therefore any time you use this command manually at the command prompt, that information is also saved for use with the Forward/Back options.

Highlighted Text in a Source Window

The Source window can display text that is highlighted as a result of various conditions or operations, such as the following:

- Double-clicking an error message in the transcript shown during compilation
- Using **Event Traceback > Show Driver**
- Coverage-related operations.

In these cases, the relevant text in the source code is shown with a persistent highlighting. To remove this highlighted display, choose **More > Clear Highlights** from the right-click popup menu of the Source window. If the Source window is docked, you can also perform this action by selecting **Source > More > Clear Highlights** from the Main menu. If the window is undocked, select **Edit > Advanced > Clear Highlights**.

Note



Clear Highlights does not affect text that you have selected with the mouse cursor.

Example

To produce a compile error that displays highlighted text in the Source window, do the following:

1. Choose **Compile > Compile Options**
2. In the Compiler Options dialog box, click either the VHDL tab or the Verilog & SystemVerilog tab.
3. Enable Show source lines with errors and click OK.
4. Open a design file and create a known compile error (such as changing the word “entity” to “entry” or “module” to “nodule”).
5. Choose **Compile > Compile** and then complete the Compile Source Files dialog box to finish compiling the file.
6. When the compile error appears in the Transcript window, double-click on it.
7. The source window is opened (if needed), and the text containing the error is highlighted.
8. To remove the highlighting, choose **Source > More > Clear Highlights**.

Hyperlinked (Underlined) Text in a Source Window

The Source window supports hyperlinked navigation, providing links displayed as underlined text. To turn hyperlinked text on or off in the Source window, do the following:

1. Click anywhere in the Source window. This enables the display of the Simulate toolbar (see [Table 2-34](#)).
2. Click the Source Navigation button.

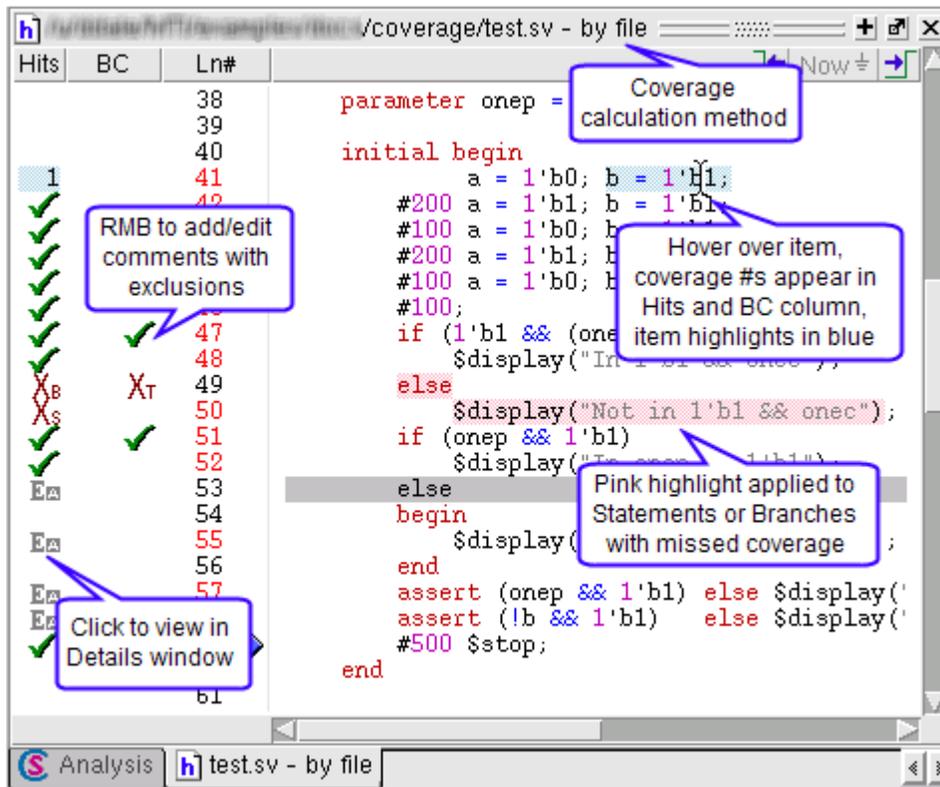
When you double-click on hyperlinked text, the selection jumps from the usage of an object to its declaration. This provides the following operations:

- Jump from the usage of a signal, parameter, macro, or a variable to its declaration.
- Jump from a module declaration to its instantiation, and vice versa.
- Navigate back and forth between visited source files.

Coverage Data in the Source Window

The [Source Window](#) includes two columns for code coverage statistics – the Hits column and the BC (Branch Coverage) column. These columns provide an immediate visual indication about how your source code is executing. The code coverage indicator icons include check marks, ‘X’s and ‘E’s. A description of each code coverage indicator icon is provided in [Table 2-92](#).

Figure 2-108. Coverage in Source Window



To see more information about any coverage item, click on the indicator icon, or in the Hits or BC column for the line of interest. In the case of a multiple-line item, this would be last line of the item. If the Coverage Details window is open, this brings up detailed coverage information for that line. If the window is not open, a right click menu option is available to open it.

For example, when you select an expression in the Code Coverage Analysis' Expression Analysis window, and you click in the column of a line containing an expression, the associated truth tables appear in the Coverage Details window. Each line in the truth table is one of the possible combinations for the expression. The expression is considered to be covered (gets a green check mark) if the entire truth table is covered.

When you hover over statements, conditions or branches in the Source window, the Hits and BC columns display the coverage numbers for that line of code. For example, in Figure 2-108, the blue highlighted line shows that the expression (b=b'b1) was hit 1 time. The value in the Hits column shows the total coverage for all items in the truth table (as shown in the Coverage Details window when you click the specific line in the hits column).

In the BC count column, only the "true" counts are given, with the exception of the AllFalse branch (if any). The AllFalse count is given next to the first "if" condition in an if-else tree that does not contain a terminating catch-all "else" branch. You can determine the branch false count by subtracting counts in the BC column from the Hits column.

Source Window Code Coverage Indicator Icons

Table 2-92. Source Window Code Coverage Indicators

Icon	Description/Indication
	All statements, branches (true), conditions, or expressions on a particular line have been executed
	Multiple kinds of coverage on the line were not executed
	True branch not executed (BC column)
	False branch not executed (BC column)
	Condition not executed (Hits column)
	Expression not executed (Hits column)
	Branch not executed (Hits column)
	Statement not executed (Hits column)
	Indicates a line of code to which active coverage exclusions have been applied. Every item on the line is excluded; none are hit.
	Some excluded items are hit.
	Some items are excluded, and all items not excluded are hit
	Some items are excluded, and some items not excluded have missing coverage
	Auto exclusions have been applied to this line. Hover the cursor over the E _A and a tool tip balloon appears with the reason for exclusion,

Coverage data presented in the Source window is either calculated “by file” or “by instance”, as indicated just after the source file name. If coverage numbers are mismatched between Code Coverage Analysis windows and the Source window, check to make sure that both are being calculated the same — either “by file” or “by instance”.

To display only numbers in Hits and BC columns, select **Tools > Code Coverage > Show Coverage Numbers**.

When the source window is active, you can skip to "missed lines" three ways:

- select **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** from the menu bar
- click the Previous zero hits and Next zero hits icons on the toolbar
- press Shift-Tab (previous miss) or Tab (next miss)

Controlling Data Displayed in a Source Window

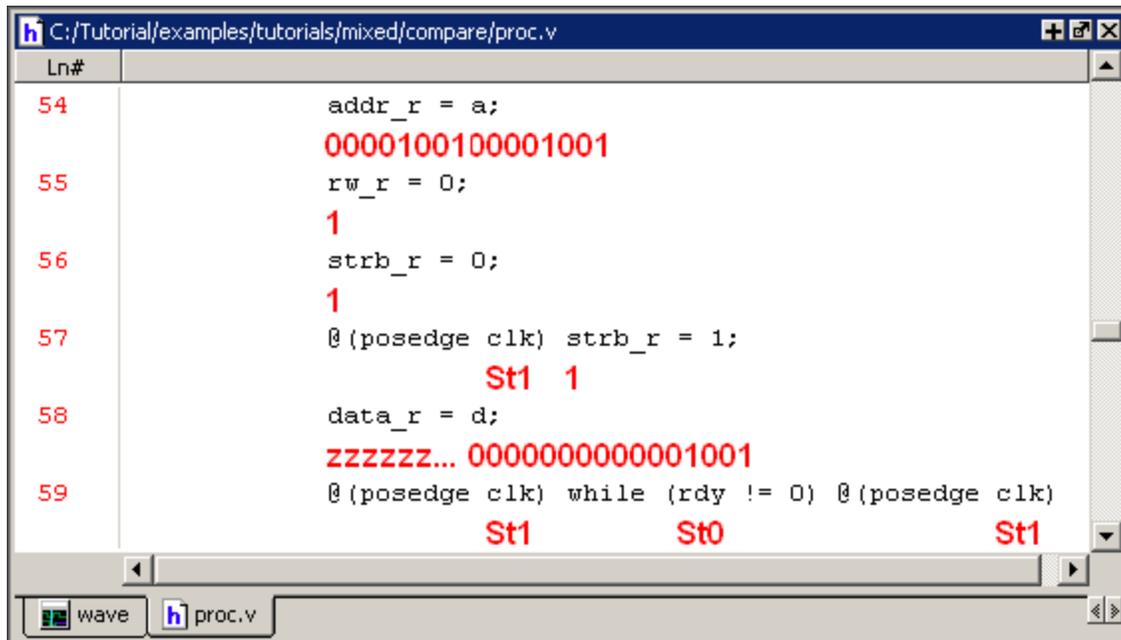
The **Tools > Code Coverage** menu contains several commands for controlling coverage data display in a Source window.

- **Hide/Show coverage data** toggles the *Hits* column off and on.
- **Hide/Show branch coverage** toggles the *BC* column off and on.
- **Hide/Show coverage numbers** displays the number of executions in the *Hits* and *BC* columns rather than check marks and Xs. When multiple statements occur on a single line an ellipsis ("...") replaces the Hits number. In such cases, hover the cursor over each statement to highlight it and display the number of executions for that statement.
- **Show coverage By Instance** displays only the number of executions for the currently selected instance in the Main window workspace.

Debugging with Source Annotation

With source annotation you can interactively debug your design by analyzing your source files in addition to using the Wave and Signal windows. Source annotation displays simulation values, including transitions, for each signal in your source file. [Figure 2-109](#) shows an example of source annotation, where the red values are added below the signals.

Figure 2-109. Source Annotation Example



Turn on source annotation by selecting **Source > Show Source Annotation** or by right-clicking a source file and selecting **Show Source Annotation**. Note that transitions are displayed only for those signals that you have logged.

To analyze the values at a given time of the simulation you can either:

- Show the signal values at the current simulation time. This is the default behavior. The window automatically updates the values as you perform a run or a single-step action.
- Show the signal values at current cursor position in the Wave window.

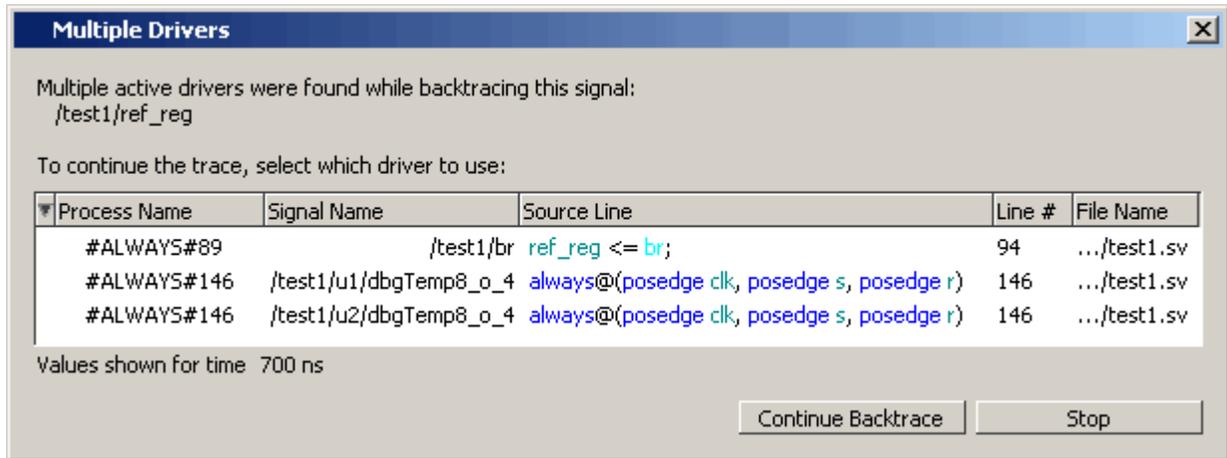
You can switch between these two settings by performing the following actions:

- When Docked:
 - **Source > Examine Now**
 - **Source > Examine Current Cursor**
- When Undocked:
 - **Tools > Options > Examine Now**
 - **Tools > Options > Examine Current Cursor**

You can highlight a specific signal in the Wave window by double-clicking on an annotation value in the source file.

If there is more than one driver for the signal, a Multiple Drivers dialog will open showing all driving processes (Figure 2-111).

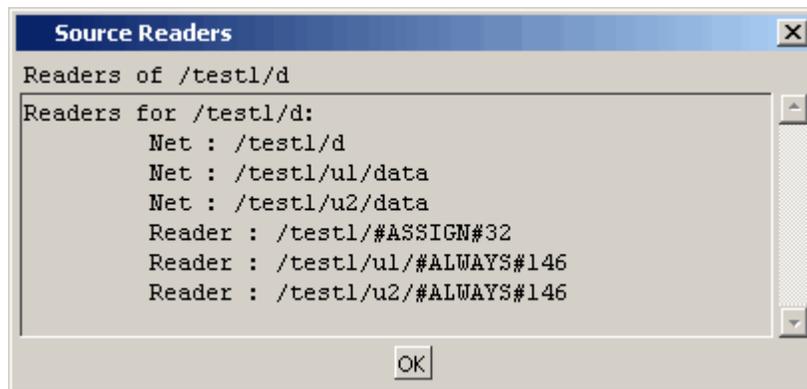
Figure 2-111. Window Shows all Driving Processes



Select any driver to open the code for that driver.

- The **Show Readers** selection opens the Source Readers window. If there is more than one reader for the signal, all will be displayed (Figure 2-112).

Figure 2-112. Source Readers Dialog Displays All Signal Readers



Limitations

The Source window's textual dataflow functions only work for pure HDL. It will not work for SystemC or for complex data types like SystemVerilog classes.

Using Language Templates

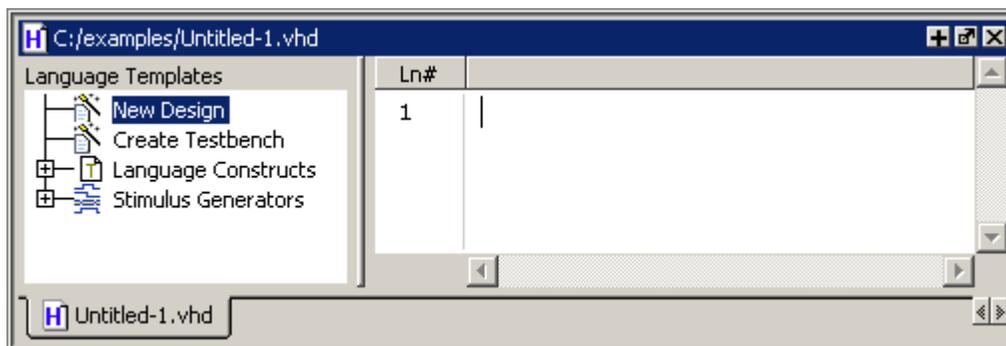
ModelSim language templates help you write code. They are a collection of wizards, menus, and dialogs that produce code for new designs, test benches, language constructs, logic blocks, and so forth.

Note

The language templates are not intended to replace thorough knowledge of coding. They are intended as an interactive reference for creating small sections of code. If you are unfamiliar with a particular language, you should attend a training class or consult one of the many available books.

To use the templates, either open an existing file, or select **File > New > Source** to create a new file. Once the file is open, select **Source > Show Language Templates** if the Source window is docked in the Main window; select **View > Show Language Templates** of the Source window is undocked. This displays a pane that shows the available templates.

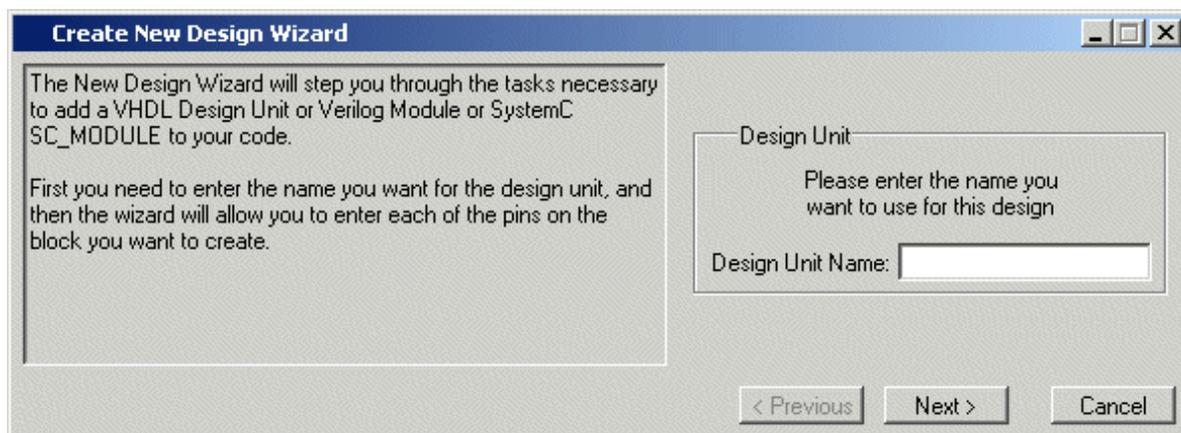
Figure 2-113. Language Templates



The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

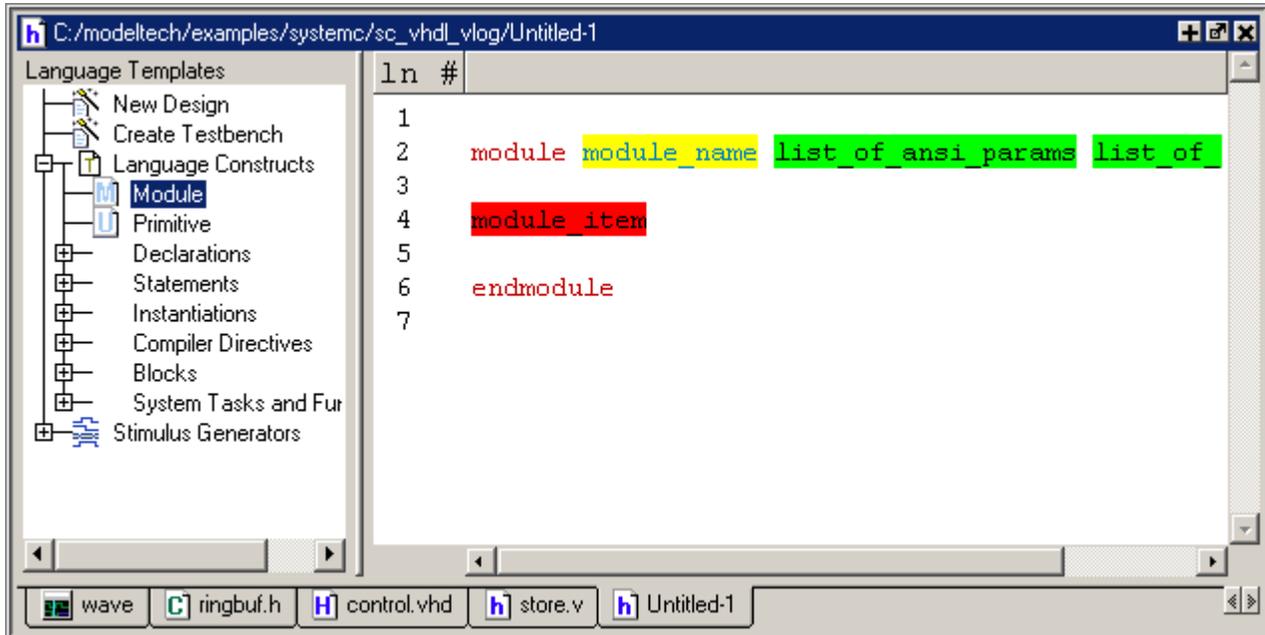
Double-click an object in the list to open a wizard or to begin creating code. Some of the objects bring up wizards while others insert code into your source file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.

Figure 2-114. Create New Design Wizard



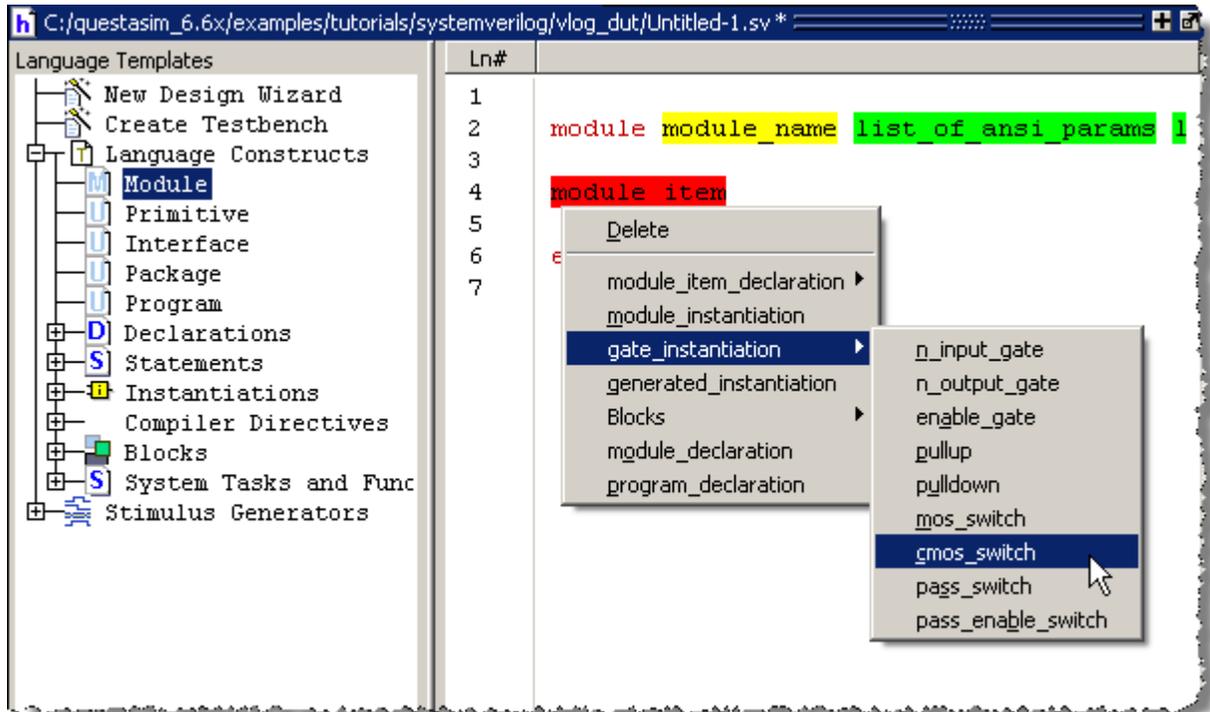
Code inserted into your source contains a variety of highlighted fields. The example below shows a module statement inserted from the Verilog template.

Figure 2-115. Inserting Module Statement from Verilog Language Template



Some of the fields, such as `module_name` in the example above, are to be replaced with names you type. Other fields can be expanded by double-clicking and still others offer a context menu of options when double-clicked. The example below shows the menu that appears when you double-click `module_item` then select `gate_instantiation`.

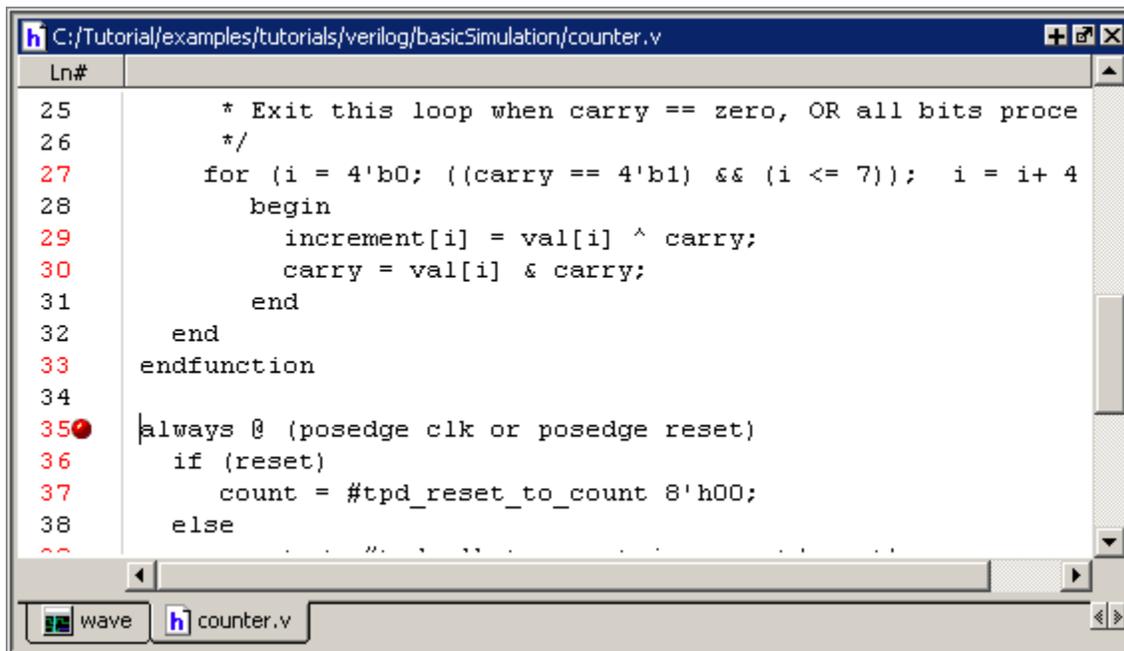
Figure 2-116. Language Template Context Menus



Setting File-Line Breakpoints with the GUI

You can easily set file-line breakpoints in your source code by clicking your mouse cursor in the line number column of a Source window. Click the left mouse button in the line number column next to a red line number and a red ball denoting a breakpoint will appear (Figure 2-117).

Figure 2-117. Breakpoint in the Source Window



The breakpoint markers are toggles. Click once to create the breakpoint; click again to disable or enable the breakpoint.

Note



When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. See [Preserving Object Visibility for Debugging Purposes](#) and [Design Object Visibility for Designs with PLI](#).

To delete the breakpoint completely, right click the red breakpoint marker, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable Breakpoint** — Deactivate the selected breakpoint.
- **Edit Breakpoint** — Open the File Breakpoint dialog to change breakpoint arguments.
- **Edit All Breakpoints** — Open the Modify Breakpoints dialog.
- **Run Until Here** — Run the simulation from the current simulation time up to the specified line of code. Refer to [Run Until Here](#) for more information.
- **Add/Remove Bookmark** — Add or remove a file-line bookmark.

Adding File-Line Breakpoints with the bp Command

Use the **bp** command to add a file-line breakpoint from the VSIM> prompt.

For example:

bp top.vhd 147

sets a breakpoint in the source file *top.vhd* at line 147.

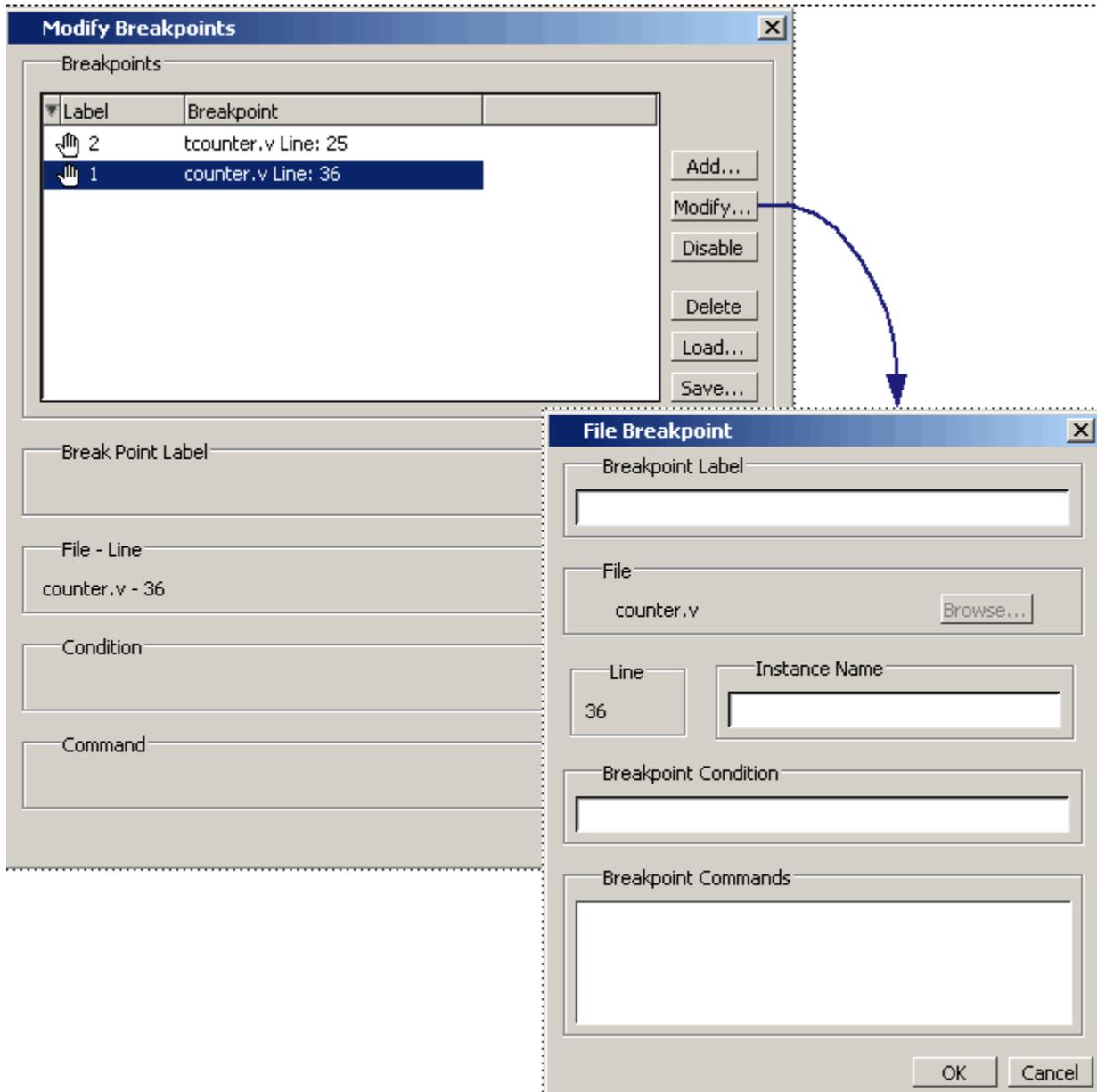
Editing File-Line Breakpoints

To modify (or add) a breakpoint according to the line number in a source file, do any one of the following:

- Select **Tools > Breakpoints** from the Main menu.
- Right-click a breakpoint and select **Edit All Breakpoints** from the popup menu.
- Click the **Edit Breakpoints** toolbar button. See [Simulate Toolbar](#).

This displays the Modify Breakpoints dialog box shown in [Figure 2-118](#).

Figure 2-118. Modifying Existing Breakpoints



The Modify Breakpoints dialog box provides a list of all breakpoints in the design. To modify a breakpoint, do the following:

1. Select a file-line breakpoint from the list.
2. Click Modify, which opens the File Breakpoint dialog box shown in [Figure 2-118](#).
3. Fill out any of the following fields to modify the selected breakpoint:
 - Breakpoint Label — Designates a label for the breakpoint.

- Instance Name — The full pathname to an instance that sets a SystemC breakpoint so it applies only to that specified instance.
- Breakpoint Condition — One or more conditions that determine whether the breakpoint is observed. If the condition is true, the simulation stops at the breakpoint. If false, the simulation bypasses the breakpoint. A condition cannot refer to a VHDL variable (only a signal). Refer to the tip below for more information on proper syntax for breakpoints entered in the GUI.
- Breakpoint Command — A string, enclosed in braces ({}) that specifies one or more commands to be executed at the breakpoint. Use a semicolon (;) to separate multiple commands.

i **Tip:** All fields in the File Breakpoint dialog box, except the Breakpoint Condition field, use the same syntax and format as the `-inst` switch and the command string of the `bp` command. Do not enclose the expression entered in the Breakpoint Condition field in quotation marks (“ ”). For more information on these command options, refer to the `bp` command in the *Questa SV/AFV Reference Manual*.

Click OK to close the File Breakpoints dialog box.

1. Click OK to close the Modify Breakpoints dialog box.

Loading and Saving Breakpoints

The Modify Breakpoints dialog (Figure 2-118) includes Load and Save buttons that allow you to load or save breakpoints.

Setting Conditional Breakpoints

In dynamic class-based code, an expression can be executed by more than one object or class instance during the simulation of a design. You set a conditional breakpoint on the line in the source file that defines the expression and specifies a condition of the expression or instance you want to examine. You can write conditional breakpoints to evaluate an absolute expression or a relative expression.

You can use the SystemVerilog keyword `this` when writing conditional breakpoints to refer to properties, parameters or methods of an instance. The value of `this` changes every time the expression is evaluated based on the properties of the current instance. Your context must be within a local method of the same class when specifying the keyword `this` in the condition for a breakpoint. Strings are not allowed.

The conditional breakpoint examples below refer to the following SystemVerilog source code file *source.sv*:

Figure 2-119. Source Code for *source.sv*

```
1 class Simple;
2   integer cnt;
3   integer id;
4   Simple next;
5
6   function new(int x);
7       id=x;
8       cnt=0
9       next=null
10  endfunction
11
12  task up;
13      cnt=cnt+1;
14      if (next) begin
15          next.up;
16      end
17  endtask
18 endclass
19
20 module test;
21   reg clk;
22   Simple a;
23   Simple b;
24
25   initial
26   begin
27       a = new(7);
28       b = new(5);
29   end
30
31   always @(posedge clk)
32   begin
33       a.up;
34       b.up;
35       a.up
36   end;
37 endmodule
```

Prerequisites

Compile and load your simulation.

Note



You must use the +acc switch when optimizing with vopt to preserve visibility of SystemVerilog class objects.

Setting a Breakpoint For a Specific Instance

Enter the following on the command line:

```
bp simple.sv 13 -cond {this.id==7}
```

Results

The simulation breaks at line 13 of the *simple.sv* source file (Figure 2-119) the first time module a hits the expression because the breakpoint is evaluating for an id of 7 (refer to line 27).

Setting a Breakpoint For a Specified Value of Any Instance.

Enter the following on the command line:

```
bp simple.sv 13 -cond {this.cnt==8}
```

Results

The simulation evaluates the expression at line 13 in the *simple.sv* source file (Figure 2-119), continuing the simulation run if the breakpoint evaluates to false. When an instance evaluates to true the simulation stops, the source is opened and highlights line 13 with a blue arrow. The first time `cnt=8` evaluates to true, the simulation breaks for an instance of module Simple b. When you resume the simulation, the expression evaluates to `cnt=8` again, but this time for an instance of module Simple a.

You can also set this breakpoint with the GUI:

1. Right-click on line 13 of the *simple.sv* source file.
2. Select Edit Breakpoint 13 from the drop menu.
3. Enter

```
this.cnt==8
```

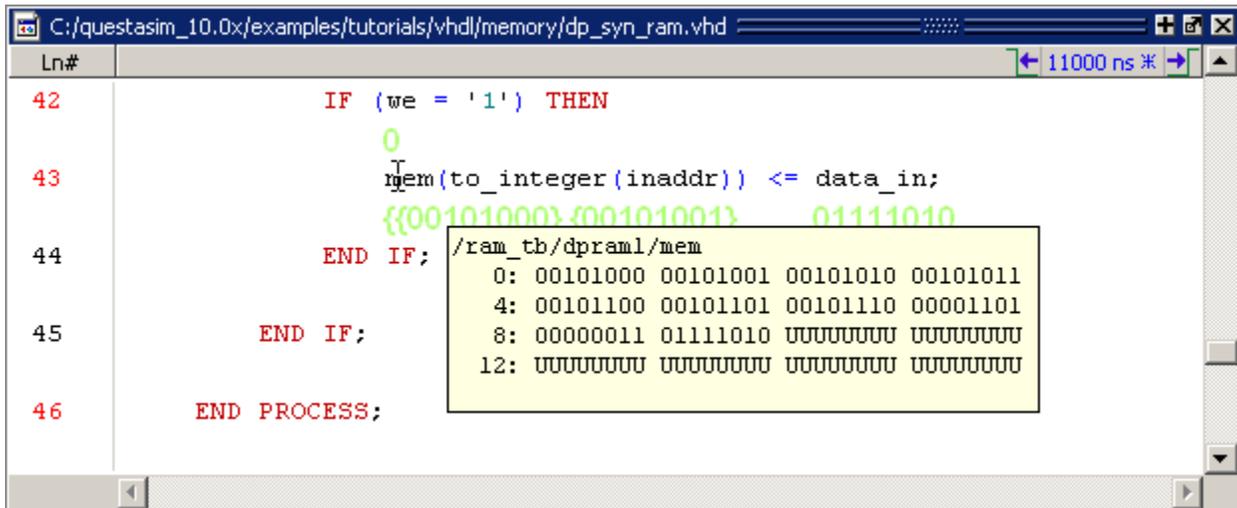
in the **Breakpoint Condition** field of the **Modify Breakpoint** dialog box. (Refer to Figure 2-118) Note that the file name and line number are automatically entered.

Checking Object Values and Descriptions

You can check the value or description of signals, indexes, macros, and other objects in the Source window. There are two quick methods to determine the value and description of an object:

- Select an object, then right-click and select **Examine** or **Describe** from the context menu.
- Pause the cursor over an object to see an examine pop-up

Figure 2-120. Source Window Description



You can select **Source > Examine Now** or **Source > Examine Current Cursor** to choose at what simulation time the object is examined or described.

You can also invoke the [examine](#) and/or [describe](#) commands on the command line or in a macro.

Marking Lines with Bookmarks

Source window bookmarks are blue flags that mark lines in a source file. These graphical icons may ease navigation through a large source file by highlighting certain lines.

As noted above in the discussion about finding text in the Source window, you can insert bookmarks on any line containing the text for which you are searching. The other method for inserting bookmarks is to right-click a line number and select **Add/Remove Bookmark**. To remove a bookmark, right-click the line number and select **Add/Remove Bookmark** again.

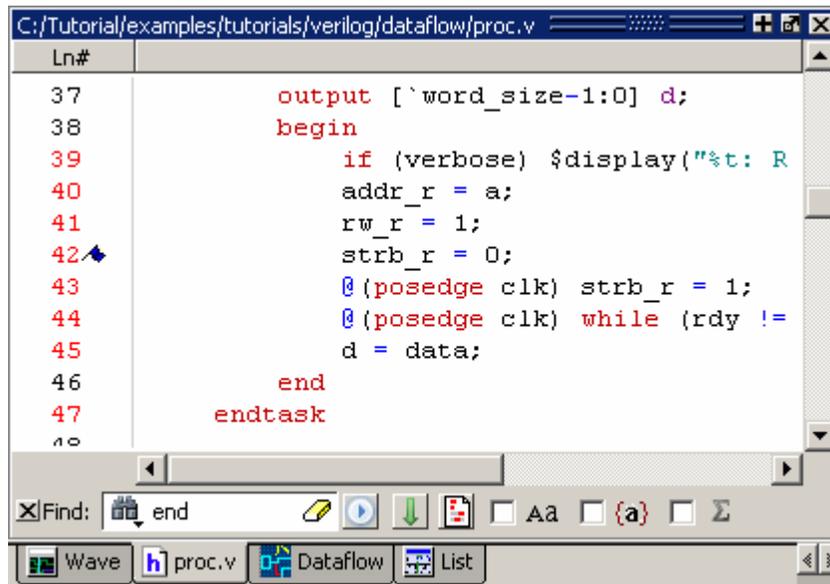
To remove all bookmarks from the Source window, select **Source > Clear Bookmarks** from the menu bar when the Source window is active.

Performing Incremental Search for Specific Code

The Source window includes a Find function that allows you to do an incremental search for specific code. To activate the Find bar ([Figure 2-121](#)) in the Source window select **Edit > Find** from the Main menus or click the **Find** icon in the Main toolbar. For more information see [Using the Find and Filter Functions](#).



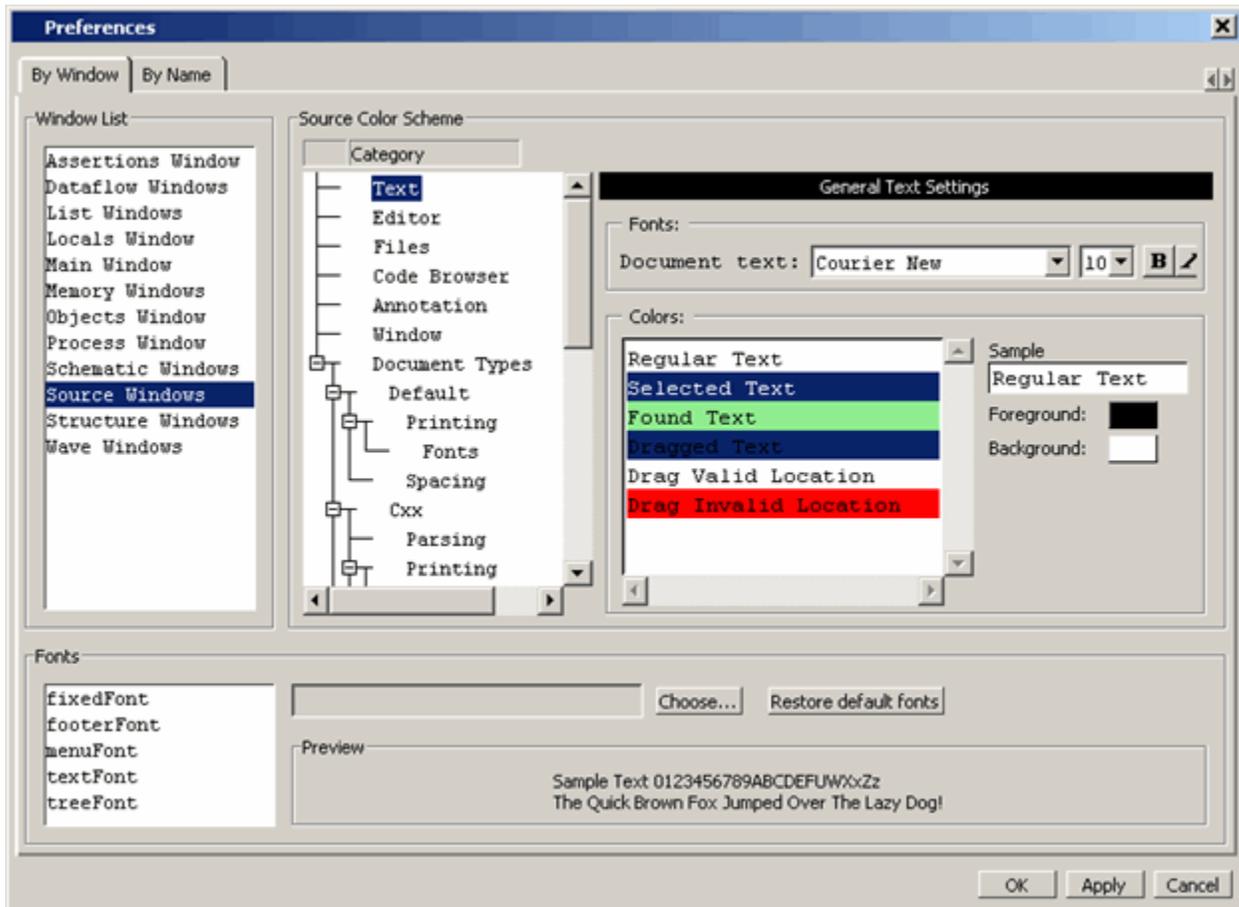
Figure 2-121. Source Window with Find Toolbar



Customizing the Source Window

You can customize a variety of settings for Source windows. For example, you can change fonts, spacing, colors, syntax highlighting, and so forth. To customize Source window settings, select **Tools > Edit Preferences**. This opens the Preferences dialog. Select **Source Windows** from the Window List.

Figure 2-122. Preferences Dialog for Customizing Source Window



Select an item from the Category list and then edit the available properties on the right. Click OK or Apply to accept the changes.

The changes will be active for the next Source window you open. The changes are saved automatically when you quit ModelSim. See [Setting Preference Variables from the GUI](#) for details.

Structure Window

Use this window to view the hierarchical structure of the active simulation.

The name of the structure window, as shown in the title bar or in the tab if grouped with other windows, can vary:

- `sim` — This is the name shown for the Structure window for the active simulation.
- `dataset_name` — The Structure window takes the name of any dataset you load through the **File > Datasets** menu item or the dataset open command.

Viewing the Structure Window

By default, the Structure window opens in a tab group with the Library windows after starting a simulation. You can also open the Structure window with the “[View Objects Window Button](#)”.

The hierarchical view includes an entry for each object within the design. When you select an object in a Structure window, it becomes the current region.

By default, the coverage statistics displayed in the columns within the Structure window are recursive. You can select to view coverage statistics for local instances by deselecting **Code Coverage > Enable Recursive Coverage Sums**. See “[Coverage Aggregation in the Structure Window](#)” for details on coverage numbers.

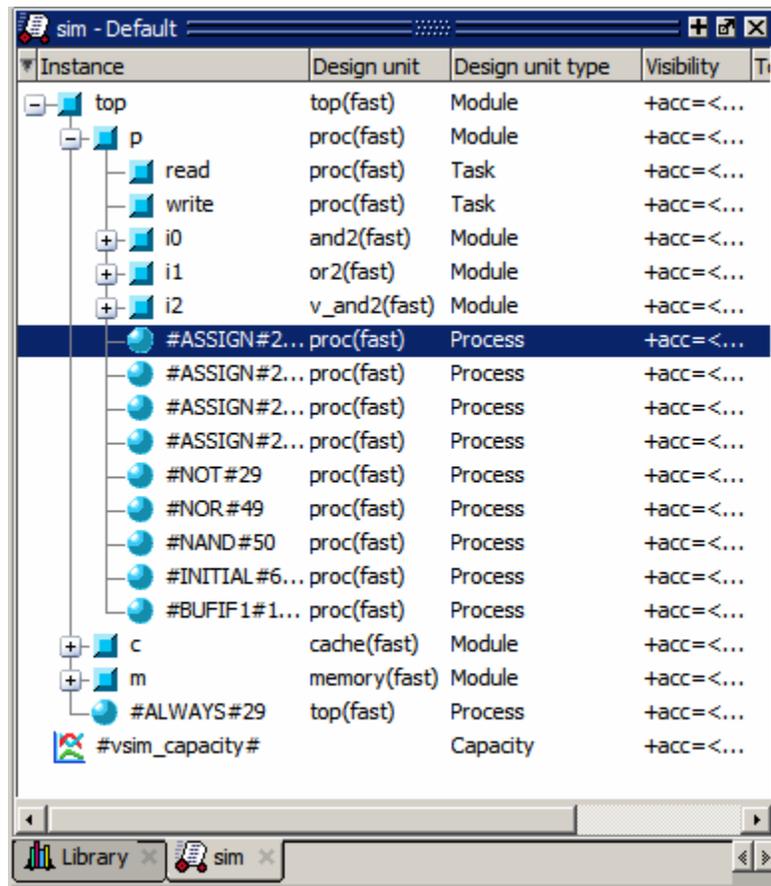
The contents of several windows automatically update based on which object you select, including the Source window, Objects window, Processes window, and Locals window.

Accessing

Access the window using any of the following:

- Menu item: **View > Structure**
- Command: view structure
- Button: [View Objects Window Button](#)

Figure 2-123. Structure Window



Structure Window Tasks

This section describes tasks for using the Structure window.

Display Source Code of a Structure Window Object

You can highlight the line of code that declares a given object in the following ways:

1. Double-click on an object — Opens the file in a new Source window, or activates the file if it is already open.
2. Single-click on an object — Highlights the code if the file is already showing in an active Source window.

Add Structure Window Objects to Other Windows

You can add objects from the Structure window to the Dataflow window, Schematic window, List window, Watch window or Wave window in the following ways:

- Mouse — Drag and drop

`/foo//bar` — search for regions containing "bar" recursively below all top level regions containing "foo".

To search for a name that contains the slash (/) character, escape the slash using a backslash (\). For example: `\bar`.

When you double-click any item in the match list that item is highlighted in the Structure window and the popup is removed. The search can be canceled by clicking on the 'x' button or by pressing the <ESC> key on your keyboard.

With 'Search While Typing' enabled (the default) each keypress that changes the pattern restarts the search immediately.

Filtering Structure Window Objects

You can control the types of information available in the Structure window through the View > Filter menu items.

- Processes — Implicit wire processes
- Functions — Verilog and VHDL Functions
- Packages — VHDL Packages
- Tasks — Verilog Tasks
- Statement — Verilog Statements
- SVClass— System Verilog class instances
- VIPackage — Verilog Packages
- VITypedef — Verilog Type Definitions
- Leaf Instances — Verilog cell instances or VHDL architecture instance.
- Capacity — Memory capacity design unit

GUI Elements of the Structure Window

This section describes GUI elements specific to this Window. For a complete list of all columns in the Structure window and a description of their contents, see [Table 2-93](#).

Column Descriptions

The table below lists columns in the Structure window with a description of their contents (Table 2-93).

Table 2-93. Columns in the Structure Window

Column name	Description
Design Unit	The name of the design unit
Design Unit Type	The type of design unit
Visibility	The +acc settings used for compilation/optimization of that design unit
Cover Options	The +cover settings used for compilation/simulation of that design unit
Total Coverage	The weighted average of all the coverage types (functional coverage and code coverage) is recursive. Deselect Code Coverage > Enable Recursive Coverage Sums to view results for the local instance. See “ Calculation of Total Coverage ” for coverage statistics details.
Covergroup %	the number of hits from the total number of covergroups, as a percentage
Cover hits	the number of cover directives whose count values are greater than or equal to the at_least value.
Cover misses	the number of cover directives whose count values are less than the at_least value
Cover %	the number of hits from the total number of cover directives, as a percentage
Cover graph	a bar chart displaying the Cover directive %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Assertion hits	Assertion hits shows different counts based on whether the -assertdebug is used: <ul style="list-style-type: none"> • with -assertdebug argument to vsim command: number of assertions whose pass count is greater than 0, and fail count is equal to 0. • without -assertdebug: number of assertions whose fail count is equal to 0.
Assertion misses	the number of assertions whose fail counts are greater than 0
Assertion %	the number of hits from the total number of assertions, as a percentage
Assertion graph	a bar chart displaying the Assertion %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Stmt count	the number of executable statements in each level and all levels under that level

Table 2-93. Columns in the Structure Window

Column name	Description
Stmts hit	the number of executable statements that were executed in each level and all levels under that level
Stmts missed	the number of executable statements that were not executed in each level and all levels under that level
Stmt %	the current ratio of Stmt hits to Stmt count
Stmt graph	a bar chart displaying the Stmt %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Branch count	Files window — the number of executable branches in each file Structure window — the number of executable branches in each level and all levels under that level
Branches hit	the number of executable branches that have been executed in the current simulation
Branches missed	the number of executable branches that were not executed in the current simulation
Branch %	the current ratio of Branch hits to Branch count
Branch graph	a bar chart displaying the Branch %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Condition rows	Files window — the number of conditions in each file Structure window — the number of conditions in each level and all levels under that level
Conditions hit	Files window — the number of times the conditions in a file have been executed Structure window — the number of times the conditions in a level, and all levels under that level, have been executed
Conditions missed	Files window — the number of conditions in a file that were not executed Structure window — the number of conditions in a level, and all levels under that level, that were not executed
Condition %	the current ratio of Condition hits to Condition rows
Condition graph	a bar chart displaying the Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Expression rows	the number of executable expressions in each level and all levels subsumed under that level

Table 2-93. Columns in the Structure Window

Column name	Description
Expressions hit	the number of times expressions in a level, and each level under that level, have been executed
Expressions missed	the number of executable expressions in a level, and all levels under that level, that were not executed
Expression %	the current ratio of Expression hits to Expression rows
Expression graph	a bar chart displaying the Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Toggle nodes	the number of points in each instance where the logic will transition from one state to another
Toggles hit	the number of nodes in each instance that have transitioned at least once
Toggles missed	the number of nodes in each instance that have not transitioned at least once
Toggle %	the current ratio of Toggle hits to Toggle nodes
Toggle graph	a bar chart displaying the Toggle %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
States	Files window — the number of states encountered in each file Structure window — the number of states encountered in each level and all levels subsumed under that level
States hit	Files window — the number of times the states were hit Structure window — the number of times states in a level, and each level under that level, have been hit
States missed	Files window — the number of states in a file that were not hit Structure window — the number of states in a level, and all levels under that level, that were not hit
State %	the current ratio of State hits to State rows
State graph	a bar chart displaying the State %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
Transitions	Files window — the number of transitions encountered in each file Structure window — the number of states encountered in each level and all levels subsumed under that level
Transitions hit	Files window — the number of times the transitions were hit Structure window — the number of times transitions in a level, and each level under that level, have been hit

Table 2-93. Columns in the Structure Window

Column name	Description
Transitions missed	Files window — the number of transitions in a file that were not hit Structure window — the number of transitions in a level, and all levels under that level, that were not hit
Transition %	the current ratio of Transition hits to Transition rows
Transition graph	a bar chart displaying the State %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
FEC Condition rows	Files window — the number of FEC conditions in each file Structure window — the number of conditions in each level and all levels under that level
FEC Conditions hit	Files window — the number of times the FEC conditions in a file have been executed Structure window — the number of times the conditions in a level, and all levels under that level, have been executed
FEC Conditions missed	Files window — the number of FEC conditions in a file that were not executed Structure window — the number of conditions in a level, and all levels under that level, that were not executed
FEC Condition %	the current ratio of FEC Condition hits to FEC Condition rows
FEC Condition graph	a bar chart displaying the FEC Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable
FEC Expression rows	Files window — the number of executable expressions in each file Structure window — the number of executable expressions in each level and all levels subsumed under that level
FEC Expressions hit	Files window — the number of times expressions in a file have been executed Structure window — the number of times expressions in a level, and each level under that level, have been executed
FEC Expressions missed	Files window — the number of executable expressions in a file that were not executed Structure window — the number of executable expressions in a level, and all levels under that level, that were not executed
FEC Expression %	the current ratio of FEC Expression hits to FEC Expression rows
FEC Expression graph	a bar chart displaying the FEC Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the PrefCoverage(cutoff) preference variable

Code Coverage in the Structure Window

The Structure window displays code coverage information in the Structure (sim) window for any datasets being simulated. When coverage is invoked, several columns for displaying coverage data are added to these windows. You can toggle columns on/off by right-clicking on a column name and selecting from the context menu that appears. [Figure 2-125](#) shows a portion of the Structure window with code coverage data displayed.

Figure 2-125. Code Coverage Data in the Structure Window

Stmt misses	Stmt %	Stmt graph	Branch count	Branch hits	Branch misses	Branch %	Branch gr
4	89.5%		34	30	4	88.2%	
1	83.3%		10	9	1	90%	

The screenshot also shows a toolbar with icons for Library, sim, Files, and Memories.

You can sort code coverage information for any column by clicking the column heading. Clicking the column heading again will reverse the order.

Coverage information in the Structure window is dynamically linked to the Code Coverage Analysis windows. Click the left mouse button on any file in the Files window to display that file's un-executed statements, branches, conditions, expressions, and toggles in the Code Coverage Analysis windows. Lines from the selected file that are excluded from coverage statistics are also displayed in the Code Coverage Analysis windows.

For details on how the Total Coverage column statistics are calculated, see [“Calculation of Total Coverage”](#).

Verification Management Browser Window

The Verification Management Browser window displays summary information for original test results in UCDBs, ranking files, and merged test results in a UCDB. It has a feature for customizing and saving the organization of the tabs. It also supports features for re-running tests, generating HTML reports from test results, and executing merges and test ranking.

For details on how the Total Coverage column statistics are calculated, see [“Calculation of Total Coverage”](#).

Accessing

Access the window using either of the following:

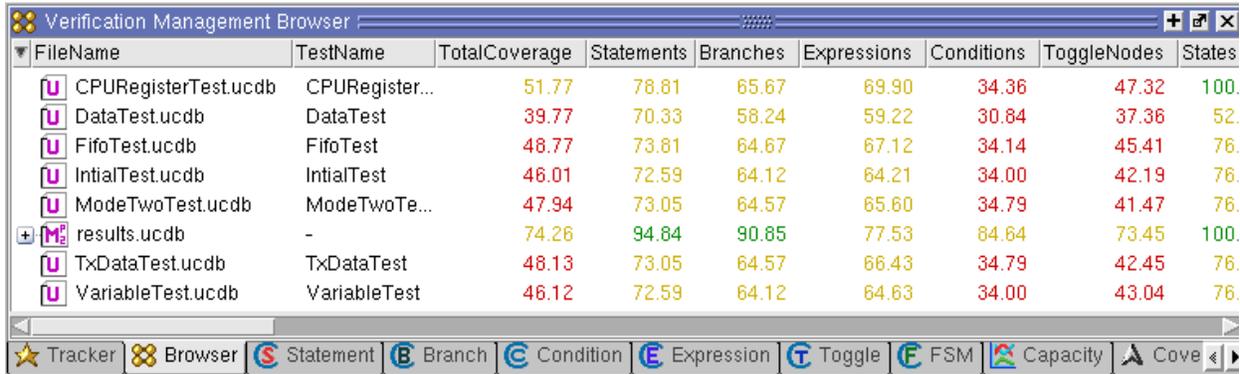
- Select **View > Verification Management > Browser**

- Execute the view command, as shown:

view testbrowser

Figure 2-126 shows the Verification Browser window using the Code Coverage column view setting, refer to [Controlling the Verification Browser Columns](#) for more information.

Figure 2-126. Browser Tab



Verification Browser Icons

The Browser uses the following icons to identify the type of file loaded into the browser:

Table 2-94. Verification Browser Icons

Browser Icon	Description
	Indicates the file is an unmerged UCDB file. A “P” notation in the upper right hand corner of the icon indicates that a Verification Plan is included in UCDB.
	Indicates the file is a rank file.
 	Indicates the file is a merged UCDB file. Notations on right hand side mean the following: <ul style="list-style-type: none"> • P - verification (test) plan is included in merged UCDB • 1 - Totals merge • 2 - Test-associated merge See “ Test-Associated Merge versus Totals Merge Algorithm ” for more information.

Controlling the Verification Browser Columns

You can customize the appearance of the Browser using either of the following methods:

- Use the “[Column Layout Toolbar](#)” to select from several pre-defined column arrangements.
- Right-click in the column headings to display a list of all column headings which allows you to toggle the columns on or off.

Saving Verification Browser Column and Filter Settings

Save your column layout and any filter settings to an external file (*browser_column_layout.do*) by selecting **File > Export > Column Layout** while the window is active. You can reload these settings with the do command. This export does not retain changes to column width.

GUI Elements of the Verification Browser Window

This section provides an overview of the GUI elements specific to this window.

Toolbar

The Browser allows access to the [Column Layout Toolbar](#) and the [Help Toolbar](#).

Menu Items

The following menu items are related to the Verification Management Browser window:

- **Add File** — adds UCDB (.ucdb) and ranking results (.rank) files to the browser. Refer to the section [Viewing Test Data in the GUI](#) for more information.
- **Remove File** — removes an entry from this window (**From Browser Only**), as well as from the file system (**Browser and File System**).
- **Remove Non-Contributing Test(s)** — operates only on ranked (.rank) files; menu selection is grayed out unless a ranked file is selected. Removes any tests that do not contribute toward the coverage.
- **Test Plan Import** — displays the XML Testplan Import Dialog Box, which allows you to import an XML test plan file into a UCDB which can then be merged with test results.
- **Merge** — displays the Merge Files Dialog Box, which allows you to merge any selected UCDB files. Refer to the section [Merging Coverage Test Data](#) for more information.
- **Rank** — displays the Rank Files Dialog Box, which allows you to create a ranking results file based on the selected UCDB files. Refer to the section [Ranking Coverage Test Data](#) for more information.
- **HTML Report** — displays the HTML Coverage Report Dialog Box, which allows you to view your coverage statistics in an HTML viewer.
- **Command Execution** — allows you to re-run simulations based on the resultant UCDB file based on the simulation settings to create the file. You can rerun any test whose test

record appears in an individual *.ucdb* file, a merged *.ucdb* file, or ranking results (*.rank*) file. See [Test Attribute Records in the UCDB](#) for more information on test records.

- Setup — Displays the Command Setup Dialog box, which allows you to create and edit your own setups which can be used to control the execution of commands. “Restore All Defaults” removes any changes you make to the list of setups and the associated commands.
- Execute on all — Executes the specified command(s) on all *.ucdb* files in the browser (through **TestReRun**), even those used in merged *.ucdb* files and *.rank* files.
- Execute on selected — Executes the specified command(s) on the selected *.ucdb* file(s) through **TestReRun**.
- **Filter** — either opens the Filter Setup Dialog Box, or applies desired filter setups.
 - Setup — opens the Filter Setup dialog that allows you to save and edit filters to apply to the data.
 - Create button — opens the Create Filter dialog which allows you to select filtering criteria, and select the tests for application of the specified filters. When you enter a Filter Name, and select “Add”, the Add/Modify Selection Criteria dialog box is displayed, where you can select the actual criteria to filter. See [“Filtering Results by User Attributes”](#) for an example.
 - Apply — applies the selected filter(s) on the data.
- **Generate Vrun Config** — generates Verification Run Management configuration file (*.rmdb*) including selected tests or all tests in the directory. Selecting either option brings up a dialog to enter the name to be used for the *.rmdb* file.
 - Save Selected Tests — Saves selected tests into a *.rmdb* file to be executed by vrun command.
 - Save All Tests — Saves all tests in the directory into a *.rmdb* file to be executed by vrun command.
- **Show Full Path** — toggles whether the FileName column shows only the filename or its full path.
- **Set Precision** — allows you to control the decimal point precision of the data in the Verification Browser window.
- **Configure Colorization** — opens the Colorization Threshold dialog box which allows you to off the colorization of coverage results displayed in the “Coverage” column, as well as set the low and high threshold coverage values for highlighting coverage values:
 - < low threshold — RED
 - > high threshold — GREEN

- > low and < high — YELLOW
- **Trend Analysis** — active only when a Trend UCDB is selected, or when multiple UCDBs are selected. Allows access to trending functionality.
 - HTML Report — opens the Coverage Trend Report (HTML) dialog box, where you set: the HTML report’s colorization threshold; the output directory path; and other HTML reporting options. See [vcover report -html](#) for detailed description of options.
 - XML Report — opens the Coverage Trend Report (XML) dialog box, where you set whether you are reporting on all (or specified) DUs, test plans, or instances; filtering; the coverage type; and the report pathname.
 - Text Report — opens the Coverage Trend Report (Text) dialog box, where you set whether you are reporting on all (or specified) DUs, test plans, or instances; filtering; the coverage type; and the report pathname.
 - Create Trend Database — opens the Create Trend Database dialog box, where you enter the name of the output trend database (the “Trend UCDB”) and input UCDBs.
 - Export — opens the Coverage Trend Report Export dialog box, where you set the options as shown for XML or HTML report, and export the report into a comma separated value (csv) or other format file.
 - View Trender — opens the Trender window in the Main window. In this window, right click on any coverage object to display a trend graph for that object.
- **Expand / Collapse Selected** — Expand or collapse selected UCDBs.
- **Expand / Collapse All** — Expand or collapse all UCDBs.
- **Save Format** — saves the current contents of the browser to a *.do* file.
- **Load** — loads a *.do* file that contains a previously saved browser layout.
- **Invoke CoverageView Mode** — opens the selected UCDB in viewcov mode, creating a new dataset. Refer to the section [Invoking Coverage View Mode](#) for more information.

Verification Results Analysis Window

The Verification Results Analysis window is used for viewing and sorting message and test data information. It requires a “qvman” license to view. For detailed information on this window, please see the Verification Management User’s Manual.

Verification Test Analysis Window

The Verification Test Analysis window is used to analyze various aspects of a particular test. It requires a “qvman” license to view. For detailed information on this window, please see the Verification Management User’s Manual.

Verification Tracker Window

The Verification Tracker window is used for test traceability analysis. This window has a feature for user-customizable column headings. It requires a “qvman” license to view. For detailed information on this window, please see the Verification Management User’s Manual.

Verification Trender Window

The Verification Trender window is used for analyzing coverage trends in verification. This window has a feature for user-customizable column headings. It requires a “qvman” license to view. For detailed information on this window, please see the Verification Management User’s Manual.

Transaction View Window

The Transaction View window displays information about a selected Questa Verification IP transaction instance. It is only available for viewing transactions for Questa Verification IPs. For detailed information on this window, please see “[Questa Verification IP Transaction Details in Transaction View Window](#)”.

Transcript Window

The Transcript window maintains a running history of commands that are invoked and messages that occur as you work with ModelSim. When a simulation is running, the Transcript displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface.

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see [Main and Source Window Mouse and Keyboard Shortcuts](#) for details).

Displaying the Transcript Window

The Transcript window is always open in the Main window and cannot be closed.

Viewing Data in the Transcript Window

The Transcript tab contains the command line interface, identified by the ModelSim prompt, and the simulation interface, identified by the VSIM prompt.

Saving the Transcript File

Variable settings determine the filename used for saving the transcript. If either **PrefMain(file)** in the *.modelsim* file or **TranscriptFile** in the *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the **TranscriptFile** variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, click in the Transcript window and then select **File > Save As**, or **File > Save**. The initial save must be made with the **Save As** selection, which stores the filename in the Tcl variable **PrefMain(saveFile)**. Subsequent saves can be made with the **Save** 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.

Refer to [Creating a Transcript File](#) for more information about creating, locating, and saving a transcript file.

Saving a Transcript File as a Macro (DO file)

1. Open a saved transcript file in a text editor.
2. Remove all commented lines leaving only the lines with commands.
3. Save the file as *<name>.do*.

Refer to the [do](#) command for information about executing a DO file.

Changing the Number of Lines Saved in the Transcript Window

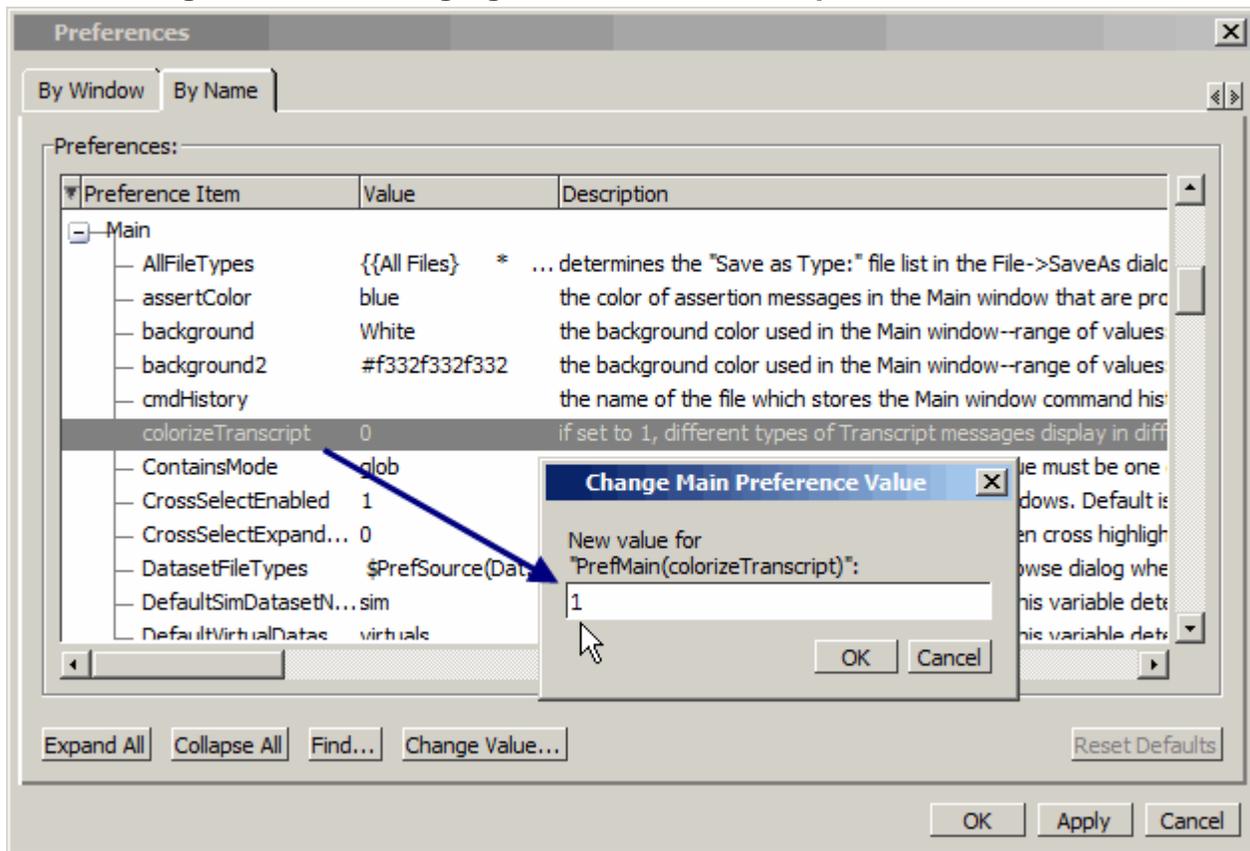
By default, the Transcript window retains the last 5000 lines of output from the transcript. You can change this default by selecting **Transcript > Saved Lines**. Setting this variable to 0 instructs the tool to retain all lines of the transcript.

Colorizing the Transcript

By default, all Transcript window messages are printed in blue. You may colorized Transcript messages according to severity as follows:

1. Select **Tools > Edit Preferences** from the Main window menus.
2. In the Preferences window select the **By Name** tab.
3. Expand the list of Preferences under "Main."
4. Select the `colorizeTranscript` preference and click the **Change Value** button.
5. Enter "1" in the Change Main Preference Value dialog and click **OK** ([Figure 2-127](#)).

Figure 2-127. Changing the colorizeTranscript Preference Value



Disabling Creation of the Transcript File

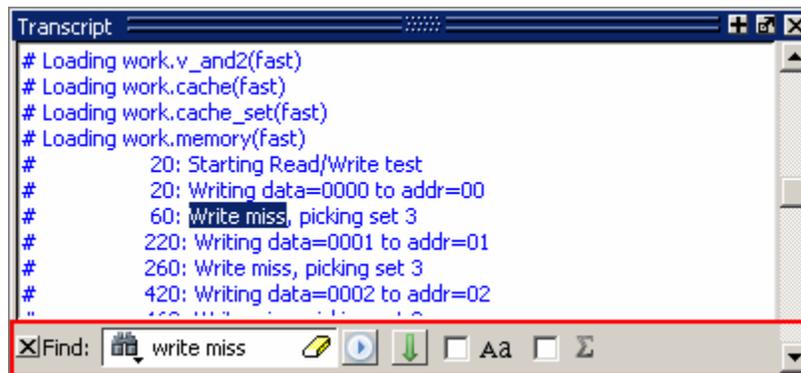
You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

Performing an Incremental Search

The Transcript tab includes an Find function (Figure 2-128) that allows you to do an incremental search for specific text. To activate the Find bar select **Edit > Find** from the menus or click the **Find** icon in the toolbar. For more information see [Using the Find and Filter Functions](#).

Figure 2-128. Transcript Window with Find Toolbar



Using Automatic Command Help

When you start typing a command at the prompt, a dropdown box appears which lists the available commands matching what has been typed so far. You may use the Up and Down arrow keys or the mouse to select the desired command. When a unique command has been entered, the command usage is presented in the drop down box.

You can toggle this feature on and off by selecting **Help > Command Completion**.

Using Transcript Menu Items

When the Transcript window is active, a "Transcript" menu selection appears in the Main window menu bar. The following items may be selected when you open the Transcript menu:

- Adjust Font Scaling — Displays the Adjust Scaling dialog box, which allows you to adjust how fonts appear for your display environment. Directions are available in the dialog box.
- Transcript File — Allows you to change the default name used when saving the transcript file. The saved transcript file will contain all the text in the current transcript file.
- Command History — Allows you to change the default name used when saving command history information. This file is saved at the same time as the transcript file.
- Save File — Allows you to change the default name used when selecting **File > Save As**.
- Saved Lines — Allows you to change how many lines of text are saved in the transcript window. Setting this value to zero (0) saves all lines.
- Line Prefix — Allows you to change the character(s) that precedes the lines in the transcript.

- Update Rate — Allows you to change the length of time (in ms) between transcript refreshes.
- ModelSim Prompt — Allows you to change the string used for the command line prompt.
- VSIM Prompt — Allows you to change the string used for the simulation prompt.
- Paused Prompt — Allows you to change the string used for when the simulation is paused.

Transcript Toolbar Items

When undocked, the Transcript window allows access to the following toolbars:

- [Standard Toolbar](#)
- [Help Toolbar](#)
- [Help Toolbar](#)

Watch Window

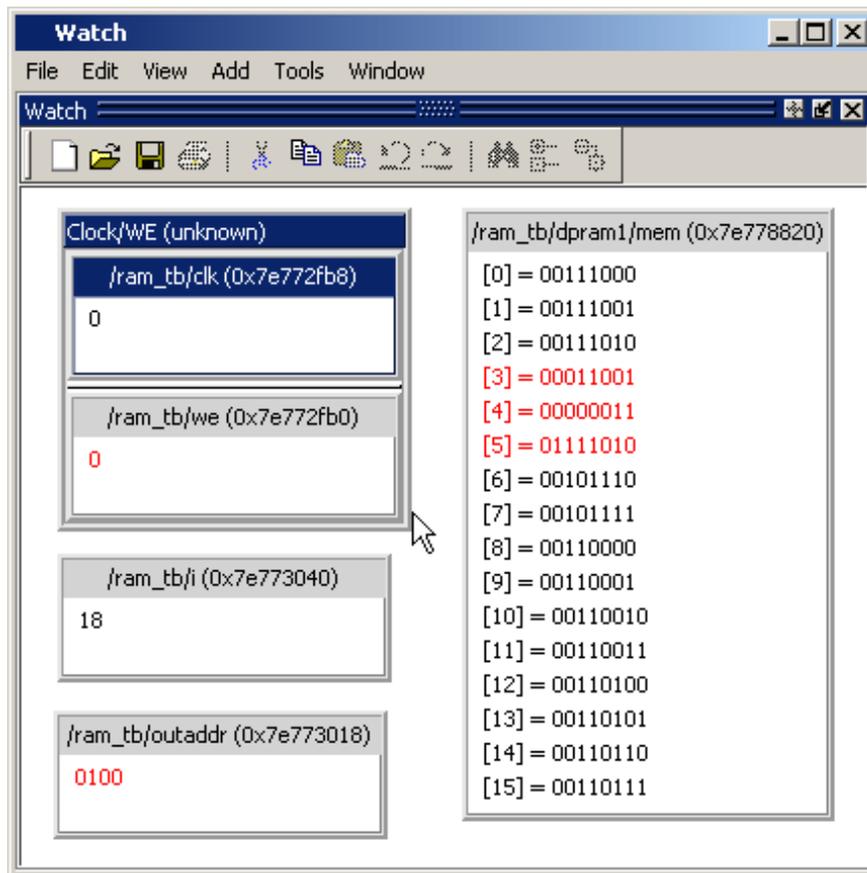
The Watch window shows values for signals and variables at the current simulation time, allows you to explore the hierarchy of object oriented designs. Unlike the Objects or Locals windows, the Watch window allows you to view any signal or variable in the design regardless of the current context. You can view the following objects:

- VHDL objects — signals, aliases, generics, constants, and variables
- Verilog objects — nets, registers, variables, named events, and module parameters
- SystemC objects — primitive channels and ports
- Virtual objects — virtual signals and virtual functions

The address of an object, if one can be obtained, is displayed in the title in parentheses as shown in [Figure 2-129](#).

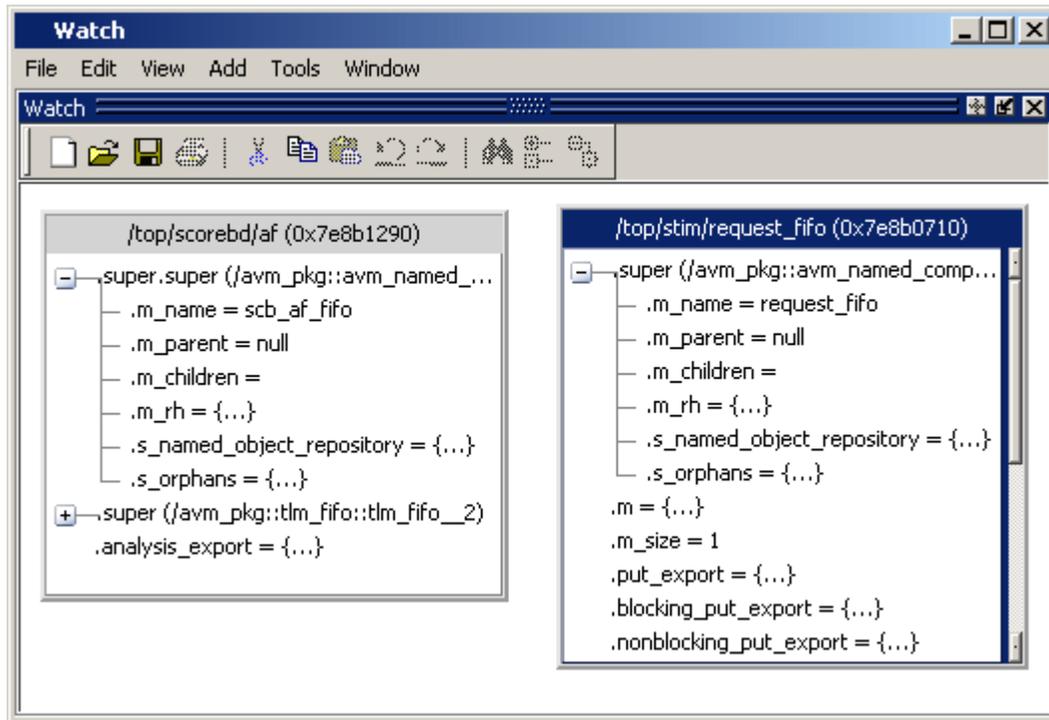
Items displayed in red are values that have changed during the previous Run command. You can change the radix of displayed values by selecting an item, right-clicking to open a popup context menu, then selecting **Properties**.

Figure 2-129. Watch Window



Items are displayed in a scrollable, hierarchical list, such as in [Figure 2-130](#) where extended SystemVerilog classes hierarchically display their super members.

Figure 2-130. Scrollable Hierarchical Display



Two Ref handles that refer to the same object will point to the same Watch window box, even if the name used to reach the object is different. This means circular references will be drawn as circular.

Selecting a line item in the window adds the item's full name to the global selection. This allows you to paste the full name in the Transcript (by simply clicking the middle mouse button) or other external application that accepts text from the global selection.

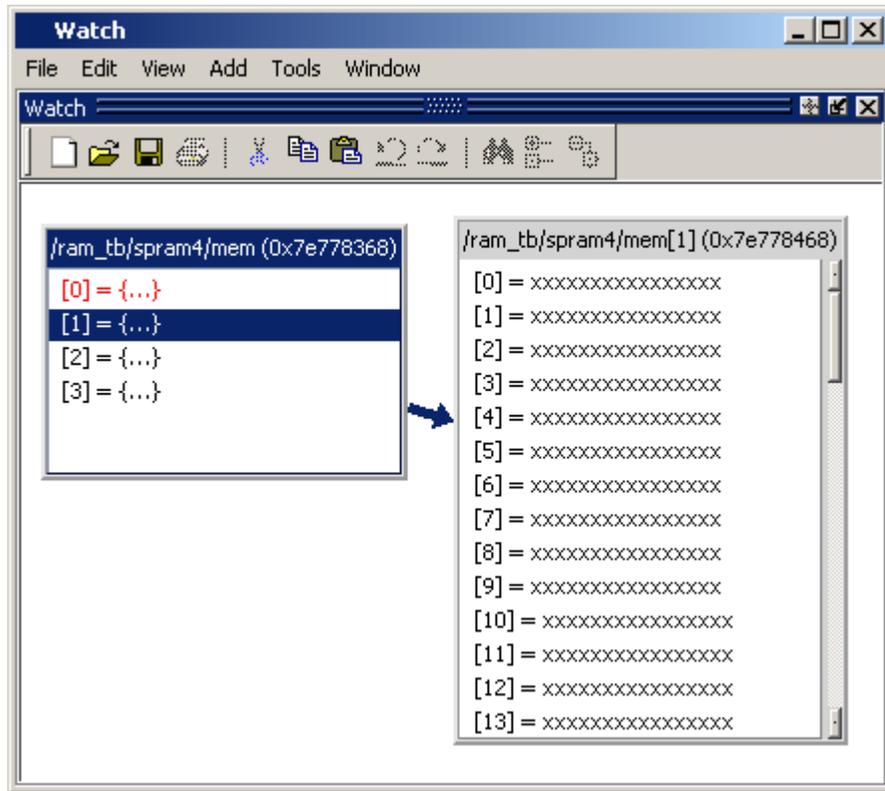
Adding Objects to the Watch Window

To add objects to the Watch window, drag -and-drop objects from the Structure window or from any of the following windows: List, Locals, Objects, Source, and Wave. You can also use the "Add Selected to Window Button". You can also use the `add watch` command.

Expanding Objects to Show Individual Bits

If you add an array or record to the window, you can view individual bit values by double-clicking the array or record. As shown in Figure 2-131, `/ram_tb/spram4/mem` has been expanded to show all the individual bit values. Notice the arrow that "ties" the array to the individual bit display.

Figure 2-131. Expanded Array

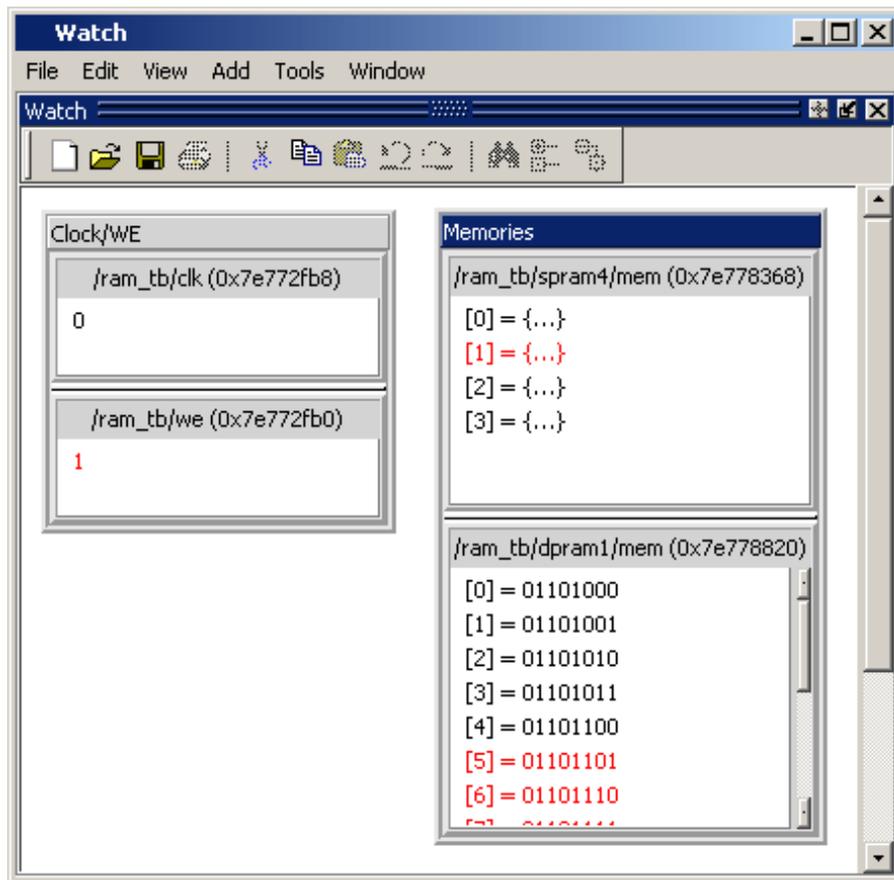


Grouping and Ungrouping Objects

You can group objects in the window so they display and move together. Select the objects, then right click one of the objects and choose **Group**.

In [Figure 2-132](#), two different sets of objects have been grouped together.

Figure 2-132. Grouping Objects in the Watch Window



To ungroup them, right-click the group and select **Ungroup**.

Saving and Reloading Format Files

You can save a format file (a DO file, actually) that will redraw the contents of the window. Right-click anywhere in the window and select **Save Format**. The default name of the format file is *watch.do*.

Once you have saved the file, you can reload it by right-clicking and selecting **Load Format**.

Wave Window

The Wave window, like the List window, allows you to view the results of your simulation. In the Wave window, however, you can see the results as waveforms and their values.

Wave Window Panes

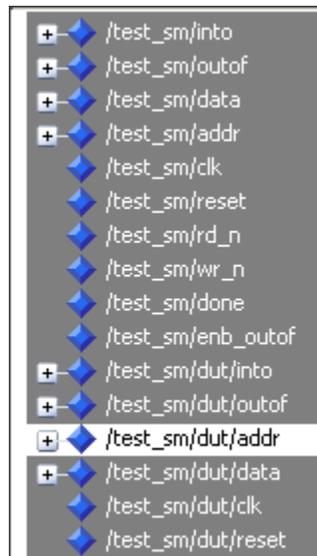
The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.

Pathname Pane

The pathname pane displays signal pathnames. Signals can be displayed with full pathnames, as shown here, or with any number of path elements. 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](#)).

Figure 2-134. Pathnames Pane



Values Pane

The 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 for all signals can be set by selecting **Simulate > Runtime Options**.

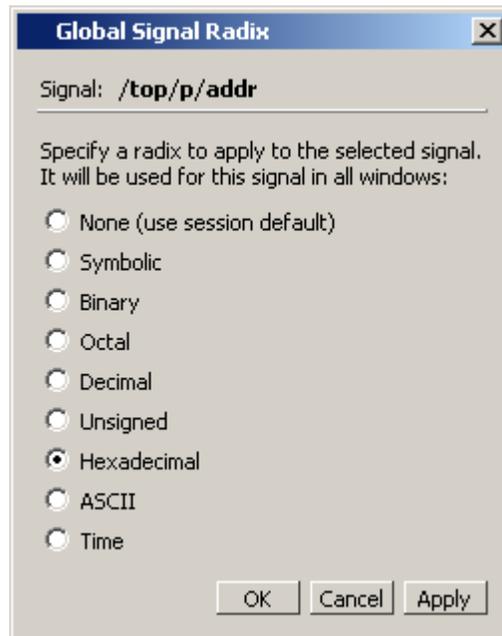
Note



When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

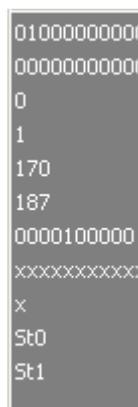
To change the radix for just the selected signal or signals, select **Wave > Format > Radix > Global Signal Radix** from the menus, or right-click the selected signal(s) and select **Radix > Global Signal Radix** from the popup menu. This opens the Global Signal Radix dialog (Figure 2-135), where you may select a radix. This sets the radix for the selected signal(s) in the Wave window and every other window where the signal appears.

Figure 2-135. Setting the Global Signal Radix from the Wave Window



The data in this pane is similar to that shown in the [Objects Window](#), except that the values change dynamically whenever a cursor in the waveform pane is moved.

Figure 2-136. Values Pane



Waveform Pane

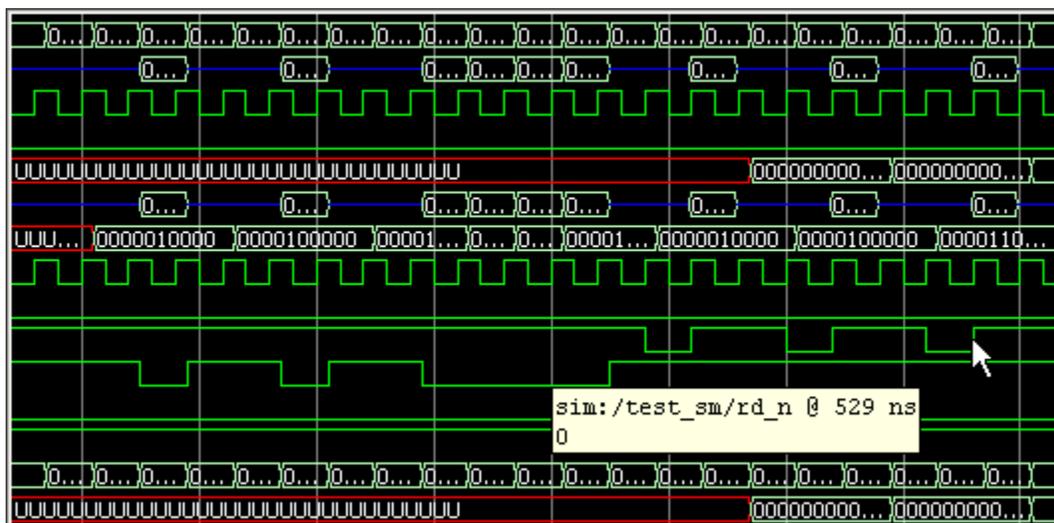
Figure 2-137 shows waveform pane, which displays waveforms that correspond to the displayed signal pathnames. It can also display as many as 20 user-defined cursors. Signal

values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. You can set the format of each signal individually by right-clicking the signal in the pathname or values panes and choosing **Format** from the popup menu. The default format is Logic.

If you place your mouse pointer on a signal in the waveform pane, a popup menu displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog box.

Dashed signal lines in the waveform pane indicate weak or ambiguous strengths of Verilog states. See [Verilog States](#) in the [Mixed-Language Simulation](#) chapter.

Figure 2-137. Waveform Pane



Analog Sidebar Toolbox

When the waveform pane contains an analog waveform, you can hover your mouse pointer over the left edge of the waveform to display the Analog Sidebar toolbox (see [Figure 2-138](#)). This toolbox shows a group of icons that gives you quick access to actions you can perform on the waveform display, as described in [Table 2-95](#).

Figure 2-138. Analog Sidebar Toolbox



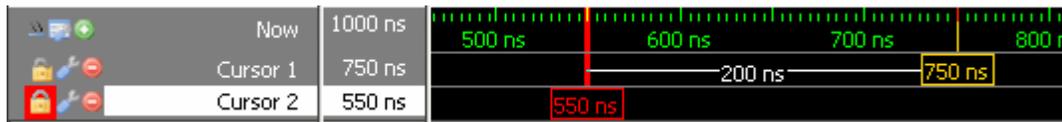
Table 2-95. Analog Sidebar Icons

Icon	Action	Description
	Open Wave Properties	Opens the Format tab of the Wave Properties dialog box, with the Analog format already selected. This dialog box duplicates the Wave Analog dialog box displayed by choosing Format > Format... > Analog (custom) from the main menu.
	Toggle Row Height	Changes the height of the row that contains the analog waveform. Toggles the height between the Min and Max values (in pixels) you specified in the Open Wave Properties dialog box under Analog Display.
	Rescale to fit Y data	Changes the waveform height so that it fits top-to-bottom within the current height of the row.
	Show menu of other actions	Displays <ul style="list-style-type: none"> • View Min Y • View Max Y • Overlay Above • Overlay Below • Colorize All • Colorize Selected
	Drag to resize waveform height	Creates an up/down dragging arrow that you can use to temporarily change the height of the row containing the analog waveform.

Cursor Pane

Figure 2-139 shows the Cursor Pane, which displays cursor names, cursor values and the cursor locations on the timeline. You can link cursors so that they move across the timeline together. See [Linking Cursors](#) in the [Waveform Analysis](#) chapter.

Figure 2-139. Cursor Pane

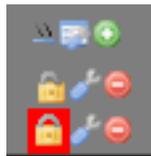


On the left side of this pane is a group of icons called the Cursor and Timeline Toolbox (see [Figure 2-140](#)). This toolbox gives you quick access to cursor and timeline features and configurations. See [Measuring Time with Cursors in the Wave Window](#) for more information.

Cursor and Timeline Toolbox

The Cursor and Timeline Toolbox displays several icons that give you quick access to cursor and timeline features.

Figure 2-140. Toolbox for Cursors and Timeline



The action for each toolbox icon is shown in [Table 2-96](#).

Table 2-96. Icons and Actions

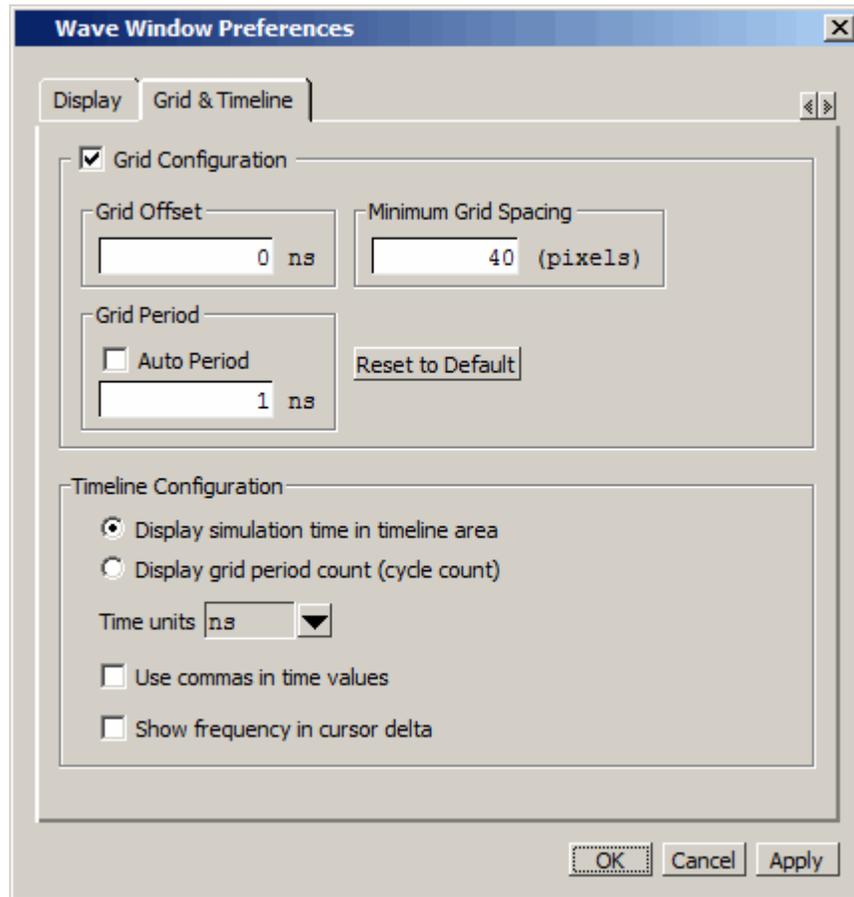
Icon	Action
	Toggle leaf name <-> full names
	Edit grid and timeline properties
	Insert cursor
	Toggle lock on cursor to prevent it from moving
	Edit this cursor
	Remove this cursor

The **Toggle leaf names <-> full names** icon allows you to switch from displaying full pathnames (the default) in the Pathnames Pane to displaying leaf or short names. You can also

control the number of path elements in the Wave Window Preferences dialog. Refer to [Hiding/Showing Path Hierarchy](#).

The **Edit grid and timeline properties** icon opens the Wave Window Properties dialog to the Grid & Timeline tab ([Figure 2-141](#)).

Figure 2-141. Editing Grid and Timeline Properties



The Grid Configuration selections allow you to set grid offset, minimum grid spacing, and grid period; or you can reset the grid configuration to default values.

The Timeline Configuration selections give you a user-definable time scale. You can display simulation time on the timeline or a clock cycle count. The time value is scaled appropriately for the selected unit.

By default, the timeline will display time delta between any two adjacent cursors. By clicking the **Show frequency in cursor delta** box, you can display the cursor delta as a frequency instead.

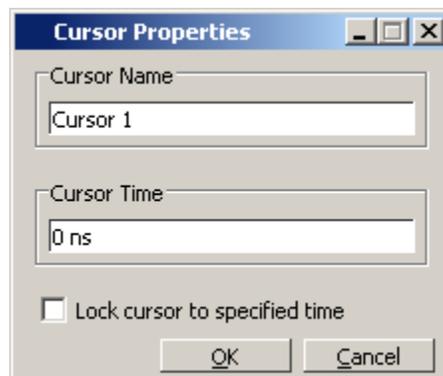
Adding Cursors to the Wave Window

You can add cursors when the Wave window is active by:

- clicking the Insert Cursor icon.
- choosing **Add > Wave > Cursor** from the menu bar
- pressing the “A” key while the mouse cursor is in the cursor pane.
- right clicking in the cursor pane at the time you want place a cursor, then selecting **New Cursor**.

Each added cursor is given a default cursor name (Cursor 2, Cursor 3, and so forth.) which you can be change by right-clicking the cursor name, then typing in a new name, or by clicking the **Edit this cursor** icon. The Edit this cursor icon opens the Cursor Properties dialog box (Figure 2-142), where you assign a cursor name and time. You can also lock the cursor to the specified time.

Figure 2-142. Cursor Properties Dialog



Messages Bar

The messages bar, located at the top of the Wave window, contains indicators pointing to the times at which a message was output from the simulator.

Figure 2-143. Wave Window - Message Bar



The message indicators (the down-pointing arrows) are color-coded as follows:

- Red — Indicates an assertion failure or error.
- Yellow — Indicates a warning.
- Green — Indicates a note.
- Grey — Indicates any other type of message.

You can use the Message bar in the following ways.

- Move the cursor to the next message — You can do this in two ways:
 - Click on the word “Messages” in the message bar to cycle the cursor to the next message after the current cursor location.
 - Click anywhere in the message bar, then use Tab or Shift+Tab to cycle the cursor between error messages either forward or backward, respectively.
- Display the [Message Viewer Window](#) — Double-click anywhere amongst the message indicators.
- Display, in the Message Viewer window, the message entry related to a specific indicator — Double-click on any message indicator.

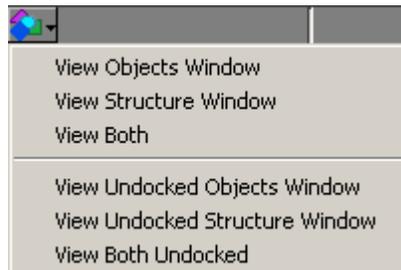
This function only works if you are using the Message Viewer in flat mode. To display your messages in flat mode:

- a. Right-click in the Message Viewer and select Display Options
- b. In the Message Viewer Display Options dialog box, deselect Display with Hierarchy.

View Objects Window Button

This button opens the Objects window with a single click. However, if you click-and-hold the button you can access additional options via a dropdown menu, as shown in [Figure 2-144](#)

Figure 2-144. View Objects Window Dropdown Menu



Assertions Debug Pane

The Assert Debug pane displays details on PSL and SVA assertion failures. See [Analyzing Assertion Failures in the Assertion Debug Pane of the Wave Window](#) for more information.

Objects You Can View in the Wave Window

The following types of objects can be viewed in the Wave window

- VHDL objects (indicated by a dark blue diamond) — signals, aliases, process variables, and shared variables
- Verilog objects (indicated by a light blue diamond) — nets, registers, variables, and named events

The GUI displays inout variables of a clocking block separately, where the output of the inout variable is appended with “_o”, for example you would see following two objects:

```
clock1.cl          /input portion of the inout cl
clock1.cl__o      /output portion of the inout cl
```

This display technique also applies to the Objects window

- Verilog and SystemVerilog transactions (indicated by a blue four point star) — see for more information
- SystemC objects
(indicated by a green diamond) — primitive channels and ports
(indicated by a green four point star) — transaction streams and their element
- Virtual objects (indicated by an orange diamond) — virtual signals, buses, and functions, see; [Virtual Objects](#) for more information
- Comparison objects (indicated by a yellow triangle) — comparison region and comparison signals; see [Waveform Compare](#) for more information
- SystemVerilog and PSL assertions (indicated by a light-blue (SVA) or magenta (PSL) triangle) — see [Viewing Assertions and Cover Directives in the Wave Window](#)
- SystemVerilog and PSL cover directives (indicated by a light-blue (SVA) or magenta (PSL) chevron) — see [Viewing Assertions and Cover Directives in the Wave Window](#)
- Questa Verification IP objects (see [Questa Verification IP Objects in the GUI](#)) — see [Questa Verification IP Transaction Viewing in the GUI](#) for more information
- Created waveforms (indicated by a red dot on a diamond) — see [Generating Stimulus with Waveform Editor](#)

The data in the object values pane is very similar to the Objects window, except that the values change dynamically whenever a cursor in the waveform pane is moved.

At the bottom of the waveform pane you can see a time line, tick marks, and the time value of each cursor's position. As you click and drag to move a cursor, the time value at the cursor location is updated at the bottom of the cursor.

You can resize the window panes by clicking on the bar between them and dragging the bar to a new location.

Waveform and signal-name formatting are easily changed via the Format menu. You can reuse any formatting changes you make by saving a Wave window format file (see [Saving the Window Format](#)).

Wave Window Toolbar

The Wave window (in the undocked Wave window) gives you quick access to the following toolbars:

- [Standard Toolbar](#)
- [Compile Toolbar](#)
- [Simulate Toolbar](#)
- [Wave Cursor Toolbar](#)
- [Wave Edit Toolbar](#)
- [Wave Toolbar](#)
- [Wave Compare Toolbar](#)
- [Zoom Toolbar](#)
- [Wave Expand Time Toolbar](#)

Chapter 3

Protecting Your Source Code

As today's IC designs increase in complexity, silicon manufacturers are leveraging third-party intellectual property (IP) to maintain or shorten design cycle times. This third-party IP is often sourced from several IP authors, each of whom may require different levels of protection in EDA tool flows. The number of protection/encryption schemes developed by IP authors has complicated the use of protected IP in design flows made up of tools from different EDA providers.

ModelSim's encryption solution allows IP authors to deliver encrypted IP code for a wide range of EDA tools and design flows. You can, for example, make module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

ModelSim supports VHDL, Verilog, and SystemVerilog IP code encryption by means of protected encryption envelopes. VHDL encryption is defined by the IEEE Std 1076-2008, section 24.1 (titled "Protect tool directives") and Annex H, section H.3 (titled "Digital envelopes"). Verilog/SystemVerilog encryption is defined by the IEEE Std 1364-2005, section 28 (titled "Protected envelopes") and Annex H, section H.3 (titled "Digital envelopes"). The protected envelopes usage model, as presented in Annex H section H.3 of both standards, is the recommended methodology for users of VHDL's ``protect` and Verilog's ``pragma protect` compiler directives. We recommend that you obtain these specifications for reference.

In addition, Questa supports the recommendations from the IEEE P1735 working group for encryption interoperability between different encryption and decryption tools. The current recommendations are denoted as "version 1" by P1735. They address use model, algorithm choices, conventions, and minor corrections to the HDL standards to achieve useful interoperability.

ModelSim also supports encryption using the `vcom/vlog -nodebug` command.

Creating Encryption Envelopes

Encryption envelopes define a region of code to be protected with [Protection Expressions](#). The protection expressions (``protect` for VHDL and ``pragma protect` for Verilog/SystemVerilog) specify the encryption algorithm used to protect the source code, the encryption key owner, the key name, and envelope attributes.

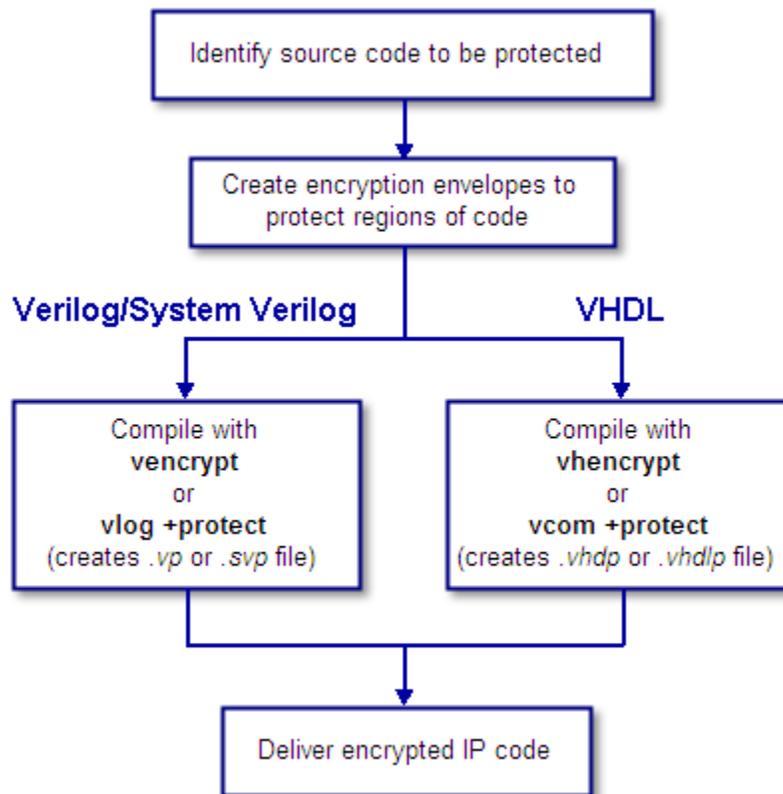
Creating encryption envelopes requires that you:

- identify the region(s) of code to be encrypted,
- enclose the code to be encrypted within protection directives, and

- compile your code with ModelSim encryption utilities - `vencrypt` for Verilog/SystemVerilog or `vhencrypt` for VHDL - or with the `vcom/vlog +protect` command.

The flow diagram for creating encryption envelopes is shown in [Figure 3-1](#).

Figure 3-1. Create an Encryption Envelope



Symmetric and asymmetric keys can be combined in encryption envelopes to provide the safety of asymmetric keys with the efficiency of symmetric keys (see [Encryption and Encoding Methods](#)). Encryption envelopes can also be used by the IP author to produce encrypted source files that can be safely decrypted by multiple authors. For these reasons, encryption envelopes are the preferred method of protection.

Configuring the Encryption Envelope

The encryption envelope may be configured two ways:

1. The encryption envelope may contain the textual design data to be encrypted ([Example 3-1](#)).

2. The encryption envelope may contain ``include` compiler directives that point to files containing the textual design data to be encrypted (Example 3-2). See [Using the ``include` Compiler Directive \(Verilog only\)](#).

Example 3-1. Encryption Envelope Contains Verilog IP Code to be Protected

```
module test_dff4(output [3:0] q, output err);
  parameter WIDTH = 4;
  parameter DEBUG = 0;
  reg [3:0] d;
  reg  clk;

  dff4 d4(q, clk, d);

  assign  err = 0;

  initial
  begin
    $dump_all_vpi;
    $dump_tree_vpi(test_dff4);
    $dump_tree_vpi(test_dff4.d4);
    $dump_tree_vpi("test_dff4");
    $dump_tree_vpi("test_dff4.d4");
    $dump_tree_vpi("test_dff4.d", "test_dff4.clk", "test_dff4.q");
    $dump_tree_vpi("test_dff4.d4.d0", "test_dff4.d4.d3");
    $dump_tree_vpi("test_dff4.d4.q", "test_dff4.d4.clk");
  end
endmodule

module dff4(output [3:0] q, input clk, input [3:0] d);
  `pragma protect data_method = "aes128-cbc"
  `pragma protect author = "IP Provider"
  `pragma protect author_info = "Widget 5 version 3.2"
  `pragma protect key_keyowner = "Mentor Graphics Corporation"
  `pragma protect key_method = "rsa"
  `pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
  `pragma protect begin
    dff_gate d0(q[0], clk, d[0]);
    dff_gate d1(q[1], clk, d[1]);
    dff_gate d2(q[2], clk, d[2]);
    dff_gate d3(q[3], clk, d[3]);
  endmodule // dff4

module dff_gate(output q, input clk, input d);
  wire preset = 1;
  wire clear = 1;

  nand #5
  g1(11,preset,14,12),
  g2(12,11,clear,clk),
  g3(13,12,clk,14),
  g4(14,13,clear,d),
  g5(q,preset,12,qbar),
  g6(qbar,q,clear,13);
endmodule
`pragma protect end
```

In this example, the Verilog code to be encrypted follows the ``pragma protect begin` expression and ends with the ``pragma protect end` expression. If the code had been written in VHDL, the code to be protected would follow a ``protect BEGIN PROTECTED` expression and would end with a ``protect END PROTECTED` expression.

Example 3-2. Encryption Envelope Contains ``include` Compiler Directives

```
`timescale 1ns / 1ps
`cell define

module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
reg q;

`pragma protect data_method = "aes128-cbc"
`pragma protect author = "IP Provider", author_info = "Widget 5 v3.2"
`pragma protect key_keyowner = "Mentor Graphics Corporation"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect begin

`include diff.v
`include prim.v
`include top.v

`pragma protect end

always @(posedge clock)
    q = d;

endmodule

`endcelldefine
```

In [Example 3-2](#), the entire contents of `diff.v`, `prim.v`, and `top.v` will be encrypted.

For a more technical explanation, see [How Encryption Envelopes Work](#).

Protection Expressions

The encryption envelope contains a number of ``pragma protect` (Verilog/SystemVerilog) or ``protect` (VHDL) expressions. The following protection expressions are expected when creating an encryption envelope:

- **data_method** — defines the encryption algorithm that will be used to encrypt the designated source text. ModelSim supports the following encryption algorithms: des-cbc, 3des-cbc, aes128-cbc, aes256-cbc, blowfish-cbc, cast128-cbc, and rsa.
- **key_keyowner** — designates the owner of the encryption key.

- **key_keyname** — specifies the keyowner's key name.
- **key_method** — specifies an encryption algorithm that will be used to encrypt the key.

Note

The combination of `key_keyowner` and `key_keyname` expressions uniquely identify a key. The `key_method` is required with these two expressions to complete the definition of the key.

- **begin** — designates the beginning of the source code to be encrypted.
- **end** — designates the end of the source code to be encrypted

Note

Encryption envelopes cannot be nested. A ``pragma protect begin/end` pair cannot bracket another ``pragma protect begin/end` pair.

Optional ``protect` (VHDL) or ``pragma protect` (Verilog/SystemVerilog) expressions that may be included are as follows:

- **author** — designates the IP provider.
- **author_info** — designates optional author information.
- **encoding** — specifies an encoding method. The default encoding method, if none is specified, is “base 64.”

If a number of protection expressions occur in a single protection directive, the expressions are evaluated in sequence from left to right. In addition, the interpretation of protected envelopes is not dependent on this sequence occurring in a single protection expression or a sequence of protection expressions. However, the most recent value assigned to a protection expression keyword will be the one used.

Unsupported Protection Expressions

Optional protection expressions that are not currently supported include:

- any `digest_*` expression
- `decrypt_license`
- `runtime_license`
- `viewport`

Using the ``include` Compiler Directive (Verilog only)

If any ``include` directives occur within a protected region of Verilog code and you use `vlog +protect` to compile, the compiler generates a copy of the include file with a `.vp` or a `.svp` extension and encrypts the entire contents of the include file. For example, if we have a header file, `header.v`, with the following source code:

```
initial begin
    a <= b;
    b <= c;
end
```

and the file we want to encrypt, `top.v`, contains the following source code:

```
module top;
    `pragma protect begin
        `include "header.v"
    `pragma protect end
endmodule
```

then, when we use the `vlog +protect` command to compile, the source code of the header file will be encrypted. If we could decrypt the resulting `work/top.vp` file it would look like:

```
module top;
    `pragma protect begin
        initial begin
            a <= b;
            b <= c;
        end
    `pragma protect end
endmodule
```

In addition, `vlog +protect` creates an encrypted version of `header.v` in `work/header.vp`.

When using the `vencrypt` compile utility (see [Delivering IP Code with Undefined Macros](#)), any ``include` statements will be treated as text just like any other source code and will be encrypted with the other Verilog/SystemVerilog source code. So, if we used the `vencrypt` utility on the `top.v` file above, the resulting `work/top.vp` file would look like the following (if we could decrypt it):

```
module top;
    `protect
        `include "header.v"
    `endprotect
endmodule
```

The `vencrypt` utility will not create an encrypted version of `header.h`.

When you use `vlog +protect` to generate encrypted files, the original source files must all be complete Verilog or SystemVerilog modules or packages. Compiler errors will result if you attempt to perform compilation of a set of parameter declarations within a module. (See also [Compiling with +protect.](#))

You can avoid such errors by creating a dummy module that includes the parameter declarations. For example, if you have a file that contains your parameter declarations and a file that uses those parameters, you can do the following:

```
module dummy;
  \protect
  \include "params.v" // contains various parameters
  \include "tasks.v" // uses parameters defined in params.v
  \endprotect
endmodule
```

Then, compile the dummy module with the `+protect` switch to generate an encrypted output file with no compile errors.

vlog +protect dummy.v

After compilation, the work library will contain encrypted versions of *params.v* and *tasks.v*, called *params.vp* and *tasks.vp*. You may then copy these encrypted files out of the work directory to more convenient locations. These encrypted files can be included within your design files; for example:

```
module main
  'include "params.vp"
  'include "tasks.vp"
  ...
endmodule
```

Using Portable Encryption for Multiple Tools

An IP author can use the concept of multiple key blocks to produce code that is secure and portable across any tool that supports Version 1 recommendations from the IEEE P1735 working group. This capability is not language-specific - it can be used for VHDL or Verilog.

To illustrate, suppose the author wants to modify the following VHDL *sample file* so the encrypted model can be decrypted and simulated by both ModelSim and by a hypothetical company named XYZ inc.

```
===== sample file =====

-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
\protect data_method = "aes128-cbc"
\protect encoding = ( enctype = "base64" )
\protect key_keyowner = "Mentor Graphics Corporation"
\protect key_keyname = "MGC-VERIF-SIM-RSA-1"
\protect key_method = "rsa"
\protect KEY_BLOCK
\protect begin
```

```
architecture a of ip1 is
...
end a;
`protect end

-- Both the entity "ip2" and its architecture "a" are completely protected
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" )
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect KEY_BLOCK
`protect begin
library ieee;
use ieee.std_logic_1164.all;
entity ip2 is
...
end ip2;
architecture a of ip2 is
...
end a;
`protect end

===== end of sample file =====
```

The author does this by writing a key block for each decrypting tool. If XYZ publishes a public key, the two key blocks in the IP source code might look like the following:

```
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_method = "rsa"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect KEY_BLOCK
`protect key_keyowner = "XYZ inc"
`protect key_method = "rsa"
`protect key_keyname = "XYZ-keyPublicKey"
`protect key_public_key = <public key of XYZ inc.>
`protect KEY_BLOCK
```

The encrypted code would look very much like the *sample file*, with the addition of another key block:

```
`protect key_keyowner = "XYZ inc"
`protect key_method = "rsa"
`protect key_keyname = "XYZ-keyPublicKey"
`protect KEY_BLOCK
    <encoded encrypted key information for "XYZ inc">
```

ModelSim uses its key block to determine the encrypted session key and XYZ Incorporated uses the second key block to determine the same key. Consequently, both implementations could successfully decrypt the code.

Note

The IP owner is responsible for obtaining the appropriate key for the specific tool(s) protected IP is intended for, and should validate the encrypted results with those tools to insure his IP is protected and will function as intended in those tools.

Compiling with +protect

To encrypt IP code with ModelSim, the **+protect** argument must be used with either the **vcom** command (for VHDL) or the **vlog** command (for Verilog and SystemVerilog). For example, if a Verilog source code file containing encryption envelopes is named *encrypt.v*, it would be compiled as follows:

```
vlog +protect encrypt.v
```

When +protect is used with **vcom** or **vlog**, encryption envelope expressions are transformed into decryption envelope expressions and decryption content expressions. Source text within encryption envelopes is encrypted using the specified key and is recorded in the decryption envelope within a `data_block`. The new encrypted file is created with the same name as the original unencrypted file but with a 'p' added to the filename extension. For Verilog, the filename extension for the encrypted file is *.vp*; for SystemVerilog it is *.svp*, and for VHDL it is *.vhdp*. This encrypted file is placed in the current work library directory.

You can designate the name of the encrypted file using the **+protect=<filename>** argument with **vcom** or **vlog** as follows:

```
vlog +protect=encrypt.vp encrypt.v
```

Example 3-3 shows the resulting source code when the Verilog IP code used in **Example 3-1** is compiled with **vlog +protect**.

Example 3-3. Results After Compiling with vlog +protect

```
module test_dff4(output [3:0] q, output err);
  parameter WIDTH = 4;
  parameter DEBUG = 0;
  reg [3:0] d;
  reg  clk;
  dff4 d4(q, clk, d);
  assign  err = 0;
  initial
  begin
    $dump_all_vpi;
    $dump_tree_vpi(test_dff4);
    $dump_tree_vpi(test_dff4.d4);
    $dump_tree_vpi("test_dff4");
    $dump_tree_vpi("test_dff4.d4");
    $dump_tree_vpi("test_dff4.d", "test_dff4.clk", "test_dff4.q");
    $dump_tree_vpi("test_dff4.d4.d0", "test_dff4.d4.d3");
    $dump_tree_vpi("test_dff4.d4.q", "test_dff4.d4.clk");
  end
```

```
endmodule

module dff4(output [3:0] q, input clk, input [3:0] d);
    `pragma protect begin_protected
    `pragma protect version = 1
    `pragma protect encrypt_agent = "Model Technology"
    `pragma protect encrypt_agent_info = "6.6a"
    `pragma protect author = "IP Provider"
    `pragma protect author_info = "Widget 5 version 3.2"
    `pragma protect data_method = "aes128-cbc"
    `pragma protect key_keyowner = "Mentor Graphics Corporation"
    `pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
    `pragma protect key_method = "rsa"
    `pragma protect key_block encoding = (enctype = "base64", line_length =
64, bytes = 128)
SdI6t9ewd9GE4va+2BgfnRuBNc45wVwjyPeSD/5qnojnbaHdpjWa/O/Tyhw0aq1T
NbDGrDg6I5dbzbLs5UQGfTB2lgOBMnE4JTpGRfV0sEqUdibBHiTpsNrbLpp1iJLi
714kQhniVnUuCx87GuqXI5AaoLGBz5rCxKyA47ElQM=
    `pragma protect data_block encoding = (enctype = "base64", line_length =
64, bytes = 496)
efkkPz4gJSO6zZfYdr37fqEoxgLZ3oTgu8y34GTYkO0ZZGKkyonE9zDQct5d0dfe
/BZwoHCWnq4xqUp2dxF4x6cw6qBJcSEifCPDY1hJASoVX+7owIPGnLh5U0P/Wohp
LvkfhIuk2FENGZh+y3rWZAC1vFYKXwDakSJ3neSg1HkwYr+T8vGviohIPKet+CPC
d/RxXO12ChI64KaMY2/fK1erXrnXV7o9ZIrJRHL/CtQ/uxY7aMioR3/WobFrnuoz
P8fH7x/I30taK25KiL6qvUN0jf7g4LiozSTvcT6iTTHXOmB0fZiC1eREMF835q8D
K51zU+rCb17Wyt8utm71WSu+2gtwvEp39G6R60fkQAuVGw+xsqtmWyyIodM+PKWl
sqeoVosBUHFY3x85F534PQNVIVAT1VzFeioMxmJWV+pft3O1rcJGqX1AxAG25CkY
M1zF77caF8LAsKbvCTgOVsHb7NEqOVTVJZzYdVY23VswClYcrxroOhPzmqNgn4pf
zqcFpP+yBnt4UELa63Os6OfsAu7DZ/4kWPawExyvaahI2ciWs3HREcZEO+aveuLT
gxEFfSm0TvBBsMwLc7UvjjC0aF1vUWhDxhwQDAjYT89r2h1G7Y0PG1G0o24s0/A2+
TjdCcOogiGsTDKx6Bxf91g==
    `pragma protect end_protected
endmodule
```

In this example, the ``pragma protect data_method` expression designates the encryption algorithm used to encrypt the Verilog IP code. The key for this encryption algorithm is also encrypted – in this case, with the RSA public key. The key is recorded in the `key_block` of the protected envelope. The encrypted IP code is recorded in the `data_block` of the envelope. ModelSim allows more than one `key_block` to be included so that a single protected envelope can be encrypted by ModelSim then decrypted by tools from different users.

The Runtime Encryption Model

After you compile with the `+protect` compile argument, all source text, identifiers, and line number information are hidden from the end user in the resulting compiled object. ModelSim cannot locate or display any information of the encrypted regions. Specifically, this means that:

- a Source window will not display the design units' source code
- a Structure window will not display the internal structure
- the Objects window will not display internal signals
- the Processes window will not display internal processes

- the Locals window will not display internal variables
- none of the hidden objects may be accessed through the Dataflow or Schematic windows or with ModelSim commands.

Language-Specific Usage Models

This section includes the following usage models that are language-specific:

- [Usage Models for Protecting Verilog Source Code](#)
 - [Delivering IP Code with Undefined Macros](#)
 - [Delivering IP Code with User-Defined Macros](#)
- [Usage Models for Protecting VHDL Source Code](#)
 - [Using the vhenrypt Utility](#)
 - [Using ModelSim Default Encryption for VHDL](#)
 - [User-Selected Encryption for VHDL](#)
 - [Using raw Encryption for VHDL](#)
 - [Encrypting Several Parts of a VHDL Source File](#)
 - [Using Portable Encryption for Multiple Tools](#)

Usage Models for Protecting Verilog Source Code

ModelSim's encryption capabilities support the following Verilog and SystemVerilog usage models for IP authors and their customers.

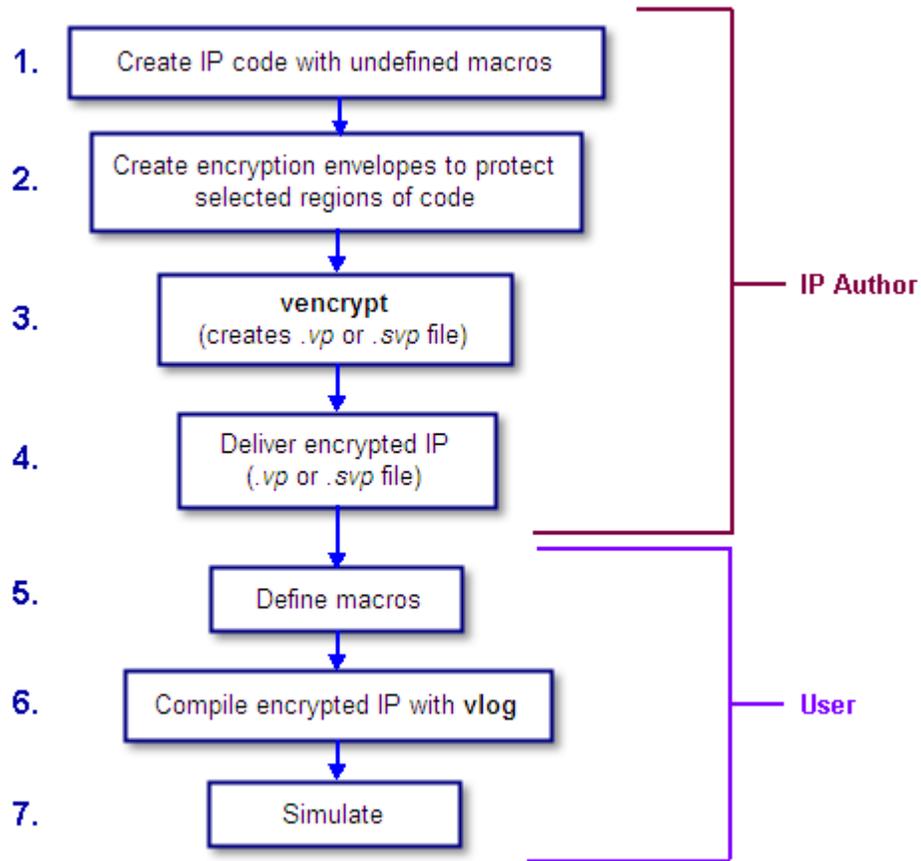
- IP authors may use the [vencrypt](#) utility to deliver Verilog and SystemVerilog code containing *undefined* macros and ``directives`. The IP user can then define the macros and ``directives` and use the code in a wide range of EDA tools and design flows. See [Delivering IP Code with Undefined Macros](#).
- IP authors may use ``pragma protect` directives to protect Verilog and SystemVerilog code containing *user-defined* macros and ``directives`. The IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows. See [Delivering IP Code with User-Defined Macros](#).

Delivering IP Code with Undefined Macros

The [vencrypt](#) utility enables IP authors to deliver VHDL and Verilog/ SystemVerilog IP code (respectively) that contains undefined macros and ``directives`. The resulting encrypted IP code can then be used in a wide range of EDA tools and design flows.

The recommended encryption usage flow is shown in [Figure 3-2](#).

Figure 3-2. Verilog/SystemVerilog Encryption Usage Flow



1. The IP author creates code that contains undefined macros and ```directives.
2. The IP author creates encryption envelopes (see [Creating Encryption Envelopes](#)) to protect selected regions of code or entire files (see [Protection Expressions](#)).
3. The IP author uses ModelSim's [vencrypt](#) utility to encrypt Verilog and SystemVerilog code contained within encryption envelopes. Macros are not pre-processed before encryption so macros and other ```directives are unchanged.

The `vencrypt` utility produces a file with a `.vp` or a `.svp` extension to distinguish it from non-encrypted Verilog and SystemVerilog files, respectively. The file extension may be changed for use with simulators other than ModelSim. The original file extension is preserved if the `-d <dirname>` argument is used with `vencrypt`, or if a ```directive is used in the file to be encrypted.

With the `-h <filename>` argument for `vencrypt` the IP author may specify a header file that can be used to encrypt a large number of files that do not contain the ``pragma protect` (or proprietary ``protect` information - see [Proprietary Source Code Encryption Tools](#)) about how to encrypt the file. Instead, encryption information is provided in the

<filename> specified by `-h <filename>`. This argument essentially concatenates the header file onto the beginning of each file and saves the user from having to edit hundreds of files in order to add in the same ``pragma protect` to every file. For example,

```
vencrypt -h encrypt_head top.v cache.v gates.v memory.v
```

concatenates the information in the `encrypt_head` file into each verilog file listed. The `encrypt_head` file may look like the following:

```
`pragma protect data_method = "aes128-cbc"  
`pragma protect author = "IP Provider"  
`pragma protect key_keyowner = "Mentor Graphics Corporation"  
`pragma protect key_method = "rsa"  
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"  
`pragma protect encoding = (enctype = "base64")  
`pragma protect begin
```

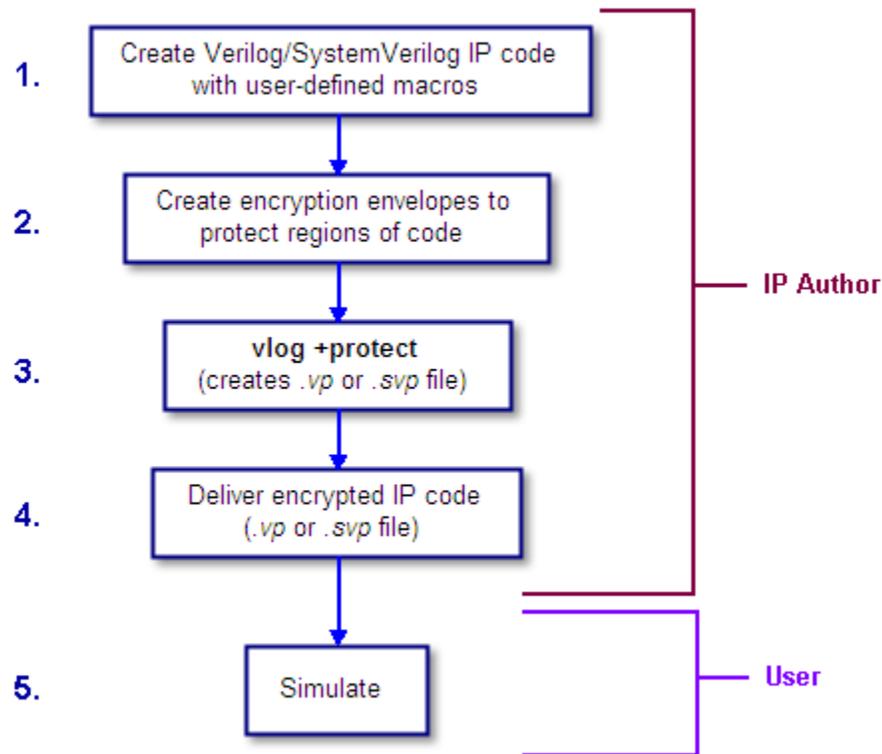
Notice, there is no ``pragma protect end` expression in the header file, just the header block that starts the encryption. The ``pragma protect end` expression is implied by the end of the file.

4. The IP author delivers encrypted IP with undefined macros and ``directives`.
5. The IP user defines macros and ``directives`.
6. The IP user compiles the design with `vlog`.
7. The IP user simulates the design with ModelSim or other simulation tools.

Delivering IP Code with User-Defined Macros

IP authors may use ``pragma protect` expressions to protect proprietary code containing user-defined macros and ``directives`. The resulting encrypted IP code can be delivered to customers for use in a wide range of EDA tools and design flows. An example of the recommended usage flow for Verilog and SystemVerilog IP is shown in [Figure 3-3](#).

Figure 3-3. Delivering IP Code with User-Defined Macros



1. The IP author creates proprietary code that contains user-defined macros and ``directives`.
2. The IP author creates encryption envelopes with ``pragma protect` expressions to protect regions of code or entire files. See [Creating Encryption Envelopes](#) and [Protection Expressions](#).
3. The IP author uses the `+protect` argument for the `vlog` command to encrypt IP code contained within encryption envelopes. The ``pragma protect` expressions are ignored unless the `+protect` argument is used during compile. (See [Compiling with +protect.](#))

The `vlog +protect` command produces a `.vp` or a `.svp` extension for the encrypted file to distinguish it from non-encrypted Verilog and SystemVerilog files, respectively. The file extension may be changed for use with simulators other than ModelSim. The original file extension is preserved if a ``directive` is used in the file to be encrypted. For more information, see [Compiling with +protect](#).

4. The IP author delivers the encrypted IP.
5. The IP user simulates the code like any other file.

When encrypting source text, any macros without parameters defined on the command line are substituted (not expanded) into the encrypted file. This makes certain macros unavailable in the encrypted source text.

ModelSim takes every simple macro that is defined with the compile command (`vlog`) and substitutes it into the encrypted text. This prevents third party users of the encrypted blocks from having access to or modifying these macros.

Note

Macros not specified with `vlog` via the `+define+` option are unmodified in the encrypted block.

For example, the code below is an example of a file that might be delivered by an IP provider. The filename for this module is *example00.sv*.

```
`pragma protect data_method = "aes128-cbc"
`pragma protect key_keyowner = "Mentor Graphics Corporation"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect author = "Mentor", author_info = "Mentor_author"
`pragma protect begin
`timescale 1 ps / 1 ps

module example00 ();
  `ifdef IPPROTECT
    reg `IPPROTECT ;
    reg otherReg ;
    initial begin
      `IPPROTECT = 1;
      otherReg    = 0;

      $display("ifdef defined as true");

      `define FOO 0
      $display("FOO is defined as: ", `FOO);
      $display("reg IPPROTECT has the value: ", `IPPROTECT );
    end
  `else
    initial begin
      $display("ifdef defined as false");
    end
  `endif

endmodule

`pragma protect end
```

We encrypt the *example00.sv* module with the `vlog` command as follows:

```
vlog +define+IPPROTECT=ip_value +protect=encrypted00.sv example00.sv
```

This creates an encrypted file called *encrypted00.sv*. We can then compile this file with a macro override for the macro “FOO” as follows:

```
vlog +define+FOO=99 encrypted00.sv
```

The macro FOO can be overridden by a customer while the macro IPPROTECT retains the value specified at the time of encryption, and the macro IPPROTECT no longer exists in the encrypted file.

Usage Models for Protecting VHDL Source Code

ModelSim's encryption capabilities support the following VHDL usage models.

- IP authors may use ``protect` directives to create an encryption envelope (see [Creating Encryption Envelopes](#)) for the VHDL code to be protected and use ModelSim's `vhencrypt` utility to encrypt the code. The encrypted IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows. See [Using the vhencrypt Utility](#).
- IP authors may use ``protect` directives to create an encryption envelope (see [Creating Encryption Envelopes](#)) for the VHDL code to be protected and use ModelSim's default encryption and decryption actions. The IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows. See [Using ModelSim Default Encryption for VHDL](#).
- IP authors may use ``protect` directives to create an encryption envelope for VHDL code and select encryption methods and encoding other than ModelSim's default methods. See [User-Selected Encryption for VHDL](#).
- IP authors may use "raw" encryption and encoding to aid debugging. See [Using raw Encryption for VHDL](#).
- IP authors may encrypt several parts of the source file, choose the encryption method for encrypting the source (the `data_method`), and use a key automatically provided by ModelSim. See [Encrypting Several Parts of a VHDL Source File](#).
- IP authors can use the concept of multiple key blocks to produce code that is secure and portable across different simulators. See [Using Portable Encryption for Multiple Tools](#).

The usage models are illustrated by examples in the sections below.

Note



VHDL encryption requires that the `KEY_BLOCK` (the sequence of `key_keyowner`, `key_keyname`, and `key_method` directives) end with a ``protect KEY_BLOCK` directive.

Using the `vhencrypt` Utility

The `vhencrypt` utility enables IP authors to deliver encrypted VHDL IP code to users. The resulting encrypted IP code can then be used in a wide range of EDA tools and design flows.

1. The IP author creates code.

2. The IP author creates encryption envelopes (see [Creating Encryption Envelopes](#)) to protect selected regions of code or entire files (see [Protection Expressions](#)).
3. The IP author uses ModelSim's `vhencrypt` utility to encrypt code contained within encryption envelopes.

The `vhencrypt` utility produces a file with a `.vhdp` or a `.vhdlp` extension to distinguish it from non-encrypted VHDL files. The file extension may be changed for use with simulators other than ModelSim. The original file extension is preserved if the `-d <dirname>` argument is used with `vhencrypt`.

With the `-h <filename>` argument for `vhencrypt` the IP author may specify a header file that can be used to encrypt a large number of files that do not contain the ``protect` information about how to encrypt the file. Instead, encryption information is provided in the `<filename>` specified by `-h <filename>`. This argument essentially concatenates the header file onto the beginning of each file and saves the user from having to edit hundreds of files in order to add in the same ``protect` to every file. For example,

```
vhencrypt -h encrypt_head top.vhd cache.vhd gates.vhd memory.vhd
```

concatenates the information in the `encrypt_head` file into each VHDL file listed. The `encrypt_head` file may look like the following:

```
`protect data_method = "aes128-cbc"  
`protect author = "IP Provider"  
`protect encoding = (enctype = "base64")  
`protect key_keyowner = "Mentor Graphics Corporation"  
`protect key_method = "rsa"  
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"  
`protect KEY_BLOCK  
`protect begin
```

Notice, there is no ``protect end` expression in the header file, just the header block that starts the encryption. The ``protect end` expression is implied by the end of the file.

4. The IP author delivers encrypted IP.
5. The IP user compiles the design with `vcom`.
6. The IP user simulates the design with ModelSim or other simulation tools.

Using ModelSim Default Encryption for VHDL

Suppose an IP author needs to make a design entity, called IP1, visible to the user so the user can instantiate the design, but the author wants to hide the architecture implementation from the user. In addition, suppose that IP1 instantiates entity IP2, which the author wants to hide completely from the user. The easiest way to accomplish this is to surround the regions to be protected with ``protect begin` and ``protect end` directives and let ModelSim choose default actions. For this example, all the source code exists in a single file, `example1.vhd`:

```
===== file example1.vhd =====
```

```
-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
`protect begin
architecture a of ip1 is
...
end a;
`protect end

-- Both the entity "ip2" and its architecture "a" are completely protected
`protect begin
entity ip2 is
...
end ip2;
architecture a of ip2 is
...
end a;
`protect end

===== end of file example1.vhd =====
```

The IP author compiles this file with the `vcom +protect` command as follows:

```
vcom +protect=example1.vhdp example1.vhd
```

The compiler produces an encrypted file, *example1.vhdp* which looks like the following:

```
===== file example1.vhdp =====

-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
`protect BEGIN_PROTECTED
`protect version = 1
`protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect encoding = ( enctype = "base64" )
`protect KEY_BLOCK
  <encoded encrypted session key>
`protect data_method="aes128-cbc"
`protect encoding = ( enctype = "base64" , bytes = 224 )
`protect DATA_BLOCK
  <encoded encrypted IP>
`protect END_PROTECTED
```

```
-- Both the entity "ip2" and its architecture "a" are completely protected
`protect BEGIN_PROTECTED
`protect version = 1
`protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect encoding = ( enctype = "base64" )
`protect KEY_BLOCK
    <encoded encrypted session key>
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" , bytes = 224 )
`protect DATA_BLOCK
    <encoded encrypted IP>
`protect END_PROTECTED

===== end of file example1.vhdp =====
```

When the IP author surrounds a text region using only **`protect begin** and **`protect end**, ModelSim uses default values for both encryption and encoding. The first few lines following the **`protect BEGIN_PROTECTED** region in file *example1.vhdp* contain the `key_keyowner`, `key_keyname`, `key_method` and `KEY_BLOCK` directives. The session key is generated into the key block and that key block is encrypted using the “rsa” method. The `data_method` indicates that the default data encryption method is `aes128-cbc` and the “enctype” value shows that the default encoding is `base64`.

Alternatively, the IP author can compile file *example1.vhd* with the command:

```
vcom +protect example1.vhd
```

Here, the author does not supply the name of the file to contain the protected source. Instead, ModelSim creates a protected file, gives it the name of the original source file with a 'p' placed at the end of the file extension, and puts the new file in the current work library directory. With the command described above, ModelSim creates file *work/example1.vhdp*. (See [Compiling with +protect.](#))

The IP user compiles the encrypted file *work/example1.vhdp* the ordinary way. The `+protect` switch is not needed and the IP user does not have to treat the *.vhdp* file in any special manner. ModelSim automatically decrypts the file internally and keeps track of protected regions.

If the IP author compiles the file *example1.vhd* and does not use the `+protect` argument, then the file is compiled, various **`protect** directives are checked for correct syntax, but no protected file is created and no protection is supplied.

ModelSim’s default encryption methods provide an easy way for IP authors to encrypt VHDL designs while hiding the architecture implementation from the user. It should be noted that the results are only usable by ModelSim tools.

User-Selected Encryption for VHDL

Suppose that the IP author wants to produce the same code as in the *example1.vhd* file used above, but wants to provide specific values and not use any default values. To do this the author adds ``protect` directives for keys, encryption methods, and encoding, and places them before each ``protect begin` directive. The input file would look like the following:

```
===== file example2.vhd =====

-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" )
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect KEY_BLOCK
`protect begin
architecture a of ip1 is
...
end a;
`protect end

-- Both the entity "ip2" and its architecture "a" are completely protected
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" )
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect KEY_BLOCK
`protect begin
library ieee;
use ieee.std_logic_1164.all;
entity ip2 is
...
end ip2;
architecture a of ip2 is
...
end a;
`protect end

===== end of file example2.vhd =====
```

The `data_method` directive indicates that the encryption algorithm “aes128-cbc” should be used to encrypt the source code (data). The `encoding` directive selects the “base64” encoding method, and the various key directives specify that the Mentor Graphic key named “MGC-VERIF-SIM-RSA-1” and the “RSA” encryption method are to be used to produce a key block containing a randomly generated session key to be used with the “aes128-cbc” method to encrypt the source code. See [Using the Mentor Graphics Public Encryption Key](#).

Using raw Encryption for VHDL

Suppose that the IP author wants to use “raw” encryption and encoding to help with debugging the following entity:

```
entity example3_ent is
    port (
        in1 : in bit;
        out1 : out bit);
end example3_ent;
```

Then the architecture the author wants to encrypt might be this:

```
===== File example3_arch.vhd
\protect data_method = "raw"
\protect encoding = ( enctype = "raw")
\protect begin
architecture arch of example3_ent is
begin
out1 <= in1 after 1 ns;
end arch;
\protect end
===== End of file example3_arch.vhd =====
```

If (after compiling the entity) the *example3_arch.vhd* file were compiled using the command:

```
vcom +protect example3_arch.vhd
```

Then the following file would be produced in the work directory

```
===== File work/example3_arch.vhdp =====
\protect data_method = "raw"
\protect encoding = ( enctype = "raw")
\protect BEGIN_PROTECTED
\protect version = 1
\protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
\protect data_method = "raw"
\protect encoding = ( enctype = "raw", bytes = 81 )
\protect DATA_BLOCK
architecture arch of example3_ent is
begin
out1 <= in1 after 1 ns;
end arch;
\protect END_PROTECTED
```

```
===== End of file work/example3_arch.vhdp
```

Notice that the protected file is very similar to the original file. The differences are that ``protect begin` is replaced by ``protect BEGIN_PROTECTED`, ``protect end` is replaced by ``protect END_PROTECTED`, and some additional encryption information is supplied after the `BEGIN PROTECTED` directive.

See [Encryption and Encoding Methods](#) for more information about raw encryption and encoding.

Encrypting Several Parts of a VHDL Source File

This example shows the use of symmetric encryption. (See [Encryption and Encoding Methods](#) for more information on symmetric and asymmetric encryption and encoding.) It also demonstrates another common use model, in which the IP author encrypts several parts of a source file, chooses the encryption method for encrypting the source code (the `data_method`), and uses a key automatically provided by ModelSim. (This is very similar to the proprietary ``protect` method in Verilog - see [Proprietary Source Code Encryption Tools](#).)

```
===== file example4.vhd =====  
  
entity ex4_ent is  
  
end ex4_ent;  
  
architecture ex4_arch of ex4_ent is  
    signal s1: bit;  
    `protect data_method = "aes128-cbc"  
    `protect begin  
        signal s2: bit;  
    `protect end  
        signal s3: bit;  
  
begin -- ex4_arch  
  
    `protect data_method = "aes128-cbc"  
    `protect begin  
s2 <= s1 after 1 ns;  
    `protect end  
  
s3 <= s2 after 1 ns;  
  
end ex4_arch;  
  
===== end of file example4.vhd
```

If this file were compiled using the command:

```
vcom +protect example4.vhd
```

Then the following file would be produced in the work directory:

```
===== File work/example4.vhdp =====
```

```

entity ex4_ent is

end ex4_ent;

architecture ex4_arch of ex4_ent is
    signal s1: bit;
    `protect data_method = "aes128-cbc"
    `protect BEGIN_PROTECTED
    `protect version = 1
    `protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
    `protect data_method = "aes128-cbc"
    `protect encoding = ( enctype = "base64" , bytes = 18 )
    `protect DATA_BLOCK
    <encoded encrypted declaration of s2>
    `protect END_PROTECTED
    signal s3: bit;

begin -- ex4_arch

    `protect data_method = "aes128-cbc"
    `protect BEGIN_PROTECTED
    `protect version = 1
    `protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
    `protect data_method = "aes128-cbc"
    `protect encoding = ( enctype = "base64" , bytes = 21 )
    `protect DATA_BLOCK
    <encoded encrypted signal assignment to s2>
    `protect END_PROTECTED

s3 <= s2 after 1 ns;

end ex4_arch;

===== End of file work/example4.vhdp

```

The encrypted *example4.vhdp* file shows that an IP author can encrypt both declarations and statements. Also, note that the signal assignment

```
s3 <= s2 after 1 ns;
```

is not protected. This assignment compiles and simulates even though signal *s2* is protected. In general, executable VHDL statements and declarations simulate the same whether or not they refer to protected objects.

Proprietary Source Code Encryption Tools

Mentor Graphics provides two proprietary methods for encrypting source code.

- The **`protect`** / **`endprotect`** compiler directives allow you to encrypt regions within Verilog and SystemVerilog files.

- The **-nodebug** argument for the **vcom** and **vlog** compile commands allows you to encrypt entire VHDL, Verilog, or SystemVerilog source files.

Using Proprietary Compiler Directives

The proprietary **`protect** vlog compiler directive is not compatible with other simulators. Though other simulators have a **`protect** directive, the algorithm ModelSim uses to encrypt Verilog and SystemVerilog source files is different. Therefore, even though an uncompiled source file with **`protect** is compatible with another simulator, once the source is compiled in ModelSim, the resulting **.vp** or **.svp** source file is not compatible.

IP authors and IP users may use the **`protect** compiler directive to define regions of Verilog and SystemVerilog code to be protected. The code is then compiled with the **vlog +protect** command and simulated with ModelSim. The **vencrypt** utility may be used if the code contains undefined macros or **`directives**, but the code must then be compiled and simulated with ModelSim.

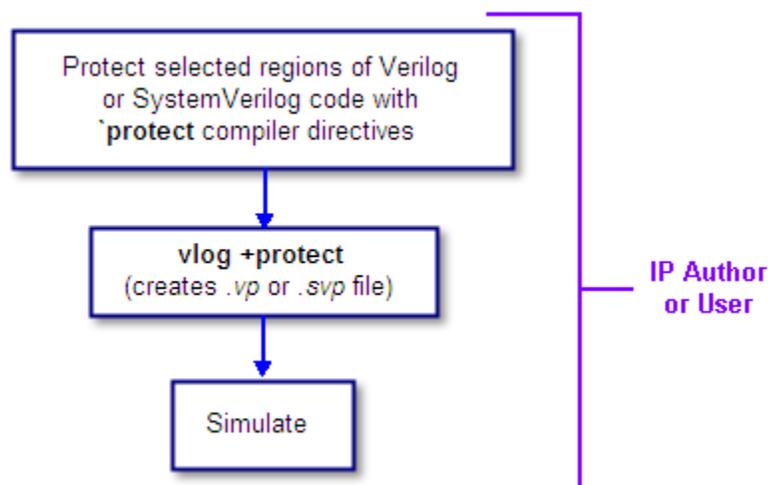
Note



While ModelSim supports both **`protect** and **`pragma protect** encryption directives, these two approaches to encryption are incompatible. Code encrypted by one type of directive cannot be decrypted by another.

The usage flow for delivering IP with the Mentor Graphics proprietary **`protect** compiler directive is as follows:

Figure 3-4. Delivering IP with `protect Compiler Directives



1. The IP author protects selected regions of Verilog or SystemVerilog IP with the **`protect / `endprotect** directive pair. The code in **`protect / `endprotect** encryption envelopes has all debug information stripped out. This behaves exactly as if using

```
vlog -nodebug=ports+pli
```

except that it applies to selected regions of code rather than the whole file.

2. The IP author uses the `vlog +protect` command to encrypt IP code contained within encryption envelopes. The ``protect` / `endprotect directives are ignored by default unless the +protect argument is used with vlog.`

Once compiled, the original source file is copied to a new file in the current work directory. The `vlog +protect` command produces a `.vp` or a `.svp` extension to distinguish it from other non-encrypted Verilog and SystemVerilog files, respectively. For example, `top.v` becomes `top.vp` and `cache.sv` becomes `cache.svp`. This new file can be delivered and used as a replacement for the original source file. (See [Compiling with +protect](#).)

Note

The `vincrypt` utility may be used if the code also contains undefined macros or ``directives`, but the code must then be compiled and simulated with ModelSim.

You can use `vlog +protect=<filename>` to create an encrypted output file, with the designated filename, in the current directory (not in the `work` directory, as in the default case where [=<filename>] is not specified). For example:

```
vlog test.v +protect=test.vp
```

If the filename is specified in this manner, all source files on the command line will be concatenated together into a single output file. Any ``include` files will also be inserted into the output file.

Caution

``protect` and ``endprotect` directives cannot be nested.

If errors are detected in a protected region, the error message always reports the first line of the protected block.

Protecting Source Code Using `-nodebug`

Verilog/SystemVerilog and VHDL IP authors and users may use the proprietary `vlog -nodebug` or `vcom -nodebug` command, respectively, to protect entire files. The `-nodebug` argument for both `vcom` and `vlog` hides internal model data, allowing you to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

Note

The `-nodebug` argument encrypts entire files. The ``protect` compiler directive allows you to encrypt regions within a file. Refer to [Compiler Directives](#) for details.

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

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

Table 3-1. Compile Options for the -nodebug Compiling

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

Note



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

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

Do not use -nodebug=ports when the parent is part of a vopt -pdu (black-box) flow or for mixed language designs, especially for Verilog modules to be instantiated inside VHDL.

Encryption Reference

This section includes reference details on:

- [Encryption and Encoding Methods](#)
- [How Encryption Envelopes Work](#)
- [Using Public Encryption Keys](#)
- [Using the Mentor Graphics Public Encryption Key](#)

Encryption and Encoding Methods

There are two basic encryption techniques: symmetric and asymmetric.

- Symmetric encryption uses the same key for both encrypting and decrypting the code region.
- Asymmetric encryption methods use two keys: a public key for encryption, and a private key for decryption.

Symmetric Encryption

For symmetric encryption, security of the key is critical and information about the key must be supplied to ModelSim. Under certain circumstances, ModelSim will generate a random key for use with a symmetric encryption method or will use an internal key.

The symmetric encryption algorithms ModelSim supports are:

- des-cbc
- 3des-cbc
- aes128-cbc
- aes192-cbc
- aes256-cbc
- blowfish-cbc
- cast128-cbc

The default symmetric encryption method ModelSim uses for encrypting IP source code is aes128-cbc.

Asymmetric Encryption

For asymmetric encryption, the public key is openly available and is published using some form of key distribution system. The private key is secret and is used by the decrypting tool, such as ModelSim. Asymmetric methods are more secure than symmetric methods, but take much longer to encrypt and decrypt data.

The only asymmetric method ModelSim supports is:

rsa

This method is only supported for specifying key information, not for encrypting IP source code (i.e., only for key methods, not for data methods).

For testing purposes, ModelSim also supports raw encryption, which doesn't change the protected source code (the simulator still hides information about the protected region).

All encryption algorithms (except raw) produce byte streams that contain non-graphic characters, so there needs to be an encoding mechanism to transform arbitrary byte streams into portable sequences of graphic characters which can be used to put encrypted text into source files. The encoding methods supported by ModelSim are:

- uuencode
- base64
- raw

Base 64 encoding, which is technically superior to uuencode, is the default encoding used by ModelSim, and is the recommended encoding for all applications.

Raw encoding must only be used in conjunction with raw encryption for testing purposes.

How Encryption Envelopes Work

Encryption envelopes work as follows:

1. The encrypting tool generates a random key for use with a symmetric method, called a “session key.”
2. The IP protected source code is encrypted using this session key.
3. The encrypting tool communicates the session key to the decrypting tool —which could be ModelSim or some other tool — by means of a `KEY_BLOCK`.
4. For each potential decrypting tool, information about that tool must be provided in the encryption envelope. This information includes the owner of the key (`key_keyowner`), the name of the key (`key_keyname`), the asymmetric method for encrypting/decrypting the key (`key_method`), and sometimes the key itself (`key_public_key`).
5. The encrypting tool uses this information to encrypt and encode the session key into a `KEY_BLOCK`. The occurrence of a `KEY_BLOCK` in the source code tells the encrypting tool to generate an encryption envelope.
6. The decrypting tool reads each `KEY_BLOCK` until it finds one that specifies a key it knows about. It then decrypts the associated `KEY_BLOCK` data to determine the original session key and uses that session key to decrypt the IP source code.

Note



VHDL encryption requires that the `KEY_BLOCK` (the sequence of `key_keyowner`, `key_keyname`, and `key_method` directives) end with a ``protect KEY_BLOCK` directive.

Using Public Encryption Keys

If IP authors want to encrypt for third party EDA tools, other public keys need to be specified with the `key_public_key` directive as follows.

For Verilog and SystemVerilog:

```
\pragma protect key_keyowner="Acme"
\pragma protect key_keyname="AcmeKeyName"
\pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxw1RUUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

For VHDL:

```
\protect key_keyowner="Acme"
\protect key_keyname="AcmeKeyName"
\protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxw1RUUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

This defines a new key named “AcmeKeyName” with a key owner of “Acme.” The data block following `key_public_key` directive is an example of a base64 encoded version of a public key that should be provided by a tool vendor.

Using the Mentor Graphics Public Encryption Key

The Mentor Graphics base64 encoded RSA public key is:

```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxw1RUUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

For Verilog and SystemVerilog applications, copy and paste the entire Mentor Graphics key block, as follows, into your code:

```
\pragma protect key_keyowner = "Mentor Graphics Corporation"
\pragma protect key_method = "rsa"
\pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
\pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxw1RUUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

The `vencrypt` utility will recognize the Mentor Graphics public key. If `vencrypt` is not used, you must use the `+protect` switch with the `vlog` command during compile.

For VHDL applications, copy and paste the entire Mentor Graphics key block, as follows, into your code:

```
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_method = "rsa"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_public_key
MIGfMA0GCSqGSIB3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

The [vhencrypt](#) utility will recognize the Mentor Graphics public key. If [vhencrypt](#) is not used, you must use the **+protect** switch with the [vcom](#) command during compile.

[Example 3-4](#) illustrates the encryption envelope methodology for using this key in Verilog/SystemVerilog. With this methodology you can collect the public keys from the various companies whose tools process your IP, then create a template that can be included into the files you want encrypted. During the encryption phase a new key is created for the encryption algorithm each time the source is compiled. These keys are never seen by a human. They are encrypted using the supplied RSA public keys.

Example 3-4. Using the Mentor Graphics Public Encryption Key in Verilog/SystemVerilog

```
//
// Copyright 1991-2009 Mentor Graphics Corporation
//
// All Rights Reserved.
//
// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS THE
PROPERTY OF
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//

`timescale 1ns / 1ps
`celldefine

module dff (q, d, clear, preset, clock); output q; input d, clear, preset, clock;
reg q;

`pragma protect data_method = "aes128-cbc"
`pragma protect key_keyowner = "Mentor Graphics Corporation"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIB3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB

`pragma protect key_keyowner = "XYZ inc"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "XYZ-keyPublicKey"
`pragma protect key_public_key
MIGfMA0GCSqGSIB3DQEBAQUAA4GNADCBiQKBgQDZQTj5T5j0log8ykyaxVg9B+4V+smyCJGW36ZjoEGq
6jXHxfqB2VAmIC/j9x4xRxtCaOeBxRpcrnIKTP13Y3ydHqpYW0s0+R4h5+cMwCzWqB18Fn0ibSEW+8gw/
/BP4dHzaJApEz2Ryj+IG3UinvvWVNheZd+j0ULHGMgrOQqrwIDAQAB

`pragma protect begin
always @(clear or preset)
  if (!clear)
    assign q = 0;
  else if (!preset)
    assign q = 1;
`pragma protect end
```

```
    else
      deassign q;
    `pragma protect end
    always @(posedge clock)
      q = d;

  endmodule

`endcelldefine
```


Chapter 4

Optimizing Designs with vopt

ModelSim, by default, performs built-in optimizations on your design to maximize simulator performance. These optimizations yield performance improvements over non-optimized runs. The optimizations will limit the visibility of design objects, but you can increase visibility of any objects for debugging purposes, as described in the section "[Preserving Object Visibility for Debugging Purposes](#)."

The command that performs global optimizations in ModelSim is called `vopt`. This chapter discusses the `vopt` functionality, the effects of optimization on your design, and how to customize the application of `vopt` to your design. For details on syntax and usage of this command, please refer to `vopt` in the Reference Manual.

Optimization Flows

There are two basic flows that you can use to control optimizations for your simulation run.

- [Three-Step Flow](#) — where you perform compilation, optimization, and simulation in three separate steps.
- [Two-Step Flow](#) — where you perform compilation and simulation in two separate steps and optimization is implicitly run prior to simulation.

Three-Step Flow

The three-step flow allows you to have the most control over the optimization process, where the steps refer to the following:

- Compilation — `vcom` or `vlog`
- Optimization — `vopt`
 - The optimization step, using the `vopt` command, requires you to specify the name of the generated output by using the `-o` switch. Refer to the section "[Naming the Optimized Design](#)" for additional information. You can use this optimized output for many simulation runs.
- Simulation — `vsim`

The three-step flow allows you to use ModelSim for several purposes including:

- Preoptimized Design Units — reduce the amount of time necessary for future optimization and simulation runs by preoptimizing (black-boxing) regions of your

design using the `-pdu` option, as described in the section "[Preoptimizing Regions of Your Design](#)."

- Performing a simulation for debug — preserve the highest level of visibility by specifying the `+acc` argument to `vopt`. For example:

```
vlog -work <required_files>
vopt +acc top -o dbugver
vsim dbugver
```

- Performing a simulation for regression — reduce the amount of visibility because you are not as concerned about debugging. For example:

```
vlog -work <required_files>
vopt top -o optver
vsim optver
```

Naming the Optimized Design

When using the `vopt` command, you must provide a name for the optimized design using the `-o` argument:

```
vopt testbench -o opt1
```

Note



The filename must not contain capital letters or any character that is illegal for your platform (for example, on Windows you cannot use “\”).

Incremental Compilation of Named Designs

The default operation of `vopt -o <name>` is incremental compilation: ModelSim reuses elements of the design that have not changed, resulting in a reduction of runtime for `vopt` when a design has been minimally modified.

Preserving Object Visibility for Debugging Purposes

For a debugging flow you can preserve object visibility by using the `+acc` switch to the `vopt` command. The `+acc` switch specifies which objects are to remain "accessible" for the simulation. The following examples show some common uses of the `vopt +acc` combination, refer to the [vopt](#) reference page for a description of all `+acc` options:

- Preserve visibility of all objects in the design by specifying no arguments to `+acc`:

```
vopt +acc mydesign -o mydesign_opt
```

- Preserve visibility of all objects in a specific module by specifying the name of the module as an argument to `+acc`:

```
vopt top +acc+mod1 mydesign -o mydesign_opt
```

- Preserve visibility of only registers (`=r`) within a specific module:

vopt top +acc=r+mod1 mydesign -o mydesign_opt

- Preserve access to nets (n), ports (p) and registers (r) for 3 levels downwards from a specific level of hierarchy in a design (*top.netlist1*):

vopt top +acc=npr3+top.netlist1 mydesign -o mydesign_opt

The examples in this section assume that you have set the [PathSeparator](#) variable to a period (.) for a Verilog environment.

- Preserve port access to a specific level of hierarchy in a design (*top.netlist2*):

vopt top +acc=p+top.netlist2 mydesign -o mydesign_opt

- Preserve port access recursively downward from a specific level of hierarchy in a design (*top.netlist2*):

vopt top +acc=p+top.netlist2. mydesign -o mydesign_opt

- Preserve visibility for all instances of a particular VHDL design region (*ent1*):

vopt top +acc=+ent1 mydesign -o mydesign_opt

- Preserve visibility of line numbers (=l) in addition to registers within a specific module:

vopt top +acc=lr+mod1 mydesign -o mydesign_opt

- Preserve visibility of line numbers and registers within a specific module and all children in that module by adding a period (.) after the module name:

vopt top +acc=lr+mod1. mydesign -o mydesign_opt

- Preserve visibility of a unique instance:

vopt +acc=mrp+top.u1 mydesign -o mydesign_opt

- Preserve visibility of a unique object:

vopt +acc=r+top.myreg mydesign -o mydesign_opt

Using an External File to Control Visibility Rules

You can use the -f switch to specify a file that contains your +acc arguments. This is most useful when you have numerous +acc arguments that you use regularly or because you provide a very fine control of visibility. For example:

vopt -f acc_file.txt mydesign -o mydesign_opt

where *acc_file.txt* contains:

```
// Add the following flags to the vopt command line.
+acc=rn+tb
+acc=n+tb.dut.u_core
+acc=pn+tb.dut.u_core.u_sub
+acc=pn+tb.dut.u_core.u_sub.u_bp
+acc=rpn+tb.dut.u_core.u_sub.u_bb.U_bb_compare
+acc=pn+tb.dut.u_core.u_sub.u_bb.U_bb_control
```

```
+acc=r+tb.dut.u_core.u_sub.u_bb.U_bb_control.U_bb_regs  
+acc=rpn+tb.dut.u_core.u_sub.u_bb.U_bb_delay0
```

This example assumes that you have set the [PathSeparator](#) variable to a period (.) for a Verilog environment.

Creating Specialized Designs for Parameters and Generics

You can use [vopt](#) to create specialized designs where generics or parameters are predefined by using the `-g` or `-G` switches, as shown in the following example:

```
vopt top -G TEST=1 -o test1_opt  
vopt top -G TEST=2 -o test2_opt  
  
vsim test1_opt  
vsim test2_opt
```

Increase Visibility to Retain Breakpoints

When running in full optimization mode, breakpoints can not be set. To retain visibility of breakpoints you should set the `+acc` option such that the object related to the breakpoint is visible.

Two-Step Flow

The two-step flow allows you to perform design optimizations using existing scripts, in that `vsim` automatically performs optimization. The two steps refer to the following:

1. Compile — `vcom` or `vlog` compiles all your modules.
2. Simulate — `vsim` performs the following actions:
 - a. Load — Runs `vopt` in the background when it loads the design.

You can pass arguments to `vopt` using the `-voptargs` argument to `vsim`. For example,

```
vsim mydesign -voptargs="+acc=rn"
```

The optimization step of `vsim` loads compiled design units from their libraries and regenerates optimized code.

- b. Simulate — Runs `vsim` on the optimized design unit.

Because `vopt` is called implicitly when using the two-step flow, ModelSim creates an optimized internal design for simulation. By default, the maximum number of these designs is set to three, after which `vsim` execution removes the oldest optimized design and creates a new one. You can increase this limit by using the `-unnamed_designs` argument to `vlib`. Because the `vsim` command manages `unnamed_designs` you cannot use the `-o` argument in the `-voptargs` specification to name an optimized design. For example, `vsim mydesign -voptargs="-o myoptdesign"` will generate an error message.

Preserving Object Visibility in the Two-Step Flow

With the three-step flow you can preserve object visibility by using the `+acc` argument to the `vopt` command as described in the section [Preserving Object Visibility for Debugging Purposes](#). With the two-step flow, you implement this same functionality with the `-voptargs` switch to the `vsim` command, which passes the arguments to the automatic invocation of `vopt`.

The following are some examples of how to pass optimization arguments from the `vsim` command line:

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

```
vsim -voptargs="+acc+mod1" mydesign
```

```
vsim -voptargs="+acc=rnl" mydesign
```

Using vopt and the -O Optimization Control Switches

The [Three-Step Flow](#) and [Two-Step Flow](#) are the primary use models for ModelSim performance architecture. These `vopt` flows allow for global visibility of the design, which in turn allows the compiler to make aggressive optimization decisions based on its design-wide knowledge.

The `vopt` command also allows the use of the `-O` optimization control switches. These switches control the aggressiveness of optimization decisions. Note, however, that these decisions are not particularly related to the global visibility aspects of running `vopt`. The `+acc` switch is more related to those, and is used to preserve visibility to certain categories of objects that might otherwise be optimized away. Objects that get optimized away can make your debug and analysis efforts more difficult.

You can also use the numbered `-O` switches on the `vcom` or `vlog` command lines to control optimizations independently from `vopt`.

The `-O0` and `-O1` switches can negatively impact performance and you should only use them if you are attempting to analyze the behavior of the simulator. If you require increased visibility for objects that are being optimized out of the simulation, use the `vopt +acc` functionality instead.

The `-O5` switch enables more aggressive native-code generation which can speed up many designs, but it can slow down others.

In code coverage flows, you should use the `vlog`, `vcom`, and `vopt -coveropt` switch (or the [CoverOpt](#) `modelsim.ini` variable) to control visibility and other interactions between optimization and code coverage collection.

Inlining and the Implications of Coverage Settings

When you recompile an inlined module with different coverage settings (e.g., +cover) than a previous compile, it forces a recompile of its inline parent modules. This is due to the fact that inlined modules inherit the coverage settings of their parent. This can have the effect of increasing expected compile time.

If a module has different coverage settings than its parent, that module will not be inlined.

Optimizing Parameters and Generics

During the optimization step you have several options on how parameters and generics affect the optimization of the design:

- **Override** — you can override any design parameters and generics with either the -G or -g switches to the vopt command. ModelSim optimizes your design based on how you have overridden any parameters and generics.

Once you override a parameter or generic in the optimization step you will not be able to change its value during the simulation. Therefore, if you attempt to override these same generics or parameters during the simulation, ModelSim will ignore those specifications of -g/-G.

```
vopt -o opt_top top -G timingCheck=1 -G top/a/noAssertions=0
```

The IEEE Std 1800-2005 for SystemVerilog places some limits on when you cannot override parameters. You will not be able to override parameters with the -g, -G, or +floatparameters switches in the following scenarios:

- The LRM states that local parameters (localparam) cannot be overridden.
 - The LRM states that you cannot specify a parameter in a generate scope, and that if one exists, it should be treated as a localparam statement.
 - The LRM does not provide a mechanism for overriding parameters declared inside a package or \$unit, and that if one exists, it should be treated as a localparam statement.
- **Float** — you can specify that parameters and generics should remain floating by using the +floatparameters or +floatgenerics switches, respectively, to the vopt command. ModelSim will optimize your design, retaining any information related to these floating parameters and generics so that you can override them during the simulation step.

```
vopt -o opt_top top +floatparameters+timingCheck+noAssertions
```

The +floatgenerics or +floatparameters switches do affect simulation performance. If this is a concern, it is suggested that you create an optimized design for each generic or parameter value you may need to simulate. Refer to the section "[Creating Specialized Designs for Parameters and Generics](#)" for more information.

- **Combination** — you can combine the use of the `-g/-G` and `+floatparameters/+floatgenerics` with the `vopt` command to have more control over the use of parameters and generics for the optimization and simulation steps.

Due to the fact that `-g/-G` and `+floatparameters/+floatgenerics` allow some use of wildcards, ambiguities could occur. If, based on your options, a parameter or generic is considered floating and also is overridden the override value take precedence.

```
vopt -o opt_top top +floatparameters+timingCheck
                                -G top/a/noAssertions=0
```

- **No Switches** — if you do not use any of the above scenarios, where you do not use `-g/-G` or `+floatparameters/+floatgenerics`, ModelSim optimizes the design based on how the design defines parameter and generic values. Because of optimizations performed, you may lose the opportunity to override any parameters or generics of the optimized design at simulation time.

If your design contains a Preoptimized Design Unit (black-boxed region) the `-g/-G` switches will override any floating parameters or generics in the PDU region. For example:

```
vopt -pdu -o dut_design dut +floatparameters+design.noAssertions
    ### creates a Preoptimized Design Unit of dut with design.noAssertions
    floating

vopt -o test_design test -G noAssertions=0
    ### the design test uses the PDU portion dut
    ### the vopt command overrides any occurrence of noAssertions,
    ### including the one in dut.

vsim test_design
    ### performs the simulation where noAssertions is set to 0.
```

Refer to the section [Preoptimizing Regions of Your Design](#) for more information.

Preoptimizing Regions of Your Design

The `vopt` command allows you to specify the `-pdu` argument (Preoptimize Design Unit), which instructs `vopt` to preoptimize (black-box) a region of your design. This feature is useful for providing better throughput by allowing you to optimize large portions of your design that may be static or not changing. For an example refer to [Simulating Designs with Several Different Test Benches](#).

For any future use of this preoptimized region, ModelSim automatically recognizes and uses the PDU of the design, which reduces the runtime of `vopt`.

When you are using `vopt -pdu`, you should associate the optimized name with the original name using the `-o` argument. For example:

```
vopt moda -pdu -o moda_pdu_opt
```

For the above example, any design that contains an instantiation of the module *moda*, ModelSim runs a design analysis and automatically includes the Preoptimized Design Unit *moda_bb_opt*.

When using this method, you should be aware of the following:

- When you instantiate a region that has been preoptimized (black-boxed), you do not need to run vopt on the top level module.
- During optimization, ModelSim does not descend into the PDU, allowing faster operation. However, parameters passing and hierarchical references across the PDU are restricted. You can retain visibility into a PDU by using the `-pdusavehierrefs` argument to vopt, but it can reduce simulation performance.
- You will need to manage both the original portion (*moda*) and its optimized version (*moda_pdu_opt*). Specifically, you must not remove the optimized version without also removing or recompiling the original version.

Simulating Designs with Several Different Test Benches

For multiple test benches, you would use vopt and `-pdu` to optimize the design. Then you could use the [Three-Step Flow](#) on the different test benches, which prevents having to optimize the design for each test bench. For example:

```
1  vlib work
2  vlib asic_lib
3  vlog -work asic_lib cell_lib.v
4  vlog netlist.v
5  vopt -L asic_lib -debugCellOpt +checkALL -pdu netlist -o opt_netlist

6  vlog tb.v test1.v
7  vopt tb -o opt_tb
8  vsim -c opt_tb -do sim.do

9  vlog test2.v
10 vopt tb -o opt_tb
11 vsim -c opt_tb -do sim.do
```

- Lines 3 and 4 — compile the library and netlist.
- Line 5 — enable the PDU feature and optimize the netlist.
- Line 6 — compile the remainder of the design.
- Line 7 — optimize the test bench.
- Line 8 — simulate the first test bench.
- Lines 9 through 11 — compile and optimize a second test and resimulate without recompiling or optimizing the PDU netlist.

Using Configurations with Preoptimized Design Units

This section provides simple examples for using Verilog and VHDL configurations with the `-pdu` argument to `vopt`.

Verilog Configuration Example

The following files show a simplified way for you to set up a Verilog configuration with the PDU feature of `vopt`.

topcfg.v	top.v	foo.v
<pre>config topcfg; design work.top; instance top.u0 use foo10; instance top.u1 use foo20; endconfig</pre>	<pre>module top; foo u0(); foo u1(); endmodule</pre>	<pre>module foo10; foo_lower #10 u0(); endmodule module foo20; foo_lower #20 u0(); endmodule module foo_lower; parameter N=99; initial begin \$display("N=%d", N); end endmodule</pre>

Given these files, you can use the following commands to simulate your design using Preoptimized Design Units (black-boxed regions).

```
1  vlib work
2  vlog foo.v
3  vopt -pdu foo10 -o foo10_pdu
4  vopt -pdu foo20 -o foo20_pdu
5  vlog top.v
6  vlog topcfg.v
7  vsim topcfg -do "run -all"
```

where each line does the following:

1. Creates the work library.
2. Compiles `foo.v` into the **work** library.
3. Creates a PDU named **foo10_pdu** based on the **foo10** module in `foo.v`. Whenever a part of the design is dependent upon **foo10**, the simulator will load the PDU **foo10_pdu** to speed up the elaboration process.
4. Creates a PDU named **foo20_pdu** based on the **foo20** module in `foo.v`.

5. Compiles *top.v* into the **work** library.
6. Compiles the configuration, *topcfg.v*, into the **work** library.
7. Elaborates the configuration and performs the simulation. When the simulator encounters **top.u0** it will use **foo10** (as defined in the configuration), at which point the simulator will use your PDU **foo10_pdu** (and similarly for **top.u1** and **foo20**).

VHDL Configuration Example

The following files show a simplified way for you to set up a VHDL configuration with the Preoptimized Design Unit feature of vopt.

topcfg.vhd

```
configuration topcfg of top is
  for arch
    for u0 : foo_comp
      use entity
work.foo10(arch);
    end for;
    for u1 : foo_comp
      use entity
work.foo20(arch);
    end for;
  end for;
end configuration;
```

top.vhd

```
entity top is
end top;

architecture arch of top is
  component foo_comp
  end component;
begin
  u0 : foo_comp;
  u1 : foo_comp;
end arch;
```

foo.vhd

```
entity foo_lower is
  generic ( N : integer := 99 );
end foo_lower;

architecture arch of foo_lower is
begin
  p1 : process
  begin
    assert false
      report "in " & p1'instance_name & ",
          N = " & integer'image(N)
      severity note;
    wait;
  end process;
end arch;

entity foo10 is
end foo10;

architecture arch of foo10 is
  component foo_lower
    generic ( N : integer );
  end component;
begin
  u0 : foo_lower
    generic map (N => 10);
end arch;

entity foo20 is
end foo20;

architecture arch of foo20 is
  component foo_lower
    generic ( N : integer );
  end component;
begin
  u0 : foo_lower
    generic map (N => 20);
end arch;
```

Given these files, you can use the following commands to simulate your design using a Preoptimized Design Unit.

```
1  vlib work
2  vcom foo.vhd
3  vopt -pdu foo10 -o foo10_pdu
4  vopt -pdu foo20 -o foo20_pdu
5  vcom top.vhd
6  vcom topcfg.vhd
7  vsim topcfg -do "run -all"
```

where each line does the following:

1. Creates the work library.
2. Compiles *foo.vhd* into the **work** library.
3. Creates a Preoptimized Design Unit (black-box) named **foo10_pdu** based on the **foo10** module in *foo.vhd*. Whenever a part of the design is dependent upon **foo10**, the simulator will load the optimized design unit **foo10_pdu** to speed up the elaboration process.
4. Creates a PDU named **foo20_pdu** based on the **foo20** module in *foo.vhd*.
5. Compiles *top.vhd* into the **work** library.
6. Compiles the configuration, *topcfg.vhd*, into the **work** library.
7. Elaborates the configuration and performs the simulation. When the simulator encounters **u0 : foo_comp** it will use **foo10** (as defined in the configuration), at which point the simulator will use your Preoptimized Design Unit **foo10_pdu** (and similarly for **u1 : foo_comp** and **foo20**).

Resolving Preoptimized Design Unit Loading Errors

Compiling libraries and optimizing design units on one machine, then simulating the design on another machine (such as in a grid environment) can lead to the following scenario:

```
# vsim -L dut -L cells -c top
# Loading work.top(fast)
# Loading work.dut_post(fast)
# ** Warning: (vsim-56) The dependency file generated by the "vopt" tool has
# either # been removed, is for an older version of the tool or has been corrupted.
# Normally # re-running vopt on the design will recreate this file and solve the
# issue.
# "/tmp/build1/test/libs/cells/post_cell_bb/_deps" - file open failed.
# ** Error: (vsim-166) Failed to load a Preoptimized Design Unit(black-box)
# created using -pdu. post_cell_bb could not be loaded from
# library '/tmp/build1/test/libs/cells'.
#       Region: /top/u1
# Error loading design
```

Preoptimized Design Units do not allow mapping of libraries inside the PDUs, though mapping can be used to find the upper-level PDU library. There are several reasons for common message number 166:

- a dependency file generated by the **vopt** tool could not be found, or was generated by an older tool version.

Rerunning vopt on the design will recreate the PDU and resolve the issue.

- the physical path to a library containing a Preoptimized Design Unit (PDU) no longer points to the library.

This can occur in nested PDUs (a PDU containing a PDU). This is due to a PDU fixing all logical references (including library mappings) to physical references below it. Soft links can be used to resolve these physical links if necessary.

Specifying Coverage During Optimization

The `+cover` and `+nocover` options for the `vopt` command allow you to specify coverage to the entire design or to specific design units and instances. Refer to the Reference Manual for the proper syntax.

In cases where more than one `+cover/+nocover` options are applied to an instance and conflict, this is resolved according to the following priority:

- The option applied to the instance or design unit has higher priority than the one inherited from its parent.
- The option applied to the instance has higher priority than the one applied to the same design unit.
- If no option is specified to an instance or design unit, it inherits the option setting from its immediate parent.
- The options specified without a specific design unit or instance name (`<selection>`) have the least priority. These are options specified to `vlog/vcom` or to `vopt` without `<selection>`. The `+cover` options specified this way are collected together. The `+nocover` option specified with `vopt`, stops collecting coverage of the specified kind.
- In case of conflicting options of the same priority, the option specified later in the `vopt` command-line overrides the option specified before.

Alternate Optimization Flows

The following sections describe usage flows for optimization that use variations of the Three- and Two-Step Flows. The Three-Step Flow is recommended for primary use, but these alternatives may be useful in your environment.

Creating Locked Libraries for Multiple-User Simulation Environments

In some cases, you and other users may require access to the same library. One conflict that can arise from this scenario is that another user may try to recompile some design units in that library, which can negatively affect your environment.

You can prevent users from recompiling the library by using the `-locklib` switch to the `vlib` command. The `-locklib` switch prevents any alteration of a library.

Procedure

1. Create the library:

```
vlib work
```

2. Compile design units into the library:

```
vlog top.v a.v b.v  
vlog tb.v
```

3. Create optimized versions of your design:

```
vopt +acc top_tb -o opt_fullVis  
vopt top_tb -o opt_Regression  
vopt +acc +cover top_tb -o opt_Cover
```

You must create any optimized designs before locking the library, otherwise the `vopt` command will issue the following error:

```
# ** Fatal: (vopt-1991) Library "\user\design\work" cannot be  
modified due to a lock.
```

4. Lock the library:

```
vlib -locklib work
```

Once the library is locked, you will not be able to alter the library in any way; including the `vlib`, `vcom`, `vopt`, and `vdel` commands.

You can ensure that the library is locked by using the `vdir` command, which returns information about the library, including the following line:

```
...  
# Library locked/unlocked : locked  
...
```

If you need to recompile a design unit or create a new optimized design, you can unlock the library as follows:

```
vlib -unlocklib work
```

5. Alternatively, you can lock individual design units:

```
vlib -lock top work
vlib -lock a work
```

Optimizing Liberty Cell Libraries for Debugging

To debug designs with Liberty logic cells, you must specify the location of the Liberty cell library file before or during optimization. To set the location of the Liberty library, you can do one of the following:

- Set the [MTL_LIBERTY_PATH](#) environment variable.
- Optimize your design by specifying:

```
vopt -libertyfiles=<file_name> -debugdb
```

Liberty Library Usage Flow

The following command sequence shows basic usage of a Liberty library:

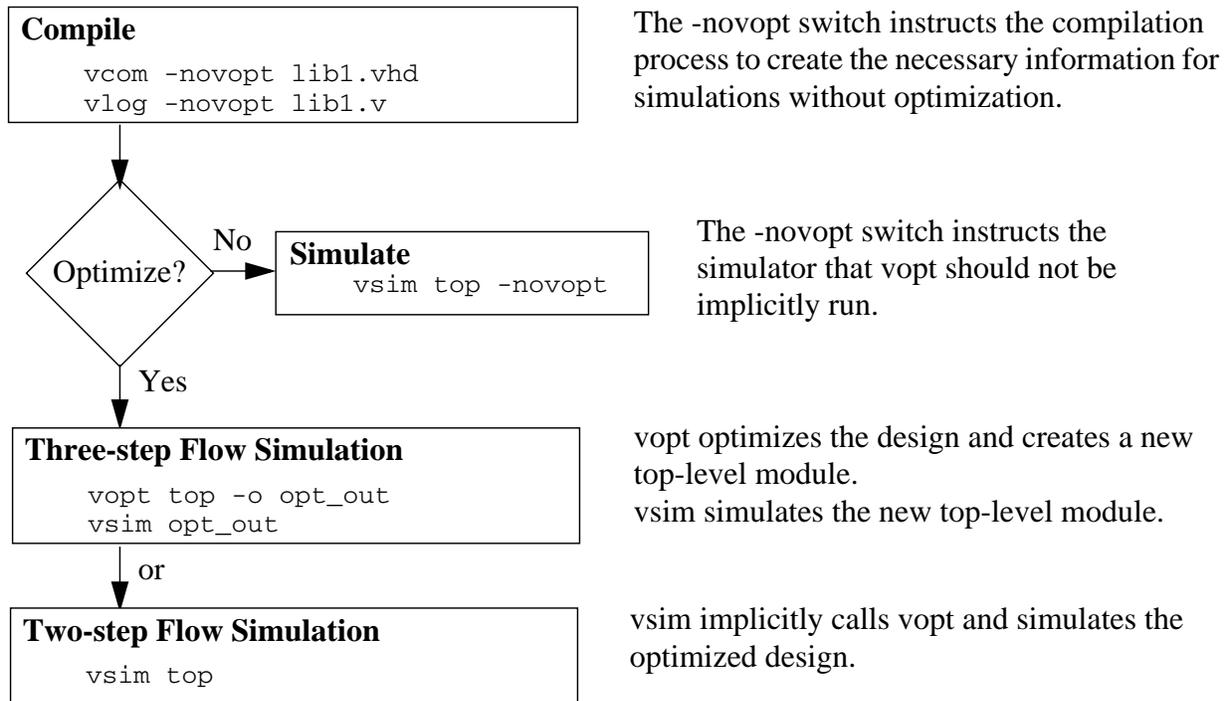
```
vlog design.v
vopt -o opt tb -libertyfiles=cells.lib -debugdb ...
vsim -c opt -debugdb ...
```

Refer to [Liberty Library Models](#) for more information on using Liberty libraries.

Creating an Environment for Optimized and Unoptimized Flows

Some work environments require that you decide whether to simulate using optimizations or no optimizations. The following diagram outlines one process for helping you make this decision.

A more efficient process is to use the [Three-Step Flow](#) and use +acc for [Preserving Object Visibility for Debugging Purposes](#).



Preserving Design Visibility with the Learn Flow

To ensure that you retain the proper level of design visibility when performing an optimized simulation (using vopt in the [Three-Step Flow](#)) you can use the -learn switch to vsim, which creates control files that include instructions for preserving visibility.

The control files created in this flow allow you to retain information during optimization for the following:

- ACC/TF PLI routines
- VPI PLI routines
- SignalSpy accesses
- force Run-time command
- FLI instances (regions), signals, ports, and variables.
- Objects specified as arguments to commands executed after -learn is started, such as add wave /top/p/*.

The following steps describe the use of the Learn Flow for preserving visibility using PLI routines:

1. Invoke the simulator

```
vsim -novopt -learn top_pli_learn -pli mypli.sl top
```

When you specify the `-learn` switch, where an argument defines the root name (`top_pli_learn`) of the generated control files, `vsim` analyzes your design as well as your PLI to determine what information needs to be retained during the optimization. Based on this analysis it then creates the following control files and places them in the current directory:

```
top_pli_learn.acc  
top_pli_learn.ocf  
top_pli_learn.ocm
```

Refer to the section “[Description of Learn Flow Control Files](#)” for a description of these files.

The learn flow is sensitive to the `PathSeparator` variable in the `modelsim.ini` file at the time of creation of the control files. Be sure to use a consistent `PathSeparator` throughout this flow.

2. Run the simulation to generate the control files (`.acc`, `.ocf`, and `.ocm`).

```
run <time_step><time_unit>
```

When running the simulation, the Learn Flow tracks and records the objects required for your PLI routines or used for commands executed before or after the run, for which you need to retain visibility. It is difficult to suggest how long you should run the simulation; your knowledge of the design and test bench should allow a guideline for you to follow.

To ensure that the simulator records every possible access, you should run a complete simulation (`run -all`).

Because the files are saved at the end of simulation, you should not restart or restore the simulation when working with the Learn Flow.

3. Create an optimized design, retaining the visibility as defined in the control files. You can determine which type of control file you wish to use. A command line example for each type include:

```
vopt -f top_pli_learn.acc -o top_opt  
vopt -ocf top_pli_learn.ocf -o top_opt  
vopt -ocf top_pli_learn.ocm -o top_opt
```

`vopt` creates the optimized design, `top_opt`, and retains visibility to the objects required by your PLI routines.

4. Simulate the optimized design.

```
vsim -pli mypli.sl top_opt
```

This performs the simulation on the optimized design, where you retained visibility to the objects required by your PLI routines.

Description of Learn Flow Control Files

The control files for the learn flow are text files that instruct vopt to retain visibility to objects required by the specified PLI routines. All three file formats are considered to be non-lossy, in that information about every object touched by the PLI during the -learn run is retained.

- .acc Learn Flow control file — This format (.acc) creates the information in the traditional +acc format used by the vopt command. However, this format does not allow for precise targeting of objects that you can get with the .ocf format.
- .ocf Learn Flow control file — This format (.ocf) is the most verbose and precisely targeted of the three control files. It is suggested that you use this file for situations where there is sparse access to objects. If you access every object in a module, this file can get considerably large.
- .ocm Learn Flow control file — This format (.ocm) is similar to the .ocf format, except that the file is factorized by design unit, which results in a smaller and more easily read file, but provides less precise targeting.

These files are text-based and can be edited by anyone.

Controlling Optimization from the GUI

Optimization (**vopt**) in the GUI is controlled from the **Simulate > Design Optimization** dialog box.

To restore total design visibility from within the GUI:

1. Select **Simulate > Design Optimization > Visibility** tab
2. Select “Apply full visibility to all modules (full debug mode)”
3. Select Design tab and select the top-level design unit to simulate
4. Specify an Output Design Name.
5. Select Start Immediately and then click OK.

Optimization Considerations for Verilog Designs

The optimization considerations for Verilog designs include:

- [Design Object Visibility for Designs with PLI](#)
- [Reporting on Gate-Level Optimizations](#)
- [Using Pre-Compiled Libraries](#)
- [Event Order and Optimized Designs](#)

- [Timing Checks in Optimized Designs](#)

Design Object Visibility for Designs with PLI

Some of the optimizations performed by **vopt** impact design object visibility. For example, many 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 use one or more **+acc** arguments in conjunction with **vopt** to enable access to specific design objects. See the **vopt** command in the Reference Manual for specific syntax of the **+acc** argument.

Automatic **+acc** for Designs with PLI

By default, if your design contains any PLI, and the automatic **vopt** flow is enabled, **vsim** automatically adds a **+acc** to the sub-invocation of **vopt**, which disables most optimizations.

If you want to override the automatic disabling of the optimizations for modules containing PLI, specify the **-no_autoacc** argument to **vsim**.

Manual **+acc** for Designs with PLI

If you are manually controlling **vopt** optimizations, and your design uses PLI applications that look for object handles in the design hierarchy, then it is likely that you will need to use the **+acc** option. For example, the built-in **\$dumpvars** system task is an internal PLI application that requires handles to nets and registers so that it can call the PLI routine **acc_vcl_add()** to monitor changes and dump the values to a VCD file. This requires that access is enabled for the nets and registers on which it operates.

Suppose you want to dump all nets and registers in the entire design, and that you have the following **\$dumpvars** call in your test bench (no arguments to **\$dumpvars** means to dump everything in the entire design):

```
initial $dumpvars;
```

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

```
vopt +acc=rn testbench
```

As another example, suppose you only need to dump nets (n) and registers (r) 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 optimize your design as follows (assuming *testbench.u1* is an instance of the module *design*):

```
vopt +acc=rn+design testbench
```

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 optimize your design as follows:

```
vopt +acc=rn+design. testbench
```

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

Performing Optimization on Designs Containing SDF

Both optimization flows ([Two-Step Flow](#) and [Three-Step Flow](#)) automatically perform SDF compilation using [sdfcom](#) if any of the following apply:

- `$sdf_annotate` system task exists in the test bench.
- `-sdfmin`, `-sdfmax`, or `-sdftyp` on the **vopt** command line in the Three-Step Flow
- `-sdfmin`, `-sdfmax`, or `-sdftyp` on the **vsim** command line in the Two-Step Flow

The following arguments to **vopt** are useful when you are dealing with SDF:

- **vopt +notimingchecks** — Allows you to simulate your gate-level design without taking into consideration timing checks, giving you performance benefits. For example:

```
vlog cells.v netlist.v tb.v
vopt tb -o tb_opt -O5 +checkALL +delay_mode_path +notimingchecks \
-debugCellOpt
vsim tb_opt
```

By default, **vopt** does not fix the `TimingChecksOn` generic in Vital models. Instead, it lets the value float to allow for overriding at simulation time. If best performance and no timing checks are desired, `+notimingchecks` should be specified with **vopt**.

```
vopt +notimingchecks topmod
```

Specifying **vopt +notimingchecks** or `-G TimingChecks=<FALSE/TRUE>` will fix the generic value for simulation. As a consequence, using **vsim +notimingchecks** at simulation may not have any effect on the simulation depending on the optimization of the model.

- **vopt { -sdfmin | -sdftyp | -sdfmax } [<instance>=]<sdf_filename>** — Annotates cells in the specified SDF files with minimum, typical, or maximum timing. This invocation will trigger the automatic SDF compilation.

- `vopt +check{ALL | AWA | CLUP | DELAY | DNET | INTRI | IPODOP | NWOT | OPRD | SUDP}` — Disables specific optimization checks (observe uppercase). Refer to the [vopt](#) reference page for details.

Reporting on Gate-Level Optimizations

You can use the `write cell_report` and the `-debugCellOpt` argument to the `vopt` command to obtain information about which cells have and have not been optimized.

`write cell_report` produces a text file that lists all modules.

```
vopt tb -o tb_opt -debugCellOpt
vsim tb_opt -do "write cell_report cell.rpt; quit -f"
```

Modules with "(cell)" following their names are optimized cells. For example,

```
Module: top
Architecture: fast

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

In this case, top was not optimized and bottom was.

Using Pre-Compiled Libraries

If the source code is unavailable for any of the modules referenced in a design, then you must search libraries for the precompiled modules using the `-L` or `-Lf` arguments to `vopt`. The `vopt` command 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 default ‘work’ library.

The `vopt` command automatically searches libraries specified in the ``uselib` directive (see [Verilog-XL uselib Compiler Directive](#)). If your design uses ``uselib` directives exclusively to reference modules in other libraries, then you do not need to specify library search arguments.

Event Order and Optimized Designs

The Verilog language does not require that the simulator execute simultaneous events in any particular order. Optimizations performed by `vopt` may expose event order dependencies that cause a design to behave differently than when run unoptimized. Event order dependencies are considered errors and should be corrected (see [Event Ordering in Verilog Designs](#) for details).

Timing Checks in Optimized Designs

Timing checks are performed whether you optimize the design or not. In general, you'll see the same results in either case. However, in a cell where there are both interconnect delays and conditional timing checks, you might see different timing check results.

- Without vopt — The conditional checks are evaluated with non-delayed values, complying with the original IEEE Std 1364-1995 specification. You can use the `-v2k_int_delays` switch with `vsim` to ensure compatibility by forcing the IEEE Std 1364-2005 implementation.
- With vopt — the conditional checks will be evaluated with delayed values, complying with the new IEEE Std 1364-2005 specification.

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

What are Projects?

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

- Source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see [Creating a Simulation Configuration](#))
- Folders (see [Organizing Projects with Folders](#))

Note



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

What are the Benefits of Projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don't need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project. Compile order is maintained for HDL-only designs.
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to source files

- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload the initial settings from the project *.mpf* file every time the project is opened

Project Conversion Between Versions

Projects are generally not backwards compatible for either number or letter releases. When you open a project created in an earlier version, you will see a message warning that the project will be converted to the newer version. You have the option of continuing with the conversion or cancelling the operation.

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

Getting Started with Projects

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

- [Step 1 — Creating a New Project](#)

This creates an *.mpf* file and a working library.

- [Step 2 — Adding Items to the Project](#)

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

- [Step 3 — Compiling the Files](#)

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

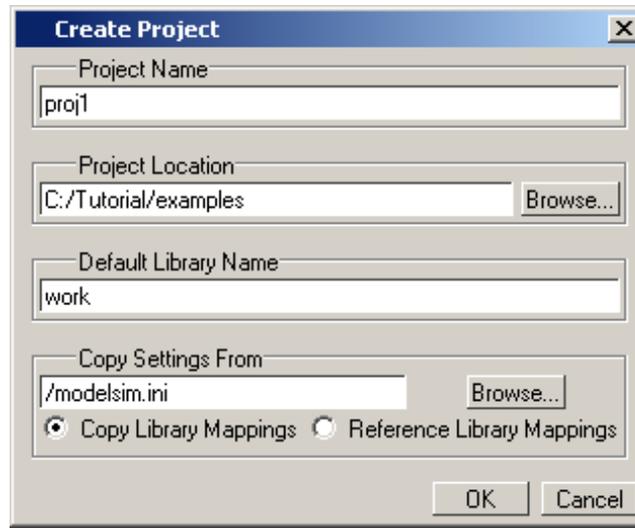
- [Step 4 — Simulating a Design](#)

This specifies the design unit you want to simulate and opens a structure tab in the Workspace pane.

Step 1 — Creating a New Project

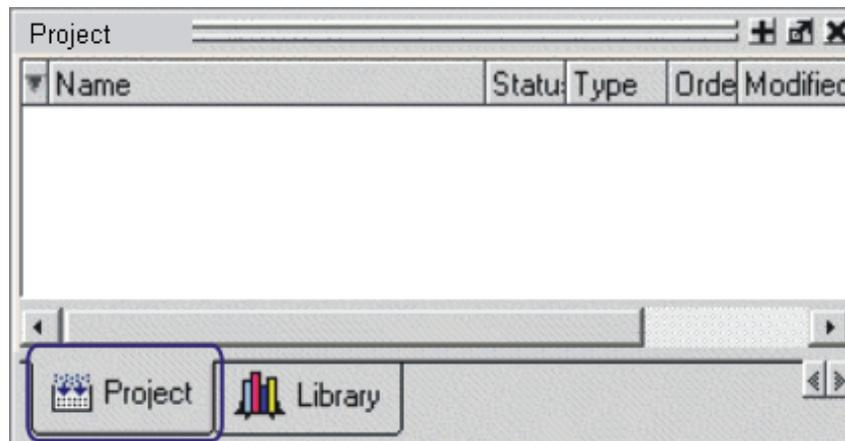
Select **File > New > Project** to create a new project. This opens the **Create Project** dialog where you can specify a project name, location, and default library name. You can generally leave the **Default Library Name** set to "work." The name you specify will be used to create a working library subdirectory within the Project Location. This dialog also allows you to reference library settings from a selected *.ini* file or copy them directly into the project.

Figure 5-1. Create Project Dialog



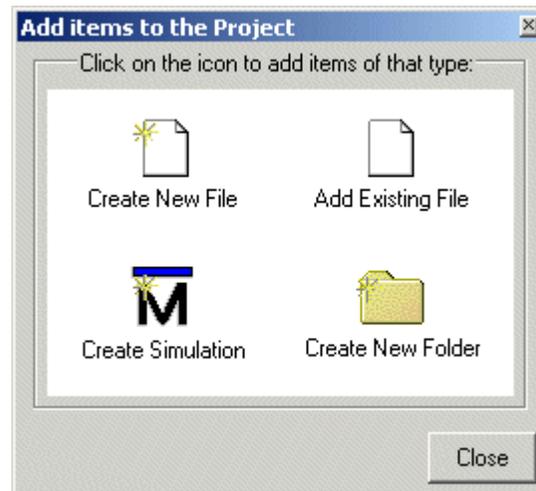
After selecting OK, you will see a blank Project window in the Main window ([Figure 5-2](#))

Figure 5-2. Project Window Detail



and the **Add Items to the Project** dialog ([Figure 5-3](#)).

Figure 5-3. Add items to the Project Dialog



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

Step 2 — Adding Items to the Project

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

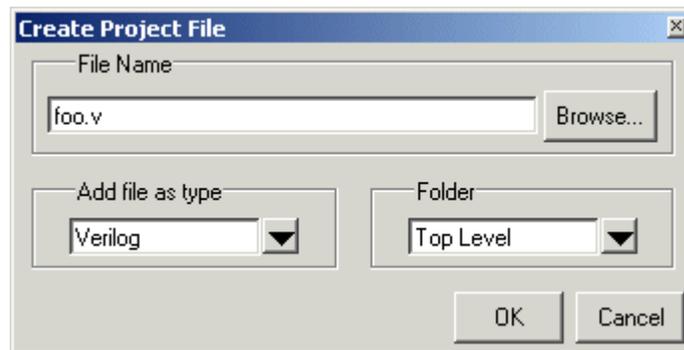
- **Create New File** — Create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor. See below for details.
- **Add Existing File** — Add an existing file. See below for details.
- **Create Simulation** — Create a Simulation Configuration that specifies source files and simulator options. See [Creating a Simulation Configuration](#) for details.
- **Create New Folder** — Create an organization folder. See [Organizing Projects with Folders](#) for details.

Create New File

The **File > New > Source** menu selections allow you to create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor.

You can also create a new project file by selecting **Project > Add to Project > New File** (the Project tab in the Workspace must be active) or right-clicking in the Project tab and selecting **Add to Project > New File**. This will open the Create Project File dialog ([Figure 5-4](#)).

Figure 5-4. Create Project File Dialog



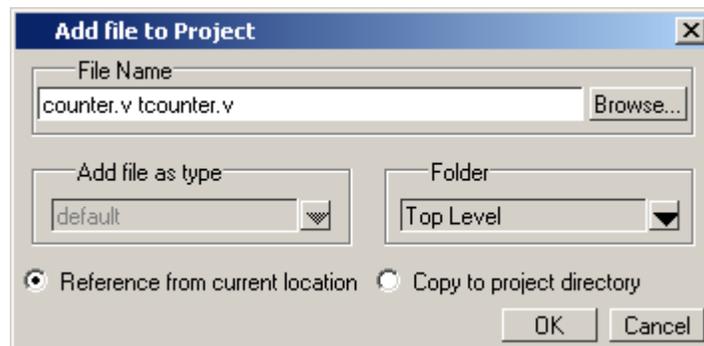
Specify a name, file type, and folder location for the new file.

When you select OK, the file is listed in the Project tab. Double-click the name of the new file and a Source editor window will open, allowing you to create source code.

Add Existing File

You can add an existing file to the project by selecting **Project > Add to Project > Existing File** or by right-clicking in the Project tab and selecting **Add to Project > Existing File**.

Figure 5-5. Add file to Project Dialog

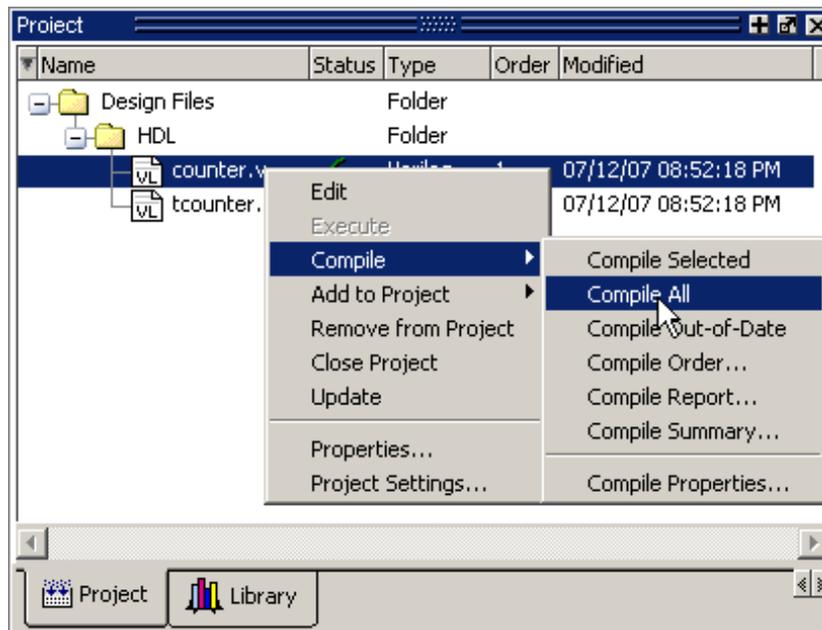


When you select OK, the file(s) is added to the Project tab.

Step 3 — Compiling the Files

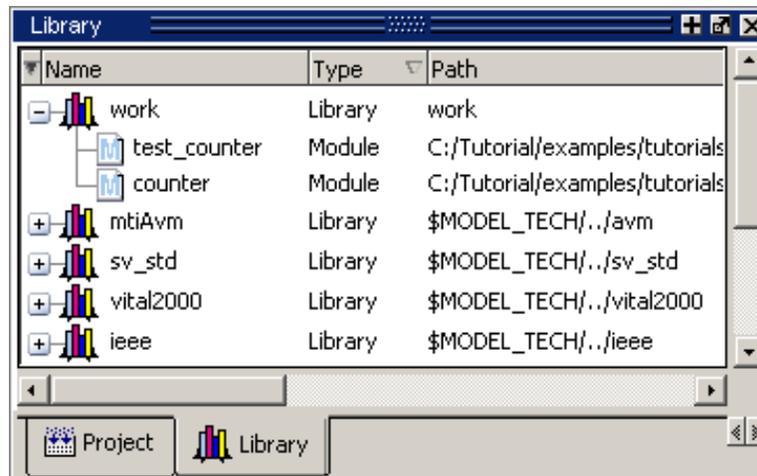
The question marks in the Status column in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile > Compile All** or right click in the Project tab and select **Compile > Compile All** (Figure 5-6).

Figure 5-6. Right-click Compile Menu in Project Window



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

Figure 5-7. Click Plus Sign to Show Design Hierarchy



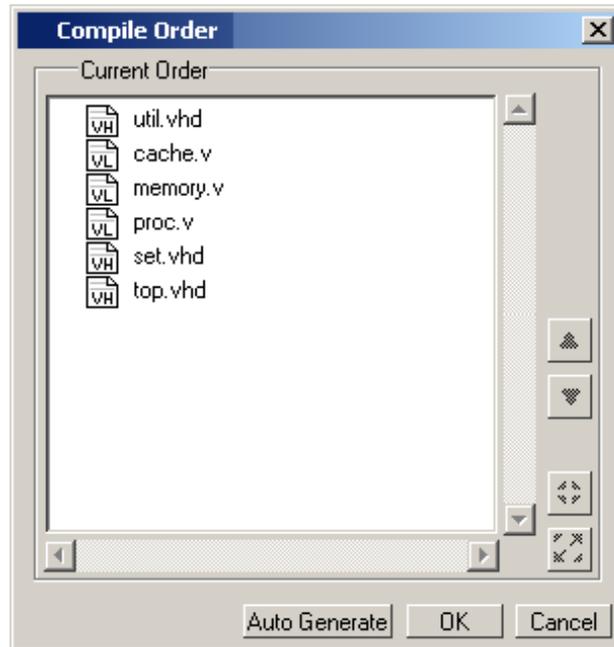
Changing Compile Order

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

To specify a custom compile order, follow these steps:

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

Figure 5-8. Setting Compile Order



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

Auto-Generating Compile Order

Auto Generate is supported for HDL-only designs. The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling the rest of the files. It continues in this manner until all files compile successfully or until a file(s) can't be compiled for reasons other than dependency.

Files can be displayed in the Project window in alphabetical or compile order (by clicking the column headings). Keep in mind that the order you see in the Project tab is not necessarily the order in which the files will be compiled.

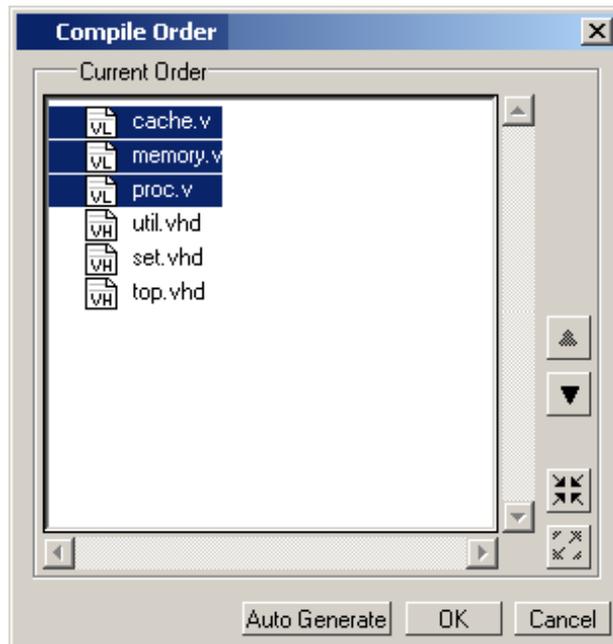
Grouping Files

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

To group files, follow these steps:

1. Select the files you want to group.

Figure 5-9. Grouping Files



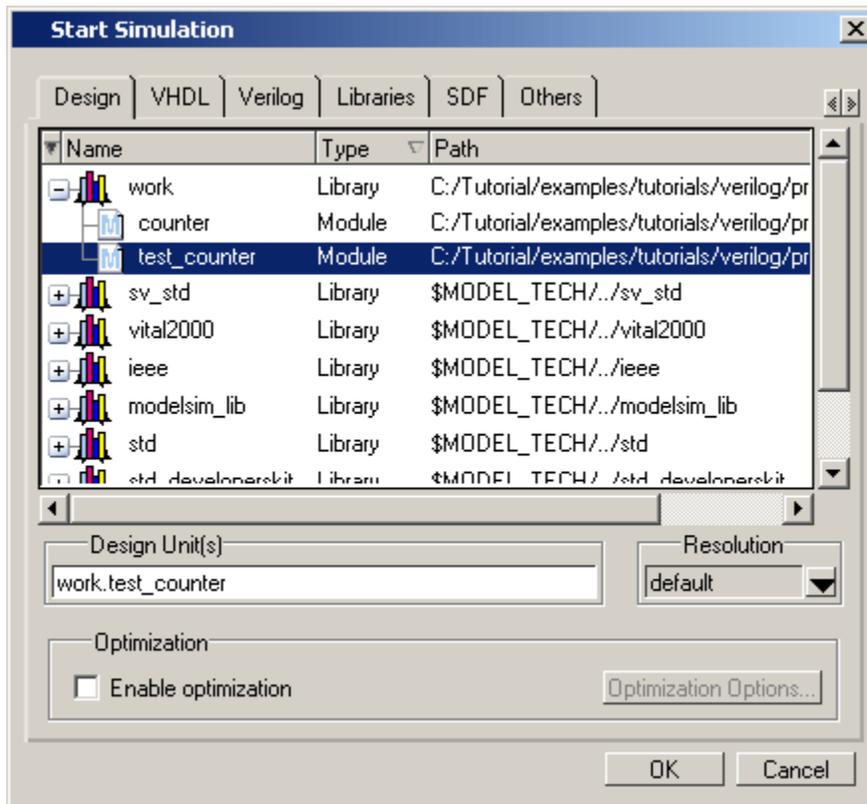
2. Click the Group button.  To ungroup files, select the group and click the Ungroup button. 

Step 4 — Simulating a Design

To simulate a design, do one of the following:

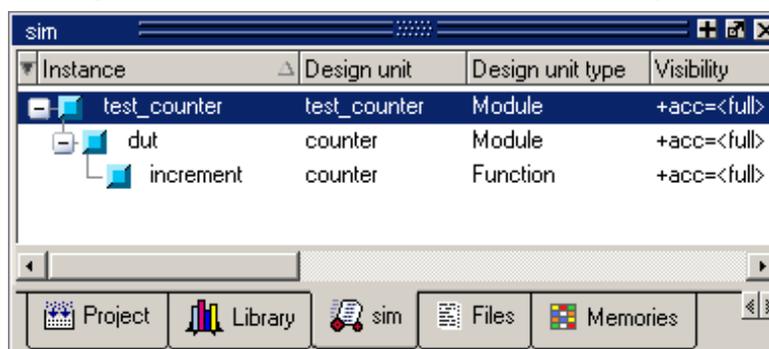
- double-click the Name of an appropriate design object (such as a test bench module or entity) in the Library window
- right-click the Name of an appropriate design object and select **Simulate** from the popup menu
- select **Simulate > Start Simulation** from the menus to open the Start Simulation dialog (Figure 5-10). Select a design unit in the Design tab. Set other options in the VHDL, Verilog, Libraries, SDF, and Others tabs. Then click OK to start the simulation.

Figure 5-10. Start Simulation Dialog



A new Structure window, named *sim*, appears that shows the structure of the active simulation (Figure 5-11).

Figure 5-11. Structure Window with Projects



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

Other Basic Project Operations

Open an Existing Project

If you previously exited ModelSim with a project open, ModelSim automatically will open that same project upon startup. You can open a different project by selecting **File > Open** and choosing Project Files from the **Files of type** drop-down.

Print the Absolute Pathnames For All Files

You can send a list of all project filenames to the transcript window by entering the command `print project filenames`. This command only works when a project is open.

Close a Project

Right-click in the Project window and select **Close Project**. This closes the Project window but leaves the Library window open. Note that you cannot close a project while a simulation is in progress.

The Project Window

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

Figure 5-12. Project Window Overview

Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/06 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/06 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/06 07:36:21 PM
counter.v	✓	Verilog	1	06/07/06 07:35:56 PM
verilog_sim		Simulation		

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

- **Type** – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.
- **Order** – The order in which the file will be compiled when you execute a Compile All command.
- **Modified** – The date and time of the last modification to the file.

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

Sorting the List

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

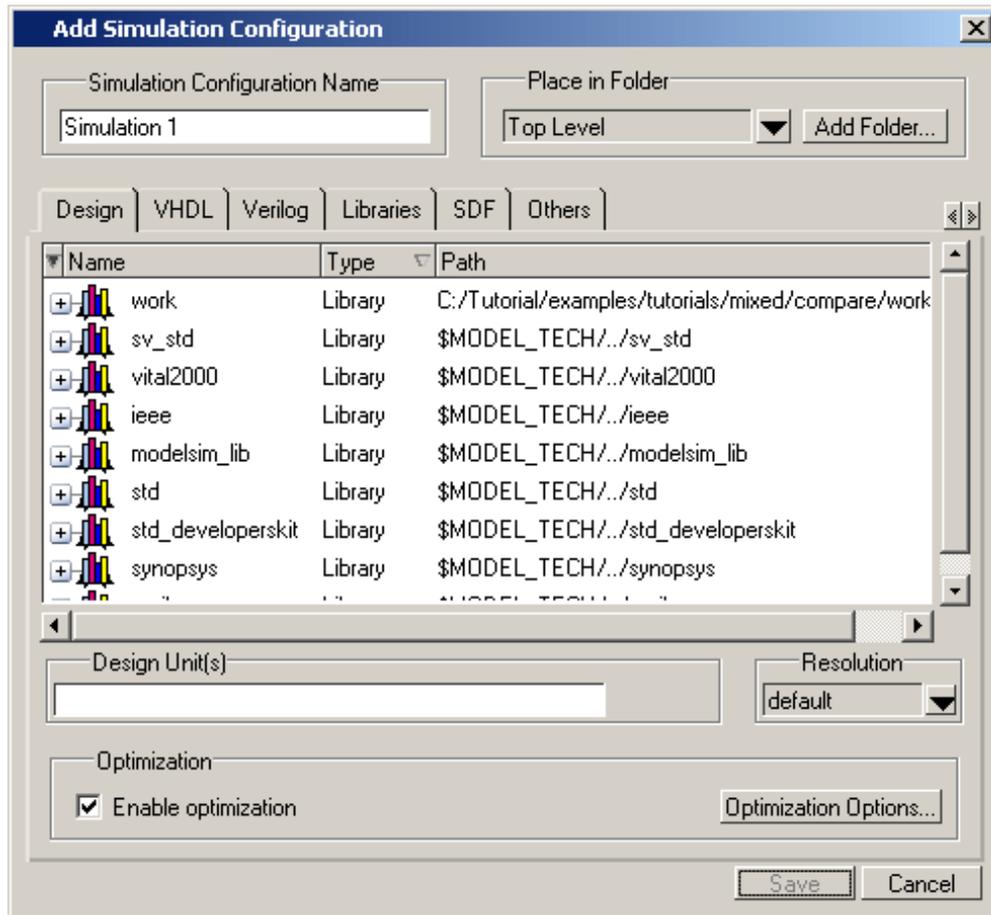
Creating a Simulation Configuration

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

To create a Simulation Configuration, follow these steps:

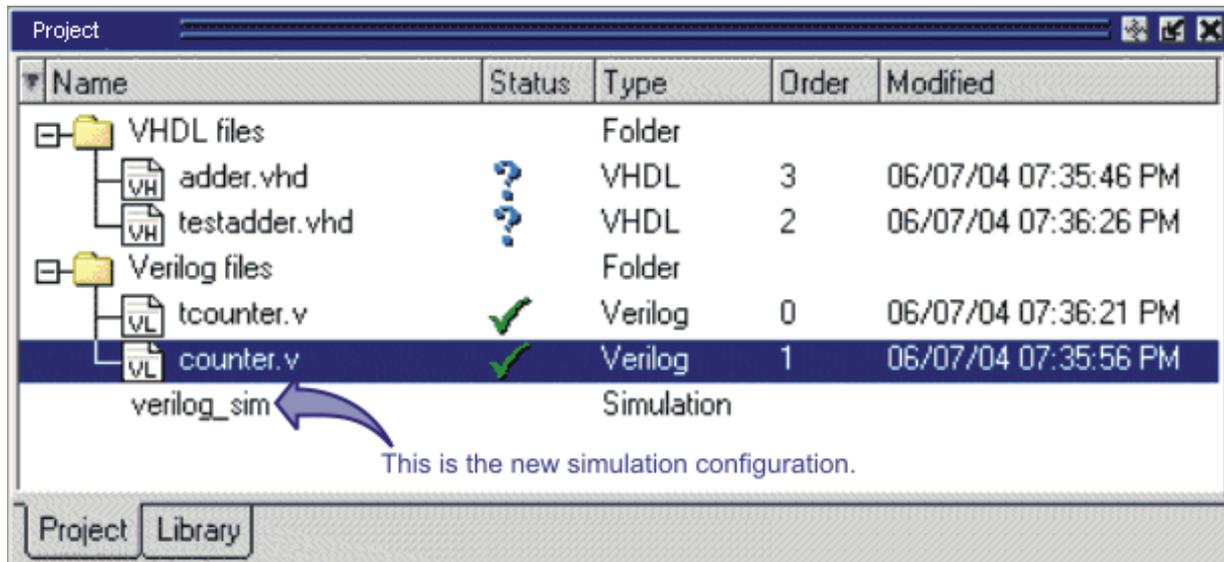
1. Select **Project > Add to Project > Simulation Configuration** from the main menu, or right-click the Project tab and select **Add to Project > Simulation Configuration** from the popup context menu in the Project window.

Figure 5-13. Add Simulation Configuration Dialog



2. Specify a name in the **Simulation Configuration Name** field.
 3. Specify the folder in which you want to place the configuration (see [Organizing Projects with Folders](#)).
 4. Select one or more design unit(s). Use the Control and/or Shift keys to select more than one design unit. The design unit names appear in the **Simulate** field when you select them.
 5. Use the other tabs in the dialog to specify any required simulation options.
- Click **OK** and the simulation configuration is added to the Project window.

Figure 5-14. Simulation Configuration in the Project Window



Double-click the Simulation Configuration *verilog_sim* to load the design.

Optimization Configurations

Similar to Simulation Configurations, Optimization Configurations are named objects that represent an optimized simulation. The process for creating and using them is similar to that for Simulation Configurations (see above). You create them by selecting **Project > Add to Project > Optimization Configuration** and specifying various options in a dialog.

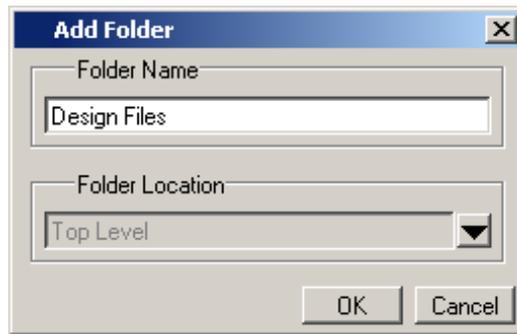
Organizing Projects with Folders

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

Adding a Folder

To add a folder to your project, select **Project > Add to Project > Folder** or right-click in the Project window and select **Add to Project > Folder** (Figure 5-15).

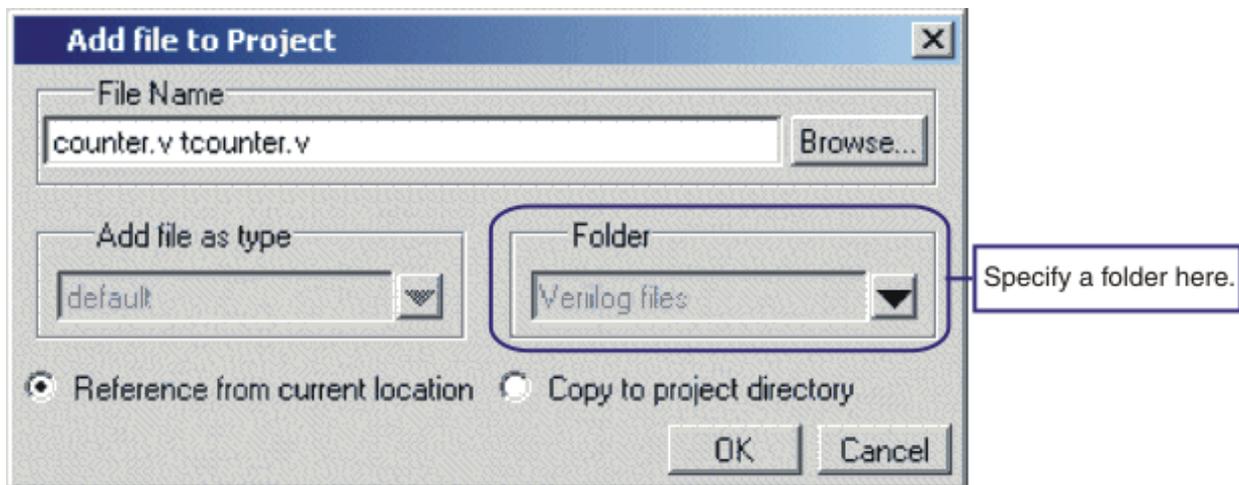
Figure 5-15. Add Folder Dialog



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

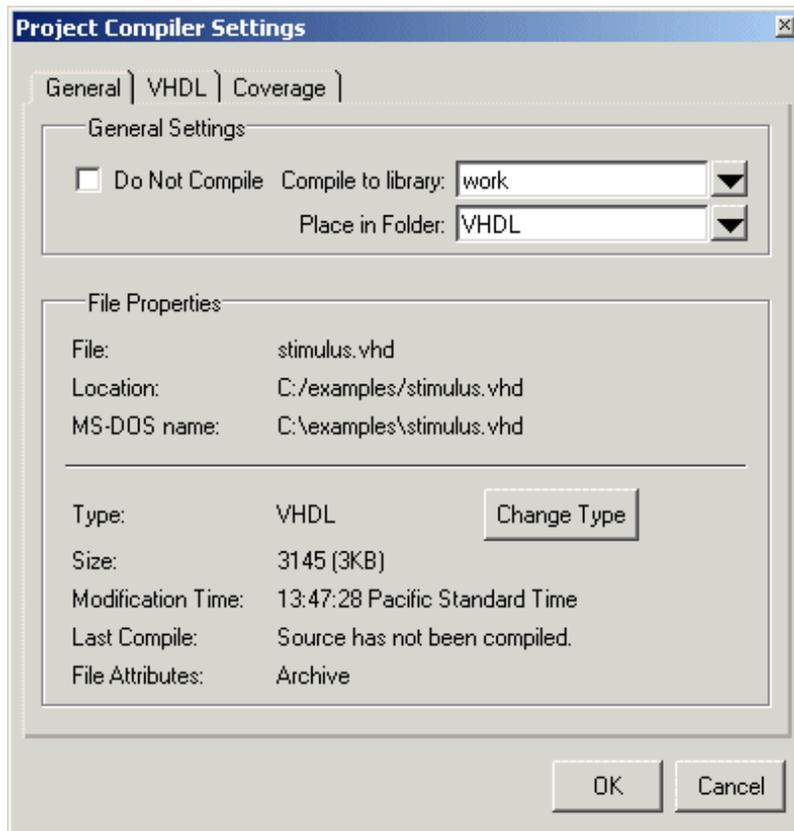
You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.

Figure 5-16. Specifying a Project Folder



If you want to move a file into a folder later on, you can do so using the Properties dialog for the file. Simply right-click on the filename in the Project window and select Properties from the context menu that appears. This will open the Project Compiler Settings Dialog (Figure 5-17). Use the Place in Folder field to specify a folder.

Figure 5-17. Project Compiler Settings Dialog



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

Specifying File Properties and Project Settings

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

File Compilation Properties

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

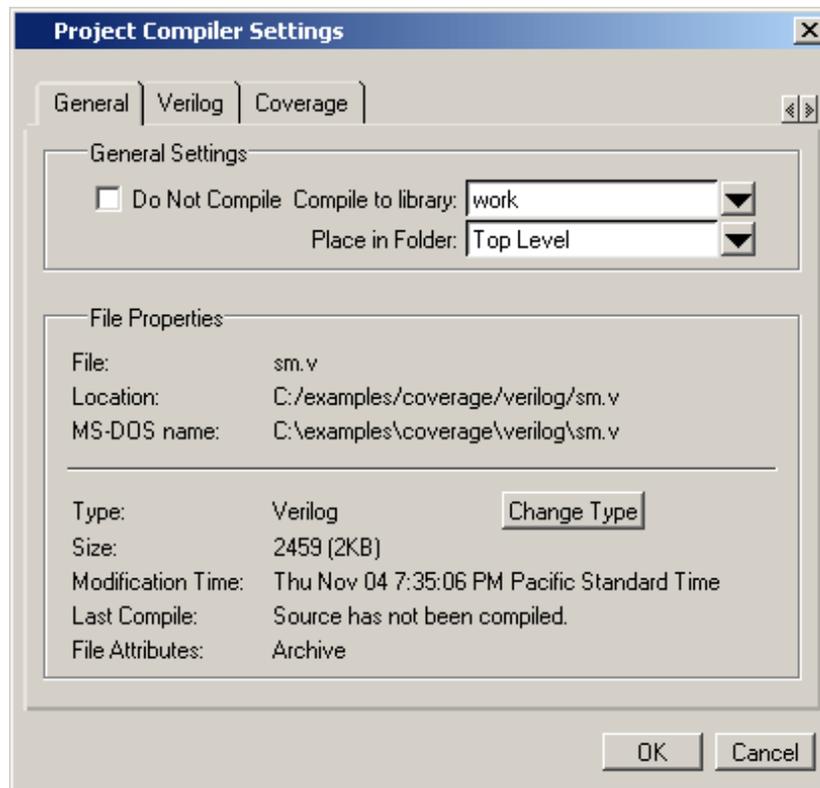
Note



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

To customize specific files, select the file(s) in the Project window, right click on the file names, and select **Properties**. The resulting Project Compiler Settings dialog (Figure 5-18) varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you will see the General tab, Coverage tab, and the VHDL or Verilog tab, respectively. If you select a SystemC file, you will see only the General tab. On the General tab, you will see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you will see all tabs but no file information on the General tab.

Figure 5-18. Specifying File Properties



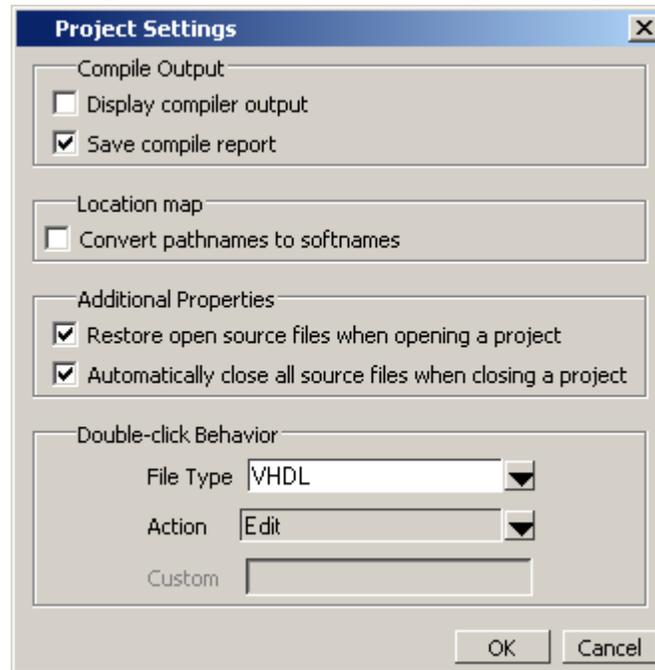
When setting options on a group of files, keep in mind the following:

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

Project Settings

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

Figure 5-19. Project Settings Dialog



Converting Pathnames to Softnames for Location Mapping

If you are using location mapping, you can convert the following into a soft pathname:

- a relative pathname
- full pathname
- pathname with an environment variable

i **Tip:** A softname is a term for a pathname that uses location mapping with `MGC_LOCATION_MAP`. The soft pathname looks like a pathname containing an environment variable, it locates the source using the location map rather than the environment.

To convert the pathname to a softname for projects using location mapping, follow these steps:

1. Right-click anywhere within the Project tab and select **Project Settings**
2. Enable the **Convert pathnames to softnames** within the Location map area of the **Project Settings** dialog box (Figure 5-19).

Once enabled, all pathnames currently in the project and any that are added later are then converted to softnames.

During conversion, if there is no softname in the mgc location map matching the entry, the pathname is converted in to a full (hardened) pathname. A pathname is hardened by removing the environment variable or the relative portion of the path. If this happens, any existing pathnames that are either relative or use environment variables are also changed: either to softnames if possible, or to hardened pathnames if not.

For more information on location mapping and pathnames, see [Using Location Mapping](#).

Accessing Projects from the Command Line

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

You can also use the [project](#) command from the command line to perform common operations on projects.

Chapter 6

Design Libraries

VHDL designs are associated with libraries, which are objects that contain compiled design units. SystemC, Verilog and SystemVerilog designs simulated within ModelSim are compiled into libraries as well.

Design Library Overview

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

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

Design Unit Information

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

- retargetable, executable code
- debugging information
- dependency information

Working Library Versus Resource Libraries

Design libraries can be used in two ways:

1. as a local working library that contains the compiled version of your design;
2. as a resource library.

The contents of your working library will change as you update your design and recompile. A resource library is typically static and serves as a parts source for your design. You can create

your own resource libraries or they may be supplied by another design team or a third party (for example, a silicon vendor).

Only one library can be the working library.

Any number of libraries can be resource libraries during a compilation. You specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched (refer to [Specifying Resource Libraries](#)).

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

The Library Named "work"

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

Archives

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

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

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

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

Working with Design Libraries

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

Creating a Library

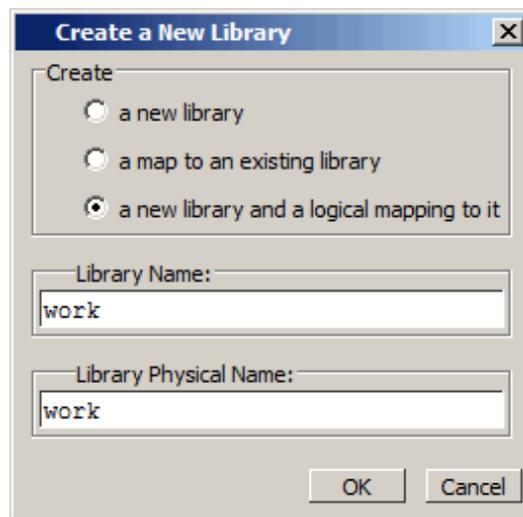
When you create a project (refer to [Getting Started with Projects](#)), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface.

From the ModelSim prompt or a UNIX/DOS prompt, use this `vlib` command:

```
vlib <directory_pathname>
```

To create a new library with the graphic interface, select **File > New > Library**.

Figure 6-1. Creating a New Library



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

The new map entry is written to the `modelsim.ini` file in the [Library] section. Refer to [modelsim.ini Variables](#) for more information.

Note



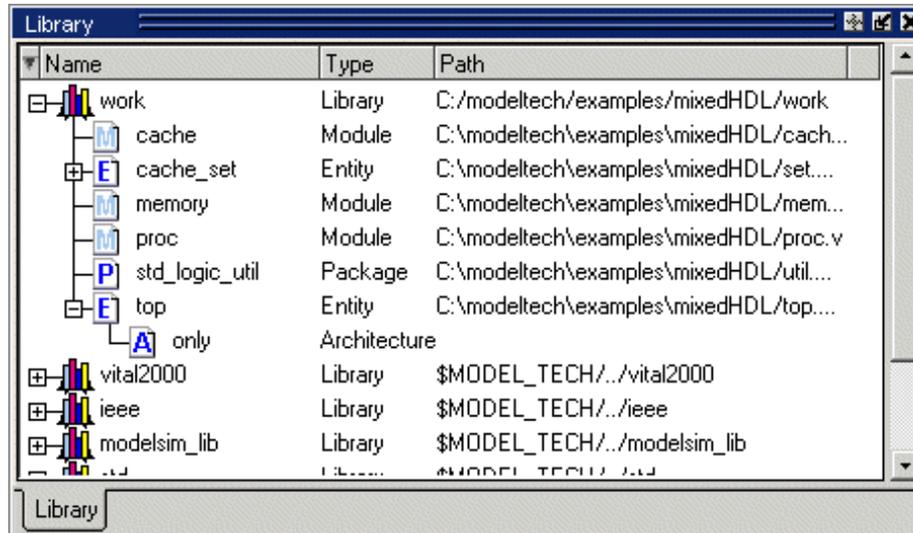
Remember that a design library is a special kind of directory. The **only** way to create a library is to use the ModelSim GUI or the `vlib` command. Do not try to create libraries using UNIX, DOS, or Windows commands.

Managing Library Contents

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

The Library window provides access to design units (configurations, modules, packages, entities, architectures, and SystemC modules) in a library. Various information about the design units is displayed in columns to the right of the design unit name.

Figure 6-2. Design Unit Information in the Workspace



The Library window has a popup menu with various commands that you access by clicking your right mouse button.

The context menu includes the following commands:

- **Simulate** — Loads and optimizes the selected design unit(s) and opens Structure (sim) and Files windows. Related command line command is `vsim -voptargs+acc`.
- **Simulate without Optimization** — Loads the selected design unit(s) without optimization. Related command line command is `vsim -novopt`.
- **Simulate with full Optimization** — Loads and optimizes the selected design unit(s). Related command line command is `vsim -vopt`.
- **Simulate with Coverage** — Loads the selected design unit(s) and collects code coverage data. Related command line command is `vsim -coverage`.
- **Edit** — Opens the selected design unit(s) in the Source window; or, if a library is selected, opens the Edit Library Mapping dialog (refer to [Library Mappings with the GUI](#)).
- **Refresh** — Rebuilds the library image of the selected library without using source code. Related command line command is `vcom` or `vlog` with the `-refresh` argument.
- **Recompile** — Recompiles the selected design unit(s). Related command line command is `vcom` or `vlog`.

- **Optimize** — Optimizes the selected Verilog design unit(s). Related command line command is `vopt`.
- **Update** — Updates the display of available libraries and design units.
- **Create Wave** — Opens a Wave window and loads the objects from the selected design unit(s) as editable waveforms. Related command line command is `wave create -pattern none`.

Assigning a Logical Name to a Design Library

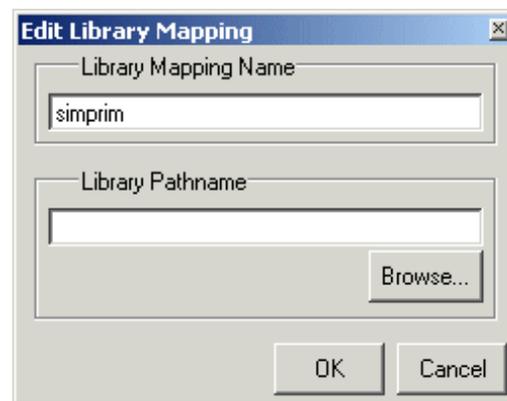
VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

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

Library Mappings with the GUI

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

Figure 6-3. Edit Library Mapping Dialog



The dialog box includes these options:

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

Library Mapping from the Command Line

You can set the mapping between a logical library name and a directory with the `vmap` command using the following syntax:

vmap <logical_name> <directory_pathname>

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

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

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

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

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

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

Unix Symbolic Links

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

ln -s <directory_pathname> <logical_name>

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

vmap <logical_name>

Library Search Rules

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

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

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

Moving a Library

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

Setting Up Libraries for Group Use

By adding an “others” clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the tool does not find a mapping in the *modelsim.ini* file, then it 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
```

You can specify only one "others" clause in the library section of a given *modelsim.ini* file.

The “others” clause only instructs the tool to look in the specified *modelsim.ini* file for a library. It does not load any other part of the specified file.

If there are two libraries with the same name mapped to two different locations – one in the current *modelsim.ini* file and the other specified by the "others" clause – the mapping specified in the current *.ini* file will take effect.

Specifying Resource Libraries

Verilog Resource Libraries

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

Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to **vlog**. Refer to [Library Usage](#) for more information.

The **LibrarySearchPath** variable in the *modelsim.ini* file (in the [vlog] section) can be used to define a space-separated list of resource library paths and/or library path variables. This behavior is identical with the **-L** argument for the **vlog** command.

```
LibrarySearchPath = <path>/lib1 <path>/lib2 <path>/lib3
```

The default for **LibrarySearchPath** is:

```
LibrarySearchPath = mtiAvm mtiOvm mtiUvm mtiUPF
```

VHDL Resource Libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the

end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The **vcom** command 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**, **env**, 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*. Refer also to, [Using the TextIO Package](#).

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

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

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

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

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

Alternate IEEE Libraries Supplied

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

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

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

Regenerating Your Design Libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (refer to [Managing Library Contents](#)), or by using the `-refresh` argument to `vcom` and `vlog`.

From the command line, you would use `vcom` with the `-refresh` argument to update VHDL design units in a library, and `vlog` with the `-refresh` argument to update Verilog design units. By default, the work library is updated. Use either `vcom` or `vlog` with the `-work <library>` argument 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
```

Note



You may specify a specific design unit name with the `-refresh` argument to `vcom` and `vlog` in order to regenerate a library image for only that design, but you may not specify a file name.

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of ModelSim. 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, directives, language constructs, or features that do not exist in the older release.

Note



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

Maintaining 32- and 64-bit Versions in the Same Library

ModelSim allows you to maintain 32-bit and 64-bit versions of a design in the same library, as long as you have not optimized them using the `vopt` command.

To do this, you must compile the design with the 32-bit version and then "refresh" the design with the 64-bit version. For example:

Using the 32-bit version of ModelSim:

```
vlog -novopt file1.v file2.v -work asic_lib
```

Next, using the 64-bit version of ModelSim:

```
vlog -novopt -work asic_lib -refresh
```

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

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

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.

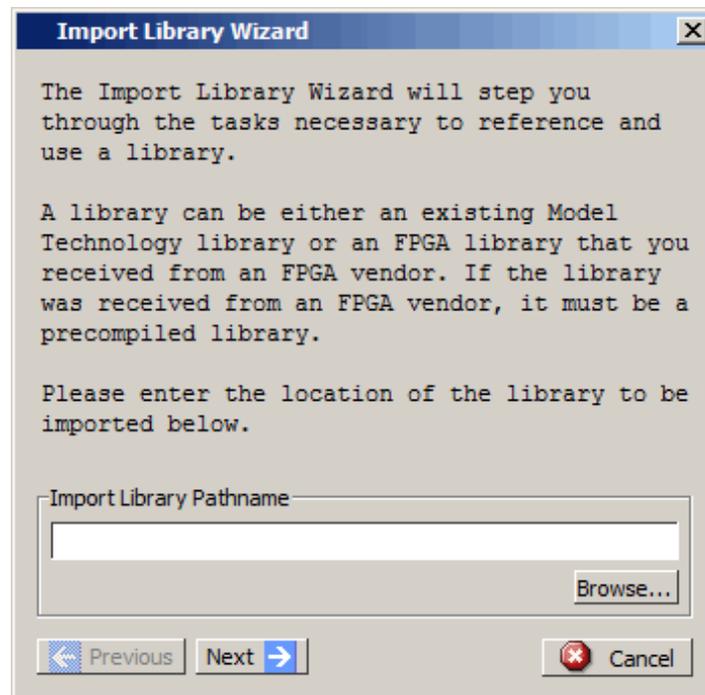
Note



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

To import an FPGA library, select **File > Import > Library**.

Figure 6-4. Import Library Wizard



Follow the instructions in the wizard to complete the import.

Protecting Source Code

The [Protecting Your Source Code](#) chapter provides details about protecting your 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.

Chapter 7

VHDL Simulation

This chapter covers the following topics related to using VHDL in a ModelSim design:

- [Basic VHDL Usage](#) — A brief outline of the steps for using VHDL in a ModelSim design.
- [Compilation and Simulation of VHDL](#) — How to compile, optimize, and simulate a VHDL design
- [Using the TextIO Package](#) — Using the TextIO package provided with ModelSim
- [VITAL Usage and Compliance](#) — Implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling
- [VHDL Utilities Package \(util\)](#) — Using the special built-in utilities package (Util Package) provided with ModelSim
- [Modeling Memory](#) — The advantages of using VHDL variables or protected types instead of signals for memory designs.

Basic VHDL Usage

Simulating VHDL designs with ModelSim consists of the following general steps:

1. Compile your VHDL code into one or more libraries using the `vcom` command. Refer to [Compiling a VHDL Design—the vcom Command](#) for more information.
2. (Optional) Elaborate and optimize your design using the `vopt` command. Refer to Chapter 4, [Optimizing Designs with vopt](#) for more information.
3. Load your design with the `vsim` command. Refer to [Simulating a VHDL Design](#).
4. Simulate the loaded design, then debug as needed.

Compilation and Simulation of VHDL

Creating a Design Library for VHDL

Before you can compile your VHDL source files, you must create a library in which to store the compilation results. Use `vlib` to create a new library. For example:

```
vlib work
```

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

The work library is actually a subdirectory named work. This subdirectory contains a special file named `_info`. Do not create a VHDL library as a directory by using a UNIX, Linux, Windows, or DOS command—always use the `vlib` command.

See [Design Libraries](#) for additional information on working with VHDL libraries.

Compiling a VHDL Design—the `vcom` Command

ModelSim compiles one or more VHDL design units with a single invocation of the `vcom` command, 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 written with the following versions of VHDL:

- 1076-1987
- 1076-1993
- 1076-2002
- 1076-2008

To do so you need to compile units from each VHDL version separately.

The `vcom` command compiles using 1076-2002 rules by default; use the `-87`, `-93`, or `-2008` arguments to `vcom` to compile units written with version 1076-1987, 1076-1993, or 1076-2008 respectively. You can also change the default by modifying the `VHDL93` variable in the `modelsim.ini` file (see [modelsim.ini Variables](#) for more information).

Note



Only a limited number of VHDL 1076-2008 constructs are currently supported.

Dependency Checking

You must re-analyze dependent design units when you change the design units they depend on in the library. The `vcom` command 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. This means you do not have to recompile design units that depend on the entity.

VHDL Case Sensitivity

VHDL is a case-insensitive language for all basic identifiers. For example, `clk` and `CLK` are regarded as the same name for a given signal or variable. This differs from Verilog and SystemVerilog, which are case-sensitive.

The `vcom` command preserves both uppercase and lowercase letters of all user-defined object names in a VHDL source file.

Usage Notes

- You can make the `vcom` command convert uppercase letters to lowercase by either of the following methods:
 - Use the `-lower` argument with the `vcom` command.
 - Set the `PreserveCase` variable to 0 in your `modelsim.ini` file.
- The supplied precompiled packages in `STD` and `IEEE` have their case preserved. This results in slightly different version numbers for these packages. As a result, you may receive out-of-date reference messages when refreshing to the current release. To resolve this, use `vcom -force_refresh` instead of `vcom -refresh`.
- Mixed language interactions
 - Design unit names — Because VHDL and Verilog design units are mixed in the same library, VHDL design units are treated as if they are lowercase. This is for compatibility with previous releases. This also to provide consistent filenames in the file system for make files and scripts.
 - Verilog packages compiled with `-mixedsvvh` — not affected by VHDL uppercase conversion.
 - VHDL packages compiled with `-mixedsvvh` — not affected by VHDL uppercase conversion; VHDL basic identifiers are still converted to lowercase for compatibility with previous releases.
 - FLI — Functions that return names of an object will not have the original case unless the source is compiled using `vcom -lower`. Port and Generic names in the `mtiInterfaceListT` structure are converted to lowercase to provide compatibility with programs doing case sensitive comparisons (`strcmp`) on the generic and port names.

How Case Affects Default Binding

The following rules describe how ModelSim handles uppercase and lowercase names in default bindings.

1. All VHDL names are case-insensitive, so ModelSim always stores them in the library in lowercase to be consistent and compatible with older releases.

2. When looking for a design unit in a library, ModelSim ignores the VHDL case and looks first for the name in lowercase. If present, ModelSim uses it.
3. If no lowercase version of the design unit name exists in the library, then ModelSim checks the library, ignoring case.
 - a. If ONE match is found this way, ModelSim selects that design unit.
 - b. If NO matches or TWO or more matches are found, ModelSim does not select anything.

The following examples demonstrate these rules. Here, the VHDL compiler needs to find a design unit named Test. Because VHDL is case-insensitive, ModelSim looks for "test" because previous releases always converted identifiers to lowercase.

Example 1

Consider the following library:

```
work
  entity test
  Module TEST
```

The VHDL entity test is selected because it is stored in the library in lowercase. The original VHDL could have contained TEST, Test, or TeSt, but the library always has the entity as "test."

Example 2

Consider the following library:

```
work
  Module Test
```

No design unit named "test" exists, but "Test" matches when case is ignored, so ModelSim selects it.

Example 3

Consider the following library:

```
work
  Module Test
  Module TEST
```

No design unit named "test" exists, but both "Test" and "TEST" match when case is ignored, so ModelSim does not select either one.

Range and Index Checking

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

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the `vcom` command. Or, you can use the `NoRangeCheck` and `NoIndexCheck` variables in the `modelsim.ini` file to specify whether or not they are performed. See [modelsim.ini Variables](#).

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

Subprogram Inlining

ModelSim attempts to inline subprograms at compile time to improve simulation performance. This happens automatically and should be largely transparent. However, you can disable automatic inlining two ways:

- Invoke `vcom` with the `-O0` or `-O1` argument
- Use the `mti_inhibit_inline` attribute as described below

Single-stepping through a simulation varies slightly, depending on whether inlining occurred. When single-stepping to a subprogram call that has not been inlined, the simulator stops first at the line of the call, and then proceeds to the line of the first executable statement in the called subprogram. If the called subprogram has been inlined, the simulator does not first stop at the subprogram call, but stops immediately at the line of the first executable statement.

`mti_inhibit_inline` Attribute

You can disable inlining for individual design units (a package, architecture, or entity) or subprograms with the `mti_inhibit_inline` attribute. Follow these rules to use the attribute:

- Declare the attribute within the design unit's scope as follows:

```
attribute mti_inhibit_inline : boolean;
```

- Assign the value `true` to the attribute for the appropriate scope. For example, to inhibit inlining for a particular function (for example, "foo"), add the following attribute assignment:

```
attribute mti_inhibit_inline of foo : procedure is true;
```

To inhibit inlining for a particular package (for example, "pack"), add the following attribute assignment:

```
attribute mti_inhibit_inline of pack : package is true;
```

Do similarly for entities and architectures.

Simulating a VHDL Design

A VHDL design is ready for simulation after it has been compiled with `vcom` and possibly optimized with `vopt` (see [Optimizing Designs with vopt](#)). You can then use the `vsim` command to invoke the simulator with the name(s) of the configuration or entity/architecture pair. Multiple optimized top design modules can be specified. For more information about simulation with multiple optimized design modules refer to the `<library_name>.<design_unit>` argument to `vsim`.

Note



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](#)) or the **Start Simulation** dialog box (open with **Simulate > Start Simulation** menu selection).

This example begins simulation on a design unit with an entity named `my_asic` and an architecture named `structure`:

```
vsim my_asic structure
```

Timing Specification

The `vsim` command can annotate a design using VITAL-compliant models with timing data from an SDF file. You can specify delay by invoking `vsim` with the `-sdfmin`, `-sdftyp`, or `-sdfmax` arguments. The following example uses an SDF file named `f1.sdf` in the current work directory, and an invocation of `vsim` annotating maximum timing values for the design unit `my_asic`:

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

By default, the timing checks within VITAL models are enabled. You can disable them with the `+notimingchecks` argument. For example:

```
vsim +notimingchecks topmod
```

If you specify `vsim +notimingchecks`, the generic `TimingChecksOn` is set to `FALSE` for all VITAL models with the `Vital_level0` or `Vital_level1` attribute (refer to [VITAL Usage and Compliance](#)). Setting this generic to `FALSE` disables the actual calls to the timing checks along with anything else that is present in the model's timing check block. In addition, if these models use the generic `TimingChecksOn` to control behavior beyond timing checks, this behavior will not occur. This can cause designs to simulate differently and provide different results.

By default, `vopt` does not fix the `TimingChecksOn` generic in VITAL models. Instead, it lets the value float to allow for overriding at simulation time. If best performance and no timing checks are desired, `+notimingchecks` should be specified with `vopt`.

```
vopt +notimingchecks topmod
```

Specifying `vopt +notimingchecks` or `-GTimingChecks=<FALSE/TRUE>` will fix the generic value for simulation. As a consequence, using `vsim +notimingchecks` at simulation may not have any effect on the simulation depending on the optimization of the model.

Naming Behavior of VHDL For Generate Blocks

A VHDL `for ... generate` statement, when elaborated in a design, places a given number of `for ... generate` equivalent blocks into the scope in which the statement exists; either an architecture, a block, or another generate block. The simulator constructs a design path name for each of these `for ... generate` equivalent blocks based on the original generate statement's label and the value of the generate parameter for that particular iteration. For example, given the following code:

```
g1: for I in 1 to Depth generate
  L: BLK port map (A(I), B(I+1));
end generate g1
```

the default names of the blocks in the design hierarchy would be:

```
g1(1), g1(2), ...
```

This name appears in the GUI to identify the blocks. You should use this name with any commands when referencing a block that is part of the simulation environment. The format of the name is based on the VHDL Language Reference Manual P1076-2008 section 16.2.5 Predefined Attributes of Named Entities.

If the type of the generate parameter is an enumeration type, the value within the parenthesis will be an enumeration literal of that type; such as: `g1(red)`.

For mixed-language designs, in which a Verilog hierarchical reference is used to reference something inside a VHDL `for ... generate` equivalent block, the parentheses are replaced with brackets (`[]`) to match Verilog syntax. If the name is dependent upon enumeration literals, the literal will be replaced with its position number because Verilog does not support using enumerated literals in its `for ... generate` equivalent block.

In releases prior to the 6.6 series, this default name was controlled by the `GenerateFormat` `modelsim.ini` file variable would have appeared as:

```
g1__1, g1__2, ...
```

All previously-generated scripts using this old format should work by default. However, if not, you can use the `GenerateFormat` and `OldVhdlForGenNames` `modelsim.ini` variables to ensure that the old and current names are mapped correctly.

Differences Between Versions of VHDL

There are four versions of the VHDL standard (IEEE Std 1076): 1076-1987, 1076-1993, 1076-2002, and 1076-2008. The default language version supported for ModelSim is 1076-2002.

If your code was written according to the 1987, 1993, or 2008 version, you may need to update your code or instruct ModelSim to use rules for different version.

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

- Select the appropriate version from the compiler options menu in the GUI
- Invoke `vcom` using the argument `-87`, `-93`, `-2002`, or `-2008`.
- Set the `VHDL93` variable in the `[vcom]` section of the `modelsim.ini` file to one of the following values:
 - 0, 87, or 1987 for 1076-1987
 - 1, 93, or 1993 for 1076-1993
 - 2, 02, or 2002 for 1076-2002
 - 3, 08, or 2008 for 1076-2008

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



Tip: Please refer to ModelSim Release Notes for the most current and comprehensive description of differences between supported versions of the VHDL standard.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
-- Compiling package foopack
```

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

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

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

by

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

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

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

- Sub-element association — In VHDL-87 when using individual sub-element association in an association list, associating individual sub-elements with NULL is discouraged. In VHDL-93 such association is forbidden. A typical message is:

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

- VHDL-2008 packages — ModelSim does not provide VHDL source for VHDL-2008 IEEE-defined standard packages because of copyright restrictions. You can obtain VHDL source from <http://standards.ieee.org/downloads/1076/1076-2008/> for the following packages:

```
IEEE.fixed_float_types
IEEE.fixed_generic_pkg
IEEE.fixed_pkg
IEEE.float_generic_pkg
IEEE.float_pkg
IEEE.MATH_REAL
IEEE.MATH_COMPLEX
IEEE.NUMERIC_BIT
IEEE.NUMERIC_BIT_UNSIGNED
IEEE.NUMERIC_STD
IEEE.NUMERIC_STD_UNSIGNED
IEEE.std_logic_1164
IEEE.std_logic_textio
```

Foreign Language Interface

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

The ModelSim FLI interface is described in detail in the Foreign Language Interface Reference Manual.

Simulator Resolution Limit for VHDL

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

Note



In Verilog, this representation of time units is referred to as precision or timescale.

Overriding the Resolution

To override the default resolution of ModelSim, specify a value for the **-t** argument of the **vsim** command line or select a different Simulator Resolution in the **Simulate** dialog box. Available values of simulator resolution are:

1 fs, 10 fs, 100 fs
1 ps, 10 ps, 100 ps
1 ns, 10 ns, 100 ns
1 us, 10 us, 100 us
1 ms, 10 ms, 100 ms
1 s, 10 s, 100 s

For example, the following command sets resolution to 10 ps:

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

Note that you need to take care in specifying a resolution value larger than a delay value in your design—delay values in that design unit are rounded to the closest multiple of the resolution. In the example above, a delay of 4 ps would be rounded down to 0 ps.

Choosing the Resolution for VHDL

You should specify the coarsest value for time resolution that does not result in undesired rounding of your delay times. The resolution value should not be unnecessarily small because it decreases the maximum simulation time limit and can cause longer simulations.

Default Binding

By default, ModelSim performs binding when you load the design with `vsim`. The advantage of this default binding at load time is that it provides more flexibility for compile order. Namely, VHDL entities don't necessarily have to be compiled before other entities/architectures that instantiate them.

However, you can force ModelSim to perform default binding at compile time instead. This may allow you to catch design errors (for example, entities with incorrect port lists) earlier in the flow. Use one of these two methods to change when default binding occurs:

- Specify the **-bindAtCompile** argument to `vcom`
- Set the `BindAtCompile` variable in the `modelsim.ini` to 1 (true)

Default Binding Rules

When searching for a VHDL entity to bind with, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE Std 1076-1987 contained a flaw that made it almost impossible for an entity to be directly visible if it had the same name as the component. This meant if a component was declared in an architecture, any entity with the same name above that declaration would be hidden because component/entity names cannot be overloaded. As a result, ModelSim observes the following rules for determining default binding:

- If performing default binding at load time, search the libraries specified with the **-Lf** argument to `vsim`.
- If a directly visible entity has the same name as the component, use it.
- If an entity would be directly visible in the absence of the component declaration, use it.
- If the component is declared in a package, search the library that contained the package for an entity with the same name.

If none of these methods is successful, ModelSim then does the following:

- Search the work library.
- Search all other libraries that are currently visible by means of the **library** clause.
- If performing default binding at load time, search the libraries specified with the **-L** argument to `vsim`.

Note that these last three searches are an extension to the 1076 standard.

Disabling Default Binding

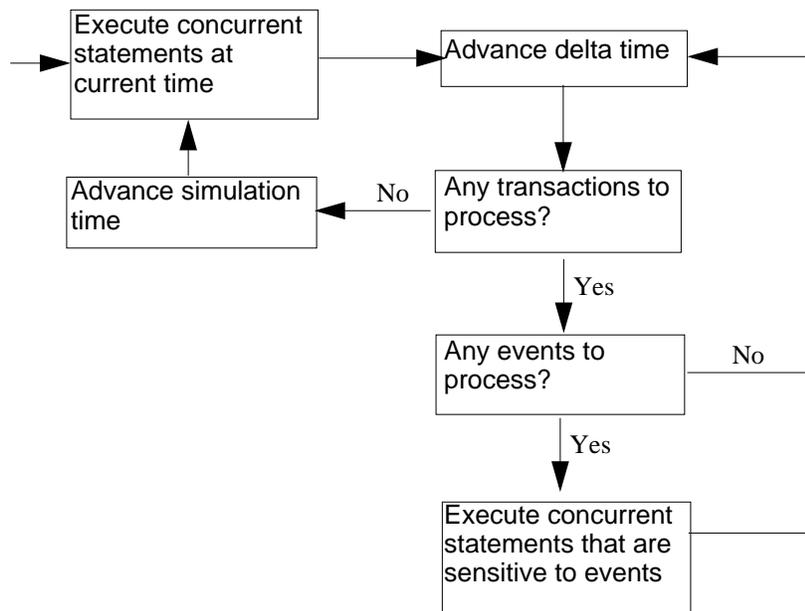
If you want default binding to occur using only configurations, you can disable normal default binding methods by setting the `RequireConfigForAllDefaultBinding` variable in the `modelsim.ini` file to 1 (true).

Delta Delays

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

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

Figure 7-1. VHDL Delta Delay Process



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

```
clk2 <= clk;

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

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

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

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

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

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

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

```
process (rst, clk)
begin
  if(rst = '0')then
    s0 <= '0';
  elsif(clk'event and clk='1') then
    s0 <= inp;
  end if;
end process;
s0_delayed <= s0;
process (rst, clk2)
begin
  if(rst = '0')then
    s1 <= '0';
  elsif(clk2'event and clk2='1') then
```

```
        s1 <= s0_delayed;  
    end if;  
end process;
```

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

Detecting Infinite Zero-Delay Loops

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

The iteration limit default value is 5000 . If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate > Runtime Options** menu or by modifying the [IterationLimit](#) variable in the *modelsim.ini*. See [modelsim.ini Variables](#) for more information on modifying the *modelsim.ini* file.

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

Using the TextIO Package

The TextIO package is defined within the IEEE Std 1076-2002, *IEEE Standard VHDL Language Reference Manual*. This package allows human-readable text input from a declared source within a VHDL file during simulation.

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 1987 syntax for a file declaration is:

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

where "file_logical_name" must be a string expression.

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

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

where "file_open_information" is:

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

You can specify a full or relative path as the file_logical_name; for example (VHDL 1987):

```
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 [modelsim.ini Variables](#) for more details.

Using STD_INPUT and STD_OUTPUT Within ModelSim

The standard VHDL1987 TextIO package contains the following file declarations:

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

Updated versions of the TextIO package contain these file declarations:

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

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

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

TextIO Implementation Issues

Writing Strings and Aggregates

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

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

will cause the following error:

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

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

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

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

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

The following procedure call also generates an error:

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

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

There are two possible solutions to this problem:

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

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

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

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

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

Reading and Writing Hexadecimal Numbers

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

To expand this functionality, ModelSim supplies hexadecimal routines in the package io_utils, which is located in the file `<install_dir>/modeltech/examples/gui/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 deallocates 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 Std 1076-2002, *IEEE Standard VHDL Language Reference Manual* — contains invalid VHDL syntax and cannot be implemented in

VHDL. This is because access values must be passed as variables, but functions do not allow variable parameters.

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

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

The ENDFILE Function

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

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

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

Using Alternative Input/Output Files

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

The VHDL1987 declaration is:

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

The VHDL1993 declaration is:

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

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

Flushing the TEXTIO Buffer

Flushing of the TEXTIO buffer is controlled by the [UnbufferedOutput](#) variable in the *modelsim.ini* file.

Providing Stimulus

You can provide an input stimulus to a design by reading data vectors from a file and assigning their values to signals. You can then verify the results of this input. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

```
<install_dir>/examples/gui/stimulus.vhd
```

VITAL Usage and Compliance

The VITAL (VHDL Initiative Towards ASIC Libraries) modeling specification is sponsored by the IEEE to promote the development of highly accurate, efficient simulation models for ASIC (Application-Specific Integrated Circuit) components in VHDL.

The IEEE Std 1076.4-2000, *IEEE Standard for VITAL ASIC Modeling Specification* is available from the Institute of Electrical and Electronics Engineers, Inc.

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

Tel: (732) 981-0060
Fax: (732) 981-1721

<http://www.ieee.org>

VITAL Source Code

The source code for VITAL packages is provided in the following ModelSim installation directories:

```
<install_dir>/vhdl_src/vital2.2b  
                  /vital95  
                  /vital2000
```

VITAL 1995 and 2000 Packages

VITAL 2000 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 1995 accelerated packages are pre-compiled into the **vital1995** library. If you need to use the older library, you either need to change the **ieee** library mapping or add a **use** clause to your VHDL code to access the VITAL 1995 packages.

To change the **ieee** library mapping, issue the following command:

```
vmap ieee <modeltech>/vital1995
```

Or, alternatively, add **use** clauses to your code:

```
LIBRARY vital1995;  
USE vital1995.vital_primitives.all;  
USE vital1995.vital_timing.all;  
USE vital1995.vital_memory.all;
```

Note that if your design uses two libraries—one that depends on **vital95** and one that depends on **vital2000**—then you will have to change the references in the source code to **vital2000**. Changing the library mapping will not work.

ModelSim VITAL built-ins are generally updated as new releases of the VITAL packages become available.

VITAL Compliance

A simulator is VITAL-compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages—as outlined in the VITAL Model Development Specification. ModelSim is compliant with IEEE Std 1076.4-2002, *IEEE Standard for VITAL ASIC Modeling Specification*. In addition, ModelSim accelerates the VITAL_Timing, VITAL_Primitives, and VITAL_memory packages. The optimized procedures are functionally equivalent to the IEEE Std 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

VITAL Compliance Checking

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

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking `vcom` with the argument **-novitalcheck**.

You can turn off compliance checking for VITAL 1995 and VITAL 2000 as well, but it strongly recommended that you leave checking on to ensure optimal simulation.

VITAL Compliance Warnings

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

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

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

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

You can control the visibility of VITAL compliance-check warnings in your `vcom` transcript. To suppress them, use the `vcom -nowarn` command. For example, `vcom -nowarn 6`, where the number 6 represents the warning level to be displayed as part of the warning: `** WARNING: [6]`. You can also add the following line to your `modelsim.ini` file in the `vcom` section:

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

See [modelsim.ini Variables](#) for more information.

Compiling and Simulating with Accelerated VITAL Packages

The `vcom` command automatically recognizes that a VITAL function is being referenced from the `ieee` library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

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

Compiler Options for VITAL Optimization

The `vcom` command has several arguments that control and provide feedback on VITAL optimization:

- `-novital`

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

- `-O0` | `-O4`

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

Enable optimizations with `-O4` (default).

- `-debugVA`

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

VHDL Utilities Package (util)

The util package contains various VHDL utilities that you can run as commands. The package is part of the `modelsim_lib` library, which is located in the `/modeltech` tree and is mapped in the default `modelsim.ini` file.

To include the utilities in this package, add the following lines similar to your VHDL code:

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

get_resolution

The `get_resolution` utility returns the current simulator resolution as a real number. For example, a resolution of 1 femtosecond (1 fs) 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\(\)](#)
- [to_time\(\)](#)

Example

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

```
resval := get_resolution;
```

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

init_signal_driver()

The `init_signal_driver()` utility drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (such as a test bench).

See [init_signal_driver](#) for complete details.

init_signal_spy()

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

See [init_signal_spy](#) for complete details.

signal_force()

The `signal_force()` utility forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (such as a test bench). A `signal_force` works the same as the `force` command when you set the `modelsim.ini` variable named `ForceSigNextIter` to 1. The variable `ForceSigNextIter` in the `modelsim.ini` file can be set to honor the signal update event in next iteration for all force types. Note that the `signal_force` utility cannot issue a repeating force.

See [signal_force](#) for complete details.

signal_release()

The `signal_release()` utility releases any force that was applied to an existing VHDL signal or Verilog register or net. This allows you to release signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (such as a test bench). A `signal_release` works the same as the `noforce` command.

See [signal_release](#) for complete details.

to_real()

The `to_real()` utility converts the physical type time value into a real value with respect to the current value of 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 rounded to 2.0 (that is, 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](#)
- [to_time\(\)](#)

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](#) 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()

The `to_time()` utility 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 converted 5.9 to a time and the simulator resolution was 1 ps, then the time value would be rounded to 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](#)
- [to_real\(\)](#)

Example

If the simulator resolution is set to 1 ps, and you enter the following function:

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

then the value returned to timeval would be 72 ps.

Modeling Memory

If you want to model a memory with VHDL using signals, you may encounter either of the following common problems with simulation:

- Memory allocation error, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Very long times to load, elaborate, or run.

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

As an alternative, you can model a memory design using variables or protected types instead of signals, which provides the following performance benefits:

- Reduced storage required to model the memory, by as much as one or two orders of magnitude
- Reduced startup and run times
- Elimination of associated memory allocation errors

Examples of Different Memory Models

[Example 7-1](#) shown below uses different VHDL architectures for the entity named memory to provide the following models for storing RAM:

- `bad_style_87` — uses a VHDL signal
- `style_87` — uses variables in the memory process
- `style_93` — uses variables in the architecture

For large memories, the run time for architecture `bad_style_87` is many times longer than the other two and uses much more memory. Because of this, you should avoid using VHDL signals to model 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.

Converting an Integer Into a `bit_vector`

The following code shows how to convert an integer variable into a `bit_vector`.

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

entity test is
end test;

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

Examples Using VHDL1987, VHDL1993, VHDL2002 Architectures

- [Example 7-1](#) contains two VHDL architectures that demonstrate recommended memory models: `style_93` uses shared variables as part of a process, `style_87` uses For comparison, a third architecture, `bad_style_87`, shows the use of signals.

The style_87 and style_93 architectures work with equal efficiency for this example. However, VHDL 1993 offers additional flexibility because the RAM storage can be shared among multiple processes. In the example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

- [Example 7-2](#) is a package (named conversions) that is included by the memory model in [Example 7-1](#).
- For completeness, [Example 7-3](#) shows protected types using VHDL 2002. Note that using protected types offers no advantage over shared variables.

Example 7-1. Memory Model Using VHDL87 and VHDL93 Architectures

Example functions are provided below in package “conversions.”

```
-----  
-- Source:      memory.vhd  
-- Component:   VHDL synchronous, single-port RAM  
-- Remarks:     Provides three different architectures  
-----  
  
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 := sylv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process memory;
-- illustrates a second process using the shared variable
initialize:
process (do_init)
    variable address : natural;
    begin
        if rising_edge(do_init) then
            for address in 0 to nwords-1 loop
                ram(address) := data_in;
            end loop;
        end if;
    end process initialize;
end architecture style_93;

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

```
architecture bad_style_87 of memory is
    -----
    signal ram : ram_type;
    -----
begin
memory:
process (cs)
    variable address : natural := 0;
begin
    if rising_edge(cs) then
        address := sylv_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;
```

Example 7-2. Conversions Package

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

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

package body conversions is

    function sylv_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 sylv_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;
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;
```

Example 7-3. Memory Model Using VHDL02 Architecture

```
-----  
-- Source:      sp_syn_ram_protected.vhd  
-- Component:   VHDL synchronous, single-port RAM  
-- Remarks:     Various VHDL examples: random access memory (RAM)  
-----  
  
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
USE ieee.numeric_std.ALL;  
  
ENTITY sp_syn_ram_protected IS  
    GENERIC (  
        data_width : positive := 8;  
        addr_width  : positive := 3  
    );  
    PORT (  
        inclk      : IN  std_logic;  
        outclk     : IN  std_logic;  
        we         : IN  std_logic;  
        addr       : IN  unsigned(addr_width-1 DOWNTO 0);  
        data_in    : IN  std_logic_vector(data_width-1 DOWNTO 0);  
        data_out   : OUT std_logic_vector(data_width-1 DOWNTO 0)  
    );  
  
END sp_syn_ram_protected;  
  
ARCHITECTURE intarch OF sp_syn_ram_protected IS  
  
    TYPE mem_type IS PROTECTED  
        PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);  
                          addr : IN unsigned(addr_width-1 DOWNTO 0));  
        IMPURE FUNCTION read ( addr : IN unsigned(addr_width-1 DOWNTO 0))  
        RETURN  
            std_logic_vector;  
        END PROTECTED mem_type;  
  
    TYPE mem_type IS PROTECTED BODY  
        TYPE mem_array IS ARRAY (0 TO 2**addr_width-1) OF  
            std_logic_vector(data_width-1 DOWNTO 0);  
        VARIABLE mem : mem_array;  
  
        PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);  
                          addr : IN unsigned(addr_width-1 DOWNTO 0)) IS  
        BEGIN  
            mem(to_integer(addr)) := data;  
        END;  
  
        IMPURE FUNCTION read ( addr : IN unsigned(addr_width-1 DOWNTO 0))  
        RETURN  
            std_logic_vector IS  
        BEGIN  
            return mem(to_integer(addr));  
        END;  
  
    END PROTECTED BODY mem_type;  
  
END intarch;
```

```

    SHARED VARIABLE memory : mem_type;

BEGIN

    ASSERT data_width <= 32
        REPORT "### Illegal data width detected"
        SEVERITY failure;

    control_proc : PROCESS (inclk, outclk)

    BEGIN
        IF (inclk'event AND inclk = '1') THEN
            IF (we = '1') THEN
                memory.write(data_in, addr);
            END IF;
        END IF;

        IF (outclk'event AND outclk = '1') THEN
            data_out <= memory.read(addr);
        END IF;
    END PROCESS;

END intarch;

-----
-- Source:      ram_tb.vhd
-- Component:   VHDL test bench for RAM memory example
-- Remarks:     Simple VHDL example: random access memory (RAM)
-----

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram_tb IS
END ram_tb;

ARCHITECTURE testbench OF ram_tb IS

    -----
    -- Component declaration single-port RAM
    -----

    COMPONENT sp_syn_ram_protected
        GENERIC (
            data_width : positive := 8;
            addr_width  : positive := 3
        );
        PORT (
            inclk      : IN  std_logic;
            outclk     : IN  std_logic;
            we         : IN  std_logic;
            addr       : IN  unsigned(addr_width-1 DOWNTO 0);
            data_in    : IN  std_logic_vector(data_width-1 DOWNTO 0);
            data_out   : OUT std_logic_vector(data_width-1 DOWNTO 0)
        );
    END COMPONENT;

    -----

```

```
-- Intermediate signals and constants
-----
SIGNAL  addr      : unsigned(19 DOWNTO 0);
SIGNAL  inaddr    : unsigned(3  DOWNTO 0);
SIGNAL  outaddr   : unsigned(3  DOWNTO 0);
SIGNAL  data_in   : unsigned(31 DOWNTO 0);
SIGNAL  data_in1  : std_logic_vector(7 DOWNTO 0);
SIGNAL  data_spl  : std_logic_vector(7 DOWNTO 0);
SIGNAL  we        : std_logic;
SIGNAL  clk       : std_logic;
CONSTANT clk_pd   : time := 100 ns;

BEGIN

-----
-- instantiations of single-port RAM architectures.
-- All architectures behave equivalently, but they
-- have different implementations. The signal-based
-- architecture (rtl) is not a recommended style.
-----
spraml : entity work.sp_syn_ram_protected
  GENERIC MAP (
    data_width => 8,
    addr_width => 12)
  PORT MAP (
    inclk    => clk,
    outclk   => clk,
    we       => we,
    addr     => addr(11 downto 0),
    data_in  => data_in1,
    data_out => data_spl);

-----
-- clock generator
-----
clock_driver : PROCESS
BEGIN
  clk <= '0';
  WAIT FOR clk_pd / 2;
  LOOP
    clk <= '1', '0' AFTER clk_pd / 2;
    WAIT FOR clk_pd;
  END LOOP;
END PROCESS;

-----
-- data-in process
-----
datain_drivers : PROCESS(data_in)
BEGIN
  data_in1 <= std_logic_vector(data_in(7 downto 0));
END PROCESS;

-----
-- simulation control process
-----
ctrl_sim : PROCESS
```

```
BEGIN
  FOR i IN 0 TO 1023 LOOP
    we      <= '1';
    data_in <= to_unsigned(9000 + i, data_in'length);
    addr    <= to_unsigned(i, addr'length);
    inaddr  <= to_unsigned(i, inaddr'length);
    outaddr <= to_unsigned(i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(7 + i, data_in'length);
    addr    <= to_unsigned(1 + i, addr'length);
    inaddr  <= to_unsigned(1 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(3, data_in'length);
    addr    <= to_unsigned(2 + i, addr'length);
    inaddr  <= to_unsigned(2 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(30330, data_in'length);
    addr    <= to_unsigned(3 + i, addr'length);
    inaddr  <= to_unsigned(3 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    we      <= '0';
    addr    <= to_unsigned(i, addr'length);
    outaddr <= to_unsigned(i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(1 + i, addr'length);
    outaddr <= to_unsigned(1 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(2 + i, addr'length);
    outaddr <= to_unsigned(2 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(3 + i, addr'length);
    outaddr <= to_unsigned(3 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    END LOOP;
    ASSERT false
      REPORT "### End of Simulation!"
      SEVERITY failure;
  END PROCESS;

END testbench;
```

Affecting Performance by Cancelling Scheduled Events

Simulation performance is likely to get worse if events are scheduled far into the future but then cancelled before they take effect. This situation acts like a memory leak and slows down simulation.

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

The following shows a wait with a time-out:

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

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

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

For projected waveforms, the following would behave the same way:

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

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

Chapter 8

Verilog and SystemVerilog Simulation

This chapter describes how to compile and simulate Verilog and SystemVerilog designs with ModelSim. This chapter covers the following topics:

- [Basic Verilog Usage](#) — A brief outline of the steps for using Verilog in a ModelSim design.
- [Verilog Compilation](#) — Information on the requirements for compiling and optimizing Verilog designs and libraries.
- [Verilog Simulation](#) — Information on the requirements for running simulation.
- [Cell Libraries](#) — Criteria for using Verilog cell libraries from ASIC and FPGA vendors that are compatible with ModelSim.
- [System Tasks and Functions](#) — System tasks and functions that are built into the simulator.
- [Compiler Directives](#) — Verilog compiler directives supported for ModelSim.
- [Sparse Memory Modeling](#) — Information on how to enable sparse memory models in a design. Sparse memories are a mechanism for allocating storage for memory elements only when they are needed.
- [Verilog PLI/VPI and SystemVerilog DPI](#) — Verilog and SystemVerilog interfaces that you can use to define tasks and functions that communicate with the simulator through a C procedural interface.
- [OVM-Aware Debug](#) — OVM-aware debugging provides you, the verification or design engineer, with information, at the OVM abstraction level, that connects you to the OVM base-class library.
- [UVM-Aware Debug](#) — UVM-aware debugging provides you, the verification or design engineer, with information, at the UVM abstraction level, that connects you to the UVM base-class library.



Tip: For more information on the verification features of SystemVerilog (such as assert and cover directives, covergroups, and test bench automation), refer to the [Verification with Assertions and Cover Directives](#) and [Verification with Functional Coverage](#) chapters of the User's Manual.

Standards, Nomenclature, and Conventions

ModelSim implements the Verilog and SystemVerilog languages as defined by the following standards:

- IEEE 1364-2005 and 1364-1995 (Verilog)
- IEEE 1800-2009 and 1800-2005 (SystemVerilog)

Note



ModelSim supports partial implementation of SystemVerilog IEEE Std 1800-2009. For release-specific information on currently supported implementation, refer to the following text file located in the ModelSim installation directory:

```
<install_dir>/docs/technotes/sysvlog.note
```

SystemVerilog is built “on top of” IEEE Std 1364 for the Verilog HDL and improves the productivity, readability, and reusability of Verilog-based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing design and verification products into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained random testbench development, coverage-driven verification, and assertion-based verification.

The standard for SystemVerilog specifies extensions for a higher level of abstraction for modeling and verification with the Verilog hardware description language (HDL). This standard includes design specification methods, embedded assertions language, testbench language including coverage and assertions application programming interface (API), and a direct programming interface (DPI).

In this chapter, the following terms apply:

- “Verilog” refers to IEEE Std 1364 for the Verilog HDL.
- “Verilog-1995” refers to IEEE Std 1364-1995 for the Verilog HDL.
- “Verilog-2001” refers to IEEE Std 1364-2001 for the Verilog HDL.
- “Verilog-2005” refers to IEEE Std 1364-2005 for the Verilog HDL.
- “SystemVerilog” refers to the extensions to the Verilog standard (IEEE Std 1364) as defined in IEEE Std 1800-2009.

Note



The term “Language Reference Manual” (or LRM) is often used informally to refer to the current IEEE standard for Verilog or SystemVerilog.

Supported Variations in Source Code

It is possible to use syntax variations of constructs that are not explicitly defined as being supported in the Verilog LRM (such as “shortcuts” supported for similar constructs in another language).

for Loops

ModelSim allows using Verilog syntax that omits any or all three specifications of a for loop: initialization, termination, increment. This is similar to allowed usage in C and is shown in the following examples.

Note

If you use this variation, a suppressible warning (2252) is displayed, which you can change to an error if you use the `vlog -pedanticerrors` command.

- Missing initializer (in order to continue where you left off):

```
for (; incr < foo; incr++) begin ... end
```

- Missing incremter (in order to increment in the loop body):

```
for (ii = 0; ii <= foo; ) begin ... end
```

- Missing initializer and terminator (in order to implement a while loop):

```
for (; goo < foo; ) begin ... end
```

- Missing all specifications (in order to create an infinite loop):

```
for (;;) begin ... end
```

Alternative One-Step Flow

If you have previously been using NCSim, ModelSim provides an alternative flow that combines the compile, optimize, and simulate phases into one command. Refer to the [qverilog](#) command for more information.

Basic Verilog Usage

Simulating Verilog designs with ModelSim consists of the following general steps:

1. Compile your Verilog code into one or more libraries using the [vlog](#) command. See [Verilog Compilation](#) for details.

2. (Optional) Elaborate and optimize your design using the `vopt` command. For more information, refer to Chapter 4, [Optimizing Designs with vopt](#) and [Optimization Considerations for Verilog Designs](#).
3. Load your design with the `vsim` command. Refer to [Verilog Simulation](#).
4. Simulate the loaded design and debug as needed.

Verilog Compilation

The first time you compile a design there is a two-step process:

1. Create a working library with `vlib` or select **File > New > Library**.
2. Compile the design using `vlog` or select **Compile > Compile**.

Creating a Working Library

Before you can compile your design, you must create a library in which to store the compilation results. Use the `vlib` command or select **File > New > Library** to create a new library. For example:

```
vlib work
```

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

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX commands – always use the `vlib` command.

See [Design Libraries](#) for additional information on working with libraries.

Invoking the Verilog Compiler

The `vlog` command invokes the Verilog compiler, which compiles Verilog source code into retargetable, executable code. You can simulate your design on any supported platform without having to recompile your design; the library format is also compatible across all platforms.

As the design compiles, the resulting object code for modules and user-defined primitives (UDPs) is generated into a library. As noted above, the compiler places results into the work library by default. You can specify an alternate library with the `-work` argument of the `vlog` command.

Example 8-1. Invocation of the Verilog Compiler

The following example shows how to use the `vlog` command to invoke the Verilog compiler:

```
vlog top.v +libext+.v+.u -y vlog_lib
```

After compiling `top.v`, `vlog` searches the `vlog_lib` library for files with modules with the same name as primitives referenced, but undefined in `top.v`. The use of `+libext+.v+.u` implies filenames with a `.v` or `.u` suffix (any combination of suffixes may be used). Only referenced definitions are compiled.

Verilog Case Sensitivity

Note that Verilog and SystemVerilog are case-sensitive languages. For example, `clk` and `CLK` are regarded as different names that you can apply to different signals or variables. This differs from VHDL, which is case-insensitive.

Parsing SystemVerilog Keywords

With standard Verilog files (`<filename>.v`), `vlog` does not automatically parse SystemVerilog keywords. SystemVerilog keywords are parsed when either of the following situations exists:

- Any file within the design contains the `.sv` file extension
- You use the `-sv` argument with the `vlog` command

The following examples of the `vlog` command show how to enable SystemVerilog features and keywords in ModelSim:

```
vlog testbench.sv top.v memory.v cache.v
```

```
vlog -sv testbench.v proc.v
```

In the first example, the `.sv` extension for `testbench` automatically causes ModelSim to parse SystemVerilog keywords. In the second example, the `-sv` argument enables SystemVerilog features and keywords.

Keyword Compatibility

One of the primary goals of SystemVerilog standardization has been to ensure full backward compatibility with the Verilog standard. Questa recognizes all reserved keywords listed in Table B-1 in Annex B of IEEE Std 1800-2009.

In previous ModelSim releases, the `vlog` command read some IEEE Std 1800-2009 keywords and treated them as IEEE Std 1800-2005 keywords. However, those keywords are no longer recognized in the IEEE Std 1800-2005 keyword set.

The following reserved keywords have been added since IEEE Std 1800-2005:

accept_on	reject_on	sync_accept_on
checker	restrict	sync_reject_on
endchecker	s_always	unique0
eventually	s_eventually	until
global	s_nexttime	until_with
implies	s_until	untyped
let	s_until_with	weak
nexttime	strong	

If you use or produce SystemVerilog code that uses any of these strings as identifiers from a previous release in which they were not considered reserved keywords, you can do either of the following to avoid a compilation error:

- Use a different set of strings in your design. You can add one or more characters as a prefix or suffix (such as an underscore, `_`) to the string, which will cause the string to be read in as an identifier and not as a reserved keyword.
- Use the SystemVerilog pragmas ``begin_keywords` and ``end_keywords` to define regions where only IEEE Std 1800-2005 keywords are recognized.

Recognizing SystemVerilog Files by File Name Extension

If you use the `-sv` argument with the `vlog` command, then ModelSim assumes that all input files are SystemVerilog, regardless of their respective filename extensions.

If you do not use the `-sv` argument with the `vlog` command, then ModelSim assumes that only files with the extension `.sv`, `.svh`, or `.svp` are SystemVerilog.

File extensions of include files

Similarly, if you do not use the `-sv` argument while reading in a file that uses an ``include` statement to specify an include file, then the file extension of the include file is ignored and the language is assumed to be the same as the file containing the ``include`. For example, if you do not use the `-sv` argument:

If `a.v` included `b.sv`, then `b.sv` would be read as a Verilog file.

If `c.sv` included `d.v`, then `d.v` would be read as a SystemVerilog file.

File extension settings in modelsim.ini

You can define which file extensions indicate SystemVerilog files with the `SVFileExtensions` variable in the `modelsim.ini` file. By default, this variable is defined in `modelsim.ini` as follows:

```
; SVFileExtensions = sv svp svh
```

For example, the following command:

```
vlog a.v b.sv c.svh d.v
```

reads in `a.v` and `d.v` as a Verilog files and reads in `b.sv` and `c.svh` as SystemVerilog files.

File types affecting compilation units

Note that whether a file is Verilog or SystemVerilog can affect when ModelSim changes from one compilation unit to another.

By default, ModelSim instructs the compiler to treat all files within a compilation command line as separate compilation units (single-file compilation unit mode, which is the equivalent of using `vlog -sfcu`).

```
vlog a.v aa.v b.sv c.svh d.v
```

ModelSim would group these source files into three compilation units:

Files in first unit — `a.v`, `aa.v`, `b.sv`

File in second unit — `c.svh`

File in third unit — `d.v`

This behavior is governed by two basic rules:

- Anything read in is added to the current compilation unit.
- A compilation unit ends at the close of a SystemVerilog file.

Initializing enum Variables

By default, ModelSim initializes enum variables using the default value of the base type instead of the leftmost value. However, you can change this so that ModelSim sets the initial value of an enum variable to the left most value in the following ways:

- Run `vlog -enumfirstinit` when compiling and run `vsim -enumbaseinit` when simulating.
- Set `EnumBaseInit = 0` in the `modelsim.ini` file.

Incremental Compilation

ModelSim supports incremental compilation of Verilog designs—there is no requirement to compile an entire design in one invocation of the compiler.

You are not required to compile your design in any particular order (unless you are using SystemVerilog packages; see Note below) because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator.

Note



Compilation order may matter when using SystemVerilog packages. As stated in the section *Referencing data in packages* of IEEE Std 1800-2005: “Packages must exist in order for the items they define to be recognized by the scopes in which they are imported.”

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.

Example 8-2. Incremental Compilation Example

Contents of testbench.sv

```
module testbench;
    timeunit 1ns;
    timeprecision 10ps;
    bit d=1, clk = 0;
    wire q;
    initial
        for (int cycles=0; cycles < 100; cycles++)
            #100 clk = !clk;

    design dut(q, d, clk);
endmodule
```

Contents of design.v:

```
module design(output bit q, input bit d, clk);
    timeunit 1ns;
    timeprecision 10ps;
    always @(posedge clk)
        q = d;
endmodule
```

Compile the design incrementally as follows:

```
ModelSim> vlog testbench.sv
.
# Top level modules:
#   testbench
ModelSim> vlog -sv test1.v
.
# Top level modules:
#   dut
```

Note that the compiler lists each module as a top-level module, although, ultimately, only *testbench* 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 that you can ignore 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
```

Automatic Incremental Compilation with `-incr`

The most efficient method of incremental compilation is to manually compile only the modules that have changed. However, this is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to compile your entire design along with the `-incr` argument. This causes the compiler to automatically determine which modules have changed and generate code only for those modules.

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

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

Top level modules:
  top
```

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

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

Top level modules:
  top
```

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

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

Note



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

Library Usage

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

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

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

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

Top level modules:
    top
```

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

Library Search Rules for the vlog Command

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

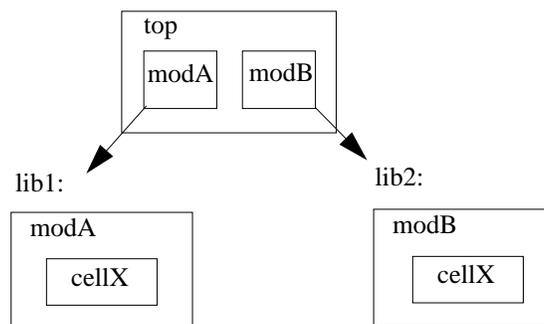
- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the [Verilog-XL *uselib* Compiler Directive](#) section.
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

Handling Sub-Modules with Common Names

Sometimes in one design you need to reference two different modules that have the same name. This situation can occur if you have hierarchical modules organized into separate libraries, and you have commonly-named sub-modules in the libraries that have different definitions. This may happen if you are using vendor-supplied libraries.

For example, say you have the following design configuration:

Example 8-3. Sub-Modules with Common Names



The normal library search rules fail in this situation. For example, if you load the design as follows:

```
vsim -L lib1 -L lib2 top
```

both instantiations of *cellX* resolve to the *lib1* version of *cellX*. On the other hand, if you specify **-L lib2 -L lib1**, both instantiations of *cellX* resolve to the *lib2* version of *cellX*.

To handle this situation, ModelSim implements a special interpretation of the expression **-L work**. When you specify **-L work** first in the search library arguments you are directing **vsim** to search for the instantiated module or UDP in the library that contains the module that does the instantiation.

In the example above you would invoke **vsim** as follows:

```
vsim -L work -L lib1 -L lib2 top
```

SystemVerilog Multi-File Compilation

Declarations in Compilation Unit Scope

SystemVerilog allows the declaration of types, variables, functions, tasks, and other constructs in compilation unit scope (\$unit). The visibility of declarations in \$unit scope does not extend outside the current compilation unit. Thus, it is important to understand how compilation units are defined by the simulator during compilation.

By default, `vlog` operates in Single File Compilation Unit mode (SFCU). This means the visibility of declarations in \$unit scope terminates at the end of each source file. Visibility does not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the declaration to the file containing the end of the declaration.

The `vlog` command also supports a non-default mode called Multi File Compilation Unit (MFCU). In MFCU mode, `vlog` compiles all files on the command line into one compilation unit. You can invoke `vlog` in MFCU mode as follows:

- For a specific, one-time compilation: `vlog -mfcu`.
- For all compilations: set the variable **MultiFileCompilationUnit = 1** in the `modelsim.ini` file.

By using either of these methods, you allow declarations in \$unit scope to remain in effect throughout the compilation of all files.

If you have made MFCU the default behavior by setting **MultiFileCompilationUnit = 1** in your `modelsim.ini` file, you can override this default behavior on a specific compilation by using `vlog -sfcu`.

Macro Definitions and Compiler Directives in Compilation Unit Scope

According to the IEEE Std 1800-2005, the visibility of macro definitions and compiler directives span the lifetime of a single compilation unit. By default, this means the definitions of macros and settings of compiler directives terminate at the end of each source file. They do not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the definition to the file containing the end of the definition.

See [Declarations in Compilation Unit Scope](#) for instructions on how to control `vlog`'s handling of compilation units.

Note

Compiler directives revert to their default values at the end of a compilation unit.

If a compiler directive is specified as an option to the compiler, this setting is used for all compilation units present in the current compilation.

Verilog-XL Compatible Compiler Arguments

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

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

Arguments Supporting Source Libraries

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

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

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

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

Verilog-XL `uselib` Compiler Directive

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

The syntax for the ``uselib` directive is:

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

where `<library_reference>` can be one or more of the following:

- **dir=<library_directory>**, which is equivalent to the command line argument:
`-y <library_directory>`
- **file=<library_file>**, which is equivalent to the command line argument:
`-v <library_file>`
- **libext=<file_extension>**, which is equivalent to the command line argument:
`+libext+<file_extension>`
- **lib=<library_name>**, which references a library for instantiated objects, specifically modules, interfaces and program blocks, but not packages. You must ensure the correct mappings are set up if the library does not exist in the current working directory. The `-compile_uselibs` argument does not affect this usage of ``uselib`.

For example, the following directive

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

is equivalent to the following command line arguments:

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

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

An important feature of ``uselib` is to allow a design to reference multiple modules having the same name, therefore 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, the simulator 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 simulator will ignore a library reference `libext=<file_extension>`. For example, the following ``uselib` directives infer the same object library:

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

In both cases the simulator assumes that the library source is compiled into the object library:

```
/h/vendorA/work
```

The simulator 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.

-compile_uselibs Argument

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

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

- The directory name specified by the **-compile_uselibs** argument. For example,

```
-compile_uselibs=./mydir
```
- The directory specified by the `MTI_USELIB_DIR` environment variable (see [Environment Variables](#))
- A directory named `mti_uselibs` that is created in the current working directory

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

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

vlog -compile_uselibs top

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

uselib is Persistent

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

vlog -compile_uselibs dut.v srtr.v

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

Verilog Configurations

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

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

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

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

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

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

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

Configurations and the Library Named work

The library named “work” is treated specially by ModelSim (see [The Library Named "work"](#) for details) for Verilog configurations.

Consider the following code example:

```
config cfg;
  design top;
  instance top.u1 use work.u1;
endconfig
```

In this case, `work.u1` indicates to load `u1` from the current library.

To create a configuration that loads an instance from a library other than the default work library, do the following:

1. Make sure the library has been created using the `vlib` command. For example:

```
vlib mylib
```

2. Define this library (`mylib`) as the new current (working) library:

```
vlog -work mylib
```

3. Load instance `u1` from the current library, which is now `mylib`:

```
config cfg;
  design top;
  instance top.u1 use mylib.u1;
endconfig
```

Verilog Generate Statements

ModelSim implements the rules adopted for Verilog 2005, because the Verilog 2001 rules for generate statements had numerous inconsistencies and ambiguities. Most of the 2005 rules are backwards compatible, but there is one key difference related to name visibility.

Name Visibility in Generate Statements

Consider the following code example:

```
module m;
  parameter p = 1;

  generate
    if (p)
      integer x = 1;
    else
      real x = 2.0;
    endgenerate

  initial $display(x);
endmodule
```

This example is legal under 2001 rules. However, it is illegal under the 2005 rules and causes an error in ModelSim. Under the new rules, you cannot hierarchically reference a name in an anonymous scope from outside that scope. In the example above, x does not propagate its visibility upwards, and each condition alternative is considered to be an anonymous scope.

For this example to simulate properly in ModelSim, change it to the following:

```
module m;
  parameter p = 1;

  if (p) begin:s
    integer x = 1;
  end
  else begin:s
    real x = 2.0;
  end

  initial $display(s.x);
endmodule
```

Because the scope is named in this example (`begin:s`), normal hierarchical resolution rules apply and the code runs without error.

In addition, note that the keyword pair `generate - endgenerate` is optional under the 2005 rules and are excluded in the second example.

Initializing Registers and Memories

For Verilog designs you can initialize registers and memories with specific values or randomly generated values. This functionality is controlled from the `vlog` and `vsim` command lines with the following switches:

- Registers: `vlog +initreg` and `vsim +initreg`
- Memories: `vlog +initmem` and `vsim +initmem`

Initialization Concepts

- **Random stability** — From run to run, it is reasonable to expect that simulation results will be consistent with the same seed value, even when the design is recompiled or different optimization switches are specified.

However, if the design changes in any way, random stability can not be ensured. These design changes include:

- Changing the source code (except for comment editing).
- Changing parameter values with `vopt -G` or `vsim -G`. This forces a different topology during design elaboration.
- Changing a `+define` switch such that different source code is compiled.
- Changing design hierarchy of the design units due to the random initial value being dependent upon the full path name of the instance.

For sequential UDPs, the simulator guarantees repeatable initial values only if the design is compiled and run with the same `vlog`, `vopt`, and `vsim` options.

- **Sequential UDPs** — An initial statement in a sequential UDP overrides all `+initreg` functionality.

Limitations

- The following are not initialized with `+initmem` or `+initreg`:
 - Variables in dynamic types, dynamic arrays, queues, or associative arrays.
 - Unpacked structs, or unpacked or tagged unions.

Requirements

- Prepare your libraries with `vlib` and `vmap` as you would normally.

Initializing with Specific Values — Enabled During Compilation

1. Compile the design unit with the `+initreg` or `+initmem` switches to the `vlog` command. Refer to the `vlog` command reference page for descriptions of the following options.
 - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
 - b. Specify the initialization value: `+{0 | 1 | X | Z}`.
2. Simulate as you would normally.

Initializing with Specific Values — Enabled During Optimization

1. Compile as you would normally

2. Optimize the design with the `+initreg` or `+initmem` switches to the `vopt` command. Refer to the `vopt` command reference page for a description of the following options.
 - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
 - b. Specify the initialization value: `+{0 | 1 | X | Z}`.
 - c. Specify design unit name: `+<selection>`
3. Simulate as you would normally.

Initializing with Random Values — Enabled During Compilation

1. Compile the design unit with the `+initreg` or `+initmem` switches to the `vlog` command. Refer to the `vlog` command reference page for descriptions of the following options.
 - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
 - b. Do not specify the initialization value. This enables the specification of a random seed during simulation.
2. Simulate as you would normally, except for adding the `+initmem+<seed>` or `+initreg+<seed>` switches. Refer to the `vsim` command reference page for a description of this switch. The random values will only include 0 or 1.

If no `+initreg` is present on the `vsim` command line, a random seed of 0 is used during initialization.

Initializing with Random Values — Enabled During Optimization

1. Compile as you would normally
2. Optimize the design with the `+initreg` or `+initmem` switches to the `vopt` command. Refer to the `vopt` command reference page for a description of the following options.
 - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
 - b. Do not specify the initialization value. This enables the specification of a random seed during simulation.
 - c. Specify design unit name: `+<selection>`
3. Simulate as you would normally, except for adding the `+initmem+<seed>` or `+initreg+<seed>` switches. Refer to the `vsim` command reference page for a description of this switch. The random values will only include 0 or 1.

If no `+initreg` is present on the `vsim` command line, a random seed of 0 is used during initialization.

Verilog Simulation

A Verilog design is ready for simulation after it has been compiled with **vlog** and possibly optimized with **vopt**. For more information on Verilog optimizations, see Chapter 4, [Optimizing Designs with vopt](#) and [Optimization Considerations for Verilog Designs](#). The simulator may then be invoked with the names of the top-level modules (many designs contain only one top-level module) or the name(s) you assigned to the optimized version(s) of the design. Multiple optimized top design modules can be specified. For more information about simulation with multiple optimized design modules refer to [vsim](#) **<library_name>.<design_unit>**. For example, if your top-level modules are “testbench” and “globals”, then invoke the simulator as follows:

```
vsim testbench globals
```

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

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

Simulator Resolution Limit (Verilog)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time (also known as the simulator resolution limit). The resolution limit defaults to the smallest time units that you specify among all of the **`timescale** compiler directives in the design.

Here is an example of a **`timescale** directive:

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

The first number (1 ns) is the time units; the second number (100 ps) is the time precision, which is the rounding factor for the specified time units. The directive above causes time values to be read as nanoseconds and rounded to the nearest 100 picoseconds.

Time units and precision can also be specified with SystemVerilog keywords as follows:

```
timeunit 1 ns  
timeprecision 100 ps
```

Modules Without Timescale Directives

Unexpected behavior may occur if your design contains some modules with timescale directives and others without. An elaboration error is issued in this situation and it is highly recommended that all modules having delays also have timescale directives to make sure that the timing of the design operates as intended.

Timescale elaboration errors may be suppressed or reduced to warnings however, there is a risk of improper design behavior and reduced performance. The `vsim +nowarnTSCALE` or `-suppress` options may be used to ignore the error, while the `-warning` option may be used to reduce the severity to a warning.

-timescale Option

The **-timescale** option can be used with the **vlog** and **vopt** commands to specify the default timescale in effect during compilation for modules that do not have an explicit ``timescale` directive. The format of the **-timescale** argument is the same as that of the ``timescale` directive:

```
-timescale <time_units>/<time_precision>
```

where `<time_units>` is `<n> <units>`. The value of `<n>` must be 1, 10, or 100. The value of `<units>` must be fs, ps, ns, us, ms, or s. In addition, the `<time_units>` must be greater than or equal to the `<time_precision>`.

For example:

```
-timescale "1ns / 1ps"
```

The argument above needs quotes because it contains white space.

Multiple Timescale Directives

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

timescale, -t, and Rounding

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

```
`timescale 1 ns / 100 ps  
  
module foo;  
  
    initial  
        #12.536 $display
```

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

- **-t** not set
The delay is rounded to 12.5 as directed by the module's 'timescale directive.
- **-t** is set to 1 fs
The delay is rounded to 12.5. Again, the module's precision is determined by the 'timescale directive. ModelSim does not override the module's precision.
- **-t** is set to 1 ns
The delay will be rounded to 13. The module's precision is determined by the **-t** setting. ModelSim can only round the module's time values because the entire simulation is operating at 1 ns.

Choosing the Resolution for Verilog

You should choose the coarsest simulator resolution limit possible that does not result in undesired rounding of your delays. For example, values smaller than the current Time Scale will be truncated to zero (0) and a warning issued. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

Event Ordering in Verilog Designs

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

Event Queues

Section 11 of IEEE Std 1364-2005 defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

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

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

The Standard (LRM) dictates that events are processed as follows:

1. All active events are processed.
2. Inactive events are moved to the active event queue and then processed.
3. Non-blocking events are moved to the active event queue and then processed.
4. Monitor events are moved to the active queue and then processed.
5. Simulation advances to the next time where there is an inactive event or a non-blocking assignment update event.

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

Assume that you have these four statements:

- always@(q) p = q;
- always @(q) p2 = not q;
- always @(p or p2) clk = p and p2;
- always @(posedge clk)

with current variable values: q = 0, p = 0, p2=1

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

Table 8-1. Evaluation 1 of always Statements

Event being processed	Active event queue
	q(0 -> 1)
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
p(0 -> 1)	3, 2
3	clk(0 -> 1), 2

Table 8-1. Evaluation 1 of always Statements (cont.)

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

Table 8-2. Evaluation 2 of always Statement

Event being processed	Active event queue
	q(0 -> 1)
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
2	p2(1 -> 0), p(0 -> 1)
p(0 -> 1)	3, p2(1 -> 0)
p2(1 -> 0)	3
3	<empty> (clk does not change)

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

Controlling Event Queues with Blocking or Non-Blocking Assignments

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

Blocking Assignments

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

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

Non-Blocking Assignments

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

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

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

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

gen2: always @(clk1)
    clk2 = clk1;

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

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

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

Debugging Event Order Issues

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

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

See the [vlog](#) command for descriptions of **-compat** and **-hazards**.

Hazard Detection

The `-hazards` argument to `vsim` detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. `vsim` detects the following kinds of hazards:

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

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

To enable hazard detection you must invoke `vlog` with the `-hazards` argument when you compile your source code and you must also invoke `vsim` with the `-hazards` argument when you simulate.

Note



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

Hazard Detection and Optimization Levels

In certain cases hazard detection results are affected by the optimization level used in the simulation. Some optimizations change the read/write operations performed on a variable if the transformation is determined to yield equivalent results. Because the hazard detection algorithm cannot determine whether the read/write operations can affect the simulation results, the optimizations can result in different hazard detection results. Generally, the optimizations reduce the number of false hazards by eliminating unnecessary reads and writes, but there are also optimizations that can produce additional false hazards.

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.
- A non-blocking assignment is not treated as a WRITE for hazard detection purposes. This is because non-blocking assignments are not normally involved in hazards. (In fact, they should be used to avoid hazards.)
- Hazards caused by simultaneous forces are not detected.

Debugging Signal Segmentation Violations

If you attempt to access a SystemVerilog object that has not been constructed with the **new** operator, you will receive a fatal error called a signal segmentation violation (SIGSEGV). For example, the following code produces a SIGSEGV fatal error:

```
class C;
  int x;
endclass

C obj;
initial obj.x = 5;
```

This attempts to initialize a property of *obj*, but *obj* has not been constructed. The code is missing the following:

```
C obj = new;
```

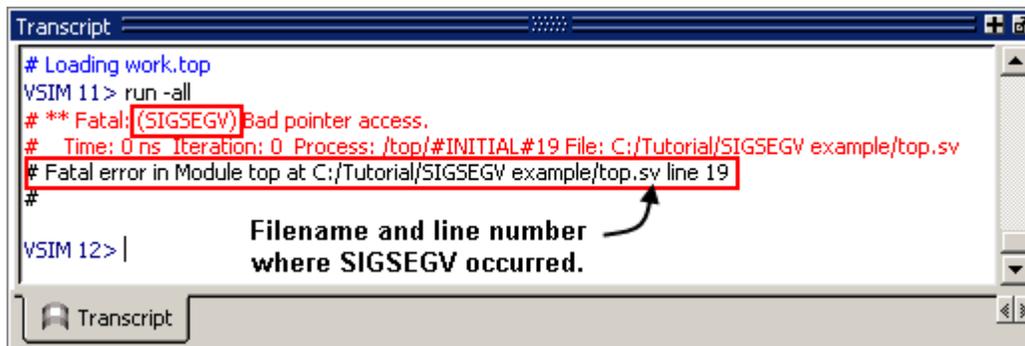
The **new** operator performs three distinct operations:

- Allocates storage for an object of type C
- Calls the “new” method in the class or uses a default method if the class does not define “new”
- Assigns the handle of the newly constructed object to “*obj*”

If the object handle *obj* is not initialized with **new**, there will be nothing to reference. ModelSim sets the variable to the value **null** and the SIGSEGV fatal error will occur.

To debug a SIGSEGV error, first look in the transcript. [Figure 8-1](#) shows an example of a SIGSEGV error message in the Transcript window.

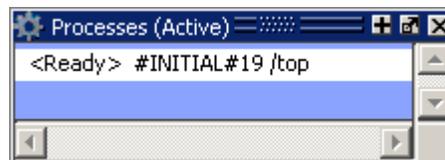
Figure 8-1. Fatal Signal Segmentation Violation (SIGSEGV)



The Fatal error message identifies the filename and line number where the code violation occurred (in this example, the file is *top.sv* and the line number is 19).

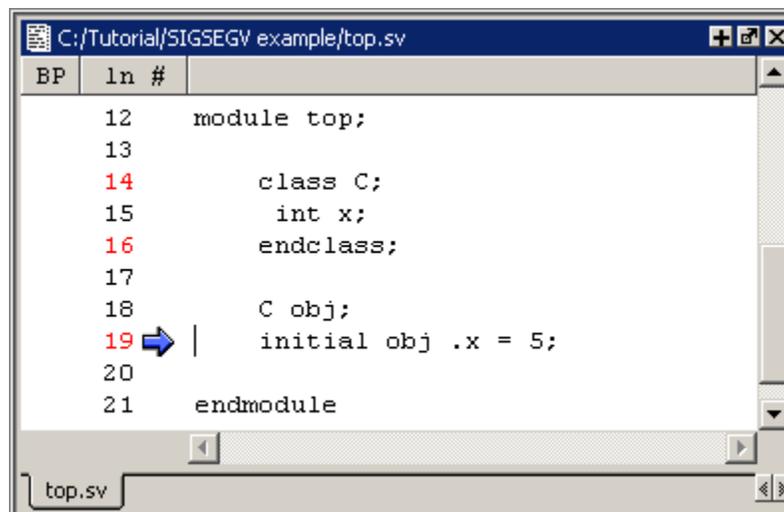
ModelSim sets the active scope to the location where the error occurred. In the Processes window, the current process is highlighted (Figure 8-2).

Figure 8-2. Current Process Where Error Occurred



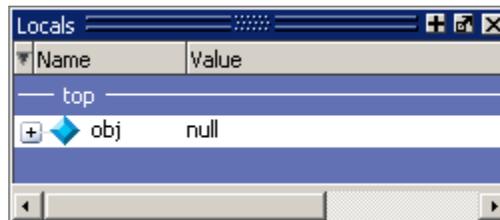
Double-click the highlighted process to open a Source window. A blue arrow will point to the statement where the simulation stopped executing (Figure 8-3).

Figure 8-3. Blue Arrow Indicating Where Code Stopped Executing



Next, look for *null* values in the ModelSim Locals window (Figure 8-4), which displays data objects declared in the local (current) scope of the active process.

Figure 8-4. Null Values in the Locals Window



The *null* value in Figure 8-4 indicates that the object handle for *obj* was not properly constructed with the **new** operator.

Negative Timing Checks

ModelSim automatically detects cells with negative timing checks and causes timing checks to be performed on the delayed versions of input ports (used when there are negative timing check limits). This is the equivalent of applying the `+delayed_timing_checks` switch with the `vsim` command.

`vsim +delayed_timing_checks`

Appropriately applying `+delayed_timing_checks` will significantly improve simulation performance.

To turn off this feature, specify `+no_autodtc` with `vsim`.

Negative Timing Check Limits

By default, ModelSim supports negative timing check limits in Verilog `$setphold` and `$recrem` system tasks. Using the `+no_neg_tcheck` argument with the `vsim` command causes all negative timing check limits to be set to zero.

Models that support negative timing check limits must be written properly if they are to be evaluated correctly. These timing checks specify delayed versions of the input ports, which are used for functional evaluation. The correct syntax for `$setphold` and `$recrem` is as follows.

`$setphold`

Syntax

```
$setphold(clk_event, data_event, setup_limit, hold_limit, [notifier], [tstamp_cond],  
          [tcheck_cond], [delayed_clk], [delayed_data])
```

Arguments

- The *clk_event* argument is required. It is a transition in a clock signal that establishes the reference time for tracking timing violations on the *data_event*. Since `$setphold` combines the functionality of the `$setup` and `$hold` system tasks, the *clk_event* sets the lower bound event for `$hold` and the upper bound event for `$setup`.

- The *data_event* argument is required. It is a transition of a data signal that initiates the timing check. The *data_event* sets the upper bound event for \$hold and the lower bound limit for \$setup.
- The *setup_limit* argument is required. It is a constant expression or specparam that specifies the minimum interval between the *data_event* and the *clk_event*. Any change to the data signal within this interval results in a timing violation.
- The *hold_limit* argument is required. It is a constant expression or specparam that specifies the interval between the *clk_event* and the *data_event*. Any change to the data signal within this interval results in a timing violation.
- The *notifier* argument is optional. It is a register whose value is updated whenever a timing violation occurs. The *notifier* can be used to define responses to timing violations.
- The *tstamp_cond* argument is optional. It 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 is optional. It 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 optional. It is a net that is continuously assigned the value of the net specified in the *clk_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.
- The *delayed_data* argument is optional. It is a net that is continuously assigned the value of the net specified in the *data_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.

You can specify negative times for either the *setup_limit* or the *hold_limit*, but the sum of the two arguments must be zero or greater. If this condition is not met, ModelSim zeroes the negative limit during elaboration or SDF annotation. To see messages about this kind of problem, use the **+ntc_warn** argument with the **vsim** command. A typical warning looks like the following:

```
** Warning: (vsim-3616) cells.v(x): Instance 'dff0' - Bad $setuphold
constraints: 5 ns and -6 ns. Negative limit(s) set to zero.
```

The *delayed_clk* and *delayed_data* arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the *delayed_clk* and *delayed_data* nets in place of the normal *clk* and *data* nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for *delayed_clk* and *delayed_data* such that the correct data is latched as long as a timing constraint has not been violated. See [Using Delayed Inputs for Timing Checks](#) for more information.

Optional arguments not included in the task must be indicated as null arguments by using commas. For example:

```
$setuphold(posedge CLK, D, 2, 4, , , tcheck_cond);
```

The \$setuphold task does not specify *notifier* or *tstamp_cond* but does include a *tcheck_cond* argument. Notice that there are no commas after the *tcheck_cond* argument. Using one or more commas after the last argument results in an error.

Note



Do not condition a \$setuphold timing check using the *tstamp_cond* or *tcheck_cond* arguments and a conditioned event. If this is attempted, only the parameters in the *tstamp_cond* or *tcheck_cond* arguments will be effective, and a warning will be issued.

\$recrem

Syntax

```
$recrem(control_event, data_event, recovery_limit, removal_limit, [notifier], [tstamp_cond],  
[tcheck_cond], [delayed_ctrl, [delayed_data]])
```

Arguments

- The *control_event* argument is required. It is an asynchronous control signal with an edge identifier to indicate the release from an active state.
- The *data_event* argument is required. It is clock or gate signal with an edge identifier to indicate the active edge of the clock or the closing edge of the gate.
- The *recovery_limit* argument is required. It is the minimum interval between the release of the asynchronous control signal and the active edge of the clock event. Any change to a signal within this interval results in a timing violation.
- The *removal_limit* argument is required. It is the minimum interval between the active edge of the clock event and the release of the asynchronous control signal. Any change to a signal within this interval results in a timing violation.
- The *notifier* argument is optional. It is a register whose value is updated whenever a timing violation occurs. The *notifier* can be used to define responses to timing violations.
- The *tstamp_cond* argument is optional. It conditions the *data_event* for the removal check and the *control_event* for the recovery check. This alternate method of conditioning precludes specifying conditions in the *control_event* and *data_event* arguments.
- The *tcheck_cond* argument is optional. It conditions the *data_event* for the recovery check and the *clk_event* for the removal check. This alternate method of conditioning precludes specifying conditions in the *control_event* and *data_event* arguments.

- The *delayed_ctrl* argument is optional. It is a net that is continuously assigned the value of the net specified in the *control_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.
- The *delayed_data* argument is optional. It is a net that is continuously assigned the value of the net specified in the *data_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.

You can specify negative times for either the *recovery_limit* or the *removal_limit*, but the sum of the two arguments must be zero or greater. If this condition is not met, ModelSim zeroes the negative limit during elaboration or SDF annotation. To see messages about this kind of problem, use the **+ntc_warn** argument with the **vsim** command.

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 *control* 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.

Optional arguments not included in the task must be indicated as null arguments by using commas. For example:

```
$recrem(posedge CLK, D, 2, 4, , , tcheck_cond);
```

The `$recrem` task does not specify *notifier* or *tstamp_cond* but does include a *tcheck_cond* argument. Notice that there are no commas after the *tcheck_cond* argument. Using one or more commas after the last argument results in an error.

Negative Timing Constraint Algorithm

The ModelSim negative timing constraint algorithm attempts to find a set of delays such that the data net is valid when the clock or control nets transition and the timing checks are satisfied. The algorithm is iterative because a set of delays that satisfies all timing checks for a pair of inputs can cause mis-ordering of another pair (where both pairs of inputs share a common input). When a set of delays that satisfies all timing checks is found, the delays are said to converge.

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

For example, in this timing check,

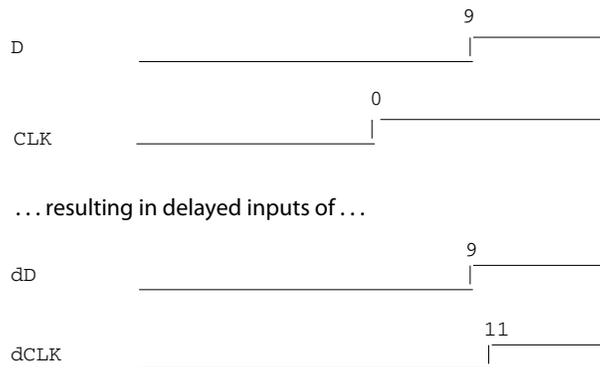
```
$setuphold(posedge CLK, D, -10, 20, notifier,, , dCLK, dD);
```

dCLK is the delayed version of the input *CLK* and *dD* is the delayed version of *D*. By default, the timing checks are performed on the inputs while the model's functional evaluation uses the delayed versions of the inputs. This posedge D-Flipflop module has a negative setup limit of -10 time units, which allows posedge *CLK* to occur up to 10 time units before the stable value of *D* is latched.



Without delaying *CLK* by 11, an old value for *D* could be latched. Note that an additional time unit of delay is added to prevent race conditions.

The inputs look like this:



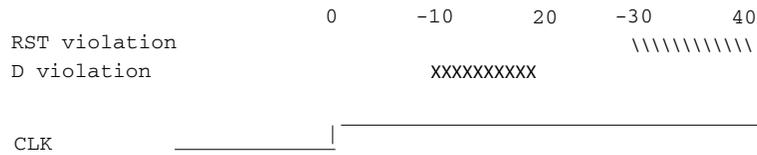
Because the posedge *CLK* transition is delayed by the amount of the negative setup limit (plus one time unit to prevent race conditions) no timing violation is reported and the new value of *D* is latched.

However, the effect of this delay could also affect other inputs with a specified timing relationship to *CLK*. The simulator is responsible for calculating the delay between all inputs and their delayed versions. The complete set of delays (delay solution convergence) must consider all timing check limits together so that whenever timing is met the correct data value is latched.

Consider the following timing checks specified relative to *CLK*:

`$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);`

`$setuphold(posedge CLK, negedge RST, -40, 50, notifier,,, dCLK, dRST);`



To solve the timing checks specified relative to *CLK* the following delay values are necessary:

	Rising	Falling
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	0

The simulator's intermediate delay solution shifts the violation regions to overlap the reference events.

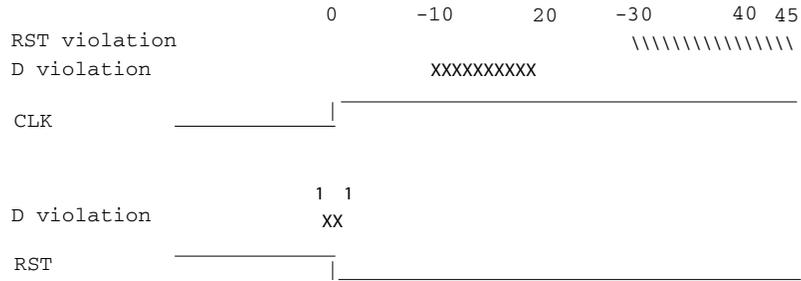


Notice that no timing is specified relative to *negedge CLK*, but the *dCLK* falling delay is set to the *dCLK* rising delay to minimize pulse rejection on *dCLK*. Pulse rejection that occurs due to delayed input delays is reported by:

"WARNING[3819] : Scheduled event on delay net dCLK was cancelled"

Now, consider the following case where a new timing check is added between *D* and *RST* and the simulator cannot find a delay solution. Some timing checks are set to zero. In this case, the new timing check is not annotated from an SDF file and a default \$setuphold limit of 1, 1 is used:

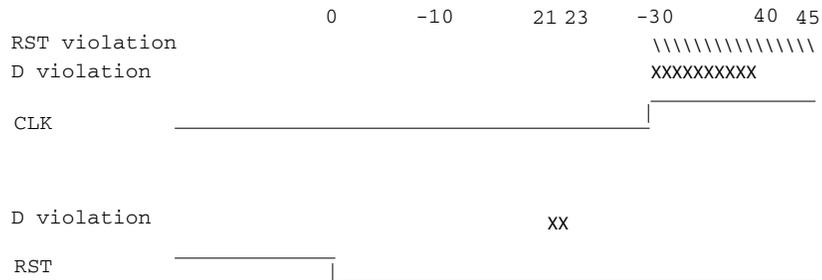
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
$setuphold(posedge CLK, negedge RST, -40, 50, notifier,,, dCLK, dRST);
$setuphold(negedge RST, D, 1, 1, notifier,,, dRST, dD);
```



As illustrated earlier, to solve timing checks on *CLK*, delays of 20 and 31 time units were necessary on *dD* and *dCLK*, respectively.

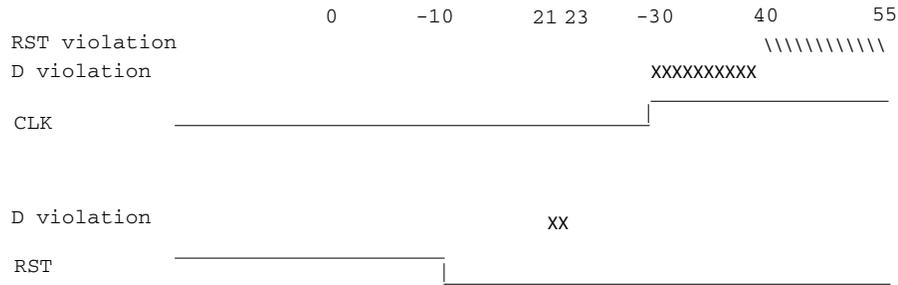
	Rising	Falling
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	0

The simulator's intermediate delay solution is:



But this is not consistent with the timing check specified between *RST* and *D*. The falling *RST* signal can be delayed by additional 10, but that is still not enough for the delay solution to converge.

	Rising	Falling
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	10



As stated above, if a delay solution cannot be determined with the specified timing check limits the smallest negative \$setup/\$recovery limit is zeroed and the calculation of delays repeated. If no negative \$setup/\$recovery limits exist, then the smallest negative \$hold/\$removal limit is zeroed. This process is repeated until a delay solution is found.

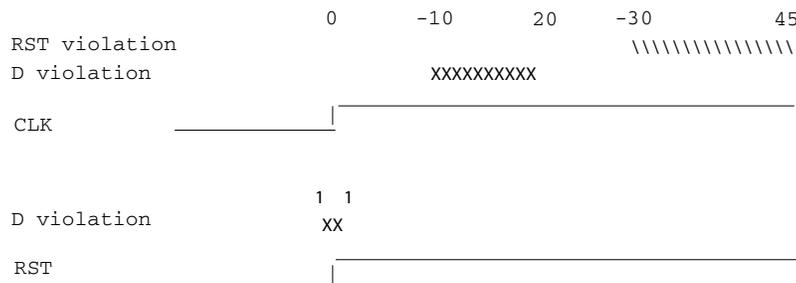
If a timing check in the design was zeroed because a delay solution was not found, a summary message like the following will be issued:

```
# ** Warning: (vsim-3316) No solution possible for some delayed timing
check nets. 1 negative limits were zeroed. Use +ntc_warn for more info.
```

Invoking `vsim` with the `+ntc_warn` option identifies the timing check that is being zeroed.

Finally consider the case where the *RST* and *D* timing check is specified on the posedge *RST*.

```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
$setuphold(posedge CLK, negedge RST, -40, 50, notifier,,, dCLK, dRST);
$setuphold(posedge RST, D, 1, 1, notifier,,, dRST, dD);
```



In this case the delay solution converges when an rising delay on *dRST* is used.

	Rising	Falling
<i>dCLK</i>	31	31

	Rising	Falling
<i>dD</i>	20	20
<i>dRST</i>	20	10



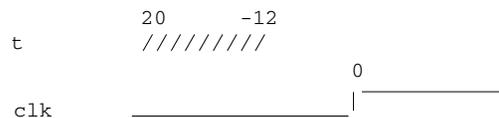
Using Delayed Inputs for Timing Checks

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

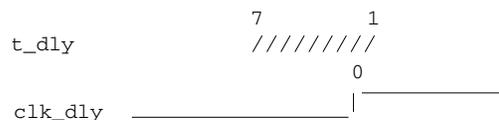
Consider an example. This timing check:

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

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



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



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

By far the greatest difference between these modes is evident when there are conditions on a delayed check event because the condition is not implicitly delayed. Also, timing checks

specified without explicit delayed signals are delayed, if necessary, when they reference an input that is delayed for a negative timing check limit.

Other simulators perform timing checks on the delayed inputs. To be compatible, ModelSim supports both methods.

Force and Release Statements in Verilog

The Verilog Language Reference Manual IEEE Std 1800-2009, dryopm 10.6.2, states that the left-hand side of a force statement cannot be a bit-select or part-select. Questa deviates from the LRM standard by supporting forcing of bit-selects, part-selects, and field-selects in your source code. The right-hand side of these force statements may not be a variable. Refer to the [force](#) command for more information.

Verilog-XL Compatible Simulator Arguments

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

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

Using Escaped Identifiers

ModelSim recognizes and maintains Verilog escaped identifier syntax. Prior to version 6.3, Verilog escaped identifiers were converted to VHDL-style extended identifiers with a backslash

at the end of the identifier. Verilog escaped identifiers then appeared as VHDL extended identifiers in simulation output and in command line interface (CLI) commands. For example, a Verilog escaped identifier like the following:

```
\ /top/dut/03
```

had to be displayed as follows:

```
\ /top/dut/03\
```

Starting in version 6.3, all object names inside the simulator appear identical to their names in original HDL source files.

Sometimes, in mixed language designs, hierarchical identifiers might refer to both VHDL extended identifiers and Verilog escaped identifiers in the same fullpath. For example, `top\VHDL*ext\ \Vlog*ext /bottom` (assuming the `PathSeparator` variable is set to '/') , or `top.\VHDL*ext\.\Vlog*ext .bottom` (assuming the `PathSeparator` variable is set to '.') Any fullpath that appears as user input to the simulator (such as on the `vsim` command line, in a `.do` file, on the `vopt` command line) should be composed of components with valid escaped identifier syntax.

A `modelsim.ini` variable called [GenerousIdentifierParsing](#) can control parsing of identifiers. If this variable is on (the variable is on by default: value = 1), either VHDL extended identifiers or Verilog escaped identifier syntax may be used for objects of either language kind. This provides backward compatibility with older `.do` files, which often contain pure VHDL extended identifier syntax, even for escaped identifiers in Verilog design regions.

Note that SDF files are always parsed in “generous mode.” SignalSpy function arguments are also parsed in “generous mode.”

On the `vsim` command line, the language-correct escaped identifier syntax should be used for top-level module names. Using incorrect escape syntax on the command line works in the incremental/debug flow, but not in the default optimized flow (see [Optimizing Designs with vopt](#)). This limitation may be removed in a future release.

Tcl and Escaped Identifiers

In Tcl, the backslash is one of a number of characters that have a special meaning. For example,

```
\n
```

creates a new line.

When a Tcl command is used in the command line interface, the TCL backslash should be escaped by adding another backslash. For example:

```
force -freeze /top/ix/iy/\ \yw\[1\]\ 10 0, 01 {50 ns} -r 100
```

The Verilog identifier, in this example, is `\yw[1]`. Here, backslashes are used to escape the square brackets (`[]`), which have a special meaning in Tcl.

For a more detailed description of special characters in Tcl and how backslashes should be used with those characters, click **Help > Tcl Syntax** in the menu bar, or simply open the `docs/tcl_help_html/TclCmd` directory in your QuestaSim installation.

Cell Libraries

Mentor Graphics has passed the Verilog test bench from the ASIC Council and achieved the “Library Tested and Approved” designation from Si2 Labs. This test bench is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors’ Verilog cell libraries are compatible with ModelSim Verilog.

The cell models generally contain Verilog “specify blocks” that describe the path delays and timing constraints for the cells. See Section 14 in the IEEE Std 1364-2005 for details on specify blocks, and Section 15 for details on timing constraints. ModelSim Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

SDF Timing Annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files. See [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 this two-input AND gate cell, the distributed delay for the AND primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even

so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364). This is the default behavior, but you can specify alternate delay modes with compiler directives and arguments. These arguments and directives are compatible with Verilog-XL. Compiler delay mode arguments take precedence over delay mode directives in the source code.

Distributed Delay Mode

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

Path Delay Mode

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

Unit Delay Mode

In unit delay mode, the non-zero distributed delays are set to one unit of simulation resolution (determined by the minimum `time_precision` argument in all `'timescale` directives in your design or the value specified with the `-t` argument to `vsim`), and the specify path delays and timing constraints are ignored. You can specify this delay mode with the **+delay_mode_unit** compiler argument or the **`delay_mode_unit** compiler directive.

Zero Delay Mode

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

Approximating Metastability

The ability to approximate metastability for Verilog gate-level designs is useful for simulating synchronizer flops used to synchronize inputs from one clock domain to another. It allows a known random state to be latched when timing between domains is violated.

Without this feature, you need to selectively use a version of the cell that does not generate unknowns from timing check violations, or disable/force notifiers to override the timing check violation toggle that introduced unknown states into the circuit.

Standard timing simulation Verilog cells have timing checks with notifier registers. The notifier registers are inputs to user defined primitives (UDPs), which are coded to generate an unknown output state when the notifier input changes.

During standard simulation, a timing check violation generates a violation message and the notifier register value is toggled. The notifier register change causes an evaluation of the UDP which generates an unknown state.

During metastable approximation the timing check operates as normal. When a timing check violation occurs, the notifier is toggled and the UDP is evaluated in a manner to approximate metastability. The metastable UDP evaluation does not generate an unknown state, but rather a random 1 or 0 logic state.

The metastability feature is implemented accepting a standard Verilog UDP description and interpreting it in a non-standard manner as follows.

Standard UDP definition for a notifier evaluation:

```
table
// clk  d    r  notif  : state : next_state
   ?    ?    ?    *      ?      x;
```

Metastable Approximation (non-standard Verilog):

```
table
// clk  d    r  notif  : state : next_state
   ?    ?    ?    *      ?      b;
```

where b is randomly generated 0 or 1 state from a specified seed.

Usage

To allow metastable approximation simulation, Verilog cells must be compiled with the proper optimization visibility (**vlog** or **vopt** command).

- **+acc=x[+<selection>[.]]** — provides the necessary visibility to allow metastability simulation of select Verilog cells.

To enable the metastable approximation during simulation, the following simulator command line option (**vsim** command) must be specified:

- **+notiftoggle01[+<seed>]** — uses the metastable UDP evaluation of enabled cells in simulation. Default seed value is 0.

System Tasks and Functions

ModelSim supports system tasks and functions as follows:

- All system tasks and functions defined in IEEE Std 1364
- Some system tasks and functions defined in SystemVerilog IEEE Std 1800-2005
- Several system tasks and functions that are specific to ModelSim

- Several non-standard, Verilog-XL system tasks

The system tasks and functions listed in this section are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI), Verilog Procedural Interface (VPI), or the SystemVerilog DPI (Direct Programming Interface). If the simulator issues warnings regarding undefined system tasks or functions, then it is likely that these tasks or functions are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 System Tasks and Functions

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

Note



You can use the [change](#) command to modify local variables in Verilog and SystemVerilog tasks and functions.

Table 8-3. IEEE Std 1364 System Tasks and Functions - 1

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

Table 8-4. IEEE Std 1364 System Tasks and Functions - 2

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

Table 8-4. IEEE Std 1364 System Tasks and Functions - 2 (cont.)

Probabilistic distribution functions	Conversion functions	Stochastic analysis tasks	Timing check tasks
\$random			\$width ¹ \$removal \$recrem

1. Verilog-XL ignores the threshold argument even though it is part of the Verilog spec. ModelSim does not ignore this argument. Be careful that you do not set the threshold argument greater-than-or-equal to the limit argument as that essentially disables the \$width check. Also, note that you cannot override the threshold argument by using SDF annotation.

Table 8-5. IEEE Std 1364 System Tasks

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
\$monitord	\$async\$or\$plane	\$dumpvars
\$monitord	\$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		

Table 8-6. IEEE Std 1364 File I/O Tasks

File I/O tasks

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

SystemVerilog System Tasks and Functions

The following system tasks and functions are supported by ModelSim and are described more completely in the Language Reference Manual (LRM) for SystemVerilog, IEEE Std 1800-2005.

Table 8-7. SystemVerilog System Tasks and Functions - 1

Expression size function	Range function
\$bits	\$isunbounded

Table 8-8. SystemVerilog System Tasks and Functions - 2

Shortreal conversions	Array querying functions	Assertion severity tasks	Assertion control tasks
\$shortrealbits	\$dimensions	\$fatal	\$asserton
\$bitstoshortreal	\$left	\$error	\$assertoff
	\$right	\$warning	\$assertkill

Table 8-8. SystemVerilog System Tasks and Functions - 2 (cont.)

Shortreal conversions	Array querying functions	Assertion severity tasks	Assertion control tasks
	\$low	\$info	
	\$high		
	\$increment		
	\$size		

Table 8-9. SystemVerilog System Tasks and Functions - 3

Assertion functions	Random number functions	Coverage functions
\$onehot	\$urandom	\$set_coverage_db_name
\$onehot0	\$urandom_range	\$load_coverage_db
\$isunknown	\$srandom	\$get_coverage
		\$coverage_save

Table 8-10. SystemVerilog System Tasks and Functions - 4

Reading packed data functions	Writing packed data functions	Other functions
\$readmemb	\$writememb	\$root
\$readmemh	\$writememh	\$unit

Using the `$coverage_save` System Function

Implementation of the `$coverage_save()` SystemVerilog system function is now compliant with the IEEE1800 standard. Existing SV calls to this function that worked in ModelSim version 10.0 and earlier will not work in version 10.1. They will fail compilation checks. The recommended action for existing designs where the existing behavior is acceptable, is to migrate to the `$coverage_save_mti()` system call to retain the exact pre-10.1 behavior (see [\\$coverage_save_mti](#)).

The new behavior of `$coverage_save()` is significantly different from the original behavior because the original implementation pre-dated the standardization process. An alternate workaround is provided for customers for whom source changes are not possible: use the `vsim -usenonstdcoveragesavesysf` switch to replace the implementation of the built-in system function with the non-standard variant. The action of this switch is global and thus affects all calls to `$coverage_save()`. As such it is not recommended as a long-term solution.

Simulator-Specific System Tasks and Functions

Table 8-11 lists system tasks and functions that are specific to ModelSim. They are not included in the IEEE Std 1364, nor are they likely supported in other simulators. Their use may limit the portability of your code.

Table 8-11. Simulator-Specific Verilog System Tasks and Functions

\$disable_signal_spy	\$sprintf()
\$enable_signal_spy	\$sdf_done
\$init_signal_driver	\$signal_force
\$init_signal_spy	\$signal_release
\$messagelog	\$wlfdumpvars()
\$coverage_save_mti	\$get_initial_random_seed

[\\$coverage_save_mti](#)

Syntax

`$coverage_save_mti(<filename>, [<instancepath>], [<xml_output>])`

Description

The `$coverage_save()` system function is defined in IEEE Std 1800, as explained above in [Using the \\$coverage_save System Function](#). The pre-standardization behavior is retained for backwards-compatibility by the `$coverage_save_mti()` system function.

The `$coverage_save_mti()` system function saves only Code Coverage information to a file during a batch run that typically would terminate with the `$finish` call. It returns a “0” to indicate that the coverage information was saved successfully or a “-1” to indicate an error (unable to open file, instance name not found, and so forth.)

If you do not specify `<instancepath>`, ModelSim saves all coverage data in the current design to the specified file. If you do specify `<instancepath>`, ModelSim saves data on that instance, and all instances below it (recursively), to the specified file.

If set to 1, the `[<xml_output>]` argument specifies that the output be saved in XML format.

See [Code Coverage](#) for more information on Code Coverage.

[\\$get_initial_random_seed](#)

Syntax

`$get_initial_random_seed`

Description

The `$get_initial_random_seed` system function returns the integer value of the initial random seed, as specified via the `Sv_Seed modelsim.ini` variable or by using the `-sv_seed vsim` command-line option.

\$messagelog

Syntax

```
$messagelog({"<message>", <value>...}[, ...]);
```

Arguments

- `<message>` — Your message, enclosed in quotation marks ("), using text and specifiers to define the output.
- `<value>` — A scope, object, or literal value that corresponds to the specifiers in the `<message>`. You must specify one `<value>` for each specifier in the `<message>`.

Specifiers

The `$messagelog` task supports all specifiers available with the `$display` system task. For more information about `$display`, refer to section 17.1 of the IEEE std 1364-2005.

The following specifiers are specific to `$messagelog`.

Note



The format of these custom specifiers differ from the `$display` specifiers. Specifically, “%:” denotes a `$messagelog` specifier and the letter denotes the type of specifier.

- `:%:C` — Group/Category

A string argument, enclosed in quotation marks ("). This attribute defines a group or category used by the message system. If you do not specify `:%:C`, the message system logs **User** as the default.

- `:%:F` — Filename

A string argument specifying a simple filename, relative path to a filename, or a full path to a filename. In the case of a simple filename or relative path to a filename, the simulator accepts what you specify in the message output, but internally it uses the current directory to complete these paths to form a full path—this allows the message viewer to link to the specified file.

If you do not include `:%:F`, the simulator automatically logs the value of the filename in which the `$messagelog` is called.

If you do include `:%:R`, `:%:F`, or `:%:L`, or a combination of any two of these, the simulator does not automatically log values for the undefined specifier(s).

- **:%I** — Message ID

A string argument. The Message Viewer displays this value in the ID column. This attribute is not used internally, therefore you do not need to be concerned about uniqueness or conflict with other message IDs.

- **:%L** — Line number

An integer argument.

If you do not include **:%L**, the simulator automatically logs the value of the line number on which the `$messagelog` is called.

If you do include **:%R**, **:%F**, or **:%L**, or a combination of any two of these, the simulator does not automatically log values for the undefined specifier(s).

- **:%O** — Object/Signal Name

A hierarchical reference to a variable or net, such as `sig1` or `top.sigx[0]`. You can specify multiple **:%O** for each `$messagelog`, which effectively forms a list of attributes of that kind, for example:

```
$messagelog("The signals are %:O, %:O, and %:O.",  
           sig1, top.sigx[0], ar [3].sig);
```

- **:%R** — Instance/Region name

A hierarchical reference to a scope, such as `top.sub1` or `sub1`. You can also specify a string argument, such as `"top.mychild"`, where the identifier inside the quotes does not need to correlate with an actual scope, it can be an artificial scope.

If you do not include **:%R**, the simulator automatically logs the instance or region in which the `$messagelog` is called.

If you do include **:%R**, **:%F**, or **:%L**, or a combination of any two of these, the simulator does not automatically log values for the undefined specifier(s).

- **:%S** — Severity Level

A case-insensitive string argument, enclosed in quotes ("), that is one of the following:

Note — This is the default if you do not specify **:%S**

Warning

Error

Fatal

Info — The error message system recognizes this as a Note

Message — The error message system recognizes this as a Note

- **:%V** — Verbosity Rating

An integer argument, where the default is zero (0). The verbosity rating allows you to specify a field you can use to sort or filter messages in the Message Viewer. In most cases you specify that this attribute is not printed, using the tilde (~) character.

Description

- Non-printing attributes (~) — You can specify that an attribute value is not to be printed in the transcribed message by placing the tilde (~) character after the percent (%) character, for example:

```
$messagelog("%::~ Do not print the Severity Level", "Warning");
```

However, the value of %:S is logged for use in the Message Viewer.

- Logging of simulation time — For each call to \$messagelog, the simulation time is logged, however the simulation time is not considered an attribute of the message system. This time is available in the Message Viewer.
- Minimum field-width specifiers — are accepted before each specifier character, for example:

```
%:0I  
%:10I
```

- Left-right justification specifier (-) — is accepted as it is for \$display.
- Macros — You can use the macros ‘__LINE__ (returns line number information) and ‘__FILE__ (returns filename information) when creating your \$messagelog tasks. For example:

```
module top;  
  
function void wrapper(string file, int line);  
    $messagelog("Hello: The caller was at %:F,%:0L", file, line);  
endfunction  
  
initial begin  
    wrapper(`__FILE__, `__LINE__);  
    wrapper(`__FILE__, `__LINE__);  
end  
  
endmodule
```

which would produce the following output

```
# Hello: The caller was at test.sv,7  
# Hello: The caller was at test.sv,8
```

Examples

- The following \$messagelog task:

```
$messagelog("hello world");
```

transcripts the message:

```
hello world
```

while logging all default attributes, but does not log a category.

- The following \$messagelog task:

```
$messagelog("%:~S%0t: PCI-X burst read started in transactor %:R",  
"Note", $time - 50, top.sysfixture.pcix);
```

transcripts the message:

```
150: PCI-X burst read started in transactor top.sysfixture.pcix
```

while silently logging the severity level of “Note”, and uses a direct reference to the Verilog scope for the %:R specifier, and does not log any attributes for %:F (filename) or %:L (line number).

- The following \$messagelog task:

```
$messagelog("%:~V%:S %:C-%:I,%:L: Unexpected AHB interrupt received  
in transactor %:R", 1, "Error", "AHB", "UNEXPINTRPT", `__LINE__,  
ahbtop.c190);
```

transcripts the message:

```
** Error: AHB-UNEXPINTRPT,238: Unexpected AHB interrupt received in  
transactor ahbtop.c190
```

where the verbosity level (%:V) is “1”, severity level (%:S) is “Error”, the category (%:C) is “AHB”, and the message identifier (%:I) is “UNEXPINTRPT”. There is a direct reference for the region (%:R) and the macro ‘__LINE__ is used for line number (%:L), resulting in no attribute logged for %:F (filename).

\$psprintf()

Syntax

```
$psprintf()
```

Description

The \$psprintf() system function behaves like the \$sformat() file I/O task except that the string result is passed back to the user as the function return value for \$psprintf(), not placed in the first argument as for \$sformat(). Thus \$psprintf() can be used where a string is valid. Note that at this time, unlike other system tasks and functions, \$psprintf() cannot be overridden by a user-defined system function in the PLI.

\$sdf_done

Syntax

```
$sdf_done
```

Description

This task is a “cleanup” function that removes internal buffers, called MIPDs, that have a delay value of zero. These MIPDs are inserted in response to the -v2k_int_delay argument to the [vsim](#) command. In general, the simulator automatically removes all zero delay MIPDs. However, if

you have `$sdf_annotate()` calls in your design that are not getting executed, the zero-delay MIPDs are not removed. Adding the `$sdf_done` task after your last `$sdf_annotate()` removes any zero-delay MIPDs that have been created.

\$wlfdumpvars()

This Verilog system task specifies variables to be logged in the current simulation's WLF file (default, *vsim.wlf*) and is called from within a Verilog design. It is equivalent to the Verilog system task `$dumpvars`, except it dumps values to the current simulation's WLF file instead of a VCD file. While it can not be called directly from within VHDL, it can log VHDL variables contained under a Verilog scope that is referenced by `$wlfdumpvars`. The `modelsim.ini` variable **WildcardFilter** will be used to filter types when a scope is logged by `$wlfdumpvars`. Multiple scopes and variables are specified as a comma separated list.

Syntax

```
$wlfdumpvars(<levels>, {<scope> | <variable>}[, <scope> | <variable>]);
```

Arguments

- `<levels>`
Specifies the number of hierarchical levels to log, if a scope is specified. Specified as a non-negative integer.
- `<scope>`
Specifies a Verilog pathname to a scope, under which all variables are logged.
- `<variable>`
Specifies a variable to log.

Examples

- Log variable "addr_bus" in the current scope

```
$wlfdumpvars(0, addr_bus);
```
- Log all variables within the scope "alu", and in any submodules

```
$wlfdumpvars(2, alu);
```
- Log all variables within the scope regfile

```
$wlfdumpvars(1, $root.top.alu.regfile)
```

Verilog-XL Compatible System Tasks and Functions

ModelSim supports a number of Verilog-XL specific system tasks and functions.

Supported Tasks and Functions Mentioned in IEEE Std 1364

The following supported system tasks and functions, though not part of the IEEE standard, are described in an annex of the IEEE Std 1364.

\$countdrivers
\$getpattern
\$readmemb
\$readmemh

Supported Tasks and Functions Not Described in IEEE Std 1364

The following system tasks are also provided for compatibility with Verilog-XL, though they are not described in the IEEE Std 1364.

\$deposit(variable, value);

This system task sets a Verilog register or net to the specified value. **variable** is the register or net to be changed; **value** is the new value for the register or net. The value remains until there is a subsequent driver transaction or another \$deposit task for the same register or net. This system task operates identically to the ModelSim **force -deposit** command.

\$disable_warnings("<keyword>"[,<module_instance>...]);

This system task instructs ModelSim to disable warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you do not specify a module instance, ModelSim disables warnings for the entire simulation.

\$enable_warnings("<keyword>"[,<module_instance>...]);

This system task enables warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you do not specify a module_instance, ModelSim enables warnings for the entire simulation.

\$system("command");

This system function takes a literal string argument, executes the specified operating system command, and displays the status of the underlying OS process. Double quotes are required for the OS command. For example, to list the contents of the working directory on Unix:

```
$system("ls -l");
```

Return value of the **\$system** function is a 32-bit integer that is set to the exit status code of the underlying OS process.

Note

There is a known issue in the return value of this system function on the win32 platform. If the OS command is built with a cygwin compiler, the exit status code may not be reported correctly when an exception is thrown, and thus the return code may be wrong. The workaround is to avoid building the application using cygwin or to use the switch **-mno-cygwin** in cygwin on the gcc command line.

\$systemf(list_of_args)

This system function can take any number of arguments. The list_of_args is treated exactly the same as with the \$display() function. The OS command that runs is the final output from \$display() given the same list_of_args. Return value of the \$systemf function is a 32-bit integer that is set to the exit status code of the underlying OS process.

Note

There is a known issue in the return value of this system function on the win32 platform. If the OS command is built with a cygwin compiler, the exit status code may not be reported correctly when an exception is thrown, and thus the return code may be wrong. The workaround is to avoid building the application using cygwin or to use the switch **-mno-cygwin** in cygwin on the gcc command line.

Extensions to Supported System Tasks

Additional functionality has been added to the \$fopen, \$setuphold, and \$recrem system tasks.

New Directory Path With \$fopen

The #fopen systemtask has been extended to create a new directory path if the path does not currently exist. You must set the [CreateDirForFileAccess](#) modelsim.ini variable to '1' to enable this feature. For example: your current directory contains the directory “dir_1 with no other directories below it and the CreateDirForFileAccess variable is set to “1”. Executing the following line of code:

```
fileno = $fopen("dir_1/nodir_2/nodir_3/testfile", "w");
```

creates the directory path nodir_2/nodir_3 and opens the file “testfile” in write mode.

Negative Timing Checks With \$setuphold and \$recrem

The \$setuphold and \$recrem system tasks have been extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as in Verilog-XL. Refer to [Negative Timing Check Limits](#) for more information.

Unsupported Verilog-XL System Tasks

The following system tasks are Verilog-XL system tasks that are not implemented in ModelSim Verilog, but have equivalent simulator commands.

\$input("filename")

This system task reads commands from the specified filename. The equivalent simulator command is **do <filename>**.

\$list[(hierarchical_name)]

This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the Structure (sim) window. The corresponding source code is displayed in a 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**.

Compiler Directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary.

Many of the compiler directives (such as **`timescale**) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default

by a ``resetall` directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line could be significant. For example, if you have a file that defines some common macros for the entire design, then you might need to place it first in the list of files to be compiled.

The ``resetall` directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364):

```
`celldefine
`default_decay_time
`default_nettype
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`protect
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

IEEE Std 1364 Compiler Directives

The following compiler directives are described in detail in the IEEE Std 1364.

```
`celldefine
`default_nettype
`define
`else
`elsif
`endcelldefine
`endif
`ifdef
`ifndef
`include
`line
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef
```

Compiler Directives for vlog

The following directives are specific to ModelSim and are not compatible with other simulators.

```
`protect ... `endprotect
```

This directive pair allows you to encrypt selected regions of your source code. The code in ``protect` regions has all debug information stripped out. This behaves exactly as if using:

```
vlog -nocode=ports+pli
```

except that it applies to selected regions of code rather than the whole file. This enables usage scenarios such as making module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

The ``protect` directive is ignored by default unless you use the `+protect` argument to `vlog`. Once compiled, the original source file is copied to a new file in the current work directory. The name of the new file is the same as the original file with a “p” appended to the suffix. For example, “top.v” is copied to “top.vp”. This new file can be delivered and used as a replacement for the original source file.

A usage scenario might be that a vendor uses the ``protect` / ``endprotect` directives on a module or a portion of a module in a file named *encrypt.v*. They compile it with `vlog +protect encrypt.v` to produce a new file named *encrypt.vp*. You can compile *encrypt.vp* just like any other verilog file. The protection is not compatible among different simulators, so the vendor must ship you a different *encrypt.vp* than they ship to someone who uses a different simulator.

You can use `vlog +protect=<filename>` to create an encrypted output file, with the designated filename, in the current directory (not in the *work* directory, as in the default case where [`=<filename>`] is not specified). For example:

```
vlog test.v +protect=test.vp
```

If the filename is specified in this manner, all source files on the command line are concatenated together into a single output file. Any ``include` files are also inserted into the output file.

``protect` and ``endprotect` directives cannot be nested.

If errors are detected in a protected region, the error message always reports the first line of the protected block.

``include`

If any ``include` directives occur within a protected region, the compiler generates a copy of the include file with a *.vp* suffix and protects the entire contents of the include file. However, when you use `vlog +protect` to generate encrypted files, the original source files must all be complete Verilog modules or packages. Compiler errors result if you attempt to perform compilation of a set of parameter declarations within a module.

You can avoid such errors by creating a dummy module that includes the parameter declarations. For example, if you have a file that contains your parameter declarations and a file that uses those parameters, you can do the following:

```
module dummy;
```

```
\protect
\include "params.v" // contains various parameters
\include "tasks.v" // uses parameters defined in params.v
\endprotect
endmodule
```

Then, compile the dummy module with the `+protect` switch to generate an encrypted output file with no compile errors.

vlog +protect dummy

After compilation, the work library contains encrypted versions of `params.v` and `tasts.v`, called `params.vp` and `tasks.vp`. You may then copy these encrypted files out of the work directory to more convenient locations. These encrypted files can be included within your design files; for example:

```
module main
  \include "params.vp"
  \include "tasks.vp"
  ...
endmodule
```

Though other simulators have a ``protect` directive, the algorithm ModelSim uses to encrypt source files is different. As a result, even though an uncompiled source file with ``protect` is compatible with another simulator, once the source is compiled in ModelSim, you could not simulate it elsewhere.

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](#) for details.

``delay_mode_path`

This directive sets distributed delays to zero in favor of path delays. See [Delay Modes](#) for details.

``delay_mode_unit`

This directive sets path delays to zero and non-zero distributed delays to one time unit. See [Delay Modes](#) for details.

``delay_mode_zero`

This directive sets path delays and distributed delays to zero. See [Delay Modes](#) for details.

`uselib

This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler arguments. See [Verilog-XL uselib Compiler Directive](#) for details.

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

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

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

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

Sparse Memory Modeling

Sparse memories are a mechanism for allocating storage for memory elements only when they are needed. You mark which memories should be treated as sparse, and ModelSim dynamically allocates memory for the accessed addresses during simulation.

Sparse memories are more efficient in terms of memory consumption, but access times to sparse memory elements during simulation are slower. Thus, sparse memory modeling should be used only on memories whose active addresses are “few and far between.”

There are two methods of enabling sparse memories:

- “Manually” by inserting attributes or meta-comments in your code
- Automatically by setting the [SparseMemThreshold](#) variable in the *modelsim.ini* file

Manually Marking Sparse Memories

You can mark memories in your code as sparse using either the *mti_sparse* attribute or the *sparse* meta-comment. For example:

```
(* mti_sparse *) reg mem [0:1023]; // Using attribute
reg /*sparse*/ [0:7] mem [0:1023]; // Using meta-comment
```

The meta-comment syntax is supported for compatibility with other simulators.

You can identify memories as “not sparse” by using the `+nosparse` switch to `vlog` or `vopt`.

Automatically Enabling Sparse Memories

Using the `SparseMemThreshold` .ini variable, you can instruct ModelSim to mark as sparse any memory that is a certain size. Consider this example:

If `SparseMemThreshold = 2048` then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

The variable `SparseMemThreshold` is set, by default, to 1048576.

Combining Automatic and Manual Modes

Because *mti_sparse* is a Verilog 2001 attribute that accepts values, you can enable automatic sparse memory modeling but still control individual memories within your code. Consider this example:

If `SparseMemThreshold = 2048` then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

However, you can override this automatic behavior using *mti_sparse* with a value:

```
(* mti_sparse = 0 *) reg mem[0:2047];
// will *not* be marked as sparse even though SparseMemThreshold = 2048

(* mti_sparse = 1*) reg mem[0:2046];
// will be marked as sparse even though SparseMemThreshold = 2048
```

Priority of Sparse Memories

The following list describes the priority in which memories are labeled as sparse or not sparse:

1. `vlog` or `vopt +nosparse[+]` — These memories are marked as “not sparse”, where `vlog` options override `vopt` options.

2. metacomment `/* sparse */` or attribute `(* mti_sparse *)` — These memories are marked “sparse” or “not sparse” depending on the attribute value.
3. SparseMemThreshold .ini variable — Memories as deep as or deeper than this threshold are marked as sparse.

Determining Which Memories Were Implemented as Sparse

To identify which memories were implemented as sparse, use this command:

write report -l

The `write report` command lists summary information about the design, including sparse memory handling. You would issue this command if you are not certain whether a memory was successfully implemented as sparse or not. For example, you might add a `/*sparse*/` metacomment above a multi-D SystemVerilog memory, which is not supported. In that case, the simulation will function correctly, but ModelSim will use a non-sparse implementation of the memory.

If you are planning to optimize your design with `vopt`, be sure to use the `+acc` argument in order to make the sparse memory visible, thus allowing the `write report -l` command to report the sparse memory.

Limitations

There are certain limitations that exist with sparse memories:

- Sparse memories can have only one packed dimension. For example:

```
reg [0:3] [2:3] mem [0:1023]
```

has two packed dimensions and cannot be marked as sparse.

- Sparse memories can have only one unpacked dimension. For example:

```
reg [0:1] mem [0:1][0:1023]
```

has two unpacked dimensions and cannot be marked as sparse.

- Dynamic and associative arrays cannot be marked as sparse.
- Memories defined within a structure cannot be marked as sparse.
- PLI functions that get the pointer to the value of a memory will not work with sparse memories. For example, using the `tf_nodeinfo()` function to implement `$fread` or `$fwrite` will not work, because ModelSim returns a NULL pointer for `tf_nodeinfo()` in the case of sparse memories.
- Memories that have parameterized dimensions like the following example:

```
parameter MYDEPTH = 2048;  
reg [31:0] mem [0:MYDEPTH-1];
```

cannot be processed as a sparse memory *unless* the design has been optimized with the `vopt` command. In optimized designs, the memory is implemented as a sparse memory, and all parameter overrides to that MYDEPTH parameter are treated correctly.

Verilog PLI/VPI and SystemVerilog DPI

ModelSim supports the use of the Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) and the SystemVerilog DPI (Direct Programming Interface). These interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. For more information on the ModelSim implementation, refer to [Verilog Interfaces to C](#).

Standards, Nomenclature, and Conventions

The product's implementation of the Verilog VPI is based on the following standards:

- IEEE 1364-2005 and 1364-2001 (Verilog)
- IEEE 1800-2005 (SystemVerilog)

ModelSim supports partial implementation of the Verilog VPI. For release-specific information on currently supported implementation, refer to the following text file located in the ModelSim installation directory:

```
<install_dir>/docs/technotes/Verilog_VPI.note
```

Extensions to SystemVerilog DPI

This section describes extensions to the SystemVerilog DPI for ModelSim.

- SystemVerilog DPI extension to support automatic DPI import tasks and functions.

You can specify the automatic lifetime qualifier to a DPI import declaration in order to specify that the DPI import task or function can be reentrant.

ModelSim supports the following addition to the SystemVerilog DPI import tasks and functions (additional support is in bold):

```
dpi_function_proto ::= function_prototype  
  
function_prototype ::= function [lifetime] data_type_or_void  
function_identifier ( [ tf_port_list ] )  
  
dpi_task_proto ::= task_prototype
```

```
task_prototype ::= task [lifetime] task_identifier  
( [ tf_port_list ] )  
  
lifetime ::= static | automatic
```

The following are a couple of examples:

```
import DPI-C cfoo = task automatic foo(input int p1);  
import DPI-C context function automatic int foo (input int p1);
```

OVM-Aware Debug

OVM-aware debugging provides you, the verification or design engineer, with information, at the OVM abstraction level, that connects you to the OVM base-class library.

Preparing Your Simulation for OVM-Aware Debug

This section describes the steps you must take to enable the OVM-aware debugging features in your OVM environment.

Prerequisites

- Design Source — SystemVerilog testbench based on the Open Verification Methodology (OVM) v2.0 or greater. Refer to ovmworld.org for more details.
- Precompiled OVM Library — You must use the precompiled OVM library (mtiOvm) provided in the installation. This library contains the necessary infrastructure used to enable the OVM-aware debugging capabilities.

Procedure

1. Compilation — You must use the OVM source included in your installation directory to take advantage of the built in debugging features. Here are three compilation scenarios that may apply to your environment.
 - If your OVM design does not require macros, you can use a command similar to:

```
vlog top.sv
```

and the compiler will use the precompiled OVM library (mtiOvm).
 - If your OVM requires macros, you must also include the ovm-2.0, or greater, source files similar to:

```
vlog top.sv \  
+includir+<install_dir>/verilog_src/ovm-<version>/src/
```
 - If you cannot use the precompiled OVM and need to compile the OVM source directly, you must specify a +define of OVM_DEBUGGER as follows:

```
vlog top.sv \  
+incdir+<install_dir>/verilog_src/ovm-<version>/src/ \  
+define+OVM_DEBUGGER \  
<install_dir>/verilog_src/ovm-<version>/src/ovm_pkg.sv
```

2. Optimization — You can explicitly or implicitly run vopt. There are no special settings to enable OVM-aware debugging.
3. Elaboration — You must specify the -OVMdebug switch on the vsim command line. Note that the switch is case-sensitive. This instructs the simulator to collect the debugging information about your OVM environment.
4. GUI — Display the OVM-aware debugging windows, [OVM Globals Window](#) and [OVM Hierarchy Window](#), by executing the command:

```
view ovm
```

You could also display these windows from the **View > OVM** menu items.

5. Simulation — The OVM Hierarchy window will be empty until the testbench creates the first OVM environment components. As soon as the simulation enters the OVM build phase, the OVM structure is built up and the OVM Hierarchy window becomes populated. You can enter the OVM build phase by running the simulation to a particular time or by setting a breakpoint in your design.

OVM-Aware Debugging Tasks

This section describes OVM-aware debugging tasks you can perform on your OVM environment.

Locating Blocking Calls in the OVM Hierarchy

If your OVM environment is at a point where there are blocking calls, the [OVM Globals Window](#) contains a tree labeled Blockers. You can use this tree to locate, in the [OVM Hierarchy Window](#), the corresponding element to a given blocker.

1. Expand the Blockers tree in the OVM Globals window.
2. Select one of the “Blocker” entries in the tree.

This adds a green arrow to the OVM Hierarchy window indicating the location of the corresponding element.

3. In the OVM Hierarchy window, continue to expand the tree until the green arrow disappears and the corresponding element to the blocker is selected. Note also that the element is also highlight green if you select any other element of the hierarchy.
4. Right-click on the selected hierarchy element and select “View Sequence Details” for additional information.

Finding Matching Get and Set Configurations

For a given OVM component you can view which get configurations have a matching set configuration, and vice versa.

1. Select an element in the OVM Hierarchy window with a type of either “component” or “sequencer”.
2. Right-click the element and select View Component Details from the pop-up menu.
This displays an OVM Component window specific to that component or sequencer.
3. In the OVM Component window, expand the Get Configurations tree.
 - Any get configuration highlighted in green can be expanded to show the location that sets the configuration.
Select any matching (green) configuration and your OVM Hierarchy window will select the corresponding element.
 - Any get configuration highlighted in red does not have a matching set configuration.
4. In the OVM Component window expand the Set Configurations tree.
 - Any set configuration highlighted in green can be expanded to show the location(s) that gets the configuration.
Select any matching (green) configuration and your OVM Hierarchy window will highlight the corresponding elements in green or directly select a single component that gets that configuration.
 - Any set configuration highlighted in red does not have a matching get configuration.

OVM-Aware Debug Windows

This section describes the four OVM windows for you to use during OVM-aware debug. The contents of these windows will change as you advance through the simulation.

OVM Globals Window

The OVM Globals window contains information that is not specific to an OVM component. Specifically it contains information about phases, barriers and blockers.

- Phases — Phases are a way to synchronize OVM components. Expand this tree to show the phases and the current phase.
- Barriers — Barriers, like phases, provide a synchronization mechanism. Unlike Phases there is no known relationship between barriers, any defined barrier will be shown.
- Blockers — During the simulation, threads may be stopped on various OVM calls. This tree shows the list of all currently blocked processes.

Display this window by selecting the **View > OVM > OVM Globals** menu item.

OVM Hierarchy Window

The OVM Hierarchy window contains hierarchical information about your OVM environment, including:

- **Name** — Provides a hierarchical view of the OVM classes used. The names are based on the class names created in your OVM environment.
- **Type** — Identifies the type of object listed in the Name column. Examples include: component, sequencer, ovm_port, tlm, fifo, sequencer, sequence.
- **Phase** — Identifies whether the object in the Name column has been started or completed.

Display this window by selecting the **View > OVM > OVM Hierarchy** menu item.

OVM Components Window

The OVM Components window contains information about the specific component, including:

- **Get Configurations** — and where they are set from.
- **Set Configurations** — and where they are used.
- **Stop Request** information

Display this window by selecting right-clicking on a component in the OVM Hierarchy window and selecting “View Component Details”.

OVM Sequence Window

The OVM Sequence window contains textual information about the selected sequence and its current state

Display this window by right-clicking on a sequence in the OVM Hierarchy window and selecting “View Sequence Details”.

UVM-Aware Debug

UVM-aware debugging provides you, the verification or design engineer, with information, at the UVM abstraction level, that connects you to the UVM base-class library.

ModelSim comes with the built-in capability of UVM-aware debug. In the ModelSim installation tree, besides including the latest qualified, pre-compiled version of the Accelera UVM package library, it also contains a pre-compiled DPI library necessary for UVM integration into ModelSim, and a ModelSim-specific UVM debugging package called

"questa_uvm_pkg". These three pieces work together to provide not only ModelSim's basic UVM support, but also its UVM-aware debug capabilities.

Preparing Your Simulation for UVM-Aware Debug

To use the built-in flow's functionality no special compile switches are required. At compile time the UVM package directory will be automatically included on the command line when the presence of a `uvm_pkg` import is detected. Even if the ModelSim optimization flow (`vopt`) is used, symbol and source information for SystemVerilog class data (only) is automatically preserved.

Simulating With UVM-Aware Debug Enabled

At design elaboration time, when a `uvm_pkg` is detected, the simulator will also automatically pull in the UVM DPI library and the `questa_uvm_pkg`. Again, no special options are required. By default, the simulator will pull in UVM component hierarchy information to populate the Structure window, and it will condition the simulator's random number generation to be consistent as long as the UVM-aware functionality is not completely disabled. One thing to remember is that UVM elaboration does not happen until after the simulator has been elaborated, at time 0. So to see the UVM component hierarchy in the Structure window you must initially issue a "run 0" command.

Besides the Structure window population and the random stability functionality, UVM-aware debug has other available features. These are all controlled with the `vsim` switch `-uvmcontrol`. In ModelSim these include: Message Viewer display of UVM messages, Wave window viewing of UVM transactions, and preparation of an elaborated design hierarchy database for the Certe tool.

ModelSim UVM-aware debug tries not to adversely impact simulation performance, so the default UVM debug setting is `-uvmcontrol=struct`, as described above. To enable all of the features of the `questa_uvm_pkg`, the `-uvmcontrol=all` option can be used. The "all" option is simple to specify but it could enable unnecessary functionality. Consequently, individual features available with `-uvmcontrol` can be enabled and disabled with a keyword list. For more information on the `-uvmcontrol` option, see the [vsim](#) command description in the Reference Manual.

If you enable UVM message logging integration (via the `vsim` the `-uvmcontrol=all` or `uvmcontrol=msglog` option) you should also enable the Message Viewing functionality with the `-msgmode` option. This is because, by default, messages will only appear in the transcript window. To have messages populate the Message Viewer window the `-msgmode` setting must be set to either "wlf" or "both".

To enable the display of UVM transactions in the Wave window use the `-uvmcontrol=trlog` option of the `vsim` command.

An orthogonal but useful and related feature is the `vsim -classdebug` feature. Among other things, it allows class handles to be used on the command line to examine class contents, and populates the Class Instance browser. For more information on the `-classdebug` option, see the [vsim](#) command description in the Reference Manual.

In summary, the fullest UVM-aware debug flow is available if `vsim` tool is invoked with the following options:

```
vsim -classdebug -msgmode both -uvmcontrol=struct,msglog,trlog \  
    <Design_Name>
```

You can set these `vsim` option as your default settings by adding them to the `[vsim]` section of the `modelsim.ini` file. For example:

```
[vsim]  
ClassDebug = 1  
msgmode = both  
UVMControl = struct,msglog,trlog
```


Chapter 9

SystemC Simulation

This chapter describes how to compile and simulate SystemC designs with ModelSim. ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) proof-of-concept SystemC simulator. This includes the Transaction Level Modeling (TLM) Library, Release 2.0. It is recommended that you obtain the OSCI functional specification, or the latest version of the IEEE Std 1666-2005, *IEEE Standard SystemC Language Reference Manual*.

In addition to the functionality described in the OSCI specification, ModelSim for SystemC includes the following features:

- Single common Graphic Interface for SystemC and HDL languages.
- Extensive support for mixing SystemC, VHDL, Verilog, and SystemVerilog in the same design (SDF annotation for HDL only). For detailed information on mixing SystemC with HDL see [Mixed-Language Simulation](#).

Supported Platforms and Compiler Versions

SystemC runs on a subset of ModelSim supported platforms. The table below shows the currently supported platforms and compiler versions:

Table 9-1. Supported Platforms for SystemC

Platform/OS	Supported compiler versions	32-bit support	64-bit support
Intel and AMD x86-based architectures (32- and 64-bit) SUSE Linux Enterprise Server 9.0, 9.1, 10, 11 Red Hat Enterprise Linux 3, 4, 5	gcc 4.3.3 gcc 4.5.0 VCO is linux (32-bit binary) VCO is linux_x86_64 (64-bit binary)	yes	yes
Windows ¹ XP, Vista and 7	Minimalist GNU for Windows (MinGW) gcc 4.2.1	yes	no

1. SystemC supported on this platform with gcc-4.2.1-mingw32vc9.

Note



ModelSim SystemC has been tested with the gcc versions provided in the install tree. It is strongly recommended to use the gcc version that came with your installation—customized versions of gcc may cause problems.

Building gcc with Custom Configuration Options

The gcc configuration for ModelSim has been qualified only for default options. If you use advanced gcc configuration options, ModelSim may not work with those options.

To use a custom gcc build, set the `CppPath` variable in the `modelsim.ini` file. This variable specifies the pathname to the compiler binary you intend to use.

When using a custom gcc, ModelSim requires that you build the custom gcc with several specific configuration options. These vary on a per-platform basis, as shown in the following table:

Table 9-2. Custom gcc Platform Requirements

Platform	Mandatory configuration options
Linux	none
Win32 (MinGW)	--with-gnu-as --with-gnu-ld

- `sjlj-exceptions` or `setjump longjump` exceptions do not work with SystemC. It can cause problems with catching exceptions thrown from `SC_THREAD` and `SC_CTHREAD`.
- Always build the compiler with `--disable-sjlj-exceptions` and never with `--enable-sjlj-exceptions`.
- `binutils-2.17` and `binutils-2.18` do not work. Do not attempt to use those on win32 at least.

If you do not have a GNU `binutils2.16` assembler and linker, you can use the `as` and `ld` programs. They are located inside the gcc in directory:

`<install_dir>/lib/gcc-lib/<gnuplatform>/<ver>`

The location of the `as` and `ld` executables has changed since gcc-3.4. For all gcc-4.x releases, `as` and `ld` are located in:

`<install_dir>/libexec/gcc/<gnuplatform>/<ver>`

By default ModelSim also uses the following options when configuring built-in gcc:

- `--disable-nls`
- `--enable-languages=c,c++`

These are not mandatory, but they do reduce the size of the gcc installation.

Usage Flow for SystemC-Only Designs

ModelSim allows users to simulate SystemC, either alone or in combination with other VHDL/Verilog modules.

The following is an overview of the usage flow for strictly SystemC designs. The remainder of this chapter provides more detailed information on how to use SystemC designs with ModelSim.

1. Create and map the working design library with the **vlib** and **vmap** statements, as needed.
2. If you are simulating **sc_main()** as the top-level, skip to Step 3. Also, refer to “[Recommendations for using sc_main at the Top Level](#),” below.

If you are simulating a SystemC top-level module instead, then modify the SystemC source code to export the top level SystemC design unit(s) using the **SC_MODULE_EXPORT** macro. Refer to “[Modifying SystemC Source Code](#)” for information and examples on how to convert **sc_main()** to an equivalent module.

3. Analyze the SystemC source using the **sccom** command, which invokes the native C++ compiler to create the C++ object files in the design library.

See [Using sccom in Addition to the Raw C++ Compiler](#) for information on when you are required to use **sccom** as opposed to another C++ compiler.

4. Perform a final link of the C++ source using **sccom -link**. This process creates a shared object file in the current work library which will be loaded by **vsim** at runtime.

You must rerun **sccom -link** before simulation if any new **sccom** compiles were performed.

5. Load the design into the simulator using the standard ModelSim **vsim** command.
6. Run the simulation using the **run** command, which you enter at the **VSIM>** command prompt.
7. Debug the design using ModelSim GUI features, including the Source and Wave windows.

Recommendations for using sc_main at the Top Level

Generally, your design should include **sc_main()** at the top level in order for ModelSim to run the SystemC/C++ source code. ModelSim executes **sc_main()** as a thread process. This allows you to use test bench code and C++ variables (including SystemC primitive channels, ports, and modules) inside **sc_main()**.

If your design does not have **sc_main()** at the top level, you must apply several modifications to your original SystemC source code—refer to “[Modifying SystemC Source Code](#).”

Example 9-1. Simple SystemC-only `sc_main()`

```
int
sc_main(int, char*[])
{
    design_top t1 = new design_top("t1");
    sc_start(-1);
    delete t1;
    return 1;
}
```

Prerequisites

- Must be running ModelSim 6.3 or higher.

Procedure

To simulate in ModelSim using `sc_main()` as the top-level in your design:

1. Run `vsim` with `sc_main` as the top-level module:

```
vsim -c sc_main
```

2. Explicitly name all simulation objects for mixed-language designs, or to enable debug support of objects created by `sc_main()`. Pass the declared name as a constructor arguments, as follows:

```
sc_signal<int> sig("sig");
top_module* top = new top("top");
```

i **Tip:** For SystemC-only designs, the simulation runs even if debug support is not enabled. Mixed language designs, however, will not elaborate if explicit naming is not performed in `sc_main()`. ModelSim issues an error message to this effect.

3. Optionally, override the default stack size (10Mb) for `sc_main()` in the `modelsim.ini` file:

```
ScMainStackSize 1 Gb
```

See [ScMainStackSize](#) variable for more information.

Concepts

- ModelSim executes `sc_main()` in two parts:
 - The code before the first call to `sc_start()` — executed during the construction phase of all other design tops.
 - The code after the first `sc_start()` or any other subsequent `sc_start()`'s — executed based on the `sc_start()` arguments.

The overall simulation is controlled by the ModelSim prompt and the `sc_start()` call does not proceed unless an appropriate **run** command is issued from the ModelSim prompt. `sc_start()` always yields to ModelSim for the time specified in its argument. Example:

```

int
sc_main(int, char*[])
{
    top t1("t1");
    sc_signal<int> reset("reset");
    t1.reset(reset);
    t2->reset(reset);
    sc_start(100, SC_NS); <----- 1st part executed during
                                construction. Yield to the kernel
                                for 100 ns.

    reset = 1;                <----- Executed only if
                                run 100 ns or more is issued
                                from batch or GUI prompt.

    sc_start(100, SC_NS); <----- Yield to the kernel for another
                                100 ns

    return 1;                <----- Executed only if
                                the simulation is run for
                                more than 200 ns.
}

```

`sc_start(-1)` in the OSCI simulator means that the simulation is run until the time it is halted by `sc_stop()`, or because there were no future events scheduled at that time. The `sc_start(-1)` in means that `sc_main()` is yielding to the ModelSim simulator until the current simulation session finishes.

- Avoid `sc_main()` going out of scope — Since `sc_main()` is run as a thread, it must not go out of scope or delete any simulation objects while the current simulation session is running. The current simulation session is active unless a quit, restart, `sc_stop`, `$finish`, or `assert` is executed, or a new design is loaded. To avoid `sc_main()` from going out of scope or deleting any simulation objects, `sc_main()` must yield control to the ModelSim simulation kernel before calling any delete and before returning from `sc_main`. In ModelSim, `sc_start(-1)` gives control to the ModelSim kernel until the current simulation session is exited. Any code after the `sc_start(-1)` is executed when the current simulation ends.

```

int
sc_main(int, char*[])
{
    top t1("t1");
    top* t2 = new top("t2");
    sc_signal<int> reset("reset");
    t1.reset(reset);
    t2->reset(reset);
    sc_start(100, SC_NS); <----- 1st part executed during
                                construction. yield to the kernel
                                for 100 ns.

    reset = 1;                <----- Will be executed only if
                                run 100 ns or more is issued
                                from batch or GUI prompt.

    sc_start(100, SC_NS); <----- Yield to the kernel for another 100 ns
}

```

```
    sc_start(-1);          <----- Will cause sc_main() to
                           suspend until the end of
                           the current simulation session

    delete t2;           <----- Will be executed at the
                           end of the current simulation
                           session.

    return 1;
}
```

If the run command specified at the simulation prompt before ending the current simulation session exceeds the cumulative `sc_start()` times inside `sc_main()`, the simulation continues to run on design elements instantiated both by `sc_main()` and outside of `sc_main()`. For example, in this case, if `sc_main()` instantiates an `sc_clock`, the clock will continue to tick if the simulation runs beyond `sc_main()`.

On the other hand, if the current simulation ends before the cumulative `sc_start()` times inside `sc_main`, the remainder of the `sc_main` will be executed before quitting the current simulation session if the `ScMainFinishOnQuit` variable is set to 1 in the `modelsim.ini` file. If this variable is set to 0, the remainder of `sc_main` will not be executed. The default value for this variable is 1. One drawback of not completely running `sc_main()` is that memory leaks might occur for objects created by `sc_main`. Also, it is possible that simulation stimulus and execution of the test bench will not be complete, and thus the simulation results will not be valid.

- `sc_cycle(sc_time)` is deprecated in SystemC 2.2. A suggested alternative to `sc_cycle` is `sc_start(sc_time)`. In case of a cycle accurate design, this will yield the same behavior. ModelSim will always convert `sc_cycle()` to `sc_start()` with a note.
- `sc_initialize()` is also deprecated in SystemC 2.2. The replacement for `sc_initialize()` is `sc_start(SC_ZERO_TIME)`. ModelSim treats `sc_initialize()` as `sc_start(SC_ZERO_TIME)`.
- ModelSim treats `sc_main()` as a top-level module and creates a hierarchy called `sc_main()` for it. Any simulation object created by `sc_main()` will be created under the `sc_main()` hierarchy in ModelSim. For example, for the `sc_main()` described above, the following hierarchy will be created:

```
 /
 |
 |-- sc_main
 |   |-- t1
 |   |-- t2
 |   |-- reset
```

Creating Shared Object Files for SystemC Code

The simulator has the ability to create intermediate shared object files for SystemC. These intermediate shared object files can then be linked together to create the final shared object file for simulation use. This method of managing can reduce both disk space and linking time for large SystemC environments.

Prerequisites

- Must be running ModelSim 6.6a or higher.

Procedure

Consider the following scenario: You have a SystemC file which is used in all of your tests, *common.cpp*, and then you have test-specific SystemC files, such as *test1.cpp*, *test2.cpp*, etc. The following procedure is an example of how you can manage your tests and common code.

1. Create a library for your intermediate shared object:

```
vlib common
```

2. Compile all common SystemC files into this library:

```
sccom -work common common.cpp
```

3. Link the files to create an intermediate shared object:

```
sccom -linkshared -work common
```

4. Create a library for your SystemC test code:

```
vlib test1
```

```
vlib test2
```

5. Compile the test-specific code into each library:

```
sccom -work test1 test1.cpp
```

```
sccom -work test2 test2.cpp
```

6. Link the test specific object files with the shared object in each of the test libraries:

```
sccom -link -libshared common -lib test1 -work test1
```

```
sccom -link -libshared common -lib test2 -work test2
```

where `-libshared` specifies the location of the intermediate shared object, `-lib` specifies the library that contains the compiled object files, and `-work` specifies the location of the final `systemc.so`.

7. Run the tests:

```
vsim -lib work1 top
```

```
vsim -lib work2 top
```

Binding to Verilog or SystemVerilog Designs

The SystemVerilog **bind** construct allows you to bind a Verilog or SystemVerilog design unit to a SystemC module. This is especially useful for binding SystemVerilog assertions to your SystemC, VHDL, Verilog and mixed designs during verification. See [Using SystemVerilog bind Construct in Mixed-Language Designs](#).

Limitations of Bind Support for SystemC

There exists certain restrictions on actual expressions when binding to SystemC targets. If the target of a bind is a SystemC module or an instance of a SystemC module, expressions and literals are not supported as actuals. These include, but are not limited to,

- bitwise binary expressions using operators `&`, `|`, `~`, `^` and `^~`
- concatenation expression
- bit select and part select expressions
- variable/constant

Distributing SystemC IP

This section describes several methods you can use to distribute your SystemC IP for others to use in a ModelSim environment.

One requirement is that you, the IP provider, must distribute the IP library for each major release version (such as 10.0 or 10.1). Patch releases (such as 10.0b or 10.1a) are mostly backward compatible, and therefore do not require you to recompile libraries with each letter release. However, sometimes the SystemC header files may be modified. Therefore, in such situations, you must distribute a recompiled library for a patch release.

- Distribute source files

You can distribute non-compiled source files that can be compiled and simulated by the IP user for use with `sccom` and `vsim` along with their user code.

- Distribute IP as archived libraries

To create a static library perform the following actions:

```
vlib <work_library>  
sccom <options> <source files>  
sccom -archive <archive_file_name>
```

where you distribute the archived library *archive_file_name* to IP users. However, you should note that there will not be any debug information generated for the IP.

- Distribute the IP as shared libraries

To create a shared library perform the following actions:

```
vlib <work library>  
sccom <options> <source files>  
sccom -linkshared -work <work_library>
```

where **sccom -linkshared** creates an intermediate SystemC shared library at `<work_library>/_sc/<platform_compiler-version>/systemc.so`.

You should inform the IP user that they must link this intermediate shared object along with their code to create final `systemc.so`, which will be loaded during simulation. Refer to the section “[Creating Shared Object Files for SystemC Code](#)” for more information.

Compiling SystemC Files

To compile SystemC designs, you must:

- Create a design library
- Modify SystemC source code if using design units as top-level
- Run `sccom` SystemC compiler
- Run SystemC linker (`sccom -link`)

Creating a Design Library for SystemC

Use `vlib` to create a new library in which to store the compilation results. For example:

```
vlib work
```

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

The **work** library is actually a subdirectory named `work`. This subdirectory contains a special file named `_info`.

Note



Do not create libraries using UNIX commands—always use the `vlib` command.

See [Design Libraries](#) for additional information on working with libraries.

Converting `sc_main()` to a Module

Since it is natural for simulators to elaborate design-unit(s) as tops, it is recommended that you use design units as your top-level rather than relying on `sc_main` based elaboration and simulation. There are a few limitations and requirements for running a `sc_main()` based simulation.

If you have a `sc_main()` based design and would like to convert it to a design-unit based one, a few modifications must be applied to your SystemC source code. To see example code containing the code modifications detailed in [Modifying SystemC Source Code](#), see [Code Modification Examples](#).

Exporting All Top-Level SystemC Modules

For SystemC designs, you must export all top level modules in your design to ModelSim. You do this with the `SC_MODULE_EXPORT(<sc_module_name>)` macro. SystemC templates are not supported as top level or boundary modules. See [Templatized SystemC Modules](#). The `sc_module_name` is the name of the top level module to be simulated in ModelSim. You must specify this macro in a C++ source (`.cpp`) file. If the macro is contained in a header file instead of a C++ source file, an error may result.

Invoking the SystemC Compiler

ModelSim compiles one or more SystemC design units with a single invocation of `sccom`, the SystemC compiler. The design units are compiled in the order that they appear on the command line. For SystemC designs, all design units must be compiled just as they would be for any C++ compilation. An example of an `sccom` command might be:

```
sccom -I ../myincludes mytop.cpp mydut.cpp
```

Compiling Optimized and/or Debug Code

By default, `sccom` invokes the C++ compiler (`g++` or `aCC`) without any optimizations. If desired, you can enter any `g++/aCC` optimization arguments at the `sccom` command line.

Also, source level debug of SystemC code is not available by default in ModelSim. To compile your SystemC code for source level debugging in ModelSim, use the `g++/aCC -g` argument on the `sccom` command line.

Reducing Compilation Time for Non-Debug Simulations

If the SystemC objects in the design need not be visible in the ModelSim simulation database, you can save compilation time by running `sccom` with the `-nodebug` argument. This bypasses the parser which creates the ModelSim debug database. However, all files containing an `SC_MODULE_EXPORT()` macro call must NOT be compiled with the `sccom -nodebug` argument, otherwise the design fails to load.

This approach is useful if you are running a design in regression mode, or creating a library (.a) from the object files (.o) created by sccom, to be linked later with the SystemC shared object.

Specifying an Alternate g++ Installation

Mentor Graphics recommends using the version of g++ that is shipped with ModelSim on its various supported platforms. However, if you want to use your own installation, you can do so by setting the **CppPath** variable in the *modelsim.ini* file to the g++ executable location.

For example, if your g++ executable is installed in */u/abc/gcc-4.2.1/bin*, then you would set the variable as follows:

```
CppPath /u/abc/gcc-4.2.1/bin/g++
```

Maintaining Portability Between OSCI and the Simulator

If you intend to simulate on both ModelSim and the OSCI proof-of-concept SystemC simulator, you can use the `MTI_SYSTEMC` macro to execute the ModelSim specific code in your design only when running ModelSim. **Sccom** defines this macro by default during compile time.

Using the original and modified code shown in the example shown in [Example 9-9](#), you might write the code as follows:

```
#ifndef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
    sc_signal<bool> mysig;
    mymod mod;

    SC_CTOR(mytop)
        : mysig("mysig"),
          mod("mod")
    {
        mod.outp(mysig);
    }
};

SC_MODULE_EXPORT(top);

#else //Otherwise, it compiles this
int sc_main(int argc, char* argv[])
{
    sc_signal<bool> mysig;
    mymod mod("mod");
    mod.outp(mysig);

    sc_start(100, SC_NS);
}
#endif
```

Using sccom in Addition to the Raw C++ Compiler

When compiling complex C/C++ test bench environments, it is common to compile code with many separate runs of the compiler. Often, you may compile code into archives (.a files), and then link the archives at the last minute using the `-L` and `-l` link options.

When using SystemC, you may also want to compile a portion of your C design using raw `g++` or `aCC` instead of **sccom**. (Perhaps you have some legacy code or some non-SystemC utility code that you want to avoid compiling with **sccom**.) You can do this; however, some cautions and rules apply.

Rules for sccom Use

The rules governing when and how you must use **sccom** are as follows:

- You must compile all code that references SystemC types or objects using **sccom**.
- When using **sccom**, you should not use the `-I` compiler option to point the compiler at any search directories containing OSCI or any other vendor supplied SystemC header files. **sccom** does this for you accurately and automatically.
- If you do use the raw C++ compiler to compile C/C++ functionality into archives or shared objects, you must then link your design using the `-L` and `-l` options with the **sccom -link** command. These options effectively pull the non-SystemC C/C++ code into a simulation image that is used at runtime.

Failure to follow the above rules can result in link-time or elaboration-time errors due to mismatches between the OSCI or any other vendor supplied SystemC header files and the ModelSim SystemC header files.

Rules for Using Raw g++ to Compile Non-SystemC C/C++ Code

If you use raw `g++` to compile your non-systemC C/C++ code, the following rules apply:

1. The `-fPIC` option to `g++` should be used during compilation with **sccom**.
2. For C++ code, you must use the built-in `g++` delivered with ModelSim, or (if using a custom `g++`) use the one you built and specified with the **CppPath** variable in the *modelsim.ini* file.

Otherwise binary incompatibilities may arise between code compiled by **sccom** and code compiled by raw `g++`.

Compiling Changed Files Only (Incremental Compilation)

You can use `sccom -incr` to enable automatic incremental compilation so that only changed files are compiled. This allows ModelSim to determine which source files have changed and recompile only those source files.

A changed file is re-compiled in the following cases:

- Its pre-processor output is different from the last time it was successfully compiled (see [Note](#) below). This includes changes in included header files and to the source code itself.
- You invoke sccom with a different set of command-line options that have an impact on the gcc command line. Preserving all settings for the gcc command ensures that ModelSim re-compiles source files when a different version of gcc is used or when a platform changes.

Note

Pre-processor output is used because it prevents compilation on a file with the following types of changes:

- Access or modification time (touch)
 - Changes to comments—except changes to the source code that affect line numbers (such as adding a comment line) will cause all affected files to be recompiled. This occurs to keep debug information current so that ModelSim can trace back to the correct areas of the source code.
-

Example

The following example shows how to compile a SystemC design with automatic incremental compilation.

1. Run sccom -incr on three files and re-link all compiled files in the design.

```
% sccom -incr top.cpp and2.cpp or2.cpp
Model Technology ModelSim SE sccom DEV compiler 2003.05 Mar  2 2008
Exported modules:
    top
% sccom -incr -link
Model Technology ModelSim SE sccom DEV compiler 2003.05 Mar  2 2008
```

2. After changing functional content of the top module, re-compile and re-link.

```
% sccom -incr top.cpp and2.cpp or2.cpp
Model Technology ModelSim SE sccom DEV compiler 2003.05 Mar  2 2008
-- Skipping file and2.cpp
-- Skipping file or2.cpp

Exported modules:
    top

% sccom -incr -link
Model Technology ModelSim SE sccom DEV compiler 2003.05 Mar  2 2008
```

3. Link again without actually changing any file.

```
% sccom -incr -link
```

```
Model Technology ModelSim SE sccom DEV compiler 2003.05 Mar  2 2008  
-- Skipping linking
```

Note



You must compile all included libraries (using `-lib`) with `-incr` for automatic incremental compilation to work in linking mode. Failing to do so generates an error.

Limitations

- Automatic incremental compile is only supported for source files compiled with `sccom`. ModelSim does not track files for changes if they are compiled directly using a C++ compiler.
- Physically moving the library that holds a shared object forces re-creating that shared object next time. This applies only to the directories holding the shared object, not to the libraries that hold object files.
- If the SystemC source file includes a static library, then any change in that static library will not cause ModelSim to recompile the source file.
- If a design file consists of more than one SystemC module, changing even one module causes ModelSim to recompile the entire source file (and all the modules contained in it), regardless of whether the other modules were changed or not.
- Automatic incremental archiving is not supported (if you use the `-archive` argument, the `-incr` argument has no effect).

Issues with C++ Templates

Templatized SystemC Modules

Templatized SystemC modules are not supported for use in the following locations:

- the top level of the design
- the boundary between SystemC and higher level HDL modules (for instance, the top level of the SystemC branch)

To convert a top level templatized SystemC module, you can either specialize the module to remove the template, or you can create a wrapper module that you can use as the top module.

For example, assume you have the following templatized SystemC module:

```
template <class T>  
class top : public sc_module  
{  
    sc_signal<T> sig1;  
    ...  
};
```

You can specialize the module by setting `T = int`, thereby removing the template, as follows:

```
class top : public sc_module
{
    sc_signal<int> sig 1;
    ...
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```
class modelsim_top : public sc_module
{
    top<int> actual_top;
    ...
};

SC_MODULE_EXPORT(modelsim_top);
```

Organizing Templated Code

Suppose you have a class template, and it contains a certain number of member functions. All those member functions must be visible to the compiler when it compiles any instance of the class. For class templates, the C++ compiler generates code for each unique instance of the class template. Unless the compiler can read the full implementation of the class template, it cannot generate code for it, which leaves the invisible parts as undefined. Since it is legal to have undefined symbols in a `.so` file, **scom -link** will not produce any errors or warnings. To make functions visible to the compiler, you must move them to the `.h` file.

Generating SystemC Verification Extensions

The data introspection for SystemC verification (SCV) depends on partial template specialization of a template called `scv_extensions`. This template extends data objects with the abstract interface `scv_extensions_if`. Each specialization of the `scv_extensions` template implements the `scv_extensions_if` interface in a way appropriate to the type in the template parameter.

This section introduces a utility (`scom -dumpscvext`) that automatically generates SCV extensions for any given type of data object.

Usage

You must include the declaration of all types (for which you want extensions to be generated) in a header file.

For example, assume you want to generate extensions for `packet_t`.

1. Define a header file similar to the following:

```
typedef struct {
    int packet_type;
```

```
int src;  
int dest;  
int payload;  
}packet_t;
```

2. Creates a C++ file (.cpp) that includes all the header files that have all the type declarations and define a global variable for each type you want to extend.

Result: The C++ file for the above type looks like this:

```
#include "test.h"  
packet_t pack;
```

3. For class templates, you need to instantiate each specialization.

For example, if `packet_t` were a class template, you could do something like this:

```
packet_t<int> pack1;  
packet_t<long> pack2;  
...
```

4. Run the `sccom -dumpscvext` command to dump SCV extensions for all the types for whom global variables have been defined in the C++ file.

`sccom -dumpscvext mypacket.cpp`

where `mypacket.cpp` is the name of the C++ file containing global variable definitions.

Result: The generated extensions are displayed in stdout (similar to the way `scgenmod` dumps a foreign module declaration).

Note



You must define global variables for all types for which extensions need to be generated. The `sccom -dumpscvext` command will cause an error out if it cannot find any global variables defined in the supplied C++ file.

The command also automatically inserts the following header in `mypacket.cpp` with the generated extensions:

```
#ifndef TYPENAME_H  
#define TYPENAME_H  
#include "scv.h"  
<generated extensions>  
  
#endif
```

Note



If extensions are generated for more than one type, the type name of the first type will be used as `TYPENAME` in the `ifndef` preprocessor.

Supported Object Types

Table 9-3 shows the target list of simple data types that are supported by the `scom -dumpscvext` command, along with the extension generated for each type.

Table 9-3. Generated Extensions for Each Object Type

SystemC Data Object Type	Generated Extension
bool	scv_extensions<bool>
char	scv_extensions<char>
short	scv_extensions<short>
int	scv_extensions<int>
long	scv_extensions<long>
long long	scv_extensions<long long>
unsigned char	scv_extensions<unsigned char>
unsigned short	scv_extensions<unsigned short>
unsigned int	scv_extensions<unsigned int>
unsigned long	scv_extensions<unsigned long>
unsigned long long	scv_extensions<unsigned long long>
float	scv_extensions<float>
double	scv_extensions<double>
string	scv_extensions<string>
pointer	scv_extensions<T*>
array	scv_extensions<T[N]>
sc_string	scv_extensions<sc_string>
sc_bit	scv_extensions<sc_bit>
sc_logic	scv_extensions<sc_logic>
sc_int	scv_extensions<sc_int<W>>
sc_uint	scv_extensions<sc_uint<W>>
sc_bigint	scv_extensions<sc_bigint<W>>
sc_biguint	scv_extensions<sc_biguint<W>>
sc_bv	scv_extensions<sc_bv<W>>
sc_lv	scv_extensions<sc_lv<W>>

SCV Extensions for User-specified Types

This section explains the rules for generating SCV extensions for user-specified types such as structures, unions, classes, and enums.

Structures and Classes

Note the following set of rules for generating a SCV extensions for a structure or class:

- Generated extensions start with macro `SCV_EXTENSIONS()`, and typename is the name of the user-specified type.
- All types in the generated extension are public and follow the same mapping table as simple types.
- Private members of the struct/class are ignored unless the extensions class is made a friend of the user-specified type. In the latter case, all private members of the class are made public in the generated extension.
- Generated extensions contain a constructor defined by the macro `SCV_EXTENSIONS_CTOR()`, and typename is the name of the user-specified type.
- A `SCV_FIELD` entry is added in constructor for each generated extension.

The following examples demonstrate the generation process for a structure and class types.

Example 9-2. Generating SCV Extensions for a Structure

```
/* SystemC type */

struct packet_t {
    sc_uint<8> addr;
    sc_uint<12> data;
};

/* Generated SCV Extention */

SCV_EXTENSIONS(packet_t) {
    public:
        scv_extensions< sc_uint<8> > addr;
        scv_extensions< sc_uint<12> > data;

        SCV_EXTENSIONS_CTOR(packet_t) {
            SCV_FIELD(addr);
            SCV_FIELD(data);
        }
};
```

Example 9-3. Generating SCV Extensions for a Class without Friend (Private Data Not Generated)

```
/* SystemC type */
```

```
class restricted_t {
    public:
        sc_uint<8> public_data;
    private:
        sc_uint<8> private_data;
};

/* Generated SCV Extension */

SCV_EXTENSIONS(restricted_t) {
    public:
        scv_extensions< sc_uint<8> > public_data;

        SCV_EXTENSIONS_CTOR(restricted_t) {
            SCV_FIELD(public_data);
        }
};
```

Example 9-4. Generating SCV Extensions for a Class with Friend (Private Data Generated)

```
/* SystemC type */

class restricted_t {
    friend class scv_extensions<restricted_t>;

    public:
        sc_uint<8> public_data;
    private:
        sc_uint<8> private_data;
};

/* Generated SCV Extension */

SCV_EXTENSIONS(restricted_t) {
    public:
        scv_extensions< sc_uint<8> > public_data;
        scv_extensions< sc_uint<8> > private_data;

        SCV_EXTENSIONS_CTOR(restricted_t) {
            SCV_FIELD(public_data);
            SCV_FIELD(private_data);
        }
};
```

Enums

Note the following set of rules for generating a SCV extensions for enumerated types:

- Generated extensions start with macro `SCV_ENUM_EXTENSIONS()`, and typename is the name of the enumerated type.

- Generated extensions consists of only a constructor defined by the macro `SCV_ENUM_CTOR()`, and `typename` is the name of the user-specified type.
- A `SCV_ENUM` entry are added in constructor for each element of the enumerated type.

The following example demonstrates the generation process for an enumerated type.

Example 9-5. Generating SCV Extensions for an Enumerated Type

```
/* SystemC type */
enum instruction_t { ADD, SUB = 201 };

/* Generated SCV Extension */

SCV_ENUM_EXTENSIONS(instruction_t) {
public:

    SCV_ENUM_CTOR(instruction_t) {
        SCV_ENUM(ADD);
        SCV_ENUM(SUB);
    }
};
```

Mentor Dynamic Extensions

The OSCI-SCV library has been modified to support Mentor Dynamic Extensions, which allow the following:

- [Named Constraints](#)
- [Dynamic Enabling and Disabling of Named Constraints](#)
- [Constrained Randomization of the Data Type for Standard Vectors](#)
- [Randomly Sized Fixed-Max Arrays](#)

Named Constraints

The open SCV API supports the following macros for creating constraint data expression initializers data member fields of a user-defined constraint, based on a derivation of class `scv_constraint_base`:

```
#define SCV_CONSTRAINT(expr)
#define SCV_SOFT_CONSTRAINT(expr)
```

The first defines a hard constraint and the second defines a soft constraint.

The following example shows a user-defined constraint that uses these macros in the SCV constraint constructor macro, `SCV_CONSTRAINT_CTOR()`.

Example 9-6. User-Defined Constraint

```
class EtherFrameConstraintT : virtual public scv_constraint_base {
public:
    scv_smart_ptr<unsigned> Type;
    scv_smart_ptr<unsigned long long> DestAddr;
    scv_smart_ptr<unsigned long long> SrcAddr;
    scv_smart_ptr<unsigned> CRC;
    scv_smart_ptr<EtherFramePayloadT> Payload;

    SCV_CONSTRAINT_CTOR( EtherFrameConstraintT ){
        printf( "Start initializing EtherFrameConstraint ...\n" );
        SCV_CONSTRAINT( Type() == (SDF_BYTE << 8 | SDF_BYTE) );
        SCV_CONSTRAINT( DestAddr() != SrcAddr() );
        SCV_CONSTRAINT( DestAddr() < 0xffLL ); // Limit to 48 bits
        SCV_CONSTRAINT( SrcAddr() < 0xffLL ); // Limit to 48 bits
    }
};
```

To augment these macros, the following macro allows a constraint field to be named:

```
#define SCV_NAMED_CONSTRAINT(type, name, expr)
```

which uses the following arguments:

- **type** — the type of the constraint (HARD/SOFT), specified as `scv_constraint_expr::scv_constraint_type`.
- **name** — the actual name given to the constraint.
- **expr** — the expression argument that is passed exactly as in the existing macros `SCV_CONSTRAINT()` and `SCV_SOFT_CONSTRAINT()`.

To support the constraint type (hard or soft), the following class is defined with an enum type that you can use specify type:

```
class scv_constraint_expr {
public:
    typedef enum { HARD, SOFT } scv_constraint_type;
    scv_constraint_expr(
        const char* name, scv_constraint_type type, scv_expression e,
        const char* file = "unknown", int line = 0 );

    const char *name() const;
    const char *file() const;
    int line() const;

    void disable();
    void enable();
    bool is_disabled() const;
};
```

When the constraint is created, the file name and line # are captured in the class `scv_constraint_expr` object so that it can be provided later to parts of the internal SCV implementation for reporting purposes, such as error messages. You can do this by referencing the ANSI C `FILE` and `LINE` directives at the point where the name constraint is constructed. Accessors `::name()`, `::file()`, and `::line()` are provided to class `scv_constraint_expr` as shown above to provide this information for messaging, if needed.

Dynamic Enabling and Disabling of Named Constraints

You can enable and disable constraints that have been created with names in accordance with the naming macro described in [Named Constraints](#) (above). To support this feature, class `scv_constraint_base` contains the following methods:

```
class scv_constraint_base {
    ...
public:
    ...
    bool disable_constraint( const char* name );
    bool enable_constraint( const char* name );
    ...
};
```

You can use these methods to enable or disable any named constraint field in a user-defined constraint object derived from class `scv_constraint_base`. The implementation of class `scv_constraint_base` can use names as lookup keys to an internal table of class `scv_constraint_expr` objects. Once looked up, you can call the `::enable()` or `::disable()` method on those objects appropriately.

Constrained Randomization of the Data Type for Standard Vectors

The data type for standard vectors (`std::vector`) has a data introspection capability in the same way that fixed arrays and other primitive data types do. This means you can define SCV extensions for vectors by specifying `scv_extensions< vector >` in a manner similar to what is supported for arrays: `scv_extensions< T[N] >`.

In addition to being able to randomize all elements of a C++ STL `std::vector`, the SCV constraint solver can randomize the number of elements of a `std::vector` (its size) to an arbitrary value.

Further, you can constrain this randomization of vector size simply by calling `vector_size.keep_only()` on the `::vector_size` member of the `scv_extensions< vector >` class. The `::keep_only()` method can be given a range of values that size can assume.

Randomly Sized Fixed-Max Arrays

This randomly sized "fixed-max" feature for array types was originally provided to test preliminary implementations of randomization support for the `std::vector` data type. However, it is also useful feature for randomization of the number of elements in an array up to a fixed maximum size denoted by the template `N` parameter of `scv_extensions< T[N] >`.

It is fully backward compatible with existing support for array (`scv_extensions< T[N] >`) support in the current open SCV API; randomization of size is disabled by default for backward compatibility, but you can enable it as needed. Default operation is for all N elements to be randomized as is done in the open SCV API.

Further, you can constrain this randomization simply by calling `vector_size.keep_only()` on the `::vector_size` member of the `scv_extensions< T[N] >` class. The `::keep_only()` method can be given a range of values that size can assume.

Linking the Compiled Source

Once the design has been compiled, you must link it using the `sccom` command with the **-link** argument.

The **sccom -link** command collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library or the library specified by the **-work** option. If you have changed your SystemC source code and recompiled it using **sccom**, then you must re-link the design by running **sccom -link** before invoking **vsim**. Otherwise, your changes to the code are not recognized by the simulator. Remember that any dependent `.a` or `.o` files should be listed on the **sccom -link** command line before the `.a` or `.o` on which it depends. For more details on dependencies and other syntax issues, refer to the [sccom command in the Reference Manual](#).

Simulating SystemC Designs

After compiling the SystemC source code, you can simulate your design with **vsim**.

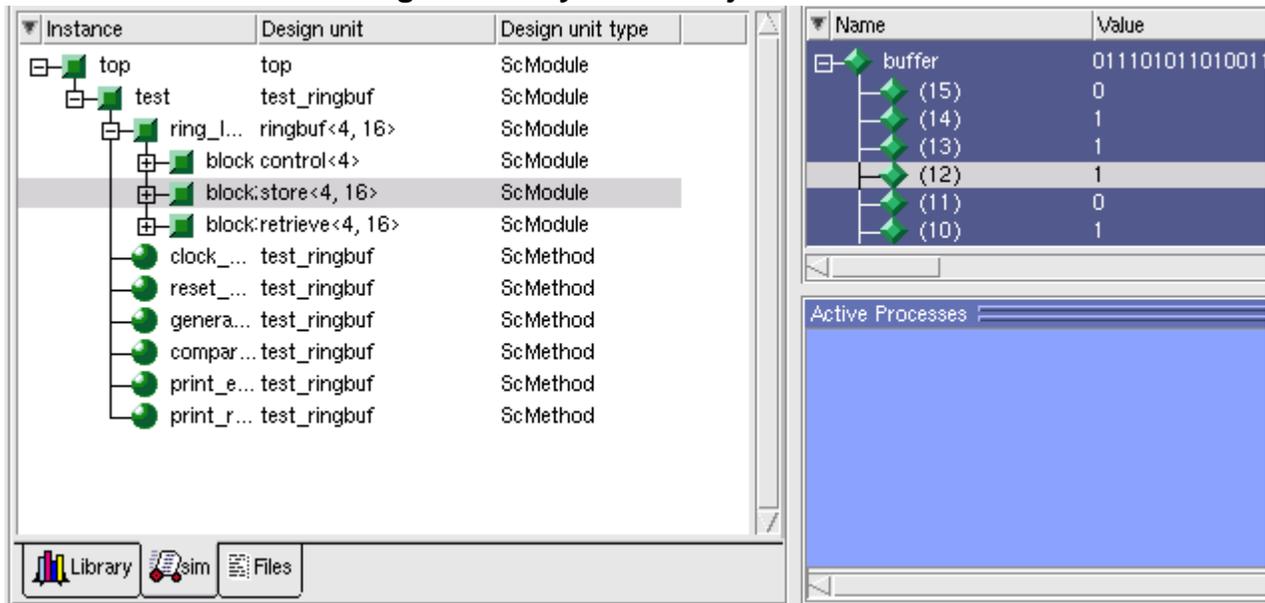
Loading the Design

For SystemC, invoke **vsim** with the top-level module(s) of the design. Multiple optimized top design modules can be specified. For more information about simulation with multiple optimized design modules refer to **vsim <library_name>.<design_unit>**. This example invokes **vsim** on a design named `top`:

```
vsim top
```

When the GUI comes up, you can expand the hierarchy of the design to view the SystemC modules. SystemC objects are denoted by green icons (see [Design Object Icons and Their Meaning](#) for more information).

Figure 9-1. SystemC Objects in GUI



To simulate from a command shell, without the GUI, invoke vsim with the -c option:

```
vsim -c <top_level_module>
```

i **Tip:** If you want to run a design with `sc_main()` as the top level, refer to [Recommendations for using `sc_main` at the Top Level](#).

Running Simulation

Run the simulation using the `run` command or select one of the **Simulate > Run** options from the menu bar.

SystemC Time Unit and Simulator Resolution

This section applies to SystemC only simulations. For simulations of mixed-language designs, the rules for how ModelSim interprets the resolution vary. See [Simulator Resolution Limit](#) for details on mixed-language simulations.

Two related yet distinct concepts are involved with determining the simulation resolution: the SystemC time unit and the simulator resolution. The following table describes the concepts, lists the default values, and defines the methods for setting/overriding the values.

Table 9-4. Time Unit and Simulator Resolution

	Description	Set by default as <i>.ini</i> file	Default value	Override default by
SystemC time unit	The unit of time used in your SystemC source code. You need to set this in cases where your SystemC default time unit is at odds with any other, non-SystemC segments of your design.	ScTimeUnit	1ns	<code>ScTimeUnit</code> <i>.ini</i> file variable or <code>sc_set_default_time_unit()</code> function before an <code>sc_clock</code> or <code>sc_time</code> statement.
Simulator resolution	The smallest unit of time measured by the simulator. If a warning is issued and your delays get truncated, set the resolution smaller; this value must be less than or equal to the UserTimeUnit	Resolution	1ns	<code>-t</code> argument to <code>vsim</code> (This overrides all other resolution settings.) or <code>sc_set_time_resolution()</code> function or GUI: Simulate > Start Simulation > Resolution

Available settings for both time unit and resolution are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

You can view the current simulator resolution by invoking the `report` command with the **simulator state** option.

Choosing Your Simulator Resolution

You should choose the coarsest simulator resolution limit possible that does not result in undesired rounding of your delays. For example, values smaller than the current Time Scale will be truncated to zero (0) and a warning issued. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

When deciding what to set the simulator's resolution to, you must keep in mind the relationship between the simulator's resolution and the SystemC time units specified in the source code. For example, with a time unit usage of:

```
sc_wait(10, SC_PS);
```

a simulator resolution of 10ps would be fine. No rounding off of the ones digits in the time units would occur. However, a specification of:

```
sc_wait(9, SC_PS);
```

would require you to set the resolution limit to 1ps in order to avoid inaccuracies caused by rounding.

Initialization and Cleanup of SystemC State-Based Code

State-based code should not be used in Constructors and Destructors. Constructors and Destructors should be reserved for creating and destroying SystemC design objects, such as `sc_modules` or `sc_signals`. State-based code should also not be used in the elaboration phase callbacks `before_end_of_elaboration()` and `end_of_elaboration()`.

The following virtual functions should be used to initialize and clean up state-based code, such as logfiles or the VCD trace functionality of SystemC. They are virtual methods of the following classes: `sc_port_base`, `sc_module`, `sc_channel`, and `sc_prim_channel`. You can think of them as phase callback routines in the SystemC language:

- `before_end_of_elaboration ()` — Called after all constructors are called, but before port binding.
- `end_of_elaboration ()` — Called at the end of elaboration after port binding.
- `start_of_simulation ()` — Called before simulation starts. Simulation-specific initialization code can be placed in this function.
- `end_of_simulation ()` — Called before ending the current simulation session.

The call sequence for these functions with respect to the SystemC object construction and destruction is as follows:

1. Constructors
2. `before_end_of_elaboration ()`
3. `end_of_elaboration ()`
4. `start_of_simulation ()`
5. `end_of_simulation ()`
6. Destructors

Usage of Callbacks

The **start_of_simulation()** callback is used to initialize any state-based code. The corresponding cleanup code should be placed in the **end_of_simulation()** callback. These callbacks are only called during simulation by **vsim** and thus, are safe.

If you have a design in which some state-based code must be placed in the constructor, destructor, or the elaboration callbacks, you can use the **mti_IsVoptMode()** function to determine if the elaboration is being run by **vopt**. You can use this function to prevent **vopt** from executing any state-based code.

Debugging the Design

You can debug SystemC designs using all the debugging features of ModelSim, with the exception of the Dataflow and Schematic windows. You must have compiled the design using the **sccom -g** argument in order to debug the SystemC objects in your design.

Viewable SystemC Types

Types (<type>) of the objects which may be viewed for debugging are the following:

Types

bool, sc_bit	short, unsigned short
sc_logic	long, unsigned long
sc_bv<width>	sc_bigint<width>
sc_lv<width>	sc_biguint<width>
sc_int<width>	sc_ufixed<W,I,Q,O,N>
sc_uint<width>	short, unsigned short
sc_fix	long long, unsigned long long
sc_fix_fast	float
sc_fixed<W,I,Q,O,N>	double
sc_fixed_fast<W,I,Q,O,N>	enum
sc_ufix	pointer
sc_ufix_fast	array
sc_ufixed	class
sc_ufixed_fast	struct
sc_signed	union
sc_unsigned	ac_int
char, unsigned char	ac_fixed
int, unsigned int	

Viewable SystemC Objects

Objects which may be viewed in SystemC for debugging purposes are as shown in the following table.

Table 9-5. Viewable SystemC Objects

Channels	Ports	Variables	Aggregates
sc_clock (a hierarchical channel) sc_event sc_export sc_mutex sc_fifo<type> sc_signal<type> sc_signal_rv<width> sc_signal_resolved tlm_fifo<type> User defined channels derived from sc_prim_channel	sc_in<type> sc_out<type> sc_inout<type> sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width> sc_in_resolved sc_out_resolved sc_inout_resolved sc_in_clk sc_out_clk sc_inout_clk sc_fifo_in sc_fifo_out User defined ports derived from sc_port<> which is : <ul style="list-style-type: none"> • connected to a built-in channel • connected to a user-defined channel derived from an sc_prim_channel¹ 	Module member variables of all C++ and SystemC built-in types (listed in the Types list below) are supported.	Aggregates of SystemC signals or ports. Only three types of aggregates are supported for debug: <ul style="list-style-type: none"> struct class array

1. You must use a special macro to make these ports viewable for debugging. For details See [MTI_SC_PORT_ENABLE_DEBUG](#).

MTI_SC_PORT_ENABLE_DEBUG

A user-defined port which is not connected to a built-in primitive channel is not viewable for debugging by default. You can make the port viewable if the actual channel connected to the port is a channel derived from an sc_prim_channel. If it is, you can add the macro MTI_SC_PORT_ENABLE_DEBUG to the channel class' public declaration area, as shown in this example:

```
class my_channel: public sc_prim_channel
{
...
public:
```

```
MTI_SC_PORT_ENABLE_DEBUG
```

```
};
```

Waveform Compare with SystemC

Waveform compare supports the viewing of SystemC signals and variables. You can compare SystemC objects to SystemC, Verilog or VHDL objects.

For pure SystemC compares, you can compare any two signals that match type and size exactly; for C/C++ types and some SystemC types, sign is ignored for compares. Thus, you can compare char to unsigned char or sc_signed to sc_unsigned. All SystemC fixed-point types may be mixed as long as the total number of bits and the number of integer bits match.

Mixed-language compares are supported as listed in the following table:

Table 9-6. Mixed-language Compares

C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg
VHDL types	bit, bit_vector, boolean, std_logic, std_logic_vector

The number of elements must match for vectors; specific indexes are ignored.

Debugging Source-Level Code

In order to debug your SystemC source code, you must compile the design for debug using the **-g** C++ compiler option. You can add this option directly to the `sccom` command line on a per run basis, with a command such as:

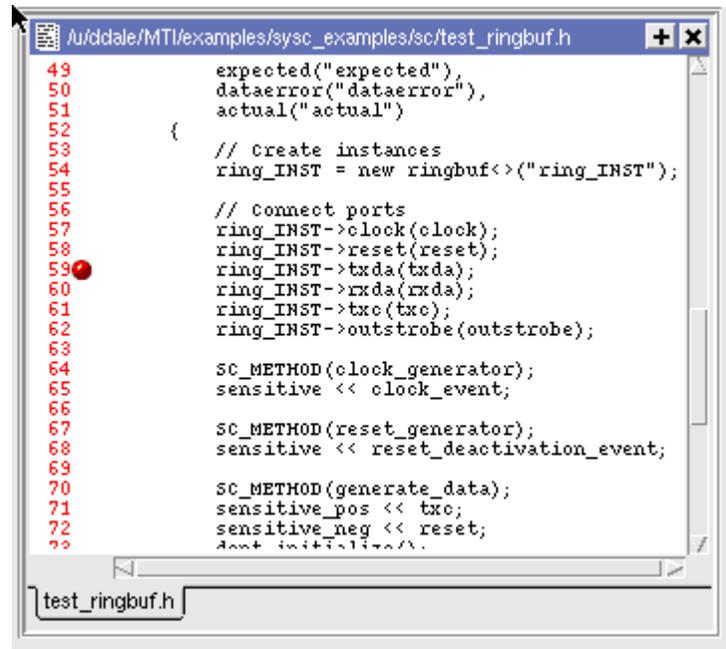
```
sccom mytop -g
```

Or, if you plan to use it every time you run the compiler, you can specify it in the `modelsim.ini` file with the `CppOptions` variable. See [modelsim.ini Variables](#) for more information.

The source code debugger, [C Debug](#), is automatically invoked when the design is compiled for debug in this way.

Figure 9-2 shows an example of how to set breakpoints in a Source window (Line 59) and single-step through your SystemC/C++ source code.

Figure 9-2. Breakpoint in SystemC Source



Note



To disallow source annotation, use the `-nodbgSYM` argument for the `sccom` command:

```
sccom -nodbgSYM
```

This disables the generation of symbols for the debugging database in the library.

Stepping Out From OSCI Library Functions

When you are using C Debug to single-step through the SystemC code, you may find that stepping through the code often ends up going inside SystemC library routines. This can be a distraction from debugging your actual code.

By default, auto-stepping out of the library for debugging is enabled, which means stepping into the library is not allowed (`cdbg allow_lib_step off`). So, if you step into a library function, execution will automatically return to your code.

You can use the `cdbg` command to disable this behavior:

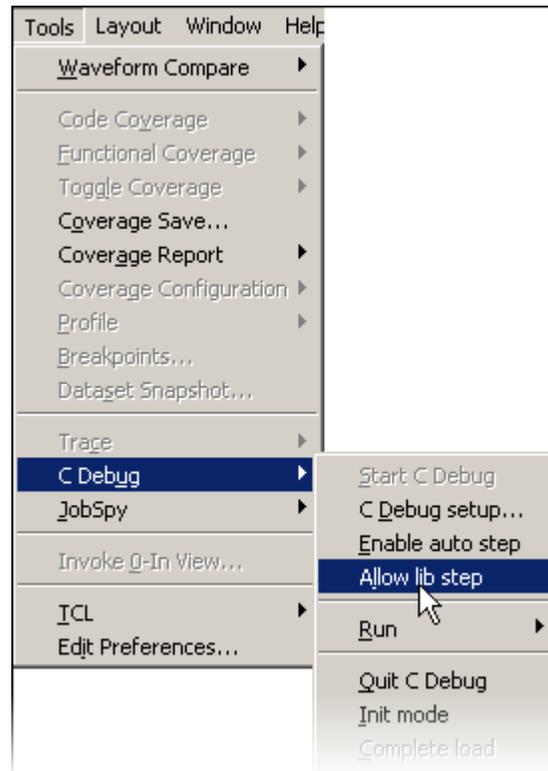
```
cdbg allow_lib_step on
```

Now, execution will not automatically step out from library functions, but it will step into the library code.

The `allow_lib_step` argument to the `cdbg` command takes a value of "on" or "off."

You can also perform this action in the GUI by selecting **Tools > CDebug > Allow lib step** from the menus (Figure 9-3).

Figure 9-3. Setting the Allow lib step Function



For example, assume that the debugger has stepped to a library function call. If this were the only library function call in the current line, execution would go the next line in your code (there would be no need for the “step out” action). However, if there are more function calls in the current line, execution comes back to the same line, and the next 'step -over' operation goes to the next line in your code. So the debugging operation always stays in your code, regardless of where it steps.

Setting Constructor/Destructor Breakpoints

You can set breakpoints in constructors and destructors of SystemC objects. Constructor breakpoints need to be set before SystemC shared library is loaded. You can set breakpoints using either the Cdebug Init mode or Automated Constructor breakpoint flow.

Cdebug Init mode

1. Start Cdebug before loading the design.

- a. Select **Tools > CDebug > Start CDebug** from the menus or use the following command:
cdbg debug_on
2. Turn on the Cdebug Init mode.
 - a. Select **Tools > CDebug > Init mode** from the menus or use the following command:
cdbg init_mode_setup
3. Load the design.

ModelSim will stop after loading the shared library.
4. Set breakpoints on constructors.

Automated Constructor breakpoint flow

1. Start ModelSim in the GUI or batch mode.
 - a. Type **vsim** at a UNIX shell prompt (**vsim -c** for batch mode) or double-click the ModelSim icon in Windows.

If the Welcome to ModelSim dialog appears, click **Close**.
2. Set the breakpoints using the following command.
bp -c [<filename>:<line> | <function_name>]

NOTE: You can also set breakpoints by opening a file in source window and clicking on a line number.
3. Load the design by entering the **vsim** command. ModelSim automatically stops after loading the shared library and sets all the constructor breakpoints. You can set additional constructor breakpoints here.
4. The **run -continue** command elaborates the design and stops the simulation at the constructor breakpoint.

You can also set destructor breakpoints using these same steps in either the Cdebug Init mode or the Automated Constructor breakpoint flow; or, after the design is loaded. If you set destructor breakpoints before loading the design, then ModelSim keeps all the breakpoints enabled even after design is loaded.

When you set a destructor breakpoint, ModelSim automatically sets up in **Stop on quit** mode (see [Debugging Functions when Quitting Simulation](#)). The debugger will stop at the breakpoint after you issue the **quit -f** command in ModelSim. This allows you to step through and examine the code. Run the **run -continue** command when you have finished examining the C code.

Because the **Stop on quit** mode is set up, when simulation completes, ModelSim automatically quits C-debugger and the GUI (whether or not a C breakpoint was hit and you return to the VSIM> prompt).

Instance Based Breakpointing

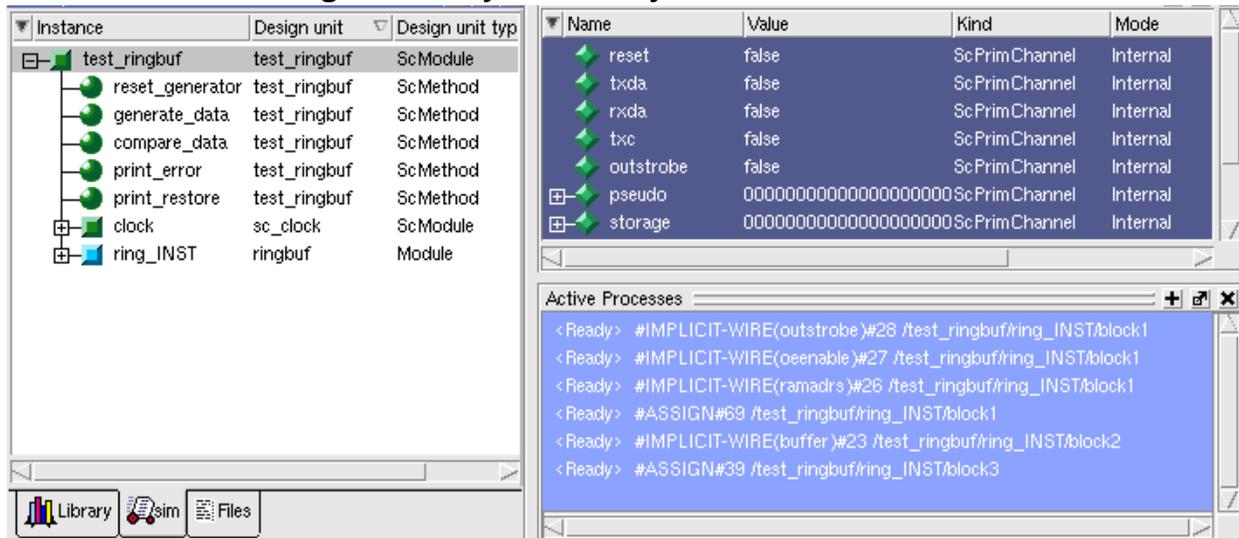
To set a SystemC breakpoints so it applies only to a specified instance, use the `-inst` argument to the `bp` command:

```
bp <filename>:<line#> -inst <instance>
```

Viewing SystemC Objects in GUI

You can view and expand SystemC objects in the Objects window and processes in the Processes pane, as shown in [Figure 9-4](#).

Figure 9-4. SystemC Objects and Processes



SystemC Object and Type Display

This section contains information on how ModelSim displays certain objects and types, as they may differ from other simulators.

Support for Globals and Statics

Globals and statics are supported for ModelSim debugging purposes, however some additional naming conventions must be followed to make them viewable.

Naming Requirement

In order to make a global viewable for debugging purposes, the name given must match the declared signal name. An example:

```
sc_signal<bool> clock("clock");
```

For statics to be viewable, the name given must be fully qualified, with the module name and declared name, as follows:

```
<module_name>::<declared_name>
```

For example, the static data member "count" is viewable in the following code excerpt:

```
SC_MODULE(top)
{
    static sc_signal<float> count; //static data member
    ....
}
sc_signal<float> top::count("top::count"); //static named in quotes
```

Viewing Global and Static Signals

ModelSim translates C++ scopes into a hierarchical arrangement. Because globals and statics exist at a level above its scope, ModelSim must add a top level, **sc_root**, to all global and static signals. Thus, to view these static or global signals in ModelSim, you need to add **sc_root** to the hierarchical name for the signal. In the case of the above examples, the debugging statements for examining "top/count" (a static) and "clock" (a global) would be:

```
VSIM> examine /sc_root/top/count  
VSIM> examine /sc_root/clock
```

Support for Aggregates

ModelSim supports aggregates of SystemC signals or ports. Three types of aggregates are supported: structures, classes, and arrays. Unions are not supported for debug. An aggregate of signals or ports will be shown as a signal of aggregate type. For example, an aggregate such as:

```
sc_signal <sc_logic> a[3];
```

is equivalent to:

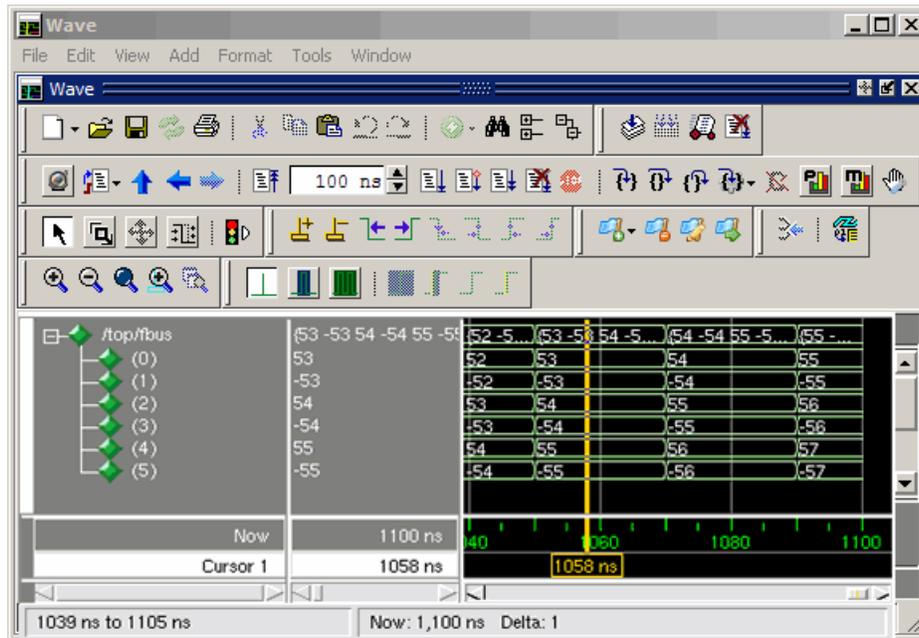
```
sc_signal <sc_lv<3>> a;
```

for debug purposes. ModelSim shows one signal - object "a" - in both cases.

The following aggregate would appear in the Wave window as shown in [Figure 9-5](#):

```
sc_signal <float> fbus [6];
```

Figure 9-5. Aggregate Data Displayed in Wave Window



SystemC Dynamic Module Array

ModelSim supports SystemC dynamic module arrays. An example of using a dynamic module array:

```

module **mod_inst;
mod_inst = new module*[2];
mod_inst[0] = new module("mod_inst[0]");
mod_inst[1] = new module("mod_inst[1]");

```

Limitations

- The instance names of modules containing dynamic arrays must match the corresponding C++ variables, such as “mod_inst[0]” and “mod_inst[1]” in the example above. If not named correctly, the module instances simulate correctly, but are not debuggable.

Viewing FIFOs

In ModelSim, the values contained in an sc_fifo appear in a definite order. The top-most or left-most value is always the next to be read from the FIFO. Elements of the FIFO that are not in use are not displayed.

Example of a signal where the FIFO has five elements:

```
# examine f_char
# {}
VSIM 4> # run 10
VSIM 6> # examine f_char
# A
VSIM 8> # run 10
VSIM 10> # examine f_char
# {A B}
VSIM 12> # run 10
VSIM 14> # examine f_char
# {A B C}
VSIM 16> # run 10
VSIM 18> # examine f_char
# {A B C D}
VSIM 20> # run 10
VSIM 22> # examine f_char
# {A B C D E}
VSIM 24> # run 10
VSIM 26> # examine f_char
# {B C D E}
VSIM 28> # run 10
VSIM 30> # examine f_char
# {C D E}
VSIM 32> # run 10
VSIM 34> # examine f_char
# {D E}
```

Viewing SystemC Memories

The ModelSim tool detects and displays SystemC memories. A memory is defined as any member variable of a SystemC module which is defined as an array of the following type:

unsigned char	sc_bit (of 2-D or more arrays only)
unsigned short	sc_logic (of 2-D or more arrays only)
unsigned int	sc_lv<N>
unsigned long	sc_bv<N>
unsigned long long	sc_int<N>
char	sc_uint<N>
short	sc_bigint<N>
int	sc_biguint<N>
float	sc_signed
double	sc_unsigned
enum	

Properly Recognizing Derived Module Class Pointers

If you declare a pointer as a base class pointer, but actually assign a derived class object to it, ModelSim still treats it as a base class pointer instead of a derived class pointer, as you intended. As such, it would be unavailable for debug. To make it available for debug, you must use the **mti_set_typename** member function to instruct that it should be treated as a derived class pointer.

To correctly associate the derived class type with an instance:

1. Use the member function **mti_set_typename** and apply it to the modules. Pass the actual derived class name to the function when an instance is constructed, as shown in [Example 9-7](#).

Example 9-7. Use of `mti_set_typename`

```
SC_MODULE(top) {  
    base_mod* inst;  
    SC_CTOR(top) {  
        if (some_condition) {  
            inst = new dl_mod("dl_inst");  
            inst->mti_set_typename("dl_mod");  
        } else {  
            inst = new d2_mod("d2_inst");  
            inst->mti_set_typename("d2_mod");  
        }  
    }  
};
```

i **Tip:** In this example, the class names are simple names, which may not be the case if the type is a class template with lots of template parameters. Look up the name in `<work>/moduleinfo.sc` file, if you are unsure of the exact names.

Here is the code for which the above `SC_MODULE` was modified:

```
class base_mod : public sc_module {  
    sc_signal<int> base_sig;  
    int          base_var;  
    ...  
};  
  
class dl_mod : public base_mod {  
    sc_signal<int> dl_sig;  
    int          dl_var;
```

```
    ...  
};  
  
class d2_mod : public base_mod {  
    sc_signal<int> d2_sig;  
    int           d2_var;  
  
    ...  
};  
  
SC_MODULE(top) {  
    base_mod* inst;  
  
    SC_CTOR(top) {  
        if (some_condition)  
            inst = new d1_mod("d1_inst");  
        else  
            inst = new d2_mod("d2_inst");  
    }  
};
```

In this unmodified code, the **sccom** compiler could only see the declarative region of a module, so it thinks "inst" is a pointer to the "base_mod" module. After elaboration, the GUI would only show "base_sig" and "base_var" in the Objects window for the instance "inst."

You really wanted to see all the variables and signals of that derived class. However, since you didn't associate the proper derived class type with the instance "inst", the signals and variables of the derived class are not debuggable, even though they exist in the kernel.

The solution is to associate the derived class type with the instance, as shown in the modified SC_MODULE above.

Custom Debugging of SystemC Channels and Variables

ModelSim offers a string-based debug solution for various simulation objects which are considered undebuggable by the SystemC compiler **sccom**. Through it, you can gain easy access for debugging to the following:

- SystemC variables of a user-defined type
- Built-in channels of a user defined type
- Built-in ports of a user defined type
- User defined channels and ports

This custom interface can be also used to debug objects that may be supported for debug natively by the simulator, but whose native debug view is too cumbersome.

Supported SystemC Objects

The custom debug interface provides debug support for the following SystemC objects (T is a user defined type, or a user-defined channel or port):

```
T
sc_signal<T>
sc_fifo<T>
tlm_fifo<T>
sc_in<T>
sc_out<T>
sc_inout<T>
```

Usage

To provide custom debug for any object:

1. Register a callback function — one for each instance of that object — with the simulator. Specify the maximum length of the string buffer to be reserved for an object instance. See [Registration and Callback Function Syntax](#).
2. The simulator calls the callback function, with the appropriate arguments, when it needs the latest value of the object.

The registration function can be called from the phase callback function **before_end_of_elaboration()**, or anytime before this function during the elaboration phase of the simulator.

3. The ModelSim simulator passes the callback function a pre-allocated string of a length specified during registration. The callback function must write the value of the object in that string, and it must be null terminated (`\0`).
4. The ModelSim simulator takes the string returned by the callback function as-is and displays it in the Objects window, Wave window, and CLI commands (such as `examine`). The describe command on custom debug objects simply reports that the object is a custom debug object of the specified length.

The macro used to register an object for debugging is `SC_MTI_REGISTER_CUSTOM_DEBUG`. Occasionally, ModelSim fails to register an object because it determines that the object cannot be debugged. In such cases, an error message is issued to that effect. If this occurs, use the `SC_MTI_REGISTER_NAMED_CUSTOM_DEBUG` to both name and register the object for debugging.

Registration and Callback Function Syntax

Registration:

```
void SC_MTI_REGISTER_CUSTOM_DEBUG
```

```
(void* obj, size_t value_len,  
mtiCustomDebugCB cb_func);  
  
void SC_MTI_REGISTER_NAMED_CUSTOM_DEBUG  
(void* obj, size_t value_len,  
mtiCustomDebugCB cb_func, const char* name);
```

Callback:

```
typedef void (*mtiCustomDebugCB)(void* obj, char* value, char  
format_char);
```

- **obj** — the handle to the object being debugged
- **value_len** — the maximum length of the debug string to be reserved for this object
- **cb_func** — the callback function to be called by the simulator for the latest value of the object being debugged
- **name** — the name of the object being debugged
- **value** — A pointer to the string value buffer in which the callback must write the string value of the object being debugged
- **format_char** — the expected format of the value: ascii ('a'), binary ('b'), decimal ('d'), hex ('h'), or octal ('o')

The callback function does not return anything.

Example 9-8. Using the Custom Interface on Different Objects

Consider an arbitrary user-defined type T as follows:

```
class myclass {  
  
private:  
  
    int x;  
    int y;  
  
public:  
    void get_string_value(char format_str, char* mti_value);  
    size_t get_value_length();  
  
    ...  
};
```

Variable of type T would be:

```
void mti_myclass_debug_cb(void* var, char* mti_value, char format_str)  
{  
    myclass* real_var = reinterpret_cast<myclass*>(var);  
    real_var->get_string_value(format_str, mti_value);  
}
```

```

}

SC_MODULE(test) {

    myclass var1;
    myclass* var2;

    SC_CTOR(test) {
        SC_MTI_REGISTER_CUSTOM_DEBUG(
            &var1,
            var1.get_value_length(),
            mti_myclass_debug_cb);

        SC_MTI_REGISTER_CUSTOM_DEBUG(
            var2,
            var2->get_value_length(),
            mti_myclass_debug_cb);
    }
};

```

`sc_signal`, `sc_fifo` and `tlm_fifo` of type `T` and Associated Ports would be:

```

void mti_myclass_debug_cb(void* var, char* mti_value, char format_str)
{
    myclass* real_var = reinterpret_cast<myclass*>(var);
    real_var->get_string_value(format_str, mti_value);
}

SC_MODULE(test) {

    sc_signal<myclass> sig1;
    sc_signal<myclass> *sig2;
    sc_fifo<myclass> fifo;

    SC_CTOR(test) {
        myclass temp;

        SC_MTI_REGISTER_CUSTOM_DEBUG(
            &sig1,
            temp.get_value_length(),
            mti_myclass_debug_cb);

        SC_MTI_REGISTER_CUSTOM_DEBUG(
            sig2,
            temp.get_value_length(),
            mti_myclass_debug_cb);

        SC_MTI_REGISTER_CUSTOM_DEBUG(
            &fifo,
            temp.get_value_length(),
            mti_myclass_debug_cb);
    }
};

```

As shown in [Example 9-8](#), although the callback function is registered on a `sc_signal<T>` or a `sc_fifo<T>` object, the callback is called on the `T` object, instead of the channel itself. The

reason for the callback on T is because `sc_signal<T>` has two sets of values, current and new value and `sc_fifo` can have more than one element in the fifo. The callback is called on each element of the fifo that is valid at any given time. For an `sc_signal<T>` the callback is called only on the current value, not the new value.

By registering the primitive channel `sc_signal<T>` for custom debug, any standard port connected to it (`sc_in<T>`, `sc_out<T>`, `sc_inout<T>`, `sc_fifo_in<T>`, and so forth) automatically is available for custom debug. It is illegal to register any built-in ports for custom debug separately.

User Defined Primitive Channels and Ports

The callback and registration mechanism for a user-defined channel derived from `sc_prim_channel` are no different than a variable of an user-defined type. Please see the section on variables of type T in [Example 9-8](#) for more details on the registration and callback mechanism for such objects.

You have two choices available to you for making user defined ports debuggable:

- Automatic debug of any port connected to a primitive channel

Any port that is connected to a channel derived from `sc_prim_channel` is automatically debuggable only if the connected channel is debuggable either natively or using custom debug. To enable this automatic debugging capability, use the following macro in the channel class:

```
MTI_SC_PORT_ENABLE_DEBUG
```

In this case, you may not separately register the port for custom debug.

- Specific port registration

Register the port separately for custom debug. To do this, simply register the specific port, without using the macro. The callback and registration mechanism is the same as a variable of type T.

Hierarchical Channels/Ports Connected to Hierarchical Channels

Hierarchical channels are basically modules, and appear in the structure pane in ModelSim. Since they are part of the design hierarchy, custom debug cannot be supported for hierarchical channels. Ports connected to hierarchical channels, however, though not supported for debug natively in ModelSim, are supported for debug with the custom interface.

Any port object registered for custom debug is treated as a variable of a user defined type. Please see [Example 9-8](#), variables of type T, for more details on the registration and callback mechanism for such objects.

Any Other Channels and Ports Connected to Such Channels

It is legal in SystemC to create a channel that implements an interface and is not derived either from `sc_channel` or `sc_prim_channel`. Take the following, for example:

```
class mychannel : public myinterface {}  
  
class myport : public sc_port<myinterface> {}
```

Channels and ports of this category are supported for debug natively in ModelSim. ModelSim treats them as variables of type T. These channels and ports can be registered for custom debug. The registration and callback mechanism is the same as for a variable of type T, as shown in [Example 9-8](#) above.

Modifying SystemC Source Code

If your design does not have `sc_main()` at the top level, you must apply several modifications to your original SystemC source code. For more information on how to make these modifications, refer to “[Code Modification Examples](#),” which contains the following examples:

- [Example 9-9: Converting sc_main to a Module](#)
- [Example 9-10: Using sc_main and Signal Assignments](#)
- [Example 9-11: Using an SCV Transaction Database](#)

Converting sc_main() to a Module

If your design does not have `sc_main()` at the top level, in order for ModelSim to run the SystemC/C++ source code, you must replace the control function of `sc_main()` with a constructor, `SC_CTOR()`, placed within a module at the top level of the design (see **mytop** in [Example 9-9](#)).

In addition, the following requirements also apply:

- Any test bench code inside `sc_main()` should be moved to a process, normally an `SC_THREAD` process.
- All C++ variables in `sc_main()`, including SystemC primitive channels, ports, and modules, must be defined as members of `sc_module`. Therefore, initialization must take place in the `SC_CTOR`. For example, all `sc_clock()` and `sc_signal()` initializations must be moved into the constructor.

Replacing sc_start() Function with Run Command and Options

ModelSim uses the **run** command and its options in place of the `sc_start()` function. If `sc_main()` has multiple `sc_start()` calls mixed in with the test bench code, then use an

`SC_THREAD()` with wait statements to emulate the same behavior. An example of this is shown in “[Code Modification Examples](#)” on page 532.

Removing Calls to `sc_initialize()`

`vsim` calls `sc_initialize()` by default at the end of elaboration, so calls to `sc_initialize()` are unnecessary.

Code Modification Examples

Example 9-9. Converting `sc_main` to a Module

[Table 9-7](#) shows a simple example of how to convert `sc_main` to a module that you can elaborate with the `vsim` command.

Table 9-7. Simple Conversion: `sc_main` to Module

Original OSCI code #1 (partial)	Modified code #1 (partial)
<pre>int sc_main(int argc, char* argv[]) { sc_signal<bool> mysig; mymod mod("mod"); mod.outp(mysig); sc_start(100, SC_NS); }</pre>	<pre>SC_MODULE(mytop) { sc_signal<bool> mysig; mymod mod; SC_CTOR(mytop) : mysig("mysig"), mod("mod") { mod.outp(mysig); } }; SC_MODULE_EXPORT(mytop);</pre>

Here, you would use the following run command for the modified code as the equivalent to the `sc_start(100, SC_NS)` statement in the original OSCI code:

run 100 ns

Example 9-10. Using `sc_main` and Signal Assignments

Table 9-8 shows a slightly more complex conversion that illustrates the use of `sc_main()` and signal assignments, and how you would get the same behavior using ModelSim.

Table 9-8. Using `sc_main` and Signal Assignments

OSCI code #2 (partial)	Modified code #2 (partial)
<pre>int sc_main(int, char**) { sc_signal<bool> reset; counter_top top("top"); sc_clock CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false); top.reset(reset); reset.write(1); sc_start(5, SC_NS); reset.write(0); sc_start(100, SC_NS); reset.write(1); sc_start(5, SC_NS); reset.write(0); sc_start(100, SC_NS); }</pre>	<pre>SC_MODULE(new_top) { sc_signal<bool> reset; counter_top top; sc_clock CLK; void sc_main_body(); SC_CTOR(new_top) : reset("reset"), top("top") CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false) { top.reset(reset); SC_THREAD(sc_main_body); } }; void new_top::sc_main_body() { reset.write(1); wait(5, SC_NS); reset.write(0); wait(100, SC_NS); reset.write(1); wait(5, SC_NS); reset.write(0); wait(100, SC_NS); sc_stop(); } SC_MODULE_EXPORT(new_top);</pre>

Example 9-11. Using an SCV Transaction Database

Table 9-9 shows a conversion that modifies a design using an SCV transaction database. ModelSim requires that you create the transaction database before calling the constructors on the design subelements.

Table 9-9. Modifications Using SCV Transaction Database

Original OSCI code # 3 (partial)	Modified ModelSim code #3 (partial)
<pre>int sc_main(int argc, char* argv[]) { scv_startup(); scv_tr_text_init(); scv_tr_db db("my_db"); scv_tr_db db::set_default_db(&db); sc_clock clk ("clk",20,0.5,0,true); sc_signal<bool> rw; test t("t"); t.clk(clk);; t.rw(rw); sc_start(100); }</pre>	<pre>SC_MODULE(top) { sc_signal<bool>* rw; test* t; SC_CTOR(top) { scv_startup(); scv_tr_text_init() scv_tr_db* db = new scv_tr_db("my_db"); scv_tr_db::set_default_db(db);; clk = new sc_clock("clk",20,0.5,0,true); rw = new sc_signal<bool> ("rw"); t = new test("t"); } }; SC_MODULE_EXPORT(new_top);</pre>

Take care to preserve the order of functions called in **sc_main()** of the original code.

You cannot place subelements in the initializer list, since the constructor body must be executed prior to their construction. Therefore, you must make the subelements as pointer types by creating them with "new" in the SC_CTOR() module.

Differences Between the Simulator and OSCI

ModelSim is based upon the OSCI proof-of-concept SystemC simulator. However, there are some minor but key differences to be aware of:

- The default time resolution of the simulator is 1ps. For **vsim** it is 1ns. You can change the value for time resolution by using the **vsim** command with the **-t** option or by modifying the value of the **Resolution** variable in the *modelsim.ini* file.
- The **run** command in ModelSim is equivalent to **sc_start()**. In the SystemC simulator, **sc_start()** runs the simulation for the duration of time specified by its argument. In ModelSim the **run** command runs the simulation for the amount of time specified by its argument.
- The **sc_cycle()**, and **sc_start()** functions are not supported in ModelSim.

- The default name for `sc_object()` is bound to the actual C object name. However, this name binding only occurs after all `sc_object` constructors are executed. As a result, any `name()` function call placed inside a constructor will not pick up the actual C object name.
- The value returned by the `name()` method prefixes OSCI-compliant hierarchical paths with "sc_main", which is ModelSim's implicit SystemC root object. For example, for the following example code:

```
#include "systemc.h"

SC_MODULE(bloc)
{
    SC_CTOR(bloc) {}
};

SC_MODULE(top)
{
    bloc b1 ;
    SC_CTOR(top) : b1("b1") { cout << b1.name() << endl ; }
};

int sc_main(int argc, char* argv[])
{
    top top_i("top_i");
    sc_start(0, SC_NS);
    return 0;
}
```

the OSCI returns:

```
top_i.b1
```

and ModelSim returns:

```
sc_main.top_i.b1
```

Fixed-Point Types

Contrary to OSCI, ModelSim compiles the SystemC kernel with support for fixed-point types. If you want to compile your own SystemC code to enable that support, you must first define the compile time macro `SC_INCLUDE_FX`. You can do this in one of two ways:

- Enter the g++/aCC argument `-DSC_INCLUDE_FX` on the `sccom` command line, such as:

```
sccom -DSC_INCLUDE_FX top.cpp
```

- Add a define statement to the C++ source code before the inclusion of the `systemc.h`, as shown below:

```
#define SC_INCLUDE_FX
#include "systemc.h"
```

Algorithmic C Datatype Support

ModelSim supports native debug for the Algorithmic-C data types **ac_int** and **ac_fixed**. The Algorithmic C data types are used in Catapult C Synthesis, a tool that generates optimized RTL from algorithms written as sequential ANSI-standard C/C++ specifications. These data types are synthesizable and run faster than their SystemC counterparts `sc_bigint`, `sc_biguint`, `sc_fixed` and `sc_ufixed`.

To use these data types in the simulator, you must obtain the datatype package and specify the path containing the Algorithmic C header files with the `-I` argument on the `sccom` command line:

```
sccom -I <path_to_AC_headers> top.cpp
```

To enable native debug support for these datatypes, you must also specify the `-DSC_INCLUDE_MTI_AC` argument on the `sccom` command line.

```
sccom -DSC_INCLUDE_MTI_AC -I <path_to_AC_headers> top.cpp
```

Native debug is only supported for Version 1.2 and above. If you do not specify `-DSC_INCLUDE_MTI_AC`, the GUI displays the C++ layout of the datatype classes.

Support for cin

The ModelSim simulator has a limited support for the C++ standard input `cin`. To enable support for `cin`, the design source files must be compiled with `-DUSE_MTI_CIN` `sccom` option. For example:

```
sccom -DUSE_MTI_CIN top.cpp
```

Limitations

ModelSim does not support `cin` when it is passed as a function parameter of type `istream`. This is true for both C++ functions and member functions of a user-defined class/struct.

For example, the following `cin` usage is not supported:

```
void getInput(istream& is)
{
    int input_data;
    ...
    is >> input_data;
    ....
}
getinput(cin);
```

A workaround for this case, the source code needs to be modified as shown below:

```
void getinput()
{
    int input_data;
```

```
    ...  
    cin >> input_data;  
    ....  
}  
getinput();
```

OSCI 2.2 Feature Implementation Details

Support for OSCI TLM Library

ModelSim includes the header files and examples from the **OSCI SystemC TLM** (Transaction Level Modeling) Library Standard version 2.0. The TLM library can be used with simulation, and requires no extra switches or files. TLM objects are not debuggable, with the exception of `tlm_fifo`.

Examples and documentation are located in *install_dir/examples/systemc/tlm*. The TLM header files (*tlm_*.h*) are located in *include/systemc*.

Phase Callback

The following functions are supported for phase callbacks:

- `before_end_of_elaboration()`
- `start_of_simulation()`
- `end_of_simulation()`

For more information regarding the use of these functions, see [Initialization and Cleanup of SystemC State-Based Code](#).

Accessing Command-Line Arguments

The following global functions allow you to gain access to command-line arguments:

- `sc_argc()` — Returns the number of arguments specified on the `vsim` command line with the `-sc_arg` argument. This function can be invoked from anywhere within SystemC code.
- `sc_argv()` — Returns the arguments specified on the `vsim` command line with the `-sc_arg` argument. This function can be invoked from anywhere within SystemC code.

Example:

When `vsim` is invoked with the following command line:

```
vsim -sc_arg "-a" -c -sc_arg "-b -c" -t ns -sc_arg -d
```

`sc_argc()` and `sc_argv()` will behave as follows:

```
int argc;
const char * const * argv;

argc = sc_argc();
argv = sc_argv();
```

The number of arguments (`argc`) is now 4.

```
argv[0] is "vopt" // if running vopt explicitly
argv[0] is "vsim" // if not
argv[1] is "-a"
argv[2] is "-b -c"
argv[3] is "-d"
```

sc_stop Behavior

When encountered during the simulation run in batch mode, the `sc_stop()` function stops the current simulation and causes ModelSim to exit. In GUI mode, a dialog box appears asking you to confirm the exit. This is the default operation of `sc_stop()`. If you want to change the default behavior of `sc_stop`, you can change the setting of the [OnFinish](#) variable in the `modelsim.ini` file. To change the behavior interactively, use the `-onfinish` argument to the `vsim` command.

Construction Parameters for SystemC Types

The information in this section applies only to SystemC signals, ports, variables, or fifos that use one of the following fixed-point types:

```
sc_signed
sc_unsigned
sc_fix
sc_fix_fast
sc_ufix
sc_ufix_fast
```

These are the only SystemC types that have construction time parameters. The default size for these types is 32. If you require values other than the default parameters, you need to read this section.

If you are using one of these types in a SystemC signal, port, fifo, or an aggregate of one of these (such as an array of `sc_signal`), you cannot pass the size parameters to the type. This is a limitation imposed by the C++ language. Instead, SystemC provides a global default size (32) that you can control.

For `sc_signed` and `sc_unsigned`, you need to use the two objects, `sc_length_param` and `sc_length_context`, and you need to use them in an unusual way. If you just want the default vector length, simply do this:

```
SC_MODULE(dut) {
    sc_signal<sc_signed> s1;
    sc_signal<sc_signed> s2;
    SC_CTOR(dut)
        : s1("s1"), s2("s2")
    {
    }
}
```

For a single setting, such as using five-bit vectors, your module and its constructor would look like the following:

```
SC_MODULE(dut) {
    sc_length_param l;
    sc_length_context c;
    sc_signal<sc_signed> s1;
    sc_signal<sc_signed> s2;
    SC_CTOR(dut)
        : l(5), c(l), s1("s1"), s2("s2")
    {
    }
}
```

Notice that the constructor initialization list sets up the length parameter first, assigns the length parameter to the context object, and then constructs the two signals. You **DO** pass the name to the signal constructor, but the name is passed to the signal object, not to the underlying type. There is no way to reach the underlying type directly. Instead, the default constructors for `sc_signed` and `sc_unsigned` reach out to the global area and get the currently defined length parameter—the one you just set.

If you need to have signals or ports with different vector sizes, you need to include a pair of parameter and context objects for each different size. For example, the following uses a five-bit vector and an eight-bit vector:

```
SC_MODULE(dut) {
    sc_length_param l1;
    sc_length_context c1;
    sc_signal<sc_signed> s1;
    sc_signal<sc_signed> s2;

    sc_length_param l2;
    sc_length_context c2;
    sc_signal<sc_signed> u1;
    sc_signal<sc_signed> u2;
    SC_CTOR(dut)
        : l1(5), c1(l1), s1("s1"), s2("s2"),
          l2(8), c2(l2), u1("u1"), u2("u2")
    {
    }
}
```

With simple variables of this type, you reuse the context object. However, you must have the extra parameter and context objects when you are using them in a constructor-initialization list because the compiler does not allow repeating an item in that list.

The four fixed-point types that use construction parameters work exactly the same way, except that they use the objects `sc_fxtype_contxt` and `sc_fxtype_params` to do the work. Also, there are more parameters you can set for fixed-point numbers. Assuming you want to set only the length of the number and the number of fractional bits, the following example is similar to the preceding example, modified for fixed-point numbers:

```
SC_MODULE(dut) {
    sc_fxtype_params p1;
    sc_fxtype_contxt c1;
    sc_signal<sc_fix> s1;
    sc_signal<sc_fix> s2;
    sc_fxtype_params p2;
    sc_fxtype_contxt c2;
    sc_signal<sc_ufix> u1;
    sc_signal<sc_ufix> u2;
    SC_CTOR(dut)
    : p1(5,0), c1(p1), s1("s1"), s2("s2"),
      p2(8,5), c2(p2), u1("u1"), u2("u2")
    {
    }
}
```

Troubleshooting SystemC Errors

In the process of modifying your SystemC design to run on ModelSim, you may encounter several common errors. This section highlights some actions you can take to correct such errors.

Unexplained Behaviors During Loading or Runtime

If your SystemC simulation behaves in otherwise unexplainable ways, you should determine whether you need to adjust the stack space ModelSim allocates for threads in your design. The required size for a stack depends on the depth of functions on that stack and the number of bytes they require for automatic (local) variables.

By default the SystemC stack size is 10,000 bytes per thread.

You may have one or more threads needing a larger stack size. If so, call the SystemC function `set_stack_size()` and adjust the stack to accommodate your needs. Note that you can ask for too much stack space and have unexplained behavior as well.

Errors During Loading

When simulating your SystemC design, you might get a "failed to load sc lib" message because of an undefined symbol, looking something like this:

```
# Loading /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so
# ** Error: (vsim-3197) Load of
"/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so" failed: ld.so.1:
/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file
```

```
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:
referenced symbol not found.
# ** Error: (vsim-3676) Could not load shared library
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so for SystemC module
'host_xtor'.
```

Source of Undefined Symbol Message

The causes for such an error could be:

- missing definition of a function/variable
- missing type
- object file or library containing the defined symbol is not linked
- mixing of C and C++ compilers to compile a testcase
- using SystemC 2.2 header files from other vendors
- bad link order specified in sccom -link
- multiply-defined symbols

Missing Definition

If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

This should appear in any header files include in your C++ sources compiled by **sccom**. It tells the compiler to expect a regular C function; otherwise the compiler decorates the name for C++ and then the symbol can't be found.

Also, be sure that you actually linked with an object file that fully defines the symbol. You can use the "nm" utility on Unix platforms to test your SystemC object files and any libraries you link with your SystemC sources. For example, assume you ran the following commands:

```
sccom test.cpp  
sccom -link libSupport.a
```

If there is an unresolved symbol and it is not defined in your sources, it should be correctly defined in any linked libraries:

```
nm libSupport.a | grep "mySymbol"
```

Missing Type

When you get errors during design elaboration, be sure that all the items in your SystemC design hierarchy, including parent elements, are declared in the declarative region of a module. If not, **sccom** ignores them.

For example, consider a design containing SystemC over VHDL. The following declaration of a child module "test" inside the constructor module of the code is not allowed and will produce an error:

```
SC_MODULE (Export)
{
    SC_CTOR (Export)
    {
        test *testInst;
        testInst = new test("test");
    }
};
```

The error results from the fact that the SystemC parse operation will not see any of the children of "test". Nor will any debug information be attached to it. Thus, the signal has no type information and cannot be bound to the VHDL port.

The solution is to move the element declaration into the declarative region of the module.

Using SystemC 2.2 Header Files Supplied by Other Vendors

SystemC 2.2 includes version control for SystemC header files. If you compile your SystemC design using a SystemC 2.2 header file that was distributed by other vendors, and then you run **sccom -link** to link the design, an error similar to the following may result upon loading the design:

```
** Error: (vsim-3197) Load of "work/systemc.so" failed: work/systemc.so:
undefined symbol: _ZN20sc_api_version_2_1_0C1Ev.
```

To resolve the error, recompile the design using **sccom**. Make sure any include paths read by **sccom** do not point to a SystemC 2.2 installation. By default, **sccom** automatically picks up the ModelSim SystemC header files.

Misplaced -link Option

The order in which you place the **-link** option within the **sccom -link** command is critical. There is a big difference between the following two commands:

```
sccom -link liblocal.a
```

and

```
sccom liblocal.a -link
```

The first command ensures that your SystemC object files are seen by the linker before the library "liblocal.a" and the second command ensures that "liblocal.a" is seen first. Some linkers can look for undefined symbols in libraries that follow the undefined reference while others can look both ways. For more information on command syntax and dependencies, see [sccom](#).

Multiple Symbol Definitions

The most common type of error found during **sccom -link** operation is the multiple symbol definition error. The error message looks something like this:

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void) ':
work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'
work/sc/test_ringbuf.o(.text+0x4): first defined here
```

This error arises when the same global symbol is present in more than one *.o* file. There are two common causes of this problem:

- A stale *.o* file in the working directory with conflicting symbol names.

In this first case, just remove the stale files with the following command:

```
vdel -lib <lib_path> -allsystemc
```

- Incorrect definition of symbols in header files.

In the second case, if you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (for instance, not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

Text in *.h* files is included into *.cpp* files by the C++ preprocessor. By the time the compiler sees the text, it's just as if you had typed the entire text from the *.h* file into the *.cpp* file. So an *.h* file included into two *.cpp* files results in lots of duplicate text being processed by the C++ compiler when it starts up. Include guards are a common technique to avoid duplicate text problems.

If an *.h* file has an out-of-line function defined, and that *.h* file is included into two *.c* files, then the out-of-line function symbol will be defined in the two corresponding *.o* files. This leads to a multiple symbol definition error during **sccom -link**.

To solve this problem, add the "inline" keyword to give the function "internal linkage." This makes the function internal to the *.o* file, and prevents the function's symbol from colliding with a symbol in another *.o* file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a *.cpp* file.

Chapter 10

Mixed-Language Simulation

ModelSim single-kernel simulation allows you to simulate designs that are written in VHDL, Verilog, SystemVerilog, and SystemC. While design units must be entirely of one language type, any design unit may instantiate design units from another language. Any instance in the design hierarchy may be a design unit from another language without restriction. In addition, ModelSim supports a procedural interface between SystemC and SystemVerilog, so you may make calls between these languages at the procedural level.

Basic Mixed-Language Flow

Simulating mixed-language designs with ModelSim includes these general steps:

1. Compile HDL source code using `vcom` or `vlog`. Compile SystemC C++ source code using `scom`. Compile all modules in the design following order-of-compile rules.
 - For SystemC designs with HDL instances — Create a SystemC foreign module declaration for all Verilog/SystemVerilog and VHDL instances (see [SystemC Foreign Module \(Verilog\) Declaration](#) or [SystemC Foreign Module \(VHDL\) Declaration](#)).
 - For Verilog/SystemVerilog/VHDL designs with SystemC instances — Export any SystemC instances that will be directly instantiated by the other language using the `SC_MODULE_EXPORT` macro. Exported SystemC modules can be instantiated just as you would instantiate any Verilog/SystemVerilog/VHDL module or design unit.
 - For binding Verilog design units to VHDL or Verilog design units or SystemC modules — See “[Using SystemVerilog bind Construct in Mixed-Language Designs](#).” When using `bind` in compilation unit scope, use the `-cname` argument with the `vlog` command (see [Handling Bind Statements in the Compilation Unit Scope](#)).
 - For VHDL that instantiates Verilog — Do not use `-nodebug=ports` during compilation of the Verilog modules because VHDL will not have the necessary access to the port information.
2. For designs containing SystemC — Link all objects in the design using `scom -link`.
3. Elaborate and optimize your design using the `vopt` command. See [Optimizing Mixed Designs](#).
4. Simulate the design with the `vsim` command.

5. Run and debug your design.

Separate Compilers with Common Design Libraries

VHDL source code is compiled by `vcom` and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in the working library. Likewise, Verilog/SystemVerilog source code is compiled by `vlog` and the resulting design units (modules and UDPs) are stored in the working library.

SystemC/C++ source code is compiled with the `sccom` command. The resulting object code is compiled into the working library.

Design libraries can store any combination of design units from any of the supported languages, provided the design unit names do not overlap (VHDL design unit names are changed to lower case). See [Design Libraries](#) for more information about library management.

Case Sensitivity

Note that VHDL and Verilog observe different rules for case sensitivity:

- VHDL is not case-sensitive. For example, `c1k` and `CLK` are regarded as the same name for the same signal or variable.
- Verilog (and SystemVerilog) are case-sensitive. For example, `c1k` and `CLK` are regarded as different names that you could apply to different signals or variables.

Caution



VHDL is not case-sensitive, so when you run `vcom -mixedsvvh` to compile the VHDL package to use in Verilog or SystemVerilog, it silently converts all names in the package to lower case (for example, `InterfaceStage` becomes `interfacestage`). Because Verilog and SystemVerilog are case-sensitive, when you run the `vlog` compiler, it looks for `InterfaceStage` in the compiled VHDL package but will not find it because it does not match `interfacestage` (which is what `vcom -mixedsvvh` produced).

This means that you must write anything in a VHDL package that SystemVerilog uses in lower case in the SystemVerilog source code, regardless of the upper/lower case used in the VHDL source code.

Using Hierarchical References

ModelSim supports the IEEE 1076-2008 standard "external name" syntax that allows you to make hierarchical references from VHDL to VHDL. Currently, these references can cross Verilog boundaries but must begin and end in VHDL.

Note



The target of an external name must be a VHDL object. The location of the VHDL external name declaration must be in VHDL but the actual path can start anywhere. This only applies to the absolute path name because the relative path name starts at the enclosing concurrent scope where the external name occurs.

The external names syntax allows references to be made to signals, constants, or variables:

```
<<SIGNAL external_pathname : subtype_indication>>
<<CONSTANT external_pathname : subtype_indication>>
<<VARIABLE external_pathname : subtype_indication>>

external_pathname <=
    absolute_pathname | relative_pathname | package_pathname
```

Notice that the standard requires the entire syntax be enclosed in double angle brackets, << >>. It also requires that you specify the type of the object you are referencing.

Here are some examples of external references:

```
REPORT "Test Pin = " & integer'image(<<SIGNAL .tb.dut.i0.tp : natural>>)
    SEVERITY note;

Q <= <<SIGNAL .tb.dut.i0.tp : std_logic_vector(3 DOWNT0 0)>>;

ALIAS test_pin IS <<SIGNAL .tb.dut.i0.tp : std_logic_vector(3 DOWNT0 0)>>;
    ...
test_pin(3) <= '1';
Q(0) <= test_pin(0);
```

To use this capability, compile your VHDL source for the IEEE 1076-2008 syntax as follows:

vcom -2008 design.vhd testbench.vhd

Note



Indexing and slicing of the name appears outside of the external name and is not part of the external path name itself. For example:

```
<< signal u1.vector : std_logic_vector>>(3)
instead of
<< signal u1.vector(3): std_logic>>
```

Note



The 1076-2002 syntax is the compiler's default.

The order of elaboration for Verilog to Verilog references that cross VHDL boundaries does not matter. However, the object referenced by a VHDL external name must be elaborated before it can be referenced.

SystemVerilog binds in VHDL scopes are translated to "equivalent" VHDL so any restrictions on VHDL external names apply to the hierarchical references in the bind statement (that is, the target must be a VHDL object.) Since binds are done after all other instances within a scope there should be no ordering issues.

Access Limitations in Mixed-Language Designs

You *cannot* directly read or change a SystemC object with a hierarchical reference within a mixed-language design. Further, you cannot directly access a Verilog/SystemVerilog object up or down the hierarchy if there is an interceding SystemC block.

To access obstructed SystemC objects, propagate the value through the ports of all design units in the hierarchy or use the control/observe functions. You can use either of the following member functions of `sc_signal` to control and observe hierarchical signals in a design:

- `control_foreign_signal()`
- `observe_foreign_signal()`

For more information on the use of control and observe, see “[Hierarchical References In Mixed HDL and SystemC Designs](#)”.

Using SystemVerilog bind Construct in Mixed-Language Designs

The SystemVerilog **bind** construct allows you to bind a Verilog design unit to another Verilog design unit or to a VHDL design unit or to a SystemC module. This is especially useful for binding SystemVerilog assertions to your SystemC, VHDL, Verilog and mixed designs during verification.

Binding one design unit to another is a simple process of creating a module that you want to bind to a target design unit, then writing a bind statement. For example, if you want to bind a SystemVerilog assertion module to a VHDL design, do the following:

1. Write assertions inside a Verilog module.
2. Designate a target VHDL entity or a VHDL entity/architecture pair.
3. Bind the assertion module to the target with a **bind** statement.

The procedure for binding a SystemVerilog assertion module to a SystemC module is similar except that in step 2 you designate a target top level SystemC module or an instance of a SystemC module in the SystemC design hierarchy.

Modules, programs, or interfaces can be bound to:

- all instances of a target SystemC module

- a specific instance of the target SystemC module
- all instances that use a certain architecture in the target module

Binding to a configuration is not allowed.

Syntax of bind Statement

To bind a SystemVerilog assertion module to a VHDL design, the syntax of the **bind** statement is:

```
bind <target_entity/architecture_name> <assertion_module_name>
<instance_name> <port connections>
```

For binding to a SystemC module, the syntax is:

```
bind <target SystemC module/full hierpath of an instance of a SystemC
module> <assertion_module_name> <instance_name> <port connections>
```

This **bind** statement will create an instance of the assertion module inside the target VHDL entity/architecture or SystemC module with the specified instance name and port connections. When the target is a VHDL entity, the **bind** instance is created under the last compiled architecture. Note that the instance being bound cannot contain another **bind** statement. In addition, a bound instance can make hierarchical reference into the design.

What Can Be Bound

The following list provides examples of what can be bound.

- Bind to all instances of a VHDL entity.

```
bind e bind_du inst(p1, p2);
```

- Bind to all instances of a VHDL entity & architecture.

```
bind \e(a) bind_du inst(p1, p2);
```

- Bind to multiple VHDL instances.

```
bind test.dut.inst1 bind_du inst(p1, p2);
bind test.dut.inst2 bind_du inst(p1, p2);
bind test.dut.inst3 bind_du inst(p1, p2);
```

- Bind to a single VHDL instance.

```
bind test.dut.inst1 bind_du inst(p1, p2);
```

- Bind to an instance where the instance path includes a for generate scope.

```
bind test.dut/forgen__4/inst1 bind_du inst(p1, p2);
```

- Bind to all instances of a VHDL entity and architecture in a library.

```
bind \mylib.e(a) bind_du inst(p1, p2);
```

- Bind to all instances of a SystemC module.

```
bind sc_mod bind_du inst(p1, p2);
```

- Bind to multiple SystemC module instances.

```
bind test.dut.sc_inst1 bind_du inst(p1, p2);  
bind test.dut.sc_inst2 bind_du inst(p1, p2);  
bind test.dut.sc_inst3 bind_du inst(p1, p2);
```

- Bind to a single SystemC module instance.

```
bind test.dut.sc_inst1 bind_du inst(p1, p2);
```

Hierarchical References to a VHDL Object from a Verilog/SystemVerilog Scope

ModelSim supports hierarchical references to VHDL Objects from a Verilog/SystemVerilog scope.

The SystemVerilog "bind" construct allows you to access VHDL or Verilog objects. The only restrictions applied are those of the bind context. For example, if you are binding into a VHDL architecture, any hierarchical references in the bind statement must have targets in VHDL. A bind into a Verilog context can have hierarchical references resolve to either VHDL or Verilog objects.

Supported Objects

The only VHDL object types that can be referenced are: signals, shared variables, constants, and generics *not* declared within processes. You cannot read VHDL process variables. In addition, VHDL functions, procedures, and types are not supported.

VHDL signals are treated as Verilog wires. You can use hierarchical references to VHDL signals in instances and left-hand sides of continuous assignments, which can be read any place a wire can be read and used in event control. Blocking assignments, non-blocking assignments, force, and release are not supported for VHDL signals.

VHDL shared variables can be read anywhere a Verilog reg can be read. VHDL variables do not have event control on them, therefore hierarchical references to VHDL shared variables used in event control are an error by default. The statement @(vhdl_entity.shared_variable) will never trigger. Because of this, you cannot use hierarchical references to VHDL shared variables in instance port maps.

You can use non-blocking assignments and blocking assignments on VHDL shared variables. VHDL constant and generics can be read anywhere. ModelSim treats them similarly to Verilog

parameters. The one exception is that they should not be used where constant expressions are required. In addition, VHDL generics cannot be changed by a defparam statement.

Supported Types

The following VHDL data types are supported for hierarchical references:

- basic scalar types
- vectors of scalar types
- fields of record that are supported types

If the VHDL type is in a package that is compiled with `vcom -mixedsvvh`, then the VHDL type will be accessible in Verilog. If the type is not in a package or not compiled `vcom -mixedsvvh` and an enum or record, then Verilog has limited access to it. It can read enum values as integers, but cannot assign to enum objects because of strict type checking.

Complex types like records are supported if there exists a matching type in the language generated with the `-mixedsvvh` switch for either the `vcom` or `vlog` commands.

Using SignalSpy for Hierarchical Access

The ModelSim Signal Spy™ technology provides hierarchical access to bound SystemVerilog objects from VHDL objects. SystemVerilog modules also can access bound VHDL objects using Signal Spy, and they can access bounded Verilog objects using standard Verilog hierarchical references. See the [Signal Spy](#) chapter for more information on the use of Signal Spy.

Mapping of Types

All SystemVerilog data types supported at the SV-VHDL boundary are supported while binding to VHDL target scopes. This includes hierarchical references in actual expressions *if* they terminate in a VHDL scope. These data-types follow the same type-mapping rules followed at the SV-VHDL mixed-language boundary for direct instantiation. See [Mapping Data Types](#).

All the types supported at the SystemC-SystemVerilog mixed language boundary are also supported when binding to a SystemC target. Please refer to [Verilog or SystemVerilog and SystemC Signal Interaction And Mappings](#) for a complete list of all supported types.

Using SV Bind With or Without vopt

SV **bind**, when using `vopt`, is fully compatible with IEEE1800-2009 LRM.

When you use SV **bind** without `vopt` (either by using the `-novopt` switch with `vsim` or by setting the `VoptFlow` variable to 0 in the `modelsim.ini` file), the actual expression in a **bind** port

map must be simple names (including hierarchical names if the target is a Verilog design unit) and Verilog literals. For example:

```
bind target checker inst(req, ack, 1;b1)
```

is a legal expression; whereas,

```
bind target checker inst(req | ack, {req1, req2})
```

is illegal because the actual expressions are neither simple names nor literals.

Additional restrictions on actual expressions exist for both Verilog and VHDL. For example, if the target of **bind** is a VHDL design unit or an instance of a VHDL design unit, the following types of actual expressions are supported only with **vopt**:

- bitwise binary expressions using operators &, |, ~, ^ and ^~
- concatenation expression
- bit select and part select expressions
- any variable/constant visible in the target scope including those defined in packages
- hierarchical references to VHDL signals/constants/variables

See [Optimizing Designs with vopt](#) for further details.

Binding to VHDL Enumerated Types

SystemVerilog infers an enumeration concept similar to VHDL enumerated types. In VHDL, the enumerated names are assigned to a fixed enumerated value, starting left-most with the value 0. In SystemVerilog, you can also explicitly define the enumerated values. As a result, the bind construct can be used for port mapping of VHDL enumerated types to Verilog port vectors.

Port mapping is supported for both input and output ports. The integer value of the enum is first converted to a bit vector before connecting to a Verilog formal port. Note that you cannot connect enum value on port actual – it has to be signal. In addition, port vectors can be of any size less than or equal to 32.

This kind of port mapping between VHDL enum and Verilog vector is only allowed when the Verilog is instantiated under VHDL through the bind construct and is not supported for normal instances.

The allowed VHDL types for port mapping to SystemVerilog port vectors are:

Table 10-1. VHDL Types Mapped To SystemVerilog Port Vectors

bit	std_logic	vl_logic
bit_vector	std_logic_vector	vl_logic_vector

See also, [Sharing User-Defined Types](#).

Example of Binding to VHDL Enumerated Types

Suppose you want to use SVA to monitor a VHDL finite state machine that uses enumerated types. With ModelSim, you can directly map VHDL enumerated types to SystemVerilog enumerated types. The same type-sharing rules are followed, which are applicable for direct instantiations at the SV-VHDL mixed-language boundary (see [Sharing User-Defined Types](#)).

Consider the following enumerated type defined in a VHDL package:

```

--/*-----pack.vhd-----*/
package pack is
    type fsm_state is(idle, send_bypass,
        load0,send0, load1,send1, load2,send2,
        load3,send3, load4,send4, load5,send5,
        load6,send6, load7,send7, load8,send8,
        load9,send9, load10,send10,
        load_bypass, wait_idle);
end package;

```

To use this at the SystemVerilog-VHDL mixed-language boundary, you must compile this package with the `-mixedsvvh` option for the `vcom` command:

vcom -mixedsvvh pack.vhd

You can include this package in your VHDL target, like a normal VHDL package:

```

use work.pack.all;
...
signal int_state : fsm_state;
signal nxt_state : fsm_state;
...

```

This package can also be imported in the SystemVerilog module containing the properties to be monitored, as if it were a SystemVerilog package.

```

import pack::*;
...
input port:
module interleaver_props (
    input clk, in_hs, out_hs,
    input fsm_state int_state
);
...
// Check for sync byte at the start of a every packet property
pkt_start_check;
- @(posedge clk) (int_state == idle && in_hs) -> (sync_in_valid);
endproperty
...

```

Now, suppose you want to implement functional coverage of the VHDL finite state machine states. With ModelSim, you can bind any SystemVerilog functionality, such as functional coverage, into a VHDL object:

```
...
covergroup sm_cvg @(posedge clk);
  coverpoint int_state
  {
    bins idle_bin = {idle};
    bins load_bins = {load_bypass, load0, load9, load10};
    bins send_bins = {send_bypass, send0, send9, send10};
    bins others = {wait_idle};
    option.at_least = 500;
  }
  coverpoint in_hs;
  in_hsXint_state: cross in_hs, int_state;
endgroup

sm_cvg sm_cvg_c1 = new;
...
```

As with monitoring VHDL components, you create a wrapper to connect the SVA to the VHDL component:

```
module interleaver_binds;
...
// Bind interleaver_props to a specific interleaver instance
// and call this instantiate interleaver_props_bind
bind interleaver_m0 interleaver_props interleaver_props_bind (
  // connect the SystemVerilog ports to VHDL ports (clk)
  // and to the internal signal (int_state)
  .clk(clk), ..
  .int_state(int_state)
);
...
endmodule
```

Again, you can use either of two options to perform the actual binding in ModelSim — instantiation or the loading of multiple top modules into the simulator.

```
vsim interleaver_tester interleaver_binds
```

Binding to a VHDL Instance

ModelSim also supports the binding of SystemVerilog into VHDL design units that are defined by generate or configuration statements.

Consider the following VHDL code:

```
architecture Structure of Test is
  signal A, B, C : std_logic_vector(0 to 3);
...
begin
```

```

TOP : for i in 0 to 3 generate
  First : if i = 0 generate
    - configure it..
    for all : thing use entity work.thing(architecture_ONE);
  begin
    Q : thing port map (A(0), B(0), C(0));
  end generate;

  Second : for i in 1 to 3 generate
    - configure it..
    for all : thing use entity work.thing(architecture_TWO);
  begin
    Q : thing port map ( A(i), B(i), C(i) );
  end generate;
end generate;
end Structure;

```

The following SystemVerilog program defines the assertion:

```

program SVA (input c, a, b);
...
sequence s1;
  @(posedge c) a ##1 b ;
endsequence
cover property (s1);
...
endprogram

```

To tie the SystemVerilog cover directive to the VHDL component, you can use a wrapper module such as the following:

```

module sva_wrapper;
  bind test.top__2.second__1.q// Bind a specific instance
  SVA// to SVA and call this
  sva_bind// instantiation sva_bind
  ( .a(A), .b(B), .c(C) );// Connect the SystemVerilog ports
  // to VHDL ports (A, B and C)
endmodule

```

You can instantiate `sva_wrapper` in the top level or simply load multiple top modules into the simulator:

```

vlib work
vlog *.sv
vcom *.vhd
vsim test sva_wrapper

```

This binds the SystemVerilog program, `SVA`, to the specific instance defined by the generate and configuration statements.

You can control the format of generate statement labels by using the [GenerateFormat](#) variable in the *modelsim.ini* file.

Handling Bind Statements in the Compilation Unit Scope

Bind statements are allowed in module, interface, and program blocks, and may exist in the compilation unit scope. ModelSim treats the compilation unit scope (\$unit) as a package – internally wrapping the content of \$unit into a package.

Before [vsim](#) elaborates a module it elaborates all packages upon which that module depends. In other words, it elaborates a \$unit package before a module in the compilation unit scope.

It should be noted that when the **bind** statement is in the compilation unit scope, the **bind** only becomes effective when \$unit package gets elaborated by [vsim](#). In addition, the package gets elaborated only when a design unit that depends on that package gets elaborated. So if you have a file in a compilation unit scope that contains only **bind** statements, you can compile that file by itself, but the **bind** statements will never be elaborated. A warning to this effect is generated by [vlog](#) if **bind** statements are found in the compilation unit scope.

The **-cname** argument for [vlog](#) gives a user-defined name to a specified compilation \$unit package (which, in the absence of **-cname**, is some implicitly generated name). You must provide this named compilation unit package with the [vsim](#) command as the top level design unit in order to force elaboration.

The **-cname** argument is used only in conjunction with the **-mfcu** argument, which instructs the compiler to treat all files within a compilation command line as a single compilation unit.

Example 10-1. Binding with -cname and -mfcu Arguments

Suppose you have a SystemVerilog module, called *checker.sv*, that contains an assertion for checking a counter:

```
module checker(clk, reset, cnt);
  parameter SIZE = 4;
  input clk;
  input reset;
  input [SIZE-1:0] cnt;
  property check_count;
    @(posedge clk)
    !reset | => cnt == ($past(cnt) + 1);
  endproperty
  assert property (check_count);
endmodule
```

You want to **bind** that to a counter module named *counter.sv*.

```

module counter(clk, reset, cnt);
parameter SIZE = 8;
input clk;
input reset;
output [SIZE-1:0] cnt;
reg [SIZE-1:0] cnt;
always @(posedge clk)
begin
    if (reset == 1'b1)
        cnt = 0;
    else
        cnt = cnt + 1;
end
endmodule

```

The **bind** statement is in a file named *bind.sv*, which will reside in the compilation unit scope.

```
bind counter checker #(SIZE) checker_inst(clk, reset, cnt);
```

This statement instructs ModelSim to create an instance of *checker* in the target module, *counter.sv*.

The final component of this design is a test bench, named *tb.sv*.

```

module testbench;
reg clk, reset;
wire [15:0] cnt;
counter #(16) inst(clk, reset, cnt);
initial
begin
    clk = 1'b0;
    reset = 1'b1;
    #500 reset = 1'b0;
    #1000 $finish;
end
always #50 clk = ~clk;
endmodule

```

If the *bind.sv* file is compiled by itself (vlog bind.sv), you will receive a Warning like this one:

```
** Warning: 'bind' found in a compilation unit scope that either does not
contain any design units or only contains design units that are
instantiated by 'bind'. The 'bind' instance will not be elaborated.
```

To fix this problem, use the **-cname** argument with **vlog** as follows:

```
vlog -cname bind_pkg -mfcu bind.sv
```

Then simulate the design with:

```
vsim testbench bind_pkg
```

If you are using the **vlog -R** commands to compile and simulate the design, this binding issue is handled properly automatically.

Limitations to Bind Support for SystemC

There exists certain restrictions on actual expressions when binding to SystemC targets. If the target of a bind is a SystemC module or an instance of a SystemC module, expressions and literals are not supported as actuals. These include, but are not limited to,

- bitwise binary expressions using operators `&`, `|`, `~`, `^` and `^~`
- concatenation expression
- bit select and part select expressions
- variable/constant

Optimizing Mixed Designs

The `vopt` command performs global optimizations to improve simulator performance. You run `vopt` on the top-level design unit. See [Optimizing Designs with vopt](#) for further details.

Simulator Resolution Limit

In a mixed-language design with only one top, the resolution of the top design unit is applied to the whole design. If the root of the mixed design is VHDL, then VHDL simulator resolution rules are used (see [Simulator Resolution Limit for VHDL](#) for VHDL details). If the root of the mixed design is Verilog or SystemVerilog, Verilog rules are used (see [Simulator Resolution Limit \(Verilog\)](#) for details). If the root is SystemC, then SystemC rules are used (see [SystemC Time Unit and Simulator Resolution](#) for details).

In the case of a mixed-language design with multiple tops, the following algorithm is used:

- If VHDL or SystemC modules are present, then the Verilog resolution is ignored. An error is issued if the Verilog resolution is finer than the chosen one.
- If both VHDL and SystemC are present, then the resolution is chosen based on which design unit is elaborated first. For example:

```
vsim sc_top vhdl_top -do vsim.do
```

In this case, the SystemC resolution (default 1 ns) is chosen.

```
vsim vhdl_top sc_top -do vsim.do
```

In this case, the VHDL resolution is chosen.

- All resolutions specified in the source files are ignored if `vsim` is invoked with the `-t` option. When set, this overrides all other resolutions.

Runtime Modeling Semantics

The ModelSim simulator is compliant with all pertinent Language Reference Manuals. To achieve this compliance, the sequence of operations in one simulation iteration (that is, delta cycle) is as follows:

- SystemC processes are run
- Signal updates are made
- HDL processes are run

The above scheduling semantics are required to satisfy both the SystemC and the HDL LRM. Namely, all processes triggered by an event in a SystemC primitive channel shall wake up at the beginning of the following delta. All processes triggered by an event on an HDL signal shall wake up at the end of the current delta. For a signal chain that crosses the language boundary, this means that processes on the SystemC side get woken up one delta later than processes on the HDL side. Consequently, one delta of skew will be introduced between such processes. However, if the processes are communicating with each other, correct system behavior will still result.

Hierarchical References to SystemVerilog

Hierarchical references to SystemVerilog properties/sequences is supported with the following restrictions.

- Clock and disable iff expressions cannot have a formal.
- Method 'matched' not supported on a hierarchically referenced sequence

Hierarchical References In Mixed HDL and SystemC Designs

A SystemC signal (including `sc_signal`, `sc_buffer`, `sc_signal_resolved`, and `sc_signal_rv`) can control or observe an HDL signal using two member functions of `sc_signal`:

```
bool control_foreign_signal(const char* name);  
bool observe_foreign_signal(const char* name);
```

The argument (`const char* name`) is a full hierarchical path to an HDL signal or port. These functions always returns "true" for all cases (even if the call failed). However, an error is issued if the call could not be completed due to any reason. See tables for Verilog/SystemVerilog ([Data Type Mapping from SystemC to Verilog or SystemVerilog](#)) and VHDL ([Data Type Mapping Between SystemC and VHDL](#)) to view a list of types supported at the mixed language boundary. If it is a supported boundary type, it is supported for hierarchical references.

Note



SystemC control/observe always return “true” for all cases (even if the call failed).

Control

When a SystemC signal calls **control_foreign_signal()** on an HDL signal, the HDL signal is considered a fanout of the SystemC signal. This means that every value change of the SystemC signal is propagated to the HDL signal. If there is a pre-existing driver on the HDL signal which has been controlled, the value of the HDL signal is the resolved value of the existing driver and the SystemC signal. This value remains in effect until a subsequent driver transaction occurs on the HDL signal, following the semantics of the **force -deposit** command.

Observe

When a SystemC signal calls **observe_foreign_signal()** on an HDL signal, the SystemC signal is considered a fanout of the HDL signal. This means that every value change of the HDL signal is propagated to the SystemC signal. If there is a pre-existing driver on the SystemC signal which has been observed, the value is changed to reflect that of the HDL signal. This value remains in effect until a subsequent driver transaction occurs on the SystemC signal, following the semantics of the **force -deposit** command.

Example:

```
SC_MODULE(test_ringbuf)
{
    sc_signal<bool> observe_sig;
    sc_signal<sc_lv<4> > control_sig;

    // HDL module instance
    ringbuf* ring_INST;

    SC_CTOR(test_ringbuf)
    {
        ring_INST = new ringbuf("ring_INST", "ringbuf");
        .....

        observe_sig.observe_foreign_signal("/test_ringbuf/ring_INST/block1_INST/buffers(0)");

        control_sig.control_foreign_signal("/test_ringbuf/ring_INST/block1_INST/sig");
    }
};
```

Signal Connections Between Mixed HDL and SystemC Designs

You can use the `scv_connect()` API function to connect a SystemC signal (including `sc_signal`, `sc_buffer`, `sc_signal_resolved`, and `sc_signal_rv`) to an HDL signal.

Note



The behavior of `scv_connect()` is identical to the behavior of the [Control](#) and [Observe](#) functions, described above in [Hierarchical References In Mixed HDL and SystemC Designs](#).

The `scv_connect()` API is provided by the SystemC Verification Standard and is defined as follows:

```
/* Function to connect an sc_signal object to an HDL signal. */
template < typename T> void scv_connect(
    sc_signal<T> & signal,
    const char * hdl_signal,
    scv_hdl_direction d = SCV_OUTPUT,
    unsigned hdl_sim_inst = 0
);

/* Function to connect an sc_signal_resolved object to an HDL signal. */
void scv_connect(
    sc_signal_resolved& signal,
    const char * hdl_signal,
    scv_hdl_direction d = SCV_OUTPUT,
    unsigned hdl_sim_inst = 0
);

/* Function connects an sc_signal_rv object to an HDL signal. */
template < int W> void scv_connect(
    sc_signal_rv<W>& signal,
    const char * hdl_signal,
    scv_hdl_direction d = SCV_OUTPUT,
    unsigned hdl_sim_inst = 0
);
```

where

- `signal` is an `sc_signal`, `sc_signal_resolved` or `sc_signal_rv` object
- `hdl_signal` is the full hierarchical path to an HDL signal or port
- `d` is the direction of the connection given by enum `scv_hdl_direction`
- `hdl_sim_inst` is not supported—any value given for this argument will be ignored

```
enum scv_hdl_direction {
```

```

    SCV_INPUT = 1, /* HDL is the only driver */
    SCV_OUTPUT = 2 /* SystemC is the only driver */
};

```

Supported Types

The `scv_connect()` function supports all datatypes supported at the SystemC-HDL mixed-language boundaries. Refer to the tables for Verilog/SystemVerilog ([Data Type Mapping from SystemC to Verilog or SystemVerilog](#)) and VHDL ([Data Type Mapping Between SystemC and VHDL](#)) to view a list of types supported at the mixed language boundary. If it is a supported boundary type, it is supported for hierarchical references.

Mapping Data Types

Cross-language (HDL) instantiation does not require additional effort on your part. As ModelSim loads a design, it detects cross-language instantiations because it can determine the language type of each design unit as it is loaded from a library. ModelSim then performs the necessary adaptations and data type conversions automatically. SystemC and HDL cross-language instantiation requires minor modification of SystemC source code (such as the addition of `SC_MODULE_EXPORT` and `sc_foreign_module`).

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. This is also true for SystemC and VHDL/Verilog/SystemVerilog ports.

The following sections describe mixed-language mappings for ModelSim:

- [Verilog and SystemVerilog to VHDL Mappings](#)
- [VHDL To Verilog and SystemVerilog Mappings](#)
- [Verilog or SystemVerilog and SystemC Signal Interaction And Mappings](#)
- [VHDL and SystemC Signal Interaction And Mappings](#)
- [Verilog or SystemVerilog and SystemC Signal Interaction And Mappings](#)

Verilog and SystemVerilog to VHDL Mappings

[Table 10-2](#) shows the mapping of data types from SystemVerilog to VHDL.

Table 10-2. SystemVerilog-to-VHDL Data Type Mapping

SystemVerilog Type	VHDL Type		Comments
	Primary mapping	Secondary mapping	
bit	bit	std_logic	2-state scalar data type

Table 10-2. SystemVerilog-to-VHDL Data Type Mapping

SystemVerilog Type	VHDL Type		Comments
	Primary mapping	Secondary mapping	
logic	std_logic	bit	4-state scalar data type
reg	std_logic	bit	4-state scalar data type
wire	std_logic	bit	A scalar wire
bit vector	bit_vector	std_logic_vector	A signed/unsigned, packed/unpacked single dimensional bit vector
reg vector	std_logic_vector	bit_vector	A signed/unsigned, packed/unpacked single dimensional logic vector
wire vector	std_logic_vector	bit_vector	A signed/unsigned, packed/unpacked single dimensional multi-bit wire
logic vector	std_logic_vector	bit_vector	A signed/unsigned, packed/unpacked single dimensional logic vector
integer	integer		4-state data type, 32-bit signed integer
integer unsigned	integer		4-state data type, 32-bit unsigned integer
int	integer		2-state data type, 32-bit signed integer
shortint	integer		2-state data type, 16-bit signed integer
longint	integer		2-state data type, 64-bit signed integer
int unsigned	integer		2-state data type, 32-bit unsigned integer
shortint unsigned	integer		2-state data type, 16-bit unsigned integer
longint unsigned	integer		2-state data type, 64-bit unsigned integer
byte	integer		2-state data type, 8-bit signed integer or ASCII character

Table 10-2. SystemVerilog-to-VHDL Data Type Mapping

SystemVerilog Type	VHDL Type		Comments
	Primary mapping	Secondary mapping	
byte unsigned	integer		2-state data type, 8-bit unsigned integer or ASCII character
enum	enum		SV enums of only 2-state int base type supported
struct	record		unpacked structure
packed struct	record	std_logic_vector bit_vector	packed structure
real	real		2-state data type, 64-bit real number
shortreal	real		2-state data type, 32-bit real number
multi-D arrays	multi-D arrays		multi-dimensional arrays of supported types

Verilog Parameters

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

Table 10-3. Verilog Parameter to VHDL Mapping

Verilog type	VHDL type
integer ¹	integer
real	real
string	string
packed vector	std_logic_vector bit_vector

1. By default, untyped Verilog parameters that are initialized with unsigned values between $2^{31}-1$ and 2^{32} are converted to VHDL integer generics. Because VHDL integer parameters are signed numbers, the Verilog values $2^{31}-1$ to 2^{32} are converted to negative VHDL values in the range from -2^{31} to -1 (the 2's complement value). To prevent this mapping, compile using the `vlog -noForceUnsignedToVhdlInteger` command.

For more information on using Verilog bit type mapping to VHDL, refer to the Usage Notes under “[VHDL Instantiation Criteria Within Verilog](#).”

Allowed VHDL Types for Verilog Ports

The following is a list of allowed VHDL types for ports connected to Verilog nets and for signals connected to Verilog ports:

bit	real	std_ulogic_vector
bit_vector	record	vl_logic
enum	shortreal	vl_logic_vector
integer	std_logic	vl_ulogic
natural	std_logic_vector	vl_ulogic_vector
positive	std_ulogic	multi-dimensional arrays

Note



Note that you can use the wildcard syntax convention (.*) when instantiating Verilog ports where the instance port name matches the connecting port name and their data types are equivalent.

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 vl_logic type is defined in the following file installed with ModelSim:

```
<install_dir>/vhdl_src/verilog/vltypes.vhd
```

Verilog States

Verilog states are mapped to std_logic and bit as follows:

Table 10-4. Verilog States Mapped to std_logic and bit

Verilog	std_logic	bit
HiZ	'Z'	'0'
Sm0	'L'	'0'
Sm1	'H'	'1'
SmX	'W'	'0'

Table 10-4. Verilog States Mapped to std_logic and bit

Verilog	std_logic	bit
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 To Verilog and SystemVerilog Mappings

Table 10-5 summarizes the mapping of data types from VHDL to SystemVerilog.

Table 10-5. VHDL to SystemVerilog Data Type Mapping

VHDL Type	SystemVerilog Type		Comments
	Primary mapping	Secondary mapping	
bit	bit	reg, logic	2-state scalar data type
boolean	bit	reg, logic	2-state enum data type
std_logic	reg	bit, logic	4-state scalar data type
bit_vector	bit vector	reg vector, wire vector, logic vector, struct packed	A signed/unsigned, packed/unpacked bit vector
std_logic_vector	reg vector	bit_vector, wire vector, logic vector, struct packed	A signed/unsigned, packed/unpacked logic vector
integer	int	integer, shortint, longint, int unsigned, shortint unsigned, longint unsigned, byte, byte unsigned	2-state data type, 32 bit signed integer
enum	enum		VHDL enumeration types
record	struct	packed struct	VHDL records
real	real	shortreal	2-state data type, 64-bit real number
multi-D arrays	multi-D arrays		multi-dimensional arrays of supported types

Mapping VHDL Generics to Verilog Types

Table 10-6 shows the mapping of VHDL Generics to Verilog types.

Table 10-6. VHDL Generics to Verilog Mapping

VHDL type	Verilog type
integer, real, time, physical, enumeration	integer or real
string	string literal
bit, st_logic, bit_vector, std_logic_vector, vl_logic, vl_logic_vector ¹	packed vector

1. Note that Verilog vectors (such as 3'b011) that can be represented as an integer value are mapped to generic of integer type (to preserve backward compatibility). Only vectors whose values cannot be represented as integers (such as 3'b0xx) are mapped to generics of this type.

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.

VHDL type bit is mapped to Verilog states as shown in [Table 10-9](#):

Table 10-7. Mapping VHDL bit to Verilog States

VHDL bit	Verilog State
'0'	St0
'1'	St1

VHDL type std_logic is mapped to Verilog states as shown in [Table 10-8](#):

Table 10-8. Mapping VHDL std_logic Type to Verilog States

VHDL std_logic	Verilog State
'U'	StX
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'_'	StX

VHDL Generics at a VHDL-SV Mixed-Language Boundary

This section describes support for overriding generics at the boundary of a VHDL-SystemVerilog design where VHDL instantiates SystemVerilog. Essentially, overriding generics while instantiating SystemVerilog inside VHDL is identical to overriding parameters while instantiating SystemVerilog inside SystemVerilog.

ModelSim overrides generics at a VHDL-SV boundary based on the style of declaration for the SystemVerilog parameters at the boundary:

- [Verilog-Style Declarations](#)
- [SystemVerilog-Style Declarations](#)
- [Miscellaneous Declarations](#)

Verilog-Style Declarations

This category is for all parameters that are defined using a Verilog-style declaration. This style of declaration does not have a type or range specification, so the type of these parameters is inferred from the final value that gets assigned to them.

Direct Entity Instantiation

The type of the formal Verilog parameter will be changed based on the type inferred from the VHDL actual. While resolving type, ModelSim gives preference to the primary type (type that is inferred from the initial value of the parameter) over other types. Further, ModelSim does not allow subelement association while overriding such generics from VHDL.

For example:

```
// SystemVerilog
parameter p1 = 10;

-- VHDL
inst1 : entity work.svmod generic map (p1 => integer'(20));
inst2 : entity work.svmod generic map (p1 => real'(2.5));
inst3 : entity work.svmod generic map (p1 => string'("Hello World"));
inst3 : entity work.svmod generic map (p1 => bit_vector'("01010101"));
```

Component Instantiation

For Verilog-style declarations, ModelSim allows you to override the default type of the generic in your component declarations.

For example:

```
// SystemVerilog
parameter p1 = 10;

-- VHDL
```

```
component svmod
  generic (p1 : std_logic_vector(7 downto 0));
end component;
...
inst1 : svmod generic map (p1 => "01010101");
```

Table 10-9. Mapping Table for Verilog-style Declarations

Type of Verilog Formal	Type of VHDL Actual
All supported types	All supported types

SystemVerilog-Style Declarations

This category is for all parameters that are defined using a SystemVerilog-style declaration. This style of declaration has an explicit type defined, which does not change based on the value that gets assigned to them.

Direct Entity Instantiation

The type of the SystemVerilog parameter is fixed. While ModelSim overrides it through VHDL, it will be an error if the type of the actual is not one of its equivalent VHDL types. [Table 10-10](#) provides a mapping table that lists equivalent types.

For example:

```
// SystemVerilog
parameter int p1 = 10;

-- VHDL
inst1 : entity work.svmod generic map (p1 => integer'(20)); -- OK
-- inst2 : entity work.svmod generic map (p1 => real'(3.5)); -- ERROR
-- inst3 : entity work.svmod generic map (p1 => string__("Hello World"));
-- ERROR
inst4 : entity work.svmod generic map (p1 => bit_vector__("010101010101"));
-- OK
```

Component Instantiation

ModelSim allows only the VHDL equivalent type of the type of the SystemVerilog parameter in the component declaration. Using any other type will result in a type-mismatch error.

For example:

```
// SystemVerilog
parameter int p1 = 10;
```

```
-- VHDL
component svmod
  generic (p1 : bit_vector(7 downto 0));
end component;
...
inst1 : svmod generic map (p1 => "01010101");
```

Table 10-10. Mapping Table for SystemVerilog-style Declarations

Type of Verilog Formal	Type of VHDL Actual
bit, logic, reg	std_logic bit boolean integer (truncate) std_logic_vector (truncate) bit_vector (truncate) real (round off to nearest integer and handle as bit vector) string (truncate)
bit/logic/reg vector	std_logic (pad with 0) bit (pad with 0) boolean (pad with 0) std_logic_vector (truncate or pad with 0) bit_vector (truncate or pad with 0) integer (truncate or pad with 0) real (round off to nearest integer and handle as bit vector) string (truncate or pad with 0)
integer, int, shortint, longint, byte	bit_vector (truncate or pad with 0) std_logic_vector (truncate or pad with 0) integer (truncate or pad with 0) bit (pad with 0) boolean (pad with 0) std_logic (pad with 0) real (round off to nearest integer) string (truncate or pad with 0)
real, shortreal	real integer
string	string

In addition to the mapping in [Table 10-10](#), ModelSim handles sign specification while overriding SystemVerilog parameters from VHDL in accordance with the following rules:

- A Verilog parameter with a range specification, but with no type specification, shall have the range of the parameter declaration and shall be unsigned. The sign and range shall not be affected by value overrides from VHDL.

- A Verilog parameter with a signed type specification and with a range specification shall be signed and shall have the range of its declaration. The sign and range shall not be affected by value overrides from VHDL.

Miscellaneous Declarations

The following types of parameter declarations require special handling, as described below.

- Untyped SystemVerilog Parameters

These parameters do not have default values and types defined in their declarations. For example:

```
parameter p1;
```

Because no default value is specified, you must specify an overriding parameter value in every instantiation of the parent SV module inside VHDL. ModelSim will consider it an error if these parameters are omitted during instantiation.

Direct Entity Instantiation

Because the parameter does not have a type of its own and takes on the type of the actual, it is important that you define the type of the actual unambiguously. If ModelSim cannot determine the type of the actual, it will be considered an error.

For example:

```
// SystemVerilog
parameter p1;

-- VHDL
inst1 : entity work.svmod generic map (p1 => integer'(20)); -- OK
inst2 : entity work.svmod generic map (p1 => real'(3.5)); -- OK
inst3 : entity work.svmod generic map (p1 => string'("Hello World"));
-- OK
```

Component Instantiation

It is your responsibility to define a type of generics corresponding to untyped SystemVerilog parameters in their component declarations. ModelSim will issue an error if an untyped SystemVerilog parameter is omitted in the component declaration.

The `vgencomp` command will dump a comment instead of the type of the generic, corresponding to an untyped parameter, and prompt you to put in your own type there.

For example:

```
// SystemVerilog
parameter p1;
```

```
-- VHDL
component svmod
  generic (p1 : bit_vector(7 downto 0) := "00000000" );
end component;
...
inst1 : svmod generic map (p1 => "01010101");
```

- **Typed SystemVerilog Parameters**

A parameter can also specify a data type, which allows modules, interfaces, or programs to have ports and data objects whose type is set for each instance. However, these types are not supported because ModelSim converts Verilog modules into VHDL entity declarations (`_primary.vhd`) and supports only those constructs that are currently handled by the VHDL language.

For example:

```
module ma #( parameter p1 = 1, parameter type p2 = shortint) (input logic
[p1:0] i, output logic [p1:0] o);
  p2 j = 0; // type of j is set by a parameter, (shortint unless redefined)
endmodule

module mb;
  logic [3:0] i,o;
  ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule
```

- **Parameters With Expressions As Default Values or No Default Values**

ModelSim provides limited support for parameters that have no default values or have their default values specified in the form of functions or expressions. If the default value expression/function can be evaluated to a constant value by the `vlog` command, that value will be used as the default value of the generic in the VHDL component. Otherwise, if the parameter is defined using Verilog-style declaration, ModelSim dumps it with a 'notype' datatype.

You can leave this type of parameter `OPEN` in entity instantiation, or omit it in component instantiation. However, if you want to override such a parameter, you can do so by applying your own data type and value (component declaration), or by using an unambiguous actual value (direct entity instantiation). If a parameter with no default value or compile-time non-constant default value is defined using SystemVerilog-style declarations, the corresponding generic on the VHDL side will have a data type, but no default value. You can also leave such generics `OPEN` in entity instantiations, or omit them in component instantiations. But if you want to override them from VHDL, you can do so in a way similar to the [Verilog-Style Declarations](#) described above—except that the data type of the overriding VHDL actual must be allowed for mapping with the Verilog formal (refer to [Table 10-10](#) for a list of allowed mappings).

Verilog or SystemVerilog and SystemC Signal Interaction And Mappings

SystemC design units are interconnected by using hierarchical and primitive channels. An `sc_signal<>` is one type of primitive channel. The following section discusses how various SystemC channel types map to Verilog wires when connected to each other across the language boundary.

Channel and Port Type Mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to Verilog modules).

Table 10-11. Channel and Port Type Mapping

Channels	Ports	Verilog mapping
<code>sc_signal<T></code>	<code>sc_in<T></code> <code>sc_out<T></code> <code>sc_inout<T></code>	Depends on type T. See table entitled Data Type Mapping from SystemC to Verilog or SystemVerilog .
<code>sc_signal_rv<W></code>	<code>sc_in_rv<W></code> <code>sc_out_rv<W></code> <code>sc_inout_rv<W></code>	wire [W-1:0]
<code>sc_signal_resolved</code>	<code>sc_in_resolved</code> <code>sc_out_resolved</code> <code>sc_inout_resolved</code>	wire [W-1:0]
<code>sc_clock</code>	<code>sc_in_clk</code> <code>sc_out_clk</code> <code>sc_inout_clk</code>	wire
<code>sc_mutex</code>	N/A	Not supported on language boundary
<code>sc_fifo</code>	<code>sc_fifo_in</code> <code>sc_fifo_out</code>	Not supported on language boundary
<code>sc_semaphore</code>	N/A	Not supported on language boundary
<code>sc_buffer</code>	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary ¹

1. User defined SystemC channels and ports derived from built-in SystemC primitive channels and ports can be connected to HDL signals. The built-in SystemC primitive channel or port must be already supported at the mixed-language boundary for the derived class connection to work.

A SystemC `sc_out` port connected to an HDL signal higher up in the design hierarchy is treated as a pure output port. A `read()` operation on such an `sc_out` port might give incorrect values. Use an `sc_inout` port to do both `read()` and `write()` operations.

Data Type Mapping from SystemC to Verilog or SystemVerilog

Table 10-12 shows the correspondence of SystemC data types to SystemVerilog data types.

Table 10-12. Data Type Mapping – SystemC to Verilog or SystemVerilog

SystemC Type	SystemVerilog Primary Mapping	SystemVerilog Secondary Mapping
enum ¹	enum	-
bool	bit	logic wire
char	byte	bit [7:0] logic [7:0] wire [7:0]
unsigned char	byte unsigned	bit [7:0] logic [7:0] wire [7:0]
short	shortint	bit [15:0] logic [15:0] wire [15:0]
unsigned short	shortint unsigned	bit [15:0] logic [15:0] wire [15:0]
int	int	integer bit [31:0] logic [31:0] wire [31:0]
unsigned int	int unsigned	integer unsigned bit [31:0] logic [31:0] wire [31:0]

Table 10-12. Data Type Mapping – SystemC to Verilog or SystemVerilog

SystemC Type	SystemVerilog Primary Mapping	SystemVerilog Secondary Mapping
long	longint (for 64 bit) int (for 32 bit)	bit [W-1:0] logic [W-1:0] wire [W-1:0], where W=64 on 64-bit W=32 on 32-bit
unsigned long	longint unsigned (64-bit) int unsigned (32-bit)	bit [W-1:0] logic [W-1:0] wire [W-1:0], where W=64 on 64-bit W=32 on 32-bit
long long	longint	bit [63:0] logic [63:0] wire [63:0]
unsigned long long	longint unsigned	bit [63:0] logic [63:0] wire [63:0]
sc_bit	bit	logic wire
sc_logic	logic	bit wire
sc_bv	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_lv	logic [W-1:0]	bit [W-1:0] wire [W-1:0]
float	shortreal	N/A
double	real	N/A
struct ^{2,3}	struct	struct packed
union ²	packed union	N/A
sc_int<W> / sc_signed ²	shortint (if W=16) int (if W=32) longint (if W=64) bit [W-1:0] (otherwise)	logic [W-1:0] wire [W-1:0]
sc_uint<W> / sc_unsigned	shortint unsigned (if W=16) int unsigned (if W=32) longint unsigned (if W=64) bit [W-1:0] (otherwise)	logic [W-1:0] wire [W-1:0]

Table 10-12. Data Type Mapping – SystemC to Verilog or SystemVerilog

SystemC Type	SystemVerilog Primary Mapping	SystemVerilog Secondary Mapping
sc_bigint<W>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_biguint<W>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_fixed<W,I,Q,O,N> sc_ufixed<W,I,Q,O,N>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_fixed_fast<W,I,Q,O,N> sc_ufixed_fast<W,I,Q,O,N>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_fix sc_ufix	bit [WL-1:0] ⁴	logic [W-1:0] wire [W-1:0]
sc_fix_fast sc_ufix_fast	bit [WL-1:0]	logic [W-1:0] wire [W-1:0]

1. Refer to [enum, struct, and union at SC-SV Mixed-Language Boundary](#) for more information on these complex types.
2. To make a port of type sc_signed or sc_unsigned of word length other than the default (32), you must use sc_length_param and sc_length_context to set the word length. For more information, see [Construction Parameters for SystemC Types](#).
3. Supports real and shortreal as field types.
4. WL (word length) is the total number of bits used in the type. It is specified during runtime. To make a port of type sc_fix, sc_ufix, sc_fix_fast, or sc_ufix_fast of word length other than the default(32), you must use sc_fxtype_params and sc_fxtype_context to set the word length. For more information, see [Construction Parameters for SystemC Types](#).

Data Type Mapping from Verilog or SystemVerilog to SystemC

Table 10-13 shows the correspondence of Verilog/SystemVerilog data types to SystemC data types.

Table 10-13. Data Type Mapping – Verilog or SystemVerilog to SystemC

Verilog/ SystemVerilog Type	SystemC Primary Mapping	SystemC Secondary Mapping
bit	bool	sc_bit sc_logic
logic	sc_logic	sc_bit bool
reg	sc_logic	sc_bit bool

Table 10-13. Data Type Mapping – Verilog or SystemVerilog to SystemC

Verilog/ SystemVerilog Type	SystemC Primary Mapping	SystemC Secondary Mapping
bit vector	sc_bv<W>	sc_lv<W> sc_int<W> sc_uint<W>
logic vector	sc_lv<W>	sc_bv<W> sc_int<W> sc_uint<W>
reg vector	sc_lv<W>	sc_bv<W> sc_int<W> sc_uint<W>
wire vector	sc_lv<W>	sc_bv<W> sc_int<W> sc_uint<W>
wire	sc_logic	sc_bit bool
integer	sc_lv<32>	int sc_int<32>
integer unsigned	sc_lv<32>	unsigned int sc_uint<32> sc_bv<32>
int	int	sc_lv<32> sc_int<32> sc_bv<32>
shortint	short	sc_lv<16> sc_int<16> sc_bv<16>
longint	long long	sc_lv<64> sc_int<64> sc_bv<64> long (for 64-bit)
longint unsigned	unsigned long long	sc_lv<64> sc_uint<64> sc_bv<64> unsigned long (for 64-bit)
byte	char	sc_lv<8> sc_int<8> sc_bv<8>
byte unsigned	unsigned char	sc_lv<8> sc_uint<8> sc_bv<8>

Table 10-13. Data Type Mapping – Verilog or SystemVerilog to SystemC

Verilog/ SystemVerilog Type	SystemC Primary Mapping	SystemC Secondary Mapping
enum ¹	enum	-
struct ¹	struct	-
packed struct	struct	
packed union ^{1, 2}	union	-
real ²	double	-
shortreal ²	float	-
multi-D array ^{2, 3}	multi-D array	-

1. Refer to [enum, struct, and union at SC-SV Mixed-Language Boundary](#) for more information on these complex types.

2. Unpacked and tagged unions are not supported at the SC-SV mixed language boundary.

3. Classes, multi-dimensional arrays, unpacked/tagged unions, strings and handles are not supported for SystemC control/observe.

enum, struct, and union at SC-SV Mixed-Language Boundary

The following guidelines apply to the use of enumerations, structures and unions at the SystemC/SystemVerilog mixed language boundary.

Enumerations

A SystemVerilog enum may be used at the SystemC - SystemVerilog language boundary if it meets the following criteria:

- Base type of the SystemVerilog enum must be int (32-bit 2-state integer).
- The value of enum elements are not ambiguous and are equal to the value of the corresponding value of enum elements on the SystemC side.

Enums with different strings are allowed at the language boundary as long as the values on both sides are identical.

- SystemVerilog enums with 'range of enumeration elements' are allowed provided the corresponding enum is correctly defined (manually) on the SystemC side.

Unions and Structures

You can use a SystemVerilog union or structure at a SystemC-SystemVerilog boundary if it meets the following criteria:

- The type of all elements of the union/structure is one of the supported types.

- The type of the corresponding elements of the SystemC union/structure follow the supported type mapping for variable ports on the SC-SV language boundary. See [Channel and Port Type Mapping](#) for mapping information.
- The number and order of elements in the definition of structures on SystemVerilog and SystemC side is the same.

For unions, the order of elements may be different, but the number of elements must be the same.

- Union must be packed and untagged. While both packed and unpacked structures are supported, only packed unions are supported at the SystemC-SystemVerilog language boundary.

Port Direction

Verilog port directions are mapped to SystemC as shown in [Table 10-14](#). Note that you can use the wildcard syntax convention (.*) when instantiating Verilog ports where the instance port name matches the connecting port name and their data types are equivalent.

Table 10-14. Mapping Verilog Port Directions to SystemC

Verilog	SystemC
input	sc_in<T> sc_in_resolved sc_in_rv<W>
output	sc_out<T> sc_out_resolved sc_out_rv<W>
inout	sc_inout<T> sc_inout_resolved sc_inout_rv<W>

Verilog to SystemC State Mappings

Verilog states are mapped to sc_logic, sc_bit, and bool as shown in [Table 10-15](#).

Table 10-15. Mapping Verilog States to SystemC States

Verilog	sc_logic	sc_bit	bool
HiZ	'Z'	'0'	false
Sm0	'0'	'0'	false
Sm1	'1'	'1'	true
SmX	'X'	'0'	false
Me0	'0'	'0'	false

Table 10-15. Mapping Verilog States to SystemC States

Verilog	sc_logic	sc_bit	bool
Me1	'1'	'1'	true
MeX	'X'	'0'	false
We0	'0'	'0'	false
We1	'1'	'1'	true
WeX	'X'	'0'	false
La0	'0'	'0'	false
La1	'1'	'1'	true
LaX	'X'	'0'	false
Pu0	'0'	'0'	false
Pu1	'1'	'1'	true
PuX	'X'	'0'	false
St0	'0'	'0'	false
St1	'1'	'1'	true
StX	'X'	'0'	false
Su0	'0'	'0'	false
Su1	'1'	'1'	true
SuX	'X'	'0'	false

For Verilog states with ambiguous strength:

- sc_bit receives '1' if the value component is 1, else it receives '0'
- bool receives true if the value component is 1, else it receives false
- sc_logic receives 'X' if the value component is X, H, or L
- sc_logic receives '0' if the value component is 0
- sc_logic receives '1' if the value component is 1

SystemC to Verilog State Mappings

SystemC type bool is mapped to Verilog states as shown in [Table 10-16](#):

Table 10-16. Mapping SystemC bool to Verilog States

bool	Verilog
false	St0

Table 10-16. Mapping SystemC bool to Verilog States

bool	Verilog
true	St1

SystemC type `sc_bit` is mapped to Verilog states as shown in [Table 10-17](#):

Table 10-17. Mapping SystemC sc_bit to Verilog States

sc_bit	Verilog
'0'	St0
'1'	St1

SystemC type `sc_logic` is mapped to Verilog states as shown in [Table 10-18](#):

Table 10-18. Mapping SystemC sc_logic to Verilog States

sc_logic	Verilog
'0'	St0
'1'	St1
'Z'	HiZ
'X'	StX

VHDL and SystemC Signal Interaction And Mappings

SystemC has a more complex signal-level interconnect scheme than VHDL. Design units are interconnected with hierarchical and primitive channels. An `sc_signal<T>` is one type of primitive channel. The following section discusses how various SystemC channel types map to VHDL types when connected to each other across the language boundary.

Port Type Mapping

[Table 10-19](#) lists port type mappings for all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to VHDL modules).

Table 10-19. SystemC Port Type Mapping

Channels	Ports	VHDL mapping
<code>sc_signal<T></code>	<code>sc_in<T></code> <code>sc_out<T></code> <code>sc_inout<T></code>	Depends on type T. See table entitled Data Type Mapping Between SystemC and VHDL .

Table 10-19. SystemC Port Type Mapping

Channels	Ports	VHDL mapping
sc_signal_rv<W>	sc_in_rv<W> sc_out_rv<W> sc_inout_rv<W>	std_logic_vector(W-1 downto 0)
sc_signal_resolved	sc_in_resolved sc_out_resolved sc_inout_resolved	std_logic
sc_clock	sc_in_clk sc_out_clk sc_inout_clk	bit/std_logic/boolean
sc_mutex	N/A	Not supported on language boundary
sc_fifo	sc_fifo_in sc_fifo_out	Not supported on language boundary
sc_semaphore	N/A	Not supported on language boundary
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary ¹

1. User defined SystemC channels and ports derived from built-in SystemC primitive channels and ports can be connected to HDL signals. The built-in SystemC primitive channel or port must be already supported at the mixed-language boundary for the derived class connection to work.

A SystemC sc_out port connected to an HDL signal higher up in the design hierarchy is treated as a pure output port. A read() operation on such an sc_out port might give incorrect values. Use an sc_inout port to do both read() and write() operations.

Data Type Mapping Between SystemC and VHDL

Table 10-20 lists the mapping between SystemC sc_signal types and VHDL types.

Table 10-20. Mapping Between SystemC sc_signal and VHDL Types

SystemC	VHDL
bool, sc_bit	bit/std_logic/boolean
sc_logic	std_logic
sc_bv<W>	bit_vector(W-1 downto 0)
sc_lv<W>	std_logic_vector(W-1 downto 0)

Table 10-20. Mapping Between SystemC sc_signal and VHDL Types (cont.)

SystemC	VHDL
sc_bv<32>, sc_lv<32>	integer
sc_bv<64>, sc_lv<64>	real
sc_int<W>, sc_uint<W>	bit_vector(W-1 downto 0) std_logic_vector(W -1 downto 0)
sc_bigint<W>, sc_biguint<W>	bit_vector(W-1 downto 0) std_logic_vector(W-1 downto 0)
sc_fixed<W,I,Q,O,N>, sc_ufixed<W,I,Q,O,N>	bit_vector(W-1 downto 0) std_logic_vector(W-1 downto 0)
sc_fixed_fast<W,I,Q,O,N>, sc_ufixed_fast<W,I,Q,O,N>	bit_vector(W-1 downto 0) std_logic_vector(W-1 downto 0)
¹ sc_fix, ¹ sc_ufix	bit_vector(WL-1 downto 0) std_logic_vector(WL- 1 downto 0)
¹ sc_fix_fast, ¹ sc_ufix_fast	bit_vector(WL-1 downto 0) std_logic_vector(WL- 1 downto 0)
² sc_signed, ² sc_unsigned	bit_vector(WL-1 downto 0) std_logic_vector(WL- 1 downto 0)
char, unsigned char	bit_vector(7 downto 0) std_logic_vector(7 downto 0)
short, unsigned short	bit_vector(15 downto 0) std_logic_vector(15 downto 0)
int, unsigned int	bit_vector(31 downto 0) std_logic_vector(7 downto 0)
long, unsigned long	bit_vector(31 downto 0) std_logic_vector(31 downto 0)
long long, unsigned long long	bit_vector(63 downto 0) std_logic_vector(63 downto 0)
float	bit_vector(31 downto 0) std_logic_vector(31 downto 0)
double	bit_vector(63 downto 0) std_logic_vector(63 downto 0) real
struct	record

Table 10-20. Mapping Between SystemC `sc_signal` and VHDL Types (cont.)

SystemC	VHDL
enum	enum
record ³	record element_declaration {element_declaration} end record [record_type_simple_name]
signal array ⁴	type signal_name array (constraint_definition) of signal_type
Not supported on language boundary (no equivalent SystemC type)	multi-D array
pointer	Not supported on language boundary (no equivalent VHDL type)
class	Not supported on language boundary (no equivalent VHDL type)
union	Not supported on language boundary (no equivalent VHDL type)
bit_fields	Not supported on language boundary (no equivalent VHDL type)
Not supported on language boundary (no equivalent SystemC type)	access
Not supported on language boundary (no equivalent SystemC type)	protected

1. WL (word length) is the total number of bits used in the type. It is specified during runtime. To make a port of type `sc_fix`, `sc_ufix`, `sc_fix_fast`, or `sc_ufix_fast` of word length other than the default(32), you must use `sc_fxtype_params` and `sc_fxtype_context` to set the word length. For more information, see [Construction Parameters for SystemC Types](#).

2. To make a port of type `sc_signed` or `sc_unsigned` of word length other than the default (32), you must use `sc_length_param` and `sc_length_context` to set the word length. For more information, see [Construction Parameters for SystemC Types](#).

3. Including nested records.

4. SystemC signal arrays are supported only for cases where VHDL instantiates a SystemC module—not vice versa.

Type Checking—Records

Two records at the SystemC-VHDL mixed-language boundary will be equivalent if all of the following conditions hold true for them:

- The number and order of elements in the definition of records on VHDL and SystemC side is the same.
- Size of each field of one record is exactly same as the size of the corresponding field in the second record.
- Type of each field of both the records is supported at the SystemC-VHDL boundary.
- Mapping between corresponding field types is permitted at the SystemC-VHDL boundary.

Type Checking—Enums

Two enumerated types at the SystemC-VHDL mixed-language boundary will be equivalent if all of the following conditions hold true for them:

- The number of elements of both enums is the same.
- The element values of both enums is the same. SystemC allows enums to have noncontinuous enum values, but VHDL allows only consecutive enum values (starting from 0) for enums. As such, this check limits the element values of SystemC enums to be consecutive integers starting from 0.
- A warning message will occur if the enum labels (enum strings) for both the enums at the SystemC-VHDL boundary are different but their values are the same.

Type Checking—Signal Arrays

SystemC signal arrays can be connected to VHDL array only if both the following conditions hold true for them:

- The number of elements in the SystemC signal array and the VHDL array is the same.
- Mapping between the type of SystemC signal array and the type of the element of the VHDL array is permitted at the SystemC-VHDL boundary.

Note



SystemC signal arrays are supported only for cases where VHDL instantiates a SystemC module—not vice versa.

Port Direction Mapping

VHDL port directions are mapped to SystemC as shown in [Table 10-21](#):

Table 10-21. Mapping VHDL Port Directions to SystemC

VHDL	SystemC
in	sc_in<T>, sc_in_resolved, sc_in_rv<W>
out	sc_out<T>, sc_out_resolved, sc_out_rv<W>
inout	sc_inout<T>, sc_inout_resolved, sc_inout_rv<W>
buffer	sc_out<T>, sc_out_resolved, sc_out_rv<W>

Note



VHDL constants are supported for port connections at a VHDL-SystemC boundary.

VHDL to SystemC State Mapping

VHDL states are mapped to sc_logic, sc_bit, and bool as shown in [Table 10-22](#):

Table 10-22. Mapping VHDL std_logic States to SystemC States

std_logic	sc_logic	sc_bit	bool
'U'	'X'	'0'	false
'X'	'X'	'0'	false
'0'	'0'	'0'	false
'1'	'1'	'1'	true
'Z'	'Z'	'0'	false
'W'	'X'	'0'	false
'L'	'0'	'0'	false
'H'	'1'	'1'	true
'-'	'X'	'0'	false

SystemC to VHDL State Mapping

SystemC type `bool` is mapped to VHDL boolean as shown in [Table 10-23](#):

Table 10-23. Mapping SystemC `bool` to VHDL Boolean States

bool	VHDL
false	false
true	true

SystemC type `sc_bit` is mapped to VHDL bit as shown in [Table 10-24](#):

Table 10-24. Mapping SystemC `sc_bit` to VHDL bit

sc_bit	VHDL
'0'	'0'
'1'	'1'

SystemC type `sc_logic` is mapped to VHDL `std_logic` states as shown in [Table 10-25](#):

Table 10-25. Mapping SystemC `sc_logic` to VHDL `std_logic`

sc_logic	std_logic
'0'	'0'
'1'	'1'
'Z'	'Z'
'X'	'X'

VHDL Instantiating Verilog or SystemVerilog

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. You can reference a Verilog module in the entity aspect of a component configuration—all you need to do is specify a module name instead of an entity name. You can also specify an optional secondary name for an optimized sub-module.

Further, you can reference a Verilog configuration in the configuration aspect of a VHDL component configuration—just specify a Verilog configuration name instead of a VHDL configuration name.

Verilog/SystemVerilog Instantiation Criteria Within VHDL

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

- The design unit is a module or configuration. UDPs are not allowed.
- The ports are named ports of type: reg, logic, bit, one-dimensional arrays of reg/logic/bit, integer, int, shortint, longint, byte, integer unsigned, int unsigned, shortint unsigned, longint unsigned, byte unsigned. (See also, [Modules with Unnamed Ports](#)).

Component Declaration for VHDL Instantiating Verilog

A Verilog module that is compiled into a library can be referenced from a VHDL design as though the module is a VHDL entity. Likewise, a Verilog configuration can be referenced as though it were a VHDL configuration.

You can extract the interface to the module from the library in the form of a component declaration by running [vgencomp](#). Given a library and module name, the vgencomp command writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose one of the following:

- bit and bit_vector
- vl_logic and vl_logic_vector

VHDL and Verilog Identifiers

The VHDL identifiers for the component name, port names, and generic names are the same as Verilog and SystemVerilog identifiers for the module name, port names, and parameter names. Except for the cases noted below, ModelSim does nothing to the Verilog identifier when it generates the entity.

ModelSim converts Verilog identifiers to VHDL 1076-1993 extended identifiers in three cases:

- The Verilog identifier is not a valid VHDL 1076-1987 identifier.
- You compile the Verilog module with the **-93** argument. One exception is a valid, lowercase identifier (for instance, topmod). Valid, lowercase identifiers will not be converted even if you compile with **-93**.
- The Verilog identifier is not unique when case is ignored. For example, if you have TopMod and topmod in the same module, ModelSim will convert the former to \TopMod\.

vgencomp Component Declaration when VHDL Instantiates Verilog

`vgencomp` 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. For example:

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. For example:

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 [W-1:0] p4;	p4 : inout std_logic_vector;

Configuration declarations are allowed to reference Verilog modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a Verilog instance to configure the instantiations within the Verilog module.

Modules with Bidirectional Pass Switches

Modules that have bidirectional pass switches (tran primitives) internally connected to their ports are not fully supported when the module is instantiated by VHDL. This is due to limitations imposed by the requirements of VHDL signal resolution.

However, full bidirectional operation is supported if the following requirements are met:

- The Verilog port is declared with mode inout.
- The connected VHDL signal is of type or subtype `std_logic`.
- The connected port hierarchy above the VHDL signal does not cross any other mixed language boundaries, and the top-level signal is also of type or subtype `std_logic`.

In all other cases, the following warning is issued at elaboration and the simulation of the Verilog port may produce incorrect results if the design actually drives in both directions across the port:

**** Warning: (vsim-3011) testfile(4): [TRAN] - Verilog net 'n' with bidirectional tran primitives might not function correctly when connected to a VHDL signal.**

If you use the port solely in a unidirectional manner, then you should explicitly declare it as either input or output (whichever matches the direction of the signal flow).

Modules with Unnamed Ports

Verilog allows modules to have unnamed ports, whereas VHDL requires that all ports have names. If any of the Verilog ports are unnamed, then all are considered to be unnamed, and it is not possible to create a matching VHDL component. In such cases, the module may not be instantiated from VHDL.

Unnamed ports occur when the module port list contains bit-selects, part-selects, or concatenations, as in the following example:

```
module m(a[3:0], b[1], b[0], {c,d});
  input [3:0] a;
  input [1:0] b;
  input c, d;
endmodule
```

Note that `a[3:0]` is considered to be unnamed even though it is a full part-select. A common mistake is to include the vector bounds in the port list, which has the undesired side effect of making the ports unnamed (which prevents you from connecting by name even in an all-Verilog design).

Most modules having unnamed ports can be easily rewritten to explicitly name the ports, thus allowing the module to be instantiated from VHDL. Consider the following example:

```
module m(y[1], y[0], a[1], a[0]);
  output [1:0] y;
  input [1:0] a;
endmodule
```

Here is the same module rewritten with explicit port names added:

```
module m(.y1(y[1]), .y0(y[0]), .a1(a[1]), .a0(a[0]));
  output [1:0] y;
  input [1:0] a;
endmodule
```

Empty Ports

Verilog modules may have "empty" ports, which are also unnamed, but they are treated differently from other unnamed ports. If the only unnamed ports are empty, then the other ports may still be connected to by name, as in the following example:

```
module m(a, , b);
  input a, b;
endmodule
```

Although this module has an empty port between ports a and b, the named ports in the module can still be connected to or from VHDL.

Verilog or SystemVerilog Instantiating VHDL

You can reference a VHDL entity or configuration from Verilog or SystemVerilog as though the design unit is a module or a configuration of the same name.

VHDL Instantiation Criteria Within Verilog

You can instantiate a VHDL design unit within Verilog or SystemVerilog if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type: bit, bit_vector, enum, integer, natural, positive, real, shortreal, std_logic, std_ulogic, std_logic_vector, std_ulogic_vector, vl_ulogic, vl_ulogic_vector, or their subtypes; unconstrained arrays; nested records; and records with fields of type integer, real, enum, and multi-dimensional arrays.

The port clause may have any mix of these types. Multi-dimensional arrays of these support types are also supported.

- The generics are of type bit, bit_vector, integer, real, std_logic, std_logic_vector, vl_logic, vl_logic_vector, time, physical, enumeration, or string.

String is the only composite type allowed.

Usage Notes

Passing a parameter values from Verilog or SystemVerilog to a VHDL generic of type `std_logic` is slightly different than other VHDL types. Note that `std_logic` is defined as a 9-state enumerated type, as follows:

```
TYPE std_ulogic IS ( 'U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-' -- Don't care
                  );
```

To be able to correctly set the VHDL generic to any of the nine states, you must set the value in the Verilog instance to the element (positional) value in the `std_logic` enum that corresponds to the `std_logic` value (that is, the position not the value itself). For example, to set the generic to a 'U', use `1'b0`, to set it to an "X", use `1'b1`, to set it to '0', use `2'b10`.

Note that this only applies to `std_logic` types—for `std_logic_vector` you can simply pass the value as you would normally expect.

For example, the following VHDL entity shows the generics of type `std_logic`:

```
entity ent is
  generic (
    a : std_logic;
    b : std_logic ;
    c : std_logic
  ) ;
```

with the following Verilog instantiation:

```
module test ;
  // here we will pass 0 to a, 1 to b and z to c
  ent #(2'b10, 2'b11, 3'b100) u_ent ()
endmodule
```

Note that this does not pass the value but the positional number corresponding to the element value in the `std_logic` enum.

Alternatively, you can use `std_logic_vector` for the generics, and you can simply pass the value as normal.

Entity and Architecture Names and Escaped Identifiers

An entity name is not case-sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise. Since instantiation bindings are not determined at compile time in Verilog, you must instruct the simulator to search your libraries when loading the design. See [Library Usage](#) for more information.

Alternatively, you can employ the escaped identifier to provide an extended form of instantiation:

```
\mylib.entity(arch) u1 (a, b, c) ;  
\mylib.entity u1 (a, b, c) ;  
\entity(arch) u1 (a, b, c) ;
```

If the escaped identifier takes the form of one of the above and is not the name of a design unit in the work library, then the instantiation is broken down as follows:

- library = mylib
- design unit = entity
- architecture = arch

Named Port Associations

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

Named port associations are not case sensitive unless a VHDL port name is an extended identifier (1076-1993). If the VHDL port name is an extended identifier, the association is case-sensitive, and the leading and trailing backslashes of the VHDL identifier are removed before comparison.

Generic Associations

Generic associations are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. Parameter assignment to generics is not case sensitive.

The **defparam** statement is not allowed for setting generic values.

SDF Annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See [SDF for Mixed VHDL and Verilog Designs](#) for more information.

Sharing User-Defined Types

Using a Common VHDL Package

With the “import” construct of SystemVerilog, you can implement user-defined types (records, enums, alias, subtypes, types, and multi-dimensional arrays) and constants from VHDL in a SystemVerilog design. (Anything other than these types is not supported.) For example:

```
import vh_pack::vh_type
```

Because VHDL is case-insensitive, design units, variables and constants will be converted to lower-case.

If you use mixed-case identifiers with its original case in your SystemVerilog code, design compilation will fail because SystemVerilog is case sensitive. For example, if your VHDL package contains an identifier named *myPacketData* the compiler will convert it to *mypacketdata*. Therefore, if you use *myPacketData* in your SystemVerilog code, compilation would fail due to a case mismatch. Because of this, it is suggested that everything in the shared package should be lower-case to avoid these mismatch issues.

In order to import a VHDL package into SystemVerilog, you must compile it using the **-mixedsvvh** argument with the **vcom** command (refer to [Usage Notes](#), below).

Note



The following types must be defined in a common package if you want to use them at the SystemVerilog-VHDL boundary:

- Records
- Enumerations
- One-dimensional array of bit, std_logic, std_ulogic, integer, natural, positive, real & time
- Multi-dimensional arrays and array of arrays of all supported types
- Subtypes of all supported types
- Alias of records, enums and arrays only
- Types (static ranges only)

Also, be sure to use vcom -mixedsvvh when compiling the common package.

ModelSim supports VHDL constants of all types currently supported at the VHDL-SystemVerilog mixed language boundary as shown in [Table 10-5](#).

Deferred constants are not supported. Only static expressions are supported as constant values.

Table 10-26 shows the mapping of literals from VHDL to SystemVerilog.

Table 10-26. Mapping Literals from VHDL to SystemVerilog

VHDL	SystemVerilog
'0' (Forcing 0)	'0'
'L' (Weak 0)	'0'
'1' (Forcing 1)	'1'
'H' (Weak 1)	'1'
'U' (Uninitialized)	'X'
'X' (Forcing Unknown)	'X'
'W' (Weak Unknown)	'X'
'-' (Don't care)	'X'
'Z' (High Impedance)	'Z'

Table 10-27 lists all supported types inside VHDL Records.

Table 10-27. Supported Types Inside VHDL Records

VHDL Type	SystemVerilog Type
bit, boolean	bit
std_logic	logic
std_ulogic	logic
bit_vector	bit vector
std_logic_vector, std_ulogic_vector, signed, unsigned	logic vector
integer, natural, positive	int
real	real
record	structure
multi-D arrays, array of arrays	multi-d arrays

Usage Notes

When using a common VHDL package at a SystemVerilog-VHDL boundary, compile the VHDL package with the **-mixedsvvh** argument with the **vcom** command, as follows:

```
vcom -mixedsvvh [[b | l | r] | i] <vhdl_package>
```

where

- b — treats all scalars and vectors in the package as SystemVerilog bit type
- l — treats all scalars and vectors in the package as SystemVerilog logic type
- r — treats all scalars and vectors in the package as SystemVerilog reg type
- i — ignores the range specified with VHDL integer types

When you compile a VHDL package with **-mixedsvvh**, the package can be included in a SystemVerilog design as if it were defined in SystemVerilog itself.

Note

If you do not specify b, l, or r with **-mixedsvvh**, then the default treatment of data types is applied:

- VHDL `bit_vector` is treated as SystemVerilog bit vector and
 - VHDL `std_logic_vector`, `std_ulogic_vector`, and `vl_logic_vector` are treated as SystemVerilog logic vectors
-

Example

Consider the following VHDL package that you want to use at a SystemVerilog-VHDL boundary:

```
--/*-----pack.vhd-----*/
package pack is
  type st_pack is record
    a: bit_vector (3 downto 0);
    b: bit;
    c: integer;
  end record;
  constant c : st_pack := (a=>"0110", b=>'0', c=>4);
end package;
```

You must compile this package with the **-mixedsvvh** argument for **vcom**:

```
vcom -mixedsvvh pack.vhd
```

This package can now be imported in the SystemVerilog design, as if it were a SystemVerilog package.

```
--/*-----VHDL_entity-----*/
use work.pack.all;
entity top is
end entity;
architecture arch of top is
  component bot
    port (in1 : in st_pack;
          in2 : bit_vector(1 to c.c);
          out1 : out st_pack);
  end component;
```

```
begin
end arch;

/*-----SV_file-----*/
import pack::*; // including the VHDL package in SV
module bot(input st_pack in1, input bit [1:c.c] in2, output st_pack out1);
endmodule
```

Using a Common SystemVerilog Package

With the “use” construct of VHDL, you can implement user-defined types (structures, enums, and multi-dimensional arrays) from SystemVerilog in a VHDL design. (Anything other than these types is not supported.) For example:

```
use work.sv_pack.sv_type
```

In order to include a SystemVerilog package in VHDL, you must compile it using the **-mixedsvvh** argument of the **vlog** command (refer to [Usage Notes](#), below).

Note



The following types must be defined in a common package if you want to use them at the SystemVerilog-VHDL boundary:

- Structures
- Enumerations with base type as 32-bit 2-state integer
- Multi-dimensional arrays of all supported types

Also, be sure to use **vcom -mixedsvvh** when compiling the common package.

[Table 10-28](#) lists all supported types inside SystemVerilog structures.

Table 10-28. Supported Types Inside SystemVerilog Structure

SystemVerilog Type	VHDL Type	Comments
bit	bit	bit types
logic, reg	std_logic	multi-valued types
enum	enum	SV enums of only 2-state int base type supported
struct	record	unpacked structure
packed struct	record	packed structure
real	real	2-state data type, 64-bit real number
shortreal	real	2-state data type, 32-bit real number
multi-D arrays	multi-D arrays	multi-dimensional arrays of supported types
byte, int, shortint, longint	integer	integer types

Usage Notes

When using a common SystemVerilog package at a SV-VHDL boundary, you should compile the SystemVerilog package with the **-mixedsvvh** argument of the **vlog** command, as follows:

```
vlog -mixedsvvh [b | s | v] <sv_package>
```

where

b treats all scalars/vectors in the package as VHDL bit/bit_vector

s treats all scalars/vectors in the package as VHDL std_logic/std_logic_vector

v treats all scalars/vectors in the package as VHDL vl_logic/vl_logic_vector

When you compile a SystemVerilog package with **-mixedsvvh**, the package can be included in a VHDL design as if it were defined in VHDL itself.

Note



If you do not specify **b**, **s**, or **v** with **-mixedsvvh**, the default treatment of data types is applied.

Example

The following SystemVerilog package contains a type named `st_pack`, which you want to use at the SV-VHDL mixed-language boundary.

```
/*-----pack.sv-----*/
package pack;
  typedef struct {
    bit [3:0] a;
    bit b;
  } st_pack;
endpackage
```

To use this package (and type) at a SystemVerilog-VHDL boundary, you must compile it using **vlog -mixedsvvh**:

```
vlog -mixedsvvh pack.sv
```

You can now include this package (`st_pack`) in the VHDL design, as if it were a VHDL package:

```
--/*-----VHDL_file-----*/
use work.pack.all; -- including the SV package in VHDL

entity top is
end entity;

architecture arch of top is
  component bot
  port(
    in1 : in st_pack; -- using type from the SV package.
```

```
        out1 : out st_pack);  
    end component;  
  
    signal sin1, sout1 : st_pack;  
begin  
    ...  
end arch;  
  
/*-----SV Module-----*/  
import pack::*;  
  
module bot(input st_pack in1, output st_pack out1);  
    ...  
endmodule
```

SystemC Instantiating Verilog or SystemVerilog

To instantiate Verilog or SystemVerilog modules into a SystemC design, you must first create a [SystemC Foreign Module \(Verilog\) Declaration](#) for each Verilog/SystemVerilog module. Once you have created the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

Verilog Instantiation Criteria Within SystemC

A Verilog/SystemVerilog design unit may be instantiated within SystemC if it meets the following criteria:

- The design unit is a module (UDPs and Verilog primitives are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- The Verilog/SystemVerilog module name must be a valid C++ identifier.
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in SystemC).

A Verilog/SystemVerilog module that is compiled into a library can be instantiated in a SystemC design as though the module were a SystemC module by passing the Verilog/SystemVerilog module name to the foreign module constructor. For an illustration of this, see [Example 10-2](#).

SystemC and Verilog Identifiers

The SystemC identifiers for the module name and port names are the same as the Verilog identifiers for the module name and port names. Verilog identifiers must be valid C++ identifiers. SystemC and Verilog are both case-sensitive. ModelSim does nothing to the SystemC identifiers when it generates the module.

Verilog Configuration Support

You can use a Verilog configuration to configure a Verilog module instantiated in SystemC. The Verilog configuration must be elaborated (with `vsim` or `vopt`) as a top-level design unit, or be part of another top-level VHDL or Verilog configuration.

SystemC Foreign Module (Verilog) Declaration

In cases where you want to run a mixed simulation with SystemC and Verilog/SystemVerilog, you must generate and declare a foreign module that stands in for each Verilog module instantiated under SystemC. You can create foreign modules in one of two ways:

- Run **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration).
- Modify your SystemC source code manually.

After you have analyzed the design, you can generate a foreign module declaration by using [scgenmod](#) as follows:

```
scgenmod mod1
```

where *mod1* can be any name of a Verilog module. A foreign module declaration for the specified module is written to stdout.

Guidelines for Manual Creation of Foreign Module Declaration

Apply the following guidelines to the creation of foreign modules. A foreign module:

- Contains ports corresponding to Verilog ports. These ports must be explicitly named in the constructor initializer list of the foreign module.
- Must not contain any internal design elements such as child instances, primitive channels, or processes.
- Must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For Verilog, the HDL name is simply the Verilog module name corresponding to the foreign module, or `[<lib>].<module>`.
- Allows inclusion of parameterized modules. Refer to [Parameter Support for SystemC Instantiating Verilog](#) for details.

Example 10-2. SystemC Instantiating Verilog - 1

A sample Verilog module to be instantiated in a SystemC design is:

```
module vcounter (clock, topcount, count);  
    input clock;  
    input topcount;  
    output count;
```

```
    reg count;
    ...
endmodule
```

The SystemC foreign module declaration for the above Verilog module is:

```
class counter : public sc_foreign_module {
public:
    sc_in<bool> clock;
    sc_in<sc_logic> topcount;
    sc_out<sc_logic> count;
    counter(sc_module_name nm)
        : sc_foreign_module(nm, "lib.vcounter"),
          clock("clock"),
          topcount("topcount"),
          count("count")
    {}
};
```

The Verilog module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the instance name of the Verilog module.

Example 10-3. SystemC Instantiating Verilog - 2

Another variation of the SystemC foreign module declaration for the same Verilog module might be:

```
class counter : public sc_foreign_module {
public:
    ...
    counter(sc_module_name nm, char* hdl_name)
        : sc_foreign_module(nm, hdl_name),
          clock("clock"),
          ...
    {}
};
```

The instantiation of this module would be:

```
counter dut("dut", "lib.counter");
```

Parameter Support for SystemC Instantiating Verilog

Since the SystemC language has no concept of parameters, parameterized values must be passed from a SystemC parent to a Verilog child through the SystemC foreign module (`sc_foreign_module`). See [SystemC Foreign Module \(Verilog\) Declaration](#) for information regarding the creation of `sc_foreign_module`.

Passing Parameters to `sc_foreign_module` (Verilog)

To instantiate a Verilog module containing parameterized values into the SystemC design, you can use one of two methods, depending on whether the parameter is an integer. If the parameter is an integer, you have two choices: passing as a template argument to the foreign module or as a constructor argument to the foreign module. Non-integer parameters must be passed to the foreign module using constructor arguments.

Passing Integer and Non-Integer Parameters as Constructor Arguments

Both integer and non-integer parameters can be passed by specifying two parameters to the `sc_foreign_module` constructor: the number of parameters (`int num_generics`), and the parameter list (`const char* generic_list`). The `generic_list` is listed as an array of `const char*`.

If you create your foreign module manually (see [Guidelines for Manual Creation of Foreign Module Declaration](#)), you must also pass the parameter information to the `sc_foreign_module` constructor. If you use `scgenmod` to create the foreign module declaration, the parameter information is detected in the HDL child and is incorporated automatically.

Example 10-4. Sample Foreign Module Declaration, with Constructor Arguments for Parameters

Following [Example 10-2](#), the following parameter information would be passed to the SystemC foreign module declaration:

```
class counter : public sc_foreign_module {
public:
    sc_in<bool> clk;
    ...
counter(sc_module_name nm, char* hdl_name
        int num_generics, const char** generic_list)
    : sc_foreign_module (nm),
      {elaborate_foreign_module(hdl_name, num_generics, generic_list);}
};
```

Example 10-5. Passing Parameters as Constructor Arguments - 1

Verilog module:

```
module counter (clk, count)

    parameter integer_param = 4;
    parameter real_param = 2.9;
    parameter str_param = "ERROR";

    output [7:0] count;
    input clk;

    ...
```

```
endmodule
```

Foreign module (created by the command: **scgenmod counter**):

```
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<8> > count;

    counter(sc_module_name nm, const char* hdl_name
            int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~counter()
    {}
};
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {
    counter* counter_inst_1; // Instantiate counter with counter_size = 20

    SC_CTOR(top)
    {
        const char* generic_list[3];
        generic_list[0] = strdup("integer_param=16");
        generic_list[1] = strdup("real_param=2.6");
        generic_list[2] = strdup("str_param=\"Hello\"");

        //Pass all parameter overrides using foreign module constructor args
        counter_inst_1 = new counter("c_inst", "work.counter", 3, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 3; i++;)
            free((char*)generic_list[i]);
    }
};
```

Passing Integer Parameters as Template Arguments

Integer parameters can be passed as template arguments to a foreign module. Doing so enables port sizes of Verilog modules to be configured using the integer template arguments. Use the `-createtemplate` option to `scgenmod` to generate a class template foreign module.

Example 10-6. SystemC Instantiating Verilog, Passing Integer Parameters as Template Arguments

Verilog module:

```
module counter (clk, count)

    parameter counter_size = 4;

    output [counter_size - 1 : 0] count;
    input clk;

    ...

endmodule
```

Foreign module (created by the command: **scgenmod -createtemplate counter**):

```
template <int counter_size = 4>
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        this->add_parameter("counter_size", counter_size);
        elaborate_foreign_module(hdl_name);
    }
    ~counter()
    {}

};
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {

    counter<20> counter_inst_1;
        // Instantiates counter with counter_size = 20
    counter    counter_inst_2;
        // Instantiates counter with default counter_size = 4

    SC_CTOR(top)
        : counter_inst_1(cinst_1, "work.counter"),
          counter_inst_2(cinst_2, "work.counter")
    {}

};
```

Example 10-7. Passing Integer Parameters as Template Arguments and Non-integer Parameters as Constructor Arguments

Verilog module:

```
module counter (clk, count)

    parameter counter_size = 4;
    parameter real_param = 2.9;
    parameter str_param = "ERROR";

    output [counter_size - 1 : 0] count;
    input clk;

    ...

endmodule
```

Foreign module (created by command: **scgenmod -createtemplate counter**):

```
template <int counter_size = 4>
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name
            int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        this->add_parameter("counter_size", counter_size);
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~counter()
    {}

};
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {

    // Instantiate counter with counter_size = 20
    counter<20>* counter_inst_1;

    SC_CTOR(top)
    {
        const char* generic_list[2];
        generic_list[0] = strdup("real_param=2.6");
        generic_list[1] = strdup("str_param=\"Hello\"");

        //
```

```
// The integer parameter override is already passed as template
// argument. Pass the overrides for the non-integer parameters
// using the foreign module constructor arguments.
//
counter_inst_1 = new counter<20>("c_inst", "work.counter", 2, \
generic_list);

// Cleanup the memory allocated for the generic list
for (int i = 0; i < 2; i++;)
    free((char*)generic_list[i]);
}
};
```

Verilog or SystemVerilog Instantiating SystemC

You can reference a SystemC module from Verilog/SystemVerilog as though the design unit is a module of the same name.

SystemC Instantiation Criteria for Verilog

A SystemC module can be instantiated in Verilog/SystemVerilog if it meets the following criteria:

- SystemC module names are case-sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- SystemC modules are exported using the `SC_MODULE_EXPORT` macro. See [Exporting SystemC Modules for Verilog](#).
- The module ports are as listed in the table shown in [Channel and Port Type Mapping](#).
- Port data type mapping must match exactly. See the table in [Data Type Mapping from SystemC to Verilog or SystemVerilog](#).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module declaration. Named port associations are case sensitive.

Exporting SystemC Modules for Verilog

To be able to instantiate a SystemC module from Verilog/SystemVerilog (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"
SC_MODULE_EXPORT(transceiver);
```

Parameter Support for Verilog Instantiating SystemC

Passing Parameters from Verilog to SystemC

To pass actual parameter values, simply use the native Verilog/SystemVerilog parameter override syntax. Parameters are passed to SystemC via the module instance parameter value list.

In addition to int, real, and string, ModelSim supports parameters with a bit range.

Named parameter association must be used for all Verilog/SystemVerilog modules that instantiate SystemC.

Retrieving Parameter Values

To retrieve parameter override information from Verilog/SystemVerilog, you can use the following functions:

```
int sc_get_param(const char* param_name, int& param_value);
int sc_get_param(const char* param_name, double& param_value);
int sc_get_param(const char* param_name, sc_string& param_value, char
format_char = 'a');
```

The first argument to `sc_get_param` defines the parameter name, the second defines the parameter value. For retrieving string values, ModelSim also provides a third optional argument, `format_char`. It is used to specify the format for displaying the retrieved string. The format can be ASCII ("a" or "A"), binary ("b" or "B"), decimal ("d" or "D"), octal ("o" or "O"), or hexadecimal ("h" or "H"). ASCII is the default. These functions return a 1 if successful, otherwise they return a 0.

Alternatively, you can use the following forms of the above functions in the constructor initializer list:

```
int sc_get_int_param(const char* param_name, int* is_successful);
double sc_get_real_param(const char* param_name, int* is_successful);
sc_string sc_get_string_param(const char* param_name, char format_char =
'a', int* is_successful);
```

Example 10-8. Verilog/SystemVerilog Instantiating SystemC, Parameter Information

Here is a complete example, ring buffer, including all files necessary for simulation.

```
// test_ringbuf.v

`timescale 1ns / 1ps
module test_ringbuf();
    reg clock;
    ...
    parameter int_param = 4;
    parameter real_param = 2.6;
    parameter str_param = "Hello World";
    parameter [7:0] reg_param = 'b001100xz;

    // Instantiate SystemC module
    ringbuf #(.int_param(int_param),
             .real_param(real_param),
             .str_param(str_param),
             .reg_param(reg_param))
            chip(.clock(clock),
              ...
              ... );
endmodule
```

```
-----

// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

#include <systemc.h>
#include "control.h"
...

SC_MODULE(ringbuf)
{
public:
    // Module ports
    sc_in clock;
    ...
    ...

    SC_CTOR(ringbuf)
        : clock("clock"),
        ...
        ...
{
    int int_param = 0;
    if (sc_get_param("int_param", int_param))
        cout << "int_param" << int_param << endl;

    double real_param = 0.0;
    int is_successful = 0;
    real_param = sc_get_real_param("real_param", &is_successful);
    if (is_successful)
        cout << "real_param" << real_param << endl;

    std::string str_param;
    str_param = sc_get_string_param("str_param", 'a', &is_successful);
    if (is_successful)
        cout << "str_param=" << str_param.c_str() << endl;
}
```

```
        str::string reg_param;
        if (sc_get_param("reg_param", 'b'))
            cout << "reg_param=" << reg_param.c_str() << endl;
    }

    ~ringbuf() {}
};

#endif
```

```
-----

// ringbuf.cpp
#include "ringbuf.h"

SC_MODULE_EXPORT(ringbuf);
```

To run the simulation, you would enter the following commands:

```
vlib work
sccom ringbuf.cpp
vlog test_ringbuf.v
sccom -link
vsim test_ringbuf
```

The simulation would return the following:

```
# int_param=4
# real_param=2.6
# str_param=Hello World
# reg_param=001100xz
```

SystemC Instantiating VHDL

To instantiate VHDL design units into a SystemC design, you must first generate a [SystemC Foreign Module \(Verilog\) Declaration](#) for each VHDL design unit you want to instantiate. Once you have generated the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

VHDL Instantiation Criteria Within SystemC

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

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type bit, bit_vector, real, std_logic, std_logic_vector, std_ulogic, std_ulogic_vector, or their subtypes. The port clause may have any mix of these types. Only locally static subtypes are allowed.

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

SystemC Foreign Module (VHDL) Declaration

In cases where you want to run a mixed simulation with SystemC and VHDL, you must create and declare a foreign module that stands in for each VHDL design unit instantiated under SystemC. You can create the foreign modules in one of two ways:

- Run **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration).
- Modify your SystemC source code manually.

After you have analyzed the design, you can generate a foreign module declaration by using **scgenmod** as follows:

```
scgenmod mod1
```

where *mod1* is any name of a VHDL entity. A foreign module declaration for the specified entity is written to stdout.

Guidelines for VHDL Complex Types

You can use VHDL complex data types with **scgenmod**. The compatible record/enum is generated, along with the foreign module declaration, subject to the following rules:

- Names of fields of the SystemC structure/enum must be same as those on the VHDL side.
- The data types of fields in the SystemC structure must follow the same type conversion (mapping) rules as normal ports.
- Additional dummy functions (`operator<<`, `sc_trace`, `operator==` functions) must be generated along with the structure definition.

Guidelines for Manual Creation in VHDL

Apply the following guidelines to the creation of foreign modules. A foreign module:

- Contains ports corresponding to VHDL ports. These ports must be explicitly named in the constructor initializer list of the foreign module.
- Must not contain any internal design elements such as child instances, primitive channels, or processes.
- Must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For VHDL, the HDL name can be in the format [`<lib>`.]<primary>[(`<secondary>`)] or [`<lib>`.]<conf>

- Can contain generics, which are supported for VHDL instantiations in SystemC designs. See [Generic Support for SystemC Instantiating VHDL](#) for more information.

Example 10-9. SystemC Design Instantiating a VHDL Design Unit

A sample VHDL design unit to be instantiated in a SystemC design is:

```
entity counter is
  port (count : buffer bit_vector(8 downto 1);
        clk   : in bit;
        reset : in bit);
end;
architecture only of counter is
  ...
  ...
end only;
```

The SystemC foreign module declaration for the above VHDL module is:

```
class counter : public sc_foreign_module {
public:
  sc_in<bool> clk;
  sc_in<bool> reset;
  sc_out<sc_logic> count;
  counter(sc_module_name nm)
    : sc_foreign_module(nm, "work.counter(only)"),
      clk("clk"),
      reset("reset"),
      count("count")
  {}
};
```

The VHDL module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the VHDL instance name.

Generic Support for SystemC Instantiating VHDL

Since the SystemC language has no concept of generics, generic values must be passed from a SystemC parent to an HDL child through the SystemC foreign module (`sc_foreign_module`). See [SystemC Foreign Module \(Verilog\) Declaration](#) for information regarding the creation of `sc_foreign_module`.

Passing Generics to `sc_foreign_module` Constructor (VHDL)

To instantiate a VHDL entity containing generics into the SystemC design, you can use one of two methods, depending on whether the generic is an integer. If the generic is an integer, you have two choices: passing as a template argument to the foreign module or as a constructor

argument to the foreign module. Non-integer generics must be passed to the foreign module using constructor arguments.

Passing Integer and Non-Integer Generics as Constructor Arguments

Both integer and non-integer parameters can be passed by specifying two generic parameters to the `sc_foreign_module` constructor: the number of generics (`int num_generics`), and the generic list (`const char* generics_list`). The `generic_list` is listed as an array of `const char*`.

If you create your foreign module manually (see [Guidelines for Manual Creation in VHDL](#)), you must also pass the generic information to the `sc_foreign_module` constructor. If you use **scgenmod** to create the foreign module declaration, the generic information is detected in the HDL child and is incorporated automatically.

Example 10-10. SystemC Instantiating VHDL, Generic Information

Following [Example 10-9](#), the generic information that would be passed to the SystemC foreign module declaration is shown below. The generic parameters passed to the constructor are shown in magenta color:

```
class counter : public sc_foreign_module {
    public:
        sc_in<bool> clk;
        ...
    counter(sc_module_name nm, char* hdl_name
           int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          {elaborate_foreign_module(hdl_name, num_generics, generic_list);}
};
```

The instantiation is:

```
dut = new counter ("dut", "work.counter", 9, generic_list);
```

Example 10-11. Passing Parameters as Constructor Arguments - 2

VHDL entity:

```
entity counter is
    generic(
        integer_gen : integer := 4,
        real_gen    : real    := 0.0,
        str_gen     : string);
    port(
        clk : in std_logic;
        count : out std_logic_vector(7 downto 0));
end counter;
```

Foreign module (created by the command: **scgenmod counter**):

```
class counter : public sc_foreign_module
```

```
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<8> > count;

    counter(sc_module_name nm, const char* hdl_name
            int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~counter()
    {}

};
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {

    counter* counter_inst_1; // Instantiate counter with counter_size = 20

    SC_CTOR(top)
    {
        const char* generic_list[3];
        generic_list[0] = strdup("integer_param=16");
        generic_list[1] = strdup("real_param=2.6");
        generic_list[2] = strdup("str_param=\"Hello\"");

        //Pass all parameter overrides using foreign module constructor args
        counter_inst_1 = new counter("c_inst", "work.counter", 3, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 3; i++;)
            free((char*)generic_list[i]);
    }
};
```

Passing Integer Generics as Template Arguments

Integer generics can be passed as template arguments to a foreign module. Doing so enables port sizes of VHDL modules to be configured using the integer template arguments. Use the `-createtemplate` option to [scgenmod](#) to generate a class template foreign module.

Example 10-12. SystemC Instantiating VHDL, Passing Integer Generics as Template Arguments

VHDL entity:

```
entity counter is
```

```
generic(counter_size : integer := 4);
port(
  clk : in std_logic;
  count : out std_logic_vector(counter_size - 1 downto 0));
end counter;
```

Foreign module (created by the command: **scgenmod -createtemplate counter**):

```
template <int counter_size = 4>
class counter : public sc_foreign_module
{
public:
  sc_in<sc_logic> clk;
  sc_out<sc_lv<counter_size-1 + 1> > count;

  counter(sc_module_name nm, const char* hdl_name)
    : sc_foreign_module(nm),
      clk("clk"),
      count("count")
  {
    this->add_parameter("counter_size", counter_size);
    elaborate_foreign_module(hdl_name);
  }
  ~counter()
  {}
}
}
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {

  counter<20> counter_inst_1;
  // Instantiates counter with counter_size = 20
  counter counter_inst_2;
  // Instantiates counter with default counter_size = 4

  SC_CTOR(top)
  : counter_inst_1(cinst_1, "work.counter"),
    counter_inst_2(cinst_2, "work.counter")
  {}

};
```

Example 10-13. Passing Integer Generics as Template Arguments and Non-integer Generics as Constructor Arguments

VHDL entity:

```
entity counter is
generic(counter_size : integer := 4);
port(
  clk : in std_logic;
  count : out std_logic_vector(counter_size - 1 downto 0));
```

```
end counter;
```

Foreign module (created by the command: **scgenmod -createtemplate counter**):

```
template <int counter_size = 4>
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name
            int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        this->add_parameter("counter_size", counter_size);
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~counter()
    {}
};
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {

    // Instantiate counter with counter_size = 20
    counter<20>* counter_inst_1;

    SC_CTOR(top)
    {
        const char* generic_list[2];
        generic_list[0] = strdup("real_param=2.6");
        generic_list[1] = strdup("str_param=\"Hello\"");

        //
        // The integer parameter override is already passed as template
        // argument. Pass the overrides for the non-integer parameters
        // using the foreign module constructor arguments.
        //
        counter_inst_1 = new counter<20>("c_inst", "work.counter", 2, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 2; i++;)
            free((char*)generic_list[i]);
    }
};
```

VHDL Instantiating SystemC

To instantiate SystemC in a VHDL design, you must create a component declaration for the SystemC module. Once you have generated the component declaration, you can instantiate the SystemC component just like any other VHDL component.

SystemC Instantiation Criteria for VHDL

A SystemC design unit may be instantiated within VHDL if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- The SystemC design unit is exported using the `SC_MODULE_EXPORT` macro.
- The module ports are as listed in the table in [Data Type Mapping Between SystemC and VHDL](#)
- Port data type mapping must match exactly. See the table in [Port Type Mapping](#).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module. Named port associations are case sensitive.

Component Declaration for VHDL Instantiating SystemC

A SystemC design unit can be referenced from a VHDL design as though it is a VHDL entity. The interface to the design unit can be extracted from the library in the form of a component declaration by running **vgencomp**. Given a library and a SystemC module name, **vgencomp** writes a component declaration to standard output.

The default component port types are:

- `std_logic`
- `std_logic_vector`

Optionally, you can choose:

- `bit` and `bit_vector`

VHDL and SystemC Identifiers

The VHDL identifiers for the component name and port names are the same as the SystemC identifiers for the module name and port names. Except for the cases noted below, ModelSim does nothing to the SystemC identifier when it generates the entity.

ModelSim converts the SystemC identifiers to VHDL 1076-1993 extended identifiers in three cases:

- The SystemC identifier is not a valid VHDL 1076-1987 identifier.
- SystemC module is compiled with `scom -93`. One exception is a valid, lowercase identifier (such as `scmod`). Valid, lowercase identifiers are not converted even if you compile the design with `scom -93`.
- The SystemC identifier is not unique when case is ignored. For example, if `ScMod` and `scmod` both appear in the same design, ModelSim will convert the former to `\ScMod\`.

vgencomp Component Declaration when VHDL Instantiates SystemC

`vgencomp` generates a component declaration according to these rules:

- Port Clause

A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named SystemC port.

You can set the VHDL port type to `bit` or `std_logic`. If the SystemC port has a range, then the VHDL port type is `bit_vector` or `std_logic_vector`. For example:

SystemC port	VHDL port
<code>sc_in<sc_logic>p1;</code>	<code>p1 : in std_logic;</code>
<code>sc_out<sc_lv<8>>p2;</code>	<code>p2 : out std_logic_vector(7 downto 0);</code>
<code>sc_inout<sc_lv<8>>p3;</code>	<code>p3 : inout std_logic_vector(7 downto 0)</code>

Configuration declarations are allowed to reference SystemC modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a SystemC instance to configure the instantiations within the SystemC module.

Exporting SystemC Modules for VHDL

To be able to instantiate a SystemC module within VHDL (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"  
SC_MODULE_EXPORT(transceiver);
```

The **scom -link** command collects the object files created in the work library, and uses them to build a shared library (*.so*) in the current work library. If you have changed your SystemC

source code and recompiled it using **scom**, then you must run **scom -link** before invoking **vsim**. Otherwise your changes to the code are not recognized by the simulator.

Passing Generics From VHDL or Verilog Down to SystemC

When you have a VHDL or Verilog module instantiating a SystemC module and want to pass generic information between the two levels, you must add custom macros to your SystemC module for registering and initializing the generic information.

Prerequisites

The SystemC module must have a Verilog or VHDL parent module

Procedure

1. Add registration macros to the declarative region of the SystemC module. The macros are:
 - `SC_GENERIC_INT(<generic_name>, <default_value>);`
<default_value> must be an integer literal
 - `SC_GENERIC_REAL(<generic_name>, <default_value>);`
<default_value> must be a real literal
 - `SC_GENERIC_STRING(<generic_name>, <default_value>);`
<default_value> must be a string literal enclosed in double quotes (“”).

For all macros, <default_value> must be a constant literal value. You cannot use variables, constants, signals or other generics.

You can use these macros multiple times to register multiple generics.

2. Add initializer macros to the initializer list of your SystemC module's constructor section. The macros are:
 - `SC_INIT_GENERIC_INT(<generic_name>)`
 - `SC_INIT_GENERIC_REAL(<generic_name>)`
 - `SC_INIT_GENERIC_STRING(<generic_name>)`

These macros will retrieve the correct generic value from the Verilog or VHDL parent module.

You can use these macros multiple times to initialize multiple generics.

3. (optional) Use a flag “<generic_name>_valid” to ensure the validity of a generic's value. This is most useful when you use a generic in a conditional block to create

underlying hierarchy. If you do not test for this validity, or continue to simulation with invalid information, you could receive the following warning.

```
# ** Warning: (vsim-6663) Instance
'/test_ringbuf/ring_INST/block1_COPY' created during elaboration in
vsim has not been created during elaboration in vopt. It is likely
that the instantiation statement corresponding to this instance is
dependent on the value of a generic propagated to SystemC from HDL.
Please check to see that the SystemC hierarchy created in vsim is
correct.
```

4. Compile using sccom as you normally would.

Example

Below are some examples of the registration and initialization macro syntax. For a complete example refer to the directory `<install_dir>/examples/systemc/vhdl_sc_generics`.

```
SC_MODULE(example)
{
public:

    SC_GENERIC_INT(generic_int, 0);
    SC_GENERIC_REAL(generic_real, 0.0);
    SC_GENERIC_STRING(generic_boolean, "true");

    SC_CTOR(example)
    : SC_INIT_GENERIC_INT(generic_int),
      SC_INIT_GENERIC_REAL(generic_real),
      SC_INIT_GENERIC_STRING(generic_boolean)
    {
        if (generic_int_valid) {
            block1_COPY = new control("block1_COPY", "control", 3,
                generic_list_1);
            block1_COPY->clock(clock);
            block1_COPY->reset(reset);
        }
    }

    ~example() {}
};
```

SystemC Procedural Interface to SystemVerilog

SystemC designs can communicate with SystemVerilog through a procedural interface, the SystemVerilog Direct Programming Interface (DPI). In contrast to a hierarchical interface, where communication is advanced through signals and ports, DPI communications consists of task and function calls passing data as arguments. This type of interface can be useful in transaction level modeling, in which bus functional models are widely used.

This section describes the use flow for using the SystemVerilog DPI to call SystemVerilog export functions from SystemC, and to call SystemC import functions from SystemVerilog.

The SystemVerilog LRM describes the details of a DPI C import and export interface. This document describes how to extend the same interface to include SystemC and C++ in general. The import and export keywords used in this document are in accordance with SystemVerilog as described in the SV LRM. An export function or task is defined in SystemVerilog, and is called by C or SystemC. An import task or function is defined in SystemC or C, and is called from SystemVerilog.

Definition of Terms

The following terms are used in this chapter.

- C++ import function

A C++ import function is defined as a free floating C++ function, either in the global or some private namespace. A C++ import function must not have any SystemC types as formal arguments. This function must be made available in the SystemC shared library.

- SystemC Import Function

A SystemC import function must be available in the SystemC shared library, and it can be either of the following:

- A free-floating C++ function, either in the global or private namespace, with formal arguments of SystemC types.
- A SystemC module member function, with or without formal arguments of SystemC types.

- Export Function

A SystemVerilog export function, as defined in the SystemVerilog LRM.

SystemC DPI Usage Flow

The usage flow of SystemC DPI depends on the function modes, whether they are import or export. The import and export calls described in the following sections can be mixed with each other in any order.

SystemC Import Functions

In order to make a SystemC import function callable from SystemVerilog, it needs to be registered from the SystemC code before it can be called from SystemVerilog. This can be thought of as exporting the function outside SystemC, thus making it callable from other languages. The registration must be done by passing a pointer to the function using an API. The registration can be done anywhere in the design but it must be done before the call happens in SystemVerilog, otherwise the call fails with undefined behavior.

Global Functions

A global function can be registered using the API below:

```
int sc_dpi_register_cpp_function(const char* function_name, PTFN
func_ptr);
```

This function takes two arguments:

- the name of the function, which can be different than the actual function name. This name must match the SystemVerilog import declaration. No two function registered using this API can have the same name: it creates an error if they do.
- a function pointer to the registered function. On successful registration, this function will return a 0. A non-zero return status means an error.

Example 10-14. Global Import Function Registration

```
int scGlobalImport(sc_logic a, sc_lv<9>* b);
sc_dpi_register_cpp_function("scGlobalImport", scGlobalImport);
```

A macro like the one shown below is provided to make the registration even more simple. In this case the ASCII name of the function will be identical to the name of the function in the source code.

```
SC_DPI_REGISTER_CPP_FUNCTION(scGlobalImport);
```

In the SystemVerilog code, the import function needs to be defined with a special marker ("DPI-SC") that tells the SV compiler that this is an import function defined in the SystemC shared library. The syntax for calling the import function remains the same as described in the SystemVerilog LRM.

Example 10-15. SystemVerilog Global Import Declaration

For the SystemC import function shown in [Example 10-14](#), the SystemVerilog import declaration is as follows:

```
import mti_scdpi::*;
import "DPI-SC" context function int scGlobalImport(
    input sc_logic a, output sc_lv[8:0] b);
```

[Example 10-16](#) shows how to register a global function by introducing a "dummy" module specifically for the purpose of the registration. This lets you do the registration in the procedural context anytime before the import function is used.

Example 10-16. Registering a Global Function

```
/*This top-level SystemC module does nothing but register DPI-SC imports
*/
SC_MODULE(dpi_sc_import)
```

```
{
    SC_CTOR(dpi_sc_import)
    {
        SC_DPI_REGISTER_CPP_FUNCTION(scGlobalImport);
        .....
    }
    ~dpi_sc_import() {};
};
SC_MODULE_EXPORT(dpi_sc_import)
```

Please refer to [Module Member Functions](#) and [Calling SystemVerilog Export Tasks / Functions from SystemC](#) for more details on the SystemC import and export task or function declaration syntax.

Module Member Functions

Registering Functions

Module member functions can be registered anytime before they are called from the SystemVerilog code. The following macro can be used to register a non-static member function if the registration is done from a module constructor or a module member function. For a static member function, the registration is accomplished using the interface `SC_DPI_REGISTER_CPP_FUNCTION`, as described in [SystemC Import Functions](#).

```
SC_DPI_REGISTER_CPP_MEMBER_FUNCTION(<function_name>, <func_ptr>);
```

Example:

```
SC_MODULE(top) {

    void sc_func() {
    }

    SC_CTOR(top) {
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_func", &top::sc_func);
    }
};
```

Note that in the above case, since the registration is done from the module constructor, the module pointer argument might be redundant. However, the module pointer argument will be required if the macro is used outside a constructor.

To register a member function from a function that is not a member of the module, the following registration function must be used:

```
int sc_dpi_register_cpp_member_function(<function_name>, <module_ptr>,
<func_ptr>);
```

This function takes three arguments. The first argument is the name of the function, which can be different than the actual function name. This is the name that must be used in the SystemVerilog import declaration. The second argument is a reference to the module instance

where the function is defined. It is illegal to pass a reference to a class other than a class derived from `sc_module` and will lead to undefined behavior. The third and final argument is a function pointer to the member function being registered. On successful registration, this function will return a 0. A non-zero return status means an error. For example, the member function `run()` of the module "top" in the example above can be registered as follows:

```
sc_module* pTop = new top("top");
sc_dpi_register_cpp_member_function("run", pTop, &top::run);
```

Setting Stack Size for Import Tasks

The tool implicitly creates a SystemC thread to execute the C++ functions declared as SystemVerilog import tasks. The default stack size is 64KBytes and may not be big enough for any C++ functions. To change the default stack size, you can use the interface `sc_dpi_set_stack_size`. You must use this interface right after the registration routine, for example:

```
SC_MODULE(top) {
    void sc_task() {
    }

    SC_CTOR(top) {
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_task", &top::sc_task);
        sc_dpi_set_stack_size(1000000); // set stack size to be 1Mbyte.
    }
}
```

For the C++ functions declared as SystemVerilog import functions, you do not need to set the stack size.

Declaring and Calling Member Import Functions in SystemVerilog

The declaration for a member import function in SystemVerilog is similar to the following:

```
import "DPI-SC" context function int scMemberImport(
    input sc_logic a, output sc_lv[8:0] b);
```

Registration of static member functions is identical to the registration of global functions using the API `sc_dpi_register_cpp_function()`.

Only one copy of the overloaded member functions is supported as a DPI import, as DPI can only identify the import function by its name. not by the function parameters.

To enable the registration of member functions, the SystemC source file must be compiled with the `-DMTI_BIND_SC_MEMBER_FUNCTION` macro.

Calling Member Import Functions in a Specific SystemC Scope

A member import function can be registered for multiple module instances, as in the case when registration routine `SC_DPI_REGISTER_CPP_MEMBER_FUNCTION()` is called from inside a SystemC module constructor. At runtime, you must specify the proper scope when the member import function call is initiated from SystemVerilog side. You can use the following two routines to manipulate the SystemC scope before making the member import function call:

```
function string scSetScopeByName(input string sc_scope_name);
```

and

```
function string scGetScopeName();
```

scSetScopeByName() expects the full hierarchical name of a valid SystemC scope as the input. The hierarchical name must use the Verilog-style path separator. The previous scope hierarchical name before setting the new scope will be returned.

scGetScopeName() returns the current SystemC scope for next member import function call.

Since both routines are predefined in ModelSim built-in package **mti_scdpi**, you need to import this package into the proper scope where the two routines are used, using the following statement:

```
import mti_scdpi::*;
```

Example 10-17. Usage of `scSetScopeByName` and `scGetScopeName`

```
//test.cpp:
SC_MODULE(scmod)
{
    void cppImportFn();

    SC_CTOR(scmod)
    {
        .....
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("cppImportFn",
        &scmod::cppImportFn);
        .....
    }
};

//test.sv:
module top();

    import mti_scdpi::*;    // where scSetScopeByName() and scGetScopeName()
    are
    defined.

    string prev_sc_scope;
```

```
string curr_sc_scope;

scmod inst1();    //scope name "top.inst1"
scmod inst2();    //scope name "top.inst2"

import "DPI-SC" function void cppImportFn();

// call DPI-SC import function under scope "top.inst1"
prev_sc_scope = scSetScopeByName("top.inst1");
curr_sc_scope = scGetScopeName();
cppImportFn();

// call DPI-SC import function under scope "top.inst2"
prev_sc_scope = scSetScopeByName("top.inst2");
curr_sc_scope = scGetScopeName();
cppImportFn();

endmodule
```

Calling SystemVerilog Export Tasks / Functions from SystemC

Unless an export call is made from an import function, you must set the scope of the export function explicitly to provide the SystemVerilog context information to the simulator. You do this by calling `svSetScope()` before each export function or task call.

An export function to be called with SystemC arguments must have an export declaration, similar to the following:

```
export "DPI-SC" context function Export;
```

The function declaration must use the SystemC type package, similar to the following:

```
import mti_scdpi::*;
function int Export(input sc_logic a, output sc_bit b);
```

The syntax for calling an export function from SystemC is the same as any other C++ function call.

SystemC Data Type Support in SystemVerilog DPI

The SystemVerilog package “`scdpi`” must be imported if a SystemC data type is used in the arguments of import and export functions.

```
import mti_scdpi::*
```

The SystemC data type names have been treated as special keywords. Avoid using these keywords for other purposes in your SystemVerilog source files.

The table below shows how each of the SystemC type will be represented in SystemVerilog. This table must be followed strictly for passing arguments of SystemC type. The SystemVerilog typedef statements, listed in the middle column of [Table 10-29](#), are automatically imported whenever the mti_scdpi package is imported.

Table 10-29. SystemC Types as Represented in SystemVerilog

SystemC Type	SV Typedef	Import/Export Declaration
sc_logic	typedef logic sc_logic	sc_logic
sc_bit	typedef bit sc_bit	sc_bit
sc_bv<N>	typedef bit sc_bv	sc_bit[N-1:0]
sc_lv<N>	typedef logic sc_lv	sc_lv[N-1:0]
sc_int<N>	typedef bit sc_int	sc_int[N-1:0]
sc_uint<N>	typedef bit sc_uint	sc_uint[N-1:0]
sc_bigint<N>	typedef bit sc_bigint	sc_bigint[N-1:0]
sc_biguint<N>	typedef bit sc_biguint	sc_biguint[N-1:0]
sc_fixed<W,I,Q,O,N>	typedef bit sc_fixed	sc_fixed[I-1:I-W]
sc_ufixed<W,I,Q,O,N>	typedef bit sc_ufixed	sc_ufixed[I-1:I-W]
sc_fixed_fast<W...>	typedef bit sc_fixed_fast	sc_fixed[I-1:I-W]
sc_ufixed_fast<W...>	typedef bit sc_ufixed_fast	sc_fixed[I-1:I-W]
sc_signed	typedef bit sc_signed	sc_signed[N-1:0]
sc_unsigned	typedef bit sc_unsigned	sc_unsigned[N-1:0]
sc_fix	typedef bit sc_fix	sc_fix[I-1:1-W]
sc_ufix	typedef bit sc_ufix	sc_ufix[I-1:1-W]
sc_fix_fast	typedef bit sc_fix_fast	sc_fix_fast[I-1:1-W]
sc_ufix_fast	typedef bit sc_ufix_fast	sc_ufix_fast[I-1:1-W]

According to the table above, a SystemC argument of type **sc_uint<32>** will be declared as **sc_uint[31:0]** in SystemVerilog “DPI-SC” declaration. Similarly, **sc_lv<9>** would be **sc_lv[8:0]**. to enable the fixed point datatypes, the SystemC source file must be compiled with **-DSC_INCLUDE_FX**.

For fixed-point types the left and right indexes of the SV vector can lead to a negative number. For example, **sc_fixed<3,0>** will translate to **sc_fixed[0-1:0-3]** which is **sc_fixed[-1:-3]**. This representation is used for fixed-point numbers in the ModelSim tool, and must be strictly followed.

For the SystemC types whose size is determined during elaboration, such as **sc_signed** and **sc_unsigned**, a parameterized array must be used on the SV side. The array size parameter value, on the SystemVerilog side, must match correctly with the constructor arguments passed to types such as **sc_signed** and **sc_unsigned** at SystemC elaboration time.

Some examples:

An export declaration with arguments of SystemC type:

```
export "DPI-SC" context function Export;
```

```
import mti_scdpi::*;  
function int Export(input sc_logic a, input sc_int[8:0] b);
```

An import function with arguments of SystemC type:

```
import mti_scdpi::*;  
import "DPI-SC" context function int scGlobalImport(  
    input sc_logic a, output sc_lv[8:0] b);
```

An export function with arguments of regular C types:

```
export "DPI-SC" context function Export;  
function int Export(input int a, output int b);
```

Using a Structure to Group Variables

Both SystemC and SystemVerilog support using a *structure*, which is a composite data type that consists of a user-defined group of variables. This grouping capability of a structure provides a convenient way to work with a large number of related variables. The structure lets you use multiple instances of these variables without having to repeat them individually for each instance.

The same typedefs supported for SystemC types as arguments to DPI-SC can be members of structures.

Example — SystemVerilog

The following structure declaration defines a group of five simple variables: direction, flags, data, addr, token_number. The name of the structure is defined as packet_sv.

```
typedef struct {  
    sc_bit          direction;  
    sc_bv[7:0]     flags;  
    sc_lv[63:0]    data;  
    bit[63:0]     addr;  
    int           token_number;  
} packet_sv;
```

You can then use this structure (packet_sv) as a data-type for arguments of DPI-SC, just like any other variable. For example:

```
import "DPI-SC" task svImportTask(input packet_sv pack_in,  
    output packet_sv pack_out);
```

Example — SystemC

An equivalent structure containing corresponding members of SystemC types are available on the SystemC side of the design. The following structure declaration defines a group of five simple variables: `direction`, `flags`, `data`, `addr`, `token_number`. The name of the structure is defined as `packet_sc`.

```
typedef struct {
    sc_bit          direction;
    sc_bv<8>       flags;
    sc_lv<64>      data;
    svBitVecVal    addr [SV_PACKED_DATA_NELEMS (64) ];
    int            token_number;
} packet_sc;
```

You can then use this structure (`packet_sc`) as a data-type for arguments of DPI-SC, just like any other variable. For example:

```
import "DPI-SC" task svImportTask(input packet_sc pack_in,
output packet_sc pack_out);
```

SystemC Function Prototype Header File (`sc_dpiheader.h`)

A SystemC function prototype header file is automatically generated for each SystemVerilog compilation. By default, this header file is named `sc_dpiheader.h`. This header file contains the C function prototype statements consistent with the "DPI-SC" import/export function/task declarations. You can include this file in the SystemC source files where the prototypes are needed for SystemC compiles and use this file as a sanity check for the SystemC function arguments and return type declaration in the SystemC source files.

When the SystemVerilog source files with the usage of "DPI-SC" spans over multiple compiles, the `sc_dpiheader.h` generated from an earlier SystemVerilog compilation will potentially be overwritten by the subsequent compiles. To avoid the name conflict, one can use the `"-scdpiheader"` argument to the `vlog` command to name the header file differently for each compilation. For example the following `vlog` command line will generate a header file called `"top_scdpi.h"`:

```
vlog -scdpiheader top_scdpi.h top.sv
```

Support for Multiple SystemVerilog Libraries

By default, only the DPI-SC usage in current work library is processed at the SystemC link time. If additional SystemVerilog libraries are used that import or export SystemC DPI routines, the names of these libraries must be provided to `sccom` at link time using the `"-dpilib"` argument.

An example of linking with multiple SystemVerilog libraries are:

```
sccom -link -dpilib dpilib1 -dpilib dpilib2 -dpilib dpilib3
```

where dpilib1, dpilib2 and dpilib3 are the logical names of SystemVerilog libraries previously compiled.

An example of a complete compile flow for compiling with multiple libraries is as follows:

```
// compile SV source files for dpilib1
vlog -work dpilib1 -scdpiheader sc_dpiheader1.h ./src/dpilib1_src.sv

// compile SV source files for dpilib2
vlog -work dpilib2 -scdpiheader sc_dpiheader2.h ./src/dpilib2_src.sv

// compile SV source files for dpilib3
vlog -work dpilib3 -scdpiheader sc_dpiheader3.h ./src/dpilib3_src.sv

// SystemC source file compilations that may include all of the above
three header files.
sccom scmod.cpp

// compile other Verilog sources file if there are any.
vlog -work work non_scdpi_source.sv

// final sccom link phase
sccom -link -dpilib dpilib1 -dpilib dpilib2 -dpilib dpilib3
```

SystemC DPI Usage Example

```
-----
hello.v:

module top;

hello c_hello();

import "DPI-SC" context function void sc_func();
export "DPI-SC" task verilog_task;

task verilog_task();
    $display("hello from verilog_task.");
endtask

initial
begin

    sc_func();

    #2000 $finish;
end
endmodule
```

hello.cpp:

```
#include "systemc.h"
#include "sc_dpiheader.h"

SC_MODULE(hello)
{
    void call_verilog_task();
    void sc_func();

    SC_CTOR(hello)
    {
        SC_THREAD(call_verilog_task);

        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_func", &hello::sc_func);
    }

    ~hello() {};
};

void hello::sc_func()
{
    printf("hello from sc_func()").
}

void hello::call_verilog_task()
{
    svSetScope(svGetScopeFromName("top"));

    for(int i = 0; i < 3; ++i)
    {
        verilog_task();
    }
}

SC_MODULE_EXPORT(hello);
```

Compilation:

```
vlog -sv hello.v
sccom -DMTI_BIND_SC_MEMBER_FUNCTION hello.cpp
sccom -link
vsim -c -do "run -all; quit -f" top
```


Chapter 11

Advanced Simulation Techniques

Checkpointing and Restoring Simulations

The `checkpoint` and `restore` commands allow you to save and restore the simulation state within the same invocation of `vsim` or between `vsim` sessions.

Table 11-1. Checkpoint and Restore Commands

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current <code>vsim</code> session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous invocation of <code>vsim</code>	vsim -restore <filename>

Checkpoint File Contents

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

- `modelsim.ini` settings
- simulation kernel state
- `vsim.wlf` file
- signals listed in the List and Wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog `$fopen` system task
- state of foreign architectures
- state of PLI/VPI/DPI code

Checkpoint Exclusions

You *cannot* checkpoint/restore the following:

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

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the Foreign Language Interface Reference Manual or [Verilog Interfaces to C](#) for more information.

Controlling Checkpoint File Compression

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

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

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

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

The Difference Between Checkpoint/Restore and Restart

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

Using Macros with Restart and Checkpoint/Restore

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

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

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a **checkpoint** and later in the same session doing a **restore** 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.

Checkpointing Foreign C Code That Works with Heap Memory

If checkpointing foreign C code (FLI/PLI/VPI/DPI) that works with heap memory, use `mti_Malloc()` rather than raw `malloc()` or `new`. Any memory allocated with `mti_Malloc()` is guaranteed to be restored correctly. Any memory allocated with raw `malloc()` will not be restored correctly, and simulator crashes can result.

Checkpointing a Running Simulation

In general you can invoke a checkpoint command only when the simulation is stopped. If you need to checkpoint without stopping the simulation, you need to write a script that utilizes the **when** command and variables from your code to trigger a checkpoint. The example below shows how this might be done with a simple Verilog design.

Keep in mind that the variable(s) in your code must be visible at the simulation time that the checkpoint will occur. Some global optimizations performed by ModelSim may limit variable visibility, and you may need to optimize your design using the **+acc** argument to **vopt**.

You would compile and run the example like this:

```
vlog when.v  
vsim -c when -do "do when.do"
```

where *when.do* is:

```
onbreak {  
    echo "Resume macro at $now"  
    resume  
}  
quietly set continueSim 1  
quietly set whenFired 0  
quietly set checkpointCntr 0  
  
when { needToSave = 1 } {  
    echo "when Stopping to allow checkpoint at $now"  
    set whenFired 1  
    stop  
}
```

```
while {$continueSim} {  
  run -all  
  if { $whenFired} {  
    set whenFired 0  
    echo "Out of run command. Do checkpoint here"  
    checkpoint cpf.n[incr checkpointCntr].cpt  
  }  
}
```

and *when.v* is:

```
module when;  
  
  reg clk;  
  reg [3:0] cnt;  
  reg needToSave;  
  
  initial  
  begin  
    needToSave = 0;  
    clk = 0;  
    cnt = 0;  
    #1000;  
    $display("Done at time %t", $time);  
    $finish;  
  end  
  
  always #10 clk = ~clk;  
  
  always @(posedge clk)  
  begin  
    cnt = cnt + 1;  
    if (cnt == 4'hF)  
    begin  
      $display("Need to Save : %b", needToSave);  
      needToSave = 1;  
    end  
  end  
  
  // Need to reset the flag, but must wait a timestep  
  // so that the when command has a chance to fire  
  
  always @(posedge needToSave)  
  #1 needToSave = 0;  
endmodule
```

and the transcript output is:

```
# vsim -do {do when.do} -c when  
# //  
# Loading work.when  
# do when.do  
# Need to Save : 0  
# when Stopping to allow checkpoint at 290 # Simulation stop requested.  
# Resume macro at 290
```

```
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 610
# Simulation stop requested.
# Resume macro at 610
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 930
# Simulation stop requested.
# Resume macro at 930
# Out of run command. Do checkpoint here
# Done at time          1000
# ** Note: $finish      : when.v(14)
#   Time: 1 us  Iteration: 0  Instance: /when
```

Simulating with an Elaboration File

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

You can generate a loadable image (elaboration file) that can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an Elaboration File?

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

Another reason to use elaboration files is for comparing performance benchmarks. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times.

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

Elaboration File Flow

To maximize the benefit of simulating elaboration files, you should observe the following flow:

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

2. Apply all normal **vsim** arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see [Modifying Stimulus](#) below).
3. Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an Elaboration File

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

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

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

Note



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

Loading an Elaboration File

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

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

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

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

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

Note

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

Modifying Stimulus

A primary use of elaboration files is to simulate the same design multiple times using a different stimulus. The following techniques allow you to modify the stimulus for each simulation run.

- Use the change command to modify parameters or generic values. This affects values only—it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.

Note that because the elaborated image is already created, the vsim -g and vsim -G arguments are ignored for simulation.

- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The **<HDLfilename>** file name, if it appears in the design as a file name (for example, a VHDL FILE object as well as some Verilog sysfuncs that take file names), is substituted with the **<NEWfilename>** file name. This mapping occurs before environment variable expansion and can't be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.
- Use **-sv_seed <integer> | random** to change the value used for the root random number generator for SystemVerilog threads.

Using With the PLI or FLI

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

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

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

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

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

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

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

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

where *vectors* is the stimulus file.

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

```
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```

Chapter 12

Recording and Viewing Transactions

This chapter discusses transactions in ModelSim: what they are, how to successfully record them, and how to view them in the GUI.

Transaction Background	641
What is a Transaction?	642
About the Source Code for Transactions	642
Viewing Transactions in the GUI	645
Viewing Transaction Objects in the Structure Window	647
Viewing Transactions in the Wave Window	647
Viewing a Transaction in the List Window	654
Viewing a Transaction in the Objects Window	655
Transactions in Designs with Questa Verification IP	657
Transaction Recording Flow	657
Transaction Recording Procedures	665
Recording Transactions in Verilog and VHDL	665
Recording Transactions in SystemC.....	669
Transaction Recording Guidelines	660
Names of Streams and Substreams	661
Stream Logging.....	661
Attribute Type.....	662
Anonymous Attributes	663
Multiple Uses of the Same Attribute	662
Transaction UIDs	662
Definition of Relationship in Transactions.....	663
The Life-cycle of a Transaction	664
Retroactive Recording / Start and End Times.....	664
Start and End Times for Phase Transactions	665
Transaction Handles and Memory Leaks	665
CLI Debugging Command Reference	676
Verilog and VHDL API System Task Reference	677

Transaction Background

What is a Transaction?	642
About the Source Code for Transactions	642
About Transaction Streams.....	643

What is a Transaction?

A *transaction* is a statement of what the design is doing between one time and another during a simulation run. It is as simple as it is powerful.

While the definition of a transaction may be simple, the word “transaction” itself can be confusing because of its association with Transaction Level Modeling (TLM). In TLM, design units pass messages across interfaces and these messages are typically called transactions.

In ModelSim, the term transaction is used in a broader sense:

- *transaction*

An abstract statement, logged in the WLF file, of what the design was doing at a specific time. The designer writes a transaction in the source code, which is then logged into the WLF file during simulation. Often, transactions represent packets of data moving around between design objects. Transactions allow users to debug and monitor the design at any level of abstraction.

As written in the source code, a transaction at a minimum consists of:

- a name
- a start time
- an end time

With that alone, you could record the transitions of a state machine, summarize the activity on a bus, and so forth. Additionally, transactions may have user-defined *attributes*, such as address, data, status, and so on.

About the Source Code for Transactions

Essentially, you record transactions by writing the transactions into your design code:

- Verilog/SystemVerilog and VHDL transactions — written using a custom API specifically developed for designs being verified with the ModelSim simulator. The term “Verilog” is used throughout this chapter to indicate both forms of the language (Verilog and SystemVerilog) unless otherwise specified.

See “[Recording Transactions in Verilog and VHDL](#)” for details on the tasks involved in recording.

- SystemC transactions — written with the SystemC Verification (SCV) library.

See “[Recording Transactions in SystemC](#)” for details on the tasks involved in recording.

See the SystemC Verification Standard Specification, Version 1.0e for SystemC API syntax for recording transactions.

You create/record transactions through the Verilog/VHDL or SystemC API calls placed in your design source code. As the simulation progresses, individual transactions are recorded into the WLF file and are available for design debug and performance analysis in both interactive debug and post-simulation debug.

See “[Verilog and VHDL API System Task Reference](#)” for the ModelSim Verilog API recording syntax.

About Transaction Streams

Transactions are recorded on *streams*, much as values are recorded on wires and signals. Streams are debuggable objects: they appear in or may be added to GUI windows such as the Objects or Wave windows.

When more than one transaction is recorded on a stream at one time the transactions are said to be “concurrent”. You can visualize concurrent transactions as if they were concurrent. The simulator creates *substreams* as needed so that concurrent transactions on the stream remain distinct (see [Figure 12-1](#)).

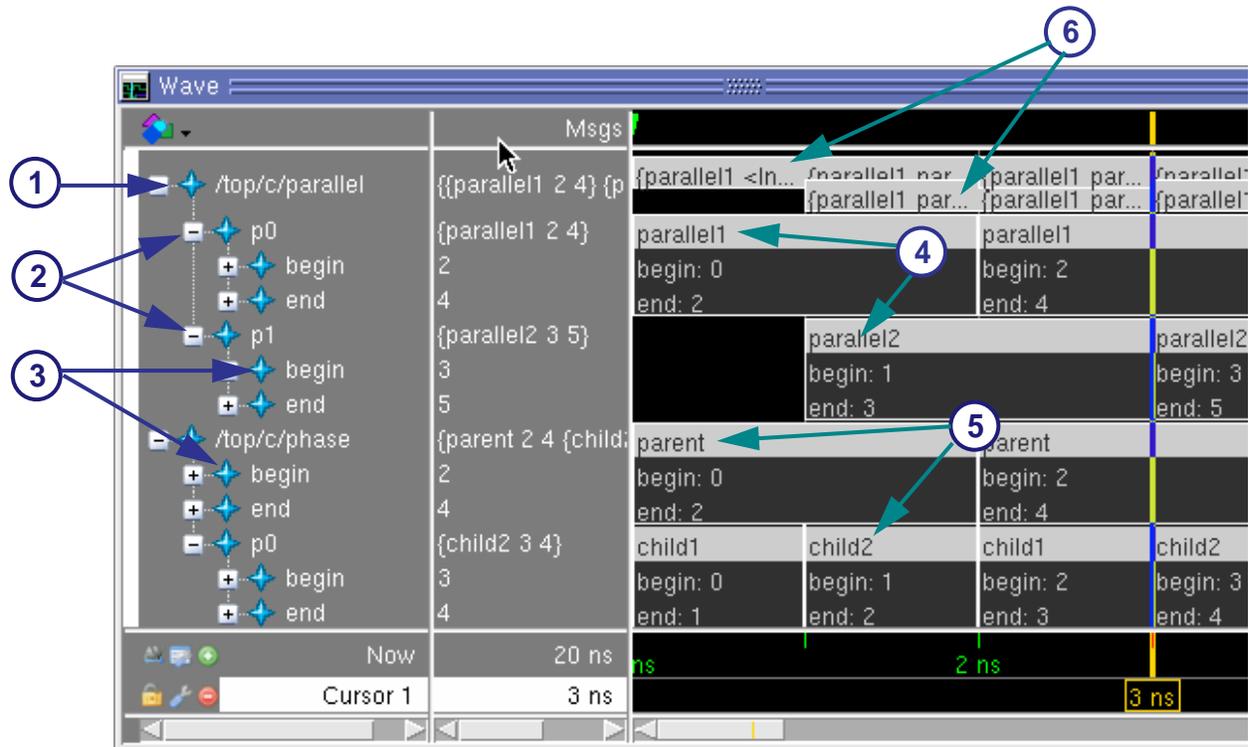
Concurrent transactions appear in the Wave window as they are recorded, either as:

- *parallel* transactions — transactions which overlap, but where no intrinsic relationship exists between the two transactions.
- *phase* transactions — where the concurrent transaction is actually a “child” of the initial transaction.

You specify whether an concurrent transaction is phase or parallel during the recording.

The simulator automatically logs the transactions, making them available for immediate viewing in the GUI. Transactions are best viewed in the Wave window. See [Viewing Transactions in the GUI](#) for procedural details.

Figure 12-1. Transaction Anatomy in Wave Window



Icon	Element
①	Transaction Stream
②	Substream
③	Attributes
④	Parallel transactions
⑤	Phase transactions
⑥	Concurrent (overlapping) transactions

Parallel Transactions

The simulator creates a separate substream for each transaction so that they are distinct from each other in the view. Expanding the substream reveals the attributes on those transactions.

Concurrent transaction instances are drawn overlapping, with a vertical offset, so that each instance can be seen.

i **Tip: Concept:** Substream Creation — Generally, you have no direct control over the creation of substreams; they are created for you during simulation as needed. The rule for substream creation is: A transaction is placed on the first substream that has no active transaction and does not have any transaction in the future of the one being logged.

Phase / Child Transactions

Phase transactions are a special type of concurrent transaction in ModelSim. See [Recording Phase Transactions](#) for information on how to specify the recording of a transaction as a phase of a parent transaction.

The simulator has an alternative way of drawing phase transactions, as they are considered to be “phases” of a parent transaction. For example, consider that a busRead transaction may have several steps or phases. Each of these could be represented as a smaller, concurrent transaction and would appear on a second substream. However, you can indicate that these are phase transactions, and by doing so, you instruct the tool to draw them specially, as shown in [Figure 12-1](#).

Concurrent transaction instances are drawn overlapping, with a vertical offset, so that each instance can be seen.

Viewing Transactions in the GUI

Once recorded, transactions can be viewed in the Wave, List, and Object windows. Transactions are best viewed in the Wave window, where their appearance is unlike any other objects in the GUI.

Prerequisites

- Transactions must be recorded (written in source code) prior to simulation in order to appear in the GUI.

See “[Recording Transactions in Verilog and VHDL](#)” and “[Recording Transactions in SystemC](#)” for details.

Transaction Viewing Commonalities

The information contained in this section relates to viewing transactions in all windows and panes where transactions appear. It explains the general viewing behavior that is consistent across the GUI.

Transactions are recorded in the WLF file, and thus are viewable, only when there are transaction instances on the stream.

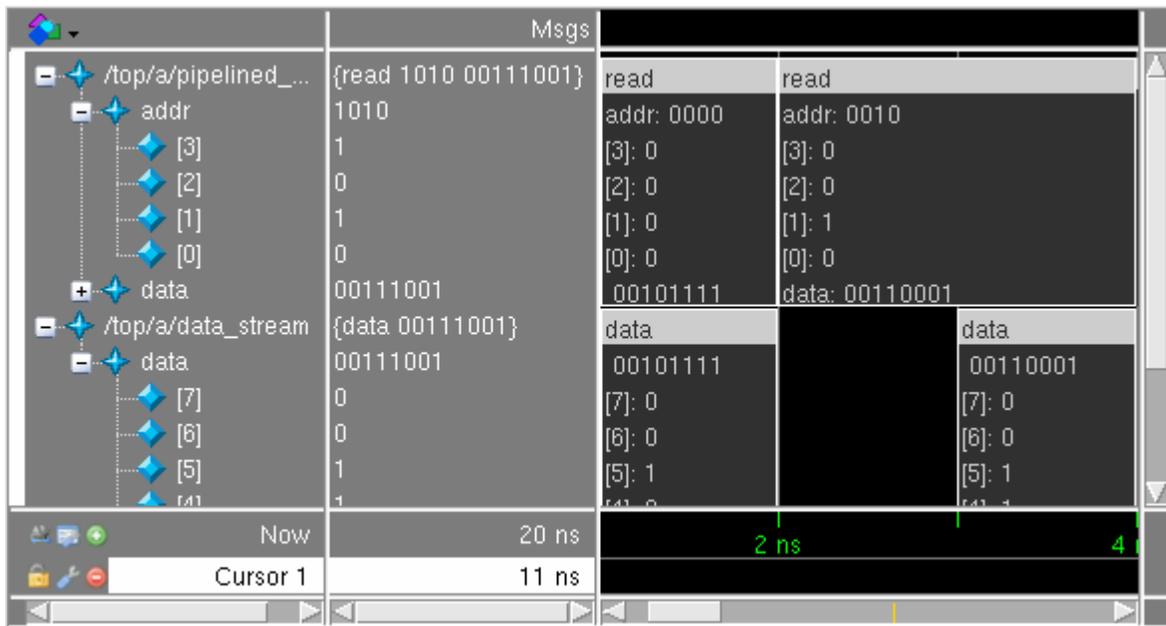
A stream has children and is expandable if:

- there have been transactions defined on it with attributes
- transactions have overlapped on the stream

Unexpanded, such a stream's value includes the names of all current active transactions, for example, "{busRetry busWrite}". The values of any attributes are available only if the stream is expanded to reveal them.

Figure 12-2 shows several streams, one of which has eight sub-streams.

Figure 12-2. Transaction Stream in Wave Window



If no transactions were defined on a particular stream, or none of the transactions have attributes, then the stream is a simple signal with the name of the current transaction as its value.

The <Inactive> value appears under the following conditions:

- When no transaction is active on a stream.
- Where an element (attribute and/or sub-stream) exists, but is not used by a transaction.

Attributes and sub-streams are additive in ModelSim. These are added to the stream's basic definition and are kept as part of the stream definition from that point onward. Even if an attribute is not used on some transaction, it remains part of the stream.

SCV allows designs to set explicit, undefined values on begin attributes and end attributes. The simulator shows these as "<Undefined>".

Transaction streams are dynamic objects under the control of the design. During simulation, the design may define new streams, define new transaction kinds, overlap transactions or create

phase transactions, add special attributes of all kinds, and so forth. In response, the simulator actively re-creates the objects in the GUI to reflect the most recent changes.

Dynamic changes are always additive: once an element is added to a stream, it remains there in all views. In post-simulation debug, all elements are shown as if they existed from the beginning of the simulation run. For both interactive and post-simulation debug, elements that did not exist at a particular simulation time are shown as if the `nolog` command had been used; their values are `"No_Data"`.

Viewing Transaction Objects in the Structure Window

Structure windows allow you to navigate through the regions in the design, much as you would use the `env` command in batch mode. Transaction objects look much like signals or nets in the design hierarchy. When you navigate to a region containing a stream, the stream is visible in the Objects window.

Viewing Transactions in the Wave Window

Prerequisite

- For transactions to be viewable in the Wave window, logging must be enabled at the simulation time when the transaction begins. Logging is automatically enabled for transactions written in SCV, Verilog, and VHDL. Logging can be disabled using the `nolog` command.

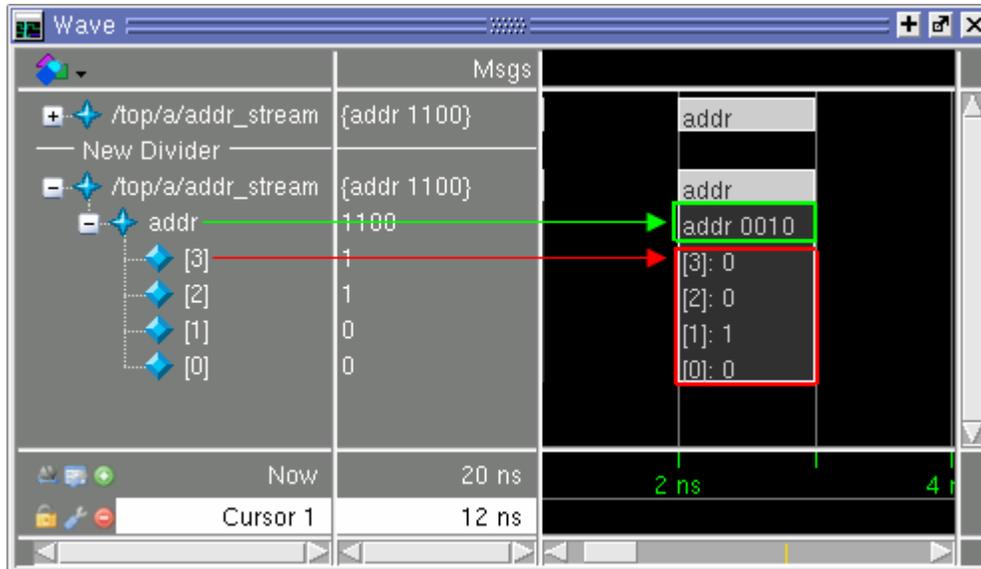
Procedure

1. Run the simulation on a design containing transactions.
`vsim top; run -all`
2. Add transactions to Wave window:
 - Drag and drop from Object or Structure (sim) windows. (The `add wave` command will be reflected in the Transcript window.)
 - Click the middle mouse button when the cursor is over a transaction.
 - Select transactions, then select **Add > To window**.
 - From the command line:
`add wave -expand top/*`
3. Select the plus icon next to streams having objects beneath them to reveal substreams and/or any attributes.

The icon for a transaction stream is a four-point star in the color of the source language for the region in which the stream is found (SystemC - green, Verilog - light blue, VHDL - dark blue).

In the waveform pane, transactions appear as boxes surrounding all the visible values for that transaction. Here's an example of a transaction on a stream with only one sub-stream where the stream is shown in its expanded and collapsed forms:

Figure 12-3. Viewing Transactions and Attributes



Each box represents an “instance” of a transaction on the stream. The horizontal line drawn between the first and second transaction indicates a period of either no activity OR a period in which logging has been disabled; there is no way to know which is the case.

When there are concurrent, parallel transactions, the stream shows concurrent values which are drawn overlapping with a vertical offset, so that each instance can be seen. Expanding the stream reveals the sub-streams, separating the transactions neatly, as in [Figure 12-4](#). Each sub-stream may expand to reveal attributes or phase sub-streams.

When you select a transaction instance, all related transactions are also highlighted, as can also be seen in [Figure 12-4](#).

Figure 12-4. Concurrent Parallel Transactions

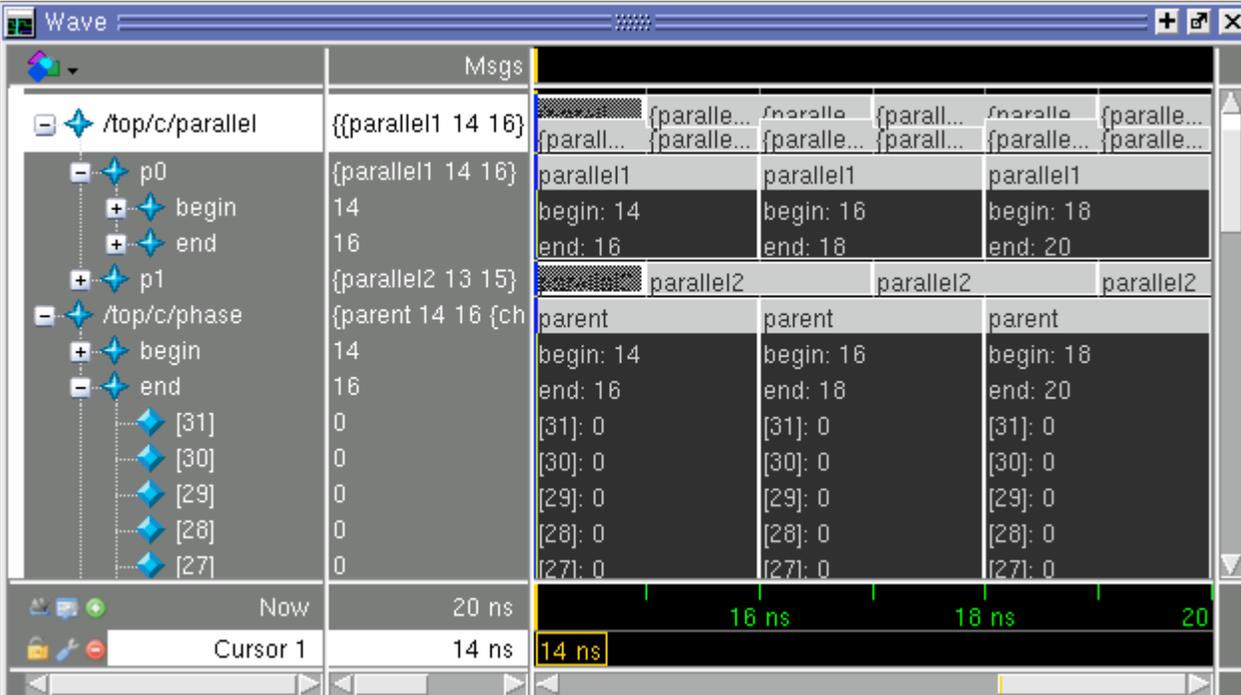
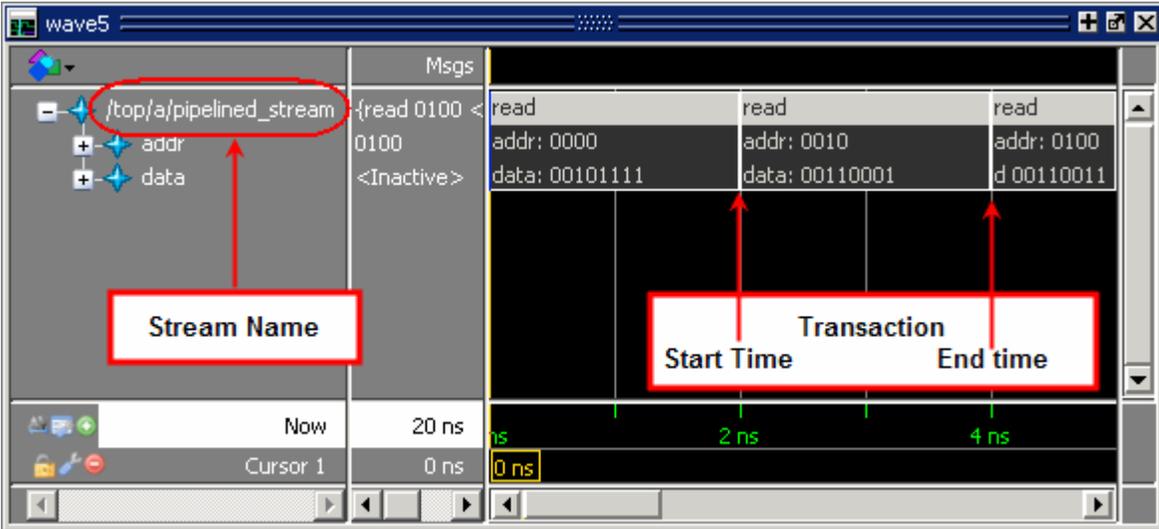


Figure 12-5 shows a simple transaction stream that includes simple, user-defined address and data attributes:

Figure 12-5. Transaction in Wave Window - Viewing



The top row of a transaction is the name of the transaction. When the transaction stream is expanded, as in Figure 12-5, additional rows are revealed that represent attributes of the transaction.



Tip: For SystemC begin/end attributes — If a begin end attribute was declared by the generator, but the value was not defined, the value appears as “Undefined” in the GUI.

Appearance of Retroactive Transactions in Wave Window

Retroactive recording refers to the recording of a transaction whose start and/or end time occurs before the current simulation time. When these transactions are drawn in the Wave window, more substreams may be shown than you might expect. This is due to the fact that even though there might be a “space” between the two transactions long enough for your retroactive transaction, the tool creates an additional substream to draw that transaction.

Logged Transactions and Retroactive Recording

A transaction is logged in the WLF file if logging is enabled at the simulation time when the design calls `::begin_transaction()` (SystemC), `$begin_transaction()` (Verilog), or `begin_transaction()` (VHDL). The effective start time of the transaction (in other words, the time passed by the design as a parameter to beginning the transaction) is irrelevant. For example, a stream could have logging disabled between T1 and T2 and still record a transaction in that period using retroactive logging after time T2. A transaction is always entirely logged or not logged.

Selecting Transactions or Streams

You can select transactions or streams with a left click of the mouse, or if the transactions are recorded with relations, you can right click for a pop-up menu for various choices.

Selecting transactions or streams with the mouse:

- Select an individual transaction: left click on the transaction. When you select a transaction, any substreams of that transaction are selected also.
Left click while holding down the SHIFT key to select multiple transactions/streams.
- Select a transaction stream: left click on the transaction name in the object name area of the Wave window.

Selecting and Viewing Related Transactions

1. Select a single transaction.
2. Right mouse click to bring up pop-up menu with the following choices:
 - Select Related — to select a transaction to which the current transaction is pointing.
 - Select Relating — to select a transaction that is pointing to the current transaction.

- **Select Chain** — to select all related and relating transactions for the current transaction. Use this to select an entire causal chain.
- **Select Meta** — to select all related and relating transactions for the current transaction. Use this to select any existing branches for the relationship.

Selecting any of these items brings up a submenu which lists all relationship names that apply.

These pop-up menu items are grayed out if more than one transaction is selected.

For information on how to record relations, see “[Specifying Relationships](#)” (SC) and “[Specifying Relationships](#)” (Verilog/VHDL).

Customizing Transaction Appearance

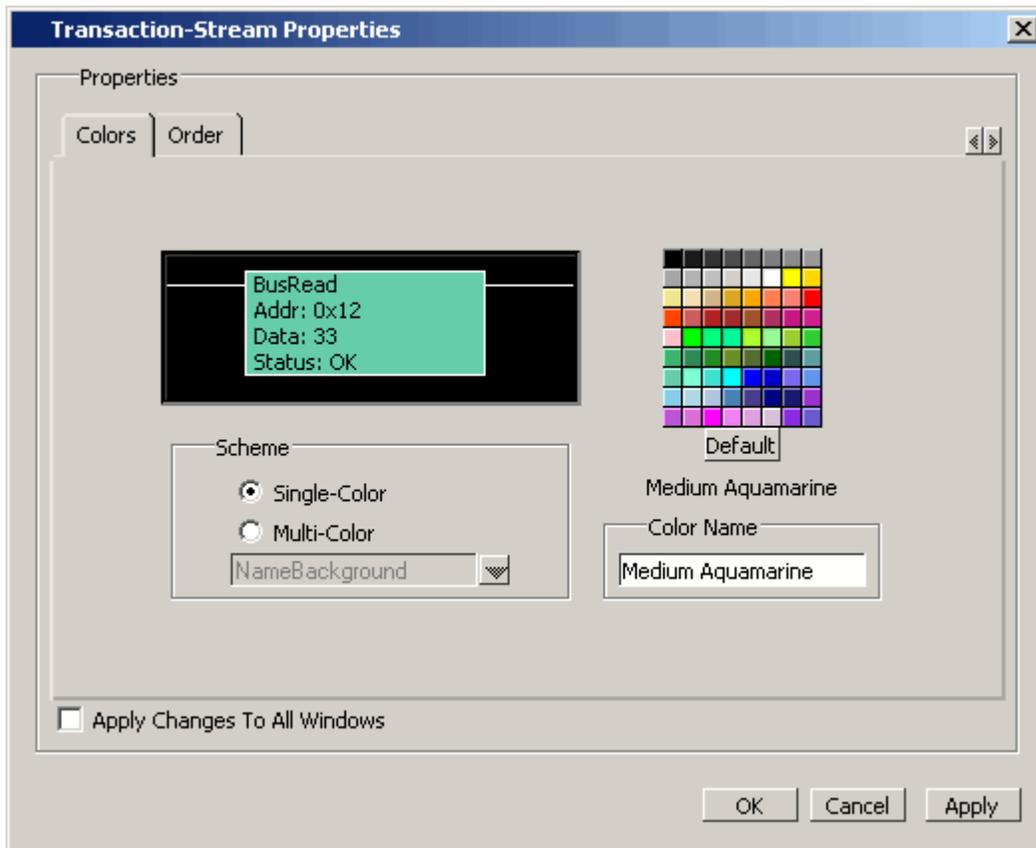
You can customize the appearance of a transaction instance, or change the look of the entire transaction stream. Further, you can apply these custom settings to either the current Wave window or all Wave windows.

Customizing Color

You can change the color of one or more transactions or streams using the GUI or the `tr color` command.

1. Right-click a transaction or stream name to open a popup menu.
2. Select Transaction Properties to open the Transaction-Stream Properties dialog box ([Figure 12-6](#)).

Figure 12-6. Transaction Stream Properties



3. Select the Colors tab.
4. In the Scheme area of the dialog box, select:
 - Single-Color to apply the color change to all elements of the stream or transaction.
 - Multi-Color to apply different colors to specific elements of the stream or transaction. When you select one of the following elements and select a color, the color is applied to that element:
 - InactiveLine - line between transactions
 - BorderLine - border around the transaction
 - NameBackground - background behind the transaction name text
 - NameText - text for the transaction name
 - AttributeBackground - background behind the attribute text
 - AttributeText - text for attribute
5. Choose a color from the palette or enter a color in the field (for example, light blue) and
6. Select **Apply** to leave dialog box open, or **OK** to apply and close it.

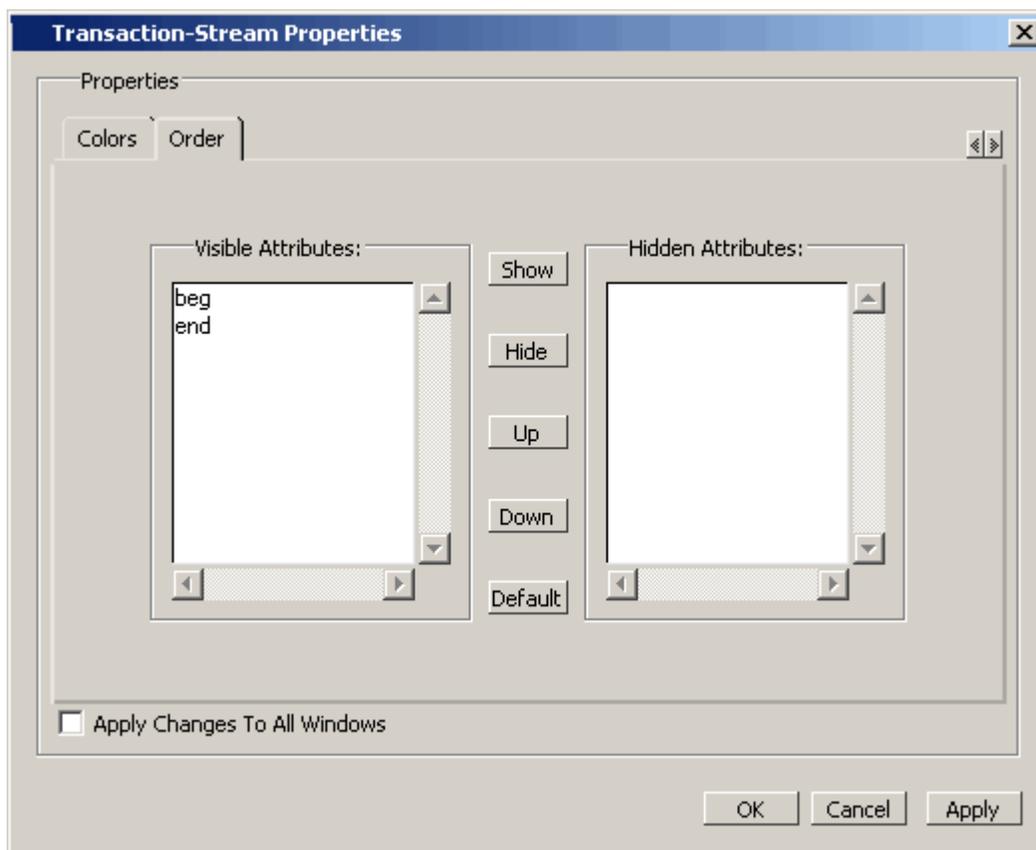
The element, transaction or entire stream of transactions changes to the chosen color.

Customizing Appearance of Attributes

You can change the order of the attributes and hide/show attributes in the stream using the GUI or the `tr order` command.

1. Right-click a transaction or stream name to open a popup menu.
2. Select Transaction Properties to open the Transaction-Stream Properties dialog box.
3. Select the Order tab (Figure 12-7).

Figure 12-7. Changing Appearance of Attributes



4. Select the attribute from the list of visible attributes and select:
 - Show - to display currently hidden attributes in stream
 - Hide - to hide attribute from view in the stream
 - Up - to move attribute up in the stream up
 - Down - to move down
 - Default - to restore original view

5. **Apply** makes the changes, leaving the dialog box open; **OK** applies the changes and exits.

Viewing a Transaction in the List Window

Prerequisites

Your design code must log the transactions for them to be visible in the GUI. See “[Recording Transactions in SystemC](#)” or “[Recording Transactions in Verilog and VHDL](#)” for instructions on transaction logging.

Procedure

1. Run simulation on a design containing transactions.
vsim top; run -all
2. Add transaction objects (transaction streams, sub-streams, attributes and attribute elements) to List window:
 - Drag and Drop from the Object or Structure (sim) windows.
 - **Select transactions**, then select the **Add Selected to Window** menu in the **Standard** toolbar > **Add to List**.
 - Menu Selection — **Add > To List**.

Results

When transactions are present in the List window, new rows are written to the List window any time a transaction's state changes. Specifically, rows are printed when a transaction starts or ends and when any attribute changes state, which can occur between time steps or deltas.

Example 12-1. Transactions in List Window

This example shows List output for a stream showing two transaction kinds. Each has a begin attribute, a special attribute and an end attribute in the style of SCV.

```
ns                               /top/abc/busMon
  delta
0 +0                               <Inactive>
1 +0                               <Inactive>
1 +0                               <Inactive>
1 +0   {busRead 1 <Inactive> <Inactive>}
3 +0       {busRead 1 100 <Inactive>}
3 +0       {busRead 1 100 10}
3 +0                               <Inactive>
4 +0                               <Inactive>
4 +0                               <Inactive>
4 +0                               <Inactive>
4 +0   {busWrite 2 <Inactive> <Inactive>}
6 +0       {busWrite 2 200 <Inactive>}
6 +0       {busWrite 2 200 20}
```

```
6 +0 <Inactive>
7 +0 {busRead 3 <Inactive> <Inactive>}
9 +0 {busRead 3 300 <Inactive>}
9 +0 {busRead 3 300 30}
9 +0 <Inactive>
```

In this example, you can see the same time/delta repeating as changes are made to the transaction. For example, at 1(0) a busRead begins with the begin attribute set to the value "1". At time 3(0), the end attribute value "100" arrives. On the next line, also at time 3(0), the special attribute's value of "10" arrives. On the next line the transaction has ended. This is followed by a number of lines showing the "<Inactive>" state as the various attributes change state internally.

Viewing a Transaction in the Objects Window

Prerequisites

- Running a design in which transactions have been written.

Procedure

1. Run simulation on a design containing transactions.

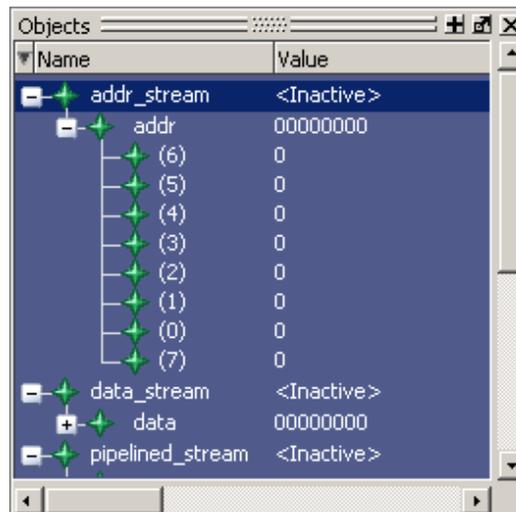
vsim top; run -all

2. Open the Objects window, if not open by default: View > Objects.

Results

Streams appear in the Objects window as simple or composite signals, depending on the complexity of transactions that have been defined for the stream. The icon for a transaction stream is a four pointed star in the color of source language for the region in which the stream is found (SystemC - green, Verilog - light blue, VHDL - dark blue).

Figure 12-8. Transactions in Objects Window



Debugging with Tcl

In order to access attributes of transactions for debugging, you must know the full path to the specific stream, substream or attribute you wish to access. After you set the path to the transaction, you can use Tcl to analyze your transactions. A full path (using the Verilog path delimiter) to an attribute would be:

<stream>.<substream>[<substream...>].<attribute>

An example of setting the path to the attribute of a transaction is as follows:

1. Set the names of streams created using TCL. For example:

```
set streamName "top.stream1"
```

ModelSim generates the names of sub streams. The name is the first character of the parent stream's name followed by a number. Sub-stream numbering starts at zero.

```
set subStream "s0"
```

2. Set the attribute name you want to access, such as:

```
set attributeName "myInteger"
```

Once you have set the stream, substream, and attribute names, you can access the variable value at any specified time. A sample examine command using the attribute in the above example might be:

```
exa -t 30 $streamName.$subStream.$attributeName
```

You can place commands such as these into a Tcl script and use it to parse the WLF database.

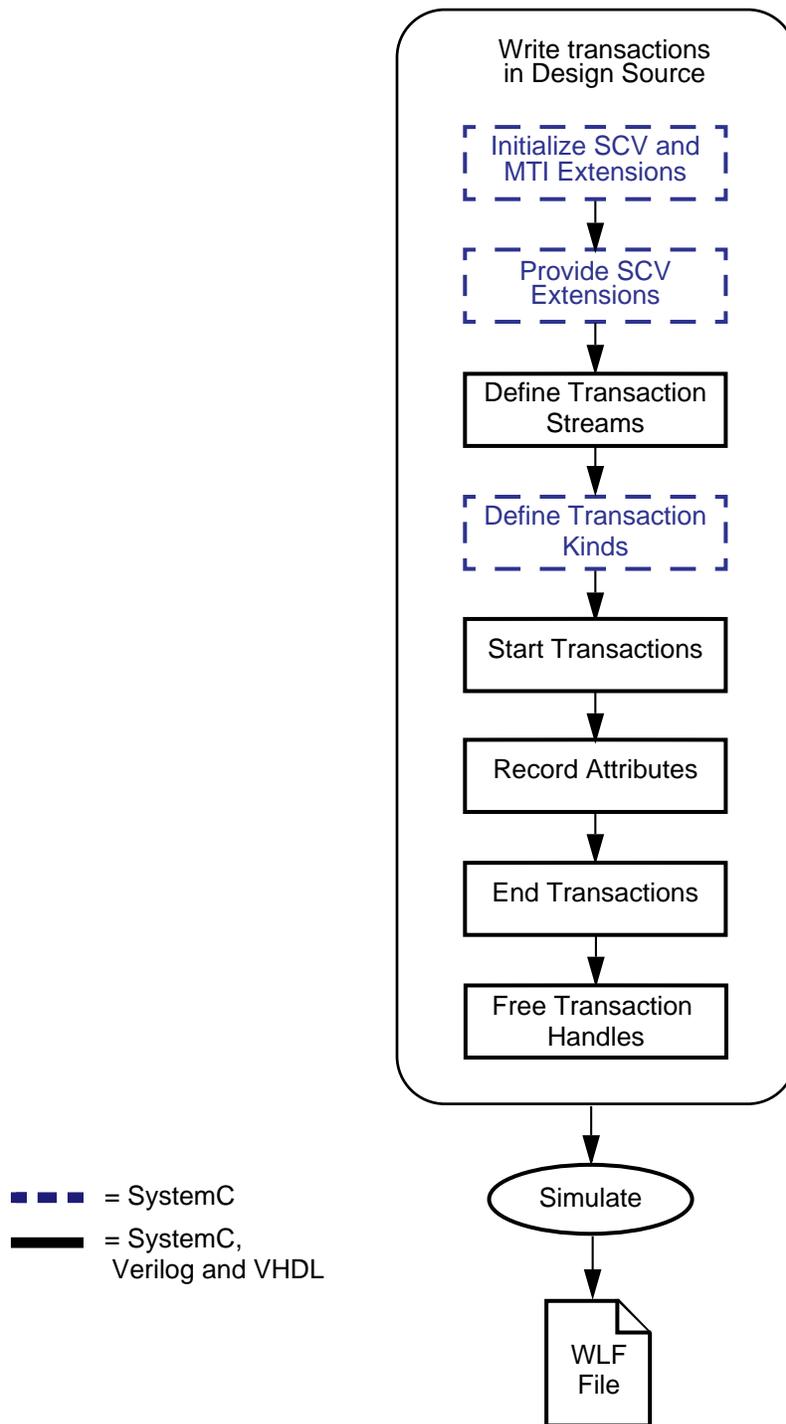
Transactions in Designs with Questa Verification IP

See “[Questa Verification IP Transaction Viewing in the GUI](#)” for details on viewing Questa Verification IP component transactions.

Transaction Recording Flow

The basic steps for recording transactions can be summarized in [Figure 12-9](#).

Figure 12-9. Recording Transactions



The SystemC tasks and Verilog API calls used in these steps are listed in [Table 12-1](#).

Table 12-1. System Tasks and API for Recording Transactions

Action	SystemC - System Task Used	Verilog/VHDL - ModelSim API Used
Initializing SCV and MTI extensions — Required - SCV only. See Initializing SCV and Creating WLF Database Object	scv_startup()	N/A
Creating database tied to WLF — Required for SCV only. See Initializing SCV and Creating WLF Database Object	scv_tr_db() scv_tr_wlf_init()	N/A
Provide SCV extensions — Required for SCV only. ¹ See Creating Transaction Generators	scv_tr_stream()	N/A
Define transaction streams — Required. SC - See Defining a transaction stream V - See Defining a transaction stream	scv_tr_generator()	create_transaction_stream
Define transaction kinds — Required for SVC only. See Defining a transaction kind (This step is Optional.)	See SCV documentation	N/A
Start the transaction — Required. SC - See Starting a Transaction V - See Starting a Transaction	::begin_transaction()	begin_transaction
Record attributes — Optional. SC - See Recording Special Attributes V - See Recording an Attribute	::record_attribute()	add_attribute
End the transaction — Required. SC - See Ending a Transaction V - See Ending a Transaction	::end_transaction()	end_transaction
Free the transaction handle — Required for Verilog and VHDL. SC - See Freeing a Transaction Handle V - See Freeing the Transaction Handle	Freed automatically when transaction goes out of scope	free_transaction
Specify relationships between transactions — Optional. SC - See Specifying Relationships V - See Specifying Relationships	::add_relation()	add_relation or begin_transaction

Table 12-1. System Tasks and API for Recording Transactions

Action	SystemC - System Task Used	Verilog/VHDL - ModelSim API Used
Specify begin and/or end times of transactions — Optional. SC - See Specifying Transaction Start and End Times V - See Specifying Transaction Start and End Times	::begin_transaction() and ::end_transaction()	begin_transaction and end_transaction
Control database logging — Optional. See Stream Logging	log / nolog and ::set_recording()	log / nolog
Simulate the design — Required in order to view transactions.	vsim <top>	vsim <top>

1. Required only for SCV user-defined types used as attributes.

Transaction Recording Guidelines

This section outlines the rules and guidelines that apply to all transaction recording, regardless of the language in which the transaction is recorded. Read these guidelines prior to recording transactions for a general understanding of recording transactions for viewing in ModelSim:

Names of Streams and Substreams	661
Stream Logging	661
Transaction UIDs	662
Attribute Type	662
Anonymous Attributes	663
Multiple Uses of the Same Attribute	662
Definition of Relationship in Transactions	663
The Life-cycle of a Transaction	664
Retroactive Recording / Start and End Times	664
Start and End Times for Phase Transactions	665
Transaction Handles and Memory Leaks	665

For language-specific instructions and deviations from these general truths, see:

- [Recording Transactions in Verilog and VHDL](#)
- [Recording Transactions in SystemC](#)

For SCV specific limitations and implementation details, see “[SCV Limitations](#)”.

Limitations

- The checkpoint/restore commands are not supported for transactions.

- Transactions appearing in a *.wlf* file that was created with 6.3 are not viewable in later versions. You must rerun the simulator in version 6.4 or above for them to be viewable.

For SCV specific limitations and implementation details, see [SCV Limitations](#).

Names of Streams and Substreams

i **Tip: Important:** A space in any name (whether a database, stream, transaction, or attribute) requires the name be enclosed by escaped or extended identifiers.

You must provide a name for streams so that they can be referenced for debug. Anonymous streams are not allowed. Stream names may be any legal C, Verilog or VHDL identifier. If the name includes white-space or is not a legal C identifier, it should be an escaped or extended identifier or you will get a warning at run time.

Note

 The simulator issues a warning for a non-standard name.

Full or Relative Pathnames

When creating a transaction stream, you can specify a full path to the region in which you want the transaction stream to appear. Assuming that the specified path leads to an existing region in the design hierarchy, that request is honored. No regions are created in the process.

You can also specify a relative path, using the tool to search upward from the calling region in the hierarchy as determined automatically by the simulator. The region where a transaction stream appears is determined according to the placement of the call to create the stream (`$create_transaction_stream`, `create_transaction_stream` or `scv_tr_stream`). For most designs, the transaction stream is placed just where you would want it to be. However, for some SystemVerilog and OVM class-based designs, the placement may not be appropriate. In these cases, full path specifications are usually safest.

Substream Names

The tool names substreams automatically. The name of any substream is the first character of the parent's name followed by a simple index number. The first substream has the index zero. If the parent stream has a non-standard name, such as one that starts with a numeral or a space, you may have difficulty with debug.

Stream Logging

By default, when your design creates a stream, logging is enabled for that stream providing that the logging is enabled at the simulation time when the design calls `::begin_transaction()`. The

effective start time of the transaction (the time passed by the design as a parameter to `::begin transaction()`) does not matter. For example, a stream could have logging disabled between T1 and T2 and still record a transaction in that period using retroactive logging after time T2. A transaction is always entirely logged or entirely ignored. You can disable the logging on transaction streams with the `nolog` command.

There is no way in the simulator to distinguish a stream whose logging has been disabled from one that is merely inactive.

Transaction UIDs

Each transaction, when created during simulation, is assigned a 64-bit serial number. This serial number, along with the logical name of the dataset in which the transaction exists, comprises the transaction's UID (unique identifier). Within the simulation run, this number is unique.

The `"tr uid"` and `"tr color"` commands use the UID to specify a specific transaction within a particular dataset in which it exists. UIDs also allow for any transaction to refer to any other transaction.

Use examples:

```
tr color -nametext "light blue" {sim 10023}
```

```
tr color -namebg red {myData 209832}
```

The first example represents a transaction in the current simulation, since `"sim"` is always the name of the current simulation dataset. The second example is a transaction from a WLF file opened with the logical name `"myData"`.

Attribute Type

On any single stream, ModelSim associates an attribute name with a data type. Attempts to overload an attribute name are not recommended; in most cases, ModelSim will issue an error. The only exception is that the same attribute name on two different sub-streams of a stream may be overloaded.

Multiple Uses of the Same Attribute

There is nothing to prevent your design from setting the same attribute (same type) many times during the transaction. However, ModelSim records only the last value of the attribute prior to the end of the transaction.

Once any attribute is used, it is considered an attribute of the parent stream from that time onward. Thus, it shows up as a parameter on all subsequent transactions, even if it is unused.

Anonymous Attributes

ModelSim requires every transaction attribute to have a name. It is possible to neglect the name in the SCV and Verilog/VHDL APIs for transaction recording, however. The simulator resolves the problem by inventing a name for the attribute.

SCV — an attribute is anonymous if the name is the empty string or the name is a NULL pointer. The simulator uses the data type to choose a new name as follows:

- If the type is a struct or class, the simulator constructs an attribute for each field or member, using the field or member name as the name of each attribute.
- If the type is anything other than a string or class, the simulator uses the type name (for example, "short", "float", etc.) as the name for the attribute.

Verilog/VHDL — an attribute is anonymous if the name parameter is ignored or is an empty string. The simulator chooses a name as follows:

- If the value of the attribute is passed through a variable, the simulator uses the name of the variable as the name for the attribute.
- If the value of the attribute is passed as a literal or the return value from a function, the simulator uses the type name of the value as the name for the attribute.

In any language, if the simulator finds an attribute already exists with the same name and type as the one it is creating, it will re-use that attribute.

Definition of Relationship in Transactions

In ModelSim, a relationship is simply a pointer from one instance in the design to another. It consists of a source transaction, a target transaction, and a name for the relationship. It can read as “<source> has the <name> relationship to <target>”. The name you assign to the relationship is arbitrary: choose a name that is meaningful to you. ModelSim interprets NO meaning from the pointer.

When ModelSim simulates the design, it records the relationship — both from the source to the target and the target to the source — in the database so it is available for transaction debug and analysis.

In the Verilog example of:

```
...  
$add_relation(hSrc, hTgt, "child");
```

the relationship is created for hSrc such that “hSrc” claims the child relationship to “hTgt”. When this relationship is recorded, a counter relationship is automatically recorded on “hTgt” to indicate that “hSrc” is claiming the child relationship with “hTgt”.

For more information on how to record a relationship, see “[Specifying Relationships](#)” (SystemC) and “[Specifying Relationships](#)” (Verilog/VHDL). For instructions on viewing related transactions, see “[Selecting and Viewing Related Transactions](#)”.

The Life-cycle of a Transaction

Any transaction has a life-cycle that can be summarized in four distinct phases (using the Verilog API as an example):

- **Creation** — This is the moment in simulation when a design calls `$begin_transaction()` or an equivalent method.
- **Start time** — Usually, the start time is determined by the call to `$begin_transaction()`, except in the case of retroactive recording, when the start time is set earlier than the creation time.
- **End time** — Typically determined by a call to `$end_transaction()`, though it, too, can be adjusted.
- **End-of-life** — The moment in simulation when the design releases its handle to the transaction, or the transaction has been deleted (Verilog/VHDL only). No more additions or changes can be made to the transaction past this point.

This life-cycle has several important implications:

- Attributes and relations can be added during the entire life-cycle, not just between the start and end times for the transaction, so long as you have a valid handle to the transaction. A valid handle is one whose returned value is non-zero. See “[Valid Handles](#)” for information about how to detect errors.
- You can enable or disable logging of transactions anytime during the life-cycle, regardless of start and end times.
- A transaction stays in memory until its handle is released. Transaction handles should be freed as soon as possible, to minimize use of memory buffering and the retroactive WLF channels. Verilog and VHDL designs must use the `free_transaction()` task explicitly for every transaction.

Retroactive Recording / Start and End Times

The only time you must specify start and end times for a transaction is when you are recording a transaction retroactively. For all other transaction types, the simulator knows the start and end times. It is illegal to start or end a transaction in the future or before time 0. If either is specified, the simulator uses the current simulation time, and issues a non-fatal error message.

Start and End Times for Phase Transactions

The start and end times for phase transactions must be entirely within the timespan of the parent transaction. Start and end times of phases can match the start or end times of parent.

Transaction Handles and Memory Leaks

When the design calls `begin_transaction()`, the transaction handle is stored in memory and remains in memory until it is freed. In Verilog and VHDL, you must free the transaction handle explicitly using `$free_transaction()` or `free_transaction()`, respectively. In SCV, the handle is freed for you when the handle goes out of scope. However, global or static transactions remain in memory for the entire simulation.

Though this memory loss may be more accurately described as “usage” rather than a “leak”, it is wasteful to use memory for transactions no longer in use. You should write your code in such a way as to free transaction handles once they are not needed.

Transaction Recording Procedures

The procedures for recording transactions in Verilog and VHDL are much simpler than those for SystemC, so they are presented first to simplify learning about the recording process.

Recording Transactions in Verilog and VHDL	665
Recording Transactions in SystemC	669

Recording Transactions in Verilog and VHDL

As there is not yet a standard for transaction recording in Verilog or VHDL, ModelSim includes a set of system tasks to perform transaction recording into a WLF file. See [Verilog and VHDL API System Task Reference](#) for specific tasks used to record the transactions.

The API is the same for Verilog and SystemVerilog. As stated previously, the name "Verilog" refers both to Verilog and SystemVerilog unless otherwise noted.

The recording APIs for Verilog and VHDL are a bit simpler than the SCV API. Specifically, in Verilog and VHDL:

- There is no database object as there is in SCV; the database is always WLF format (a *.wlf* file).
- There is no concept of begin and end attributes All attributes are recorded with the system task `$add_attribute()` or `add_attribute`.

- Your design code must free the transaction handle once the transaction is complete and all use of the handle for relations or attribute recording is complete. (In most cases, SystemC designs ignore this step since SCV frees the handle automatically.)

For a full example of recorded Verilog transactions with comments, see [Verilog API Code Example](#).

Prerequisites

- Understand the rules governing transaction recording. See the section entitled [Transaction Recording Guidelines](#).
- For VHDL, the design must include the transaction recording package supplied with ModelSim. You can find this in the `modelsim_lib` library.

```
library modelsim_lib;  
use modelsim_lib.transactions.all;
```

Procedure

This procedure is based on the Verilog API. The VHDL API is very similar.

1. Defining a transaction stream

Use `$create_transaction_stream()` to create one or more stream objects.

```
module top;  
    integer hStream  
  
    initial begin  
        hStream = $create_transaction_stream("stream", "transaction");  
        .  
        .  
    end  
    .  
    .  
endmodule
```

This example code declares the stream *stream* in the current module. The stream is part of the WLF database and the stream will appear as an object in the GUI. The stream will be logged.

In some OVM or other class-based designs, you may want to specify stream a full path to the location where you wish to the stream to appear. See [“Full or Relative Pathnames”](#) for more information.

2. Starting a Transaction

Use `$begin_transaction`, providing:

- a valid handle to a transaction stream
- a variable to hold the handle of the transaction itself

```
integer hTrans;
```

```
.  
.  
hTrans = $begin_transaction(hstream, "READ");
```

In this example, we begin a transaction named "READ" on the stream already created. The `$begin_transaction` system function accepts other parameters to specify: the start time for the transaction, and any relationship information, including its designation as a phase transaction (see “[Phase / Child Transactions](#)”). See [Verilog and VHDL API System Task Reference](#) for syntax details.

The return value is the handle for the transaction. It is needed to end the transaction or record attribute.

3. Recording an Attribute

Optional. Use the `$add_attribute` system task and provide:

- the handle of the transaction being recorded
- the name for the attribute
- a variable that holds the attribute value

```
integer address;  
.  
.  
$add_attribute(hTrans, address, "addr");
```

Be aware that nothing prevents the design from setting the same attribute many times during the transaction. However, ModelSim records only the last value of the attribute prior to the end of the transaction. Once the design uses an attribute, it becomes a permanent attribute of the parent stream from that time onward. Thus, it shows up as an element of all subsequent transactions, even if it is unused.

4. Ending a Transaction

Submit a call to `$end_transaction` and provide the handle of the transaction:

```
$end_transaction(hTrans);
```

This ends the specified transaction, though it does not invalidate the transaction handle. The handle is still valid for calls to record attributes and to define relations between transactions. As with `$begin_transaction()`, there are optional parameters for this system task. See [Verilog and VHDL API System Task Reference](#) for details.

5. Specifying Relationships

See “[Definition of Relationship in Transactions](#)”. To specify a relationship between transactions, you must provide:

- two valid transaction handles: one for the source, one for the target
- a <name> for the relation (one signifying the relationship of the <source> to the <target>). ModelSim captures the name and uses it to record to pointers, one

from the source instance to the target instance, and one from the target to the source. In the examples below, the name chosen to represent the relationship is “successor”.

- Specify relation from an existing transaction to another existing transaction:

Submit a call to `$add_relation()`, with the source, target and name:

```
integer hSrc;  
integer hTgt;  
.  
.  
$add_relation(hSrc, hTgt, "successor");
```

This method is valid any time the design has two valid transaction handles.

See “[Definition of Relationship in Transactions](#)” and “[Selecting and Viewing Related Transactions](#)” for more information.

6. Specifying Transaction Start and End Times

To specify a start and/or end time for any transaction, pass the start and end times as parameters to `$begin_transaction()` and `$end_transaction()`. The time must be the current simulation time or earlier. See “[Transaction Recording Guidelines](#)” for information on valid start and end times.

7. Freeing the Transaction Handle

i **Tip: To avoid memory leakage:** You must explicitly free all transaction handles in your design. This is a requirement for Verilog, SystemVerilog and VHDL) recording. See “[Transaction Handles and Memory Leaks](#)”.

- a. Ensure that the transaction is complete AND all use of the handle for recording attributes and relations has been completed.
- b. Submit a call to `$free_transaction`, providing the handle of the transaction being freed.

```
$free_transaction(hTrans);
```

where *hTrans* is the name of the transaction handle to be freed.

8. Deleting a Transaction

To delete a transaction, pass a valid transaction handle as the parameter to `$delete_transaction()`. This removes the specified transaction from the transaction database before it is written to the WLF file. If the transaction was visible in the Wave window (or elsewhere), it vanishes as if it never existed.

```
$delete_transaction(hTrans);
```

where *hTrans* is the handle of the transaction being deleted.

Example 12-2. Verilog API Code Example

This example is distributed with ModelSim and can be found in the `<install_dir>/examples/systemverilog/transactions/simple` directory.

```
module top;
    integer stream, tr;

    initial begin
        stream = $create_transaction_stream("Stream");
        #10;
        tr = $begin_transaction(stream, "Tran1");
        $add_attribute(tr, 10, "beg");
        $add_attribute(tr, 12, "special");
        $add_attribute(tr, 14, "end");
        #4;
        $end_transaction(tr);
        $free_transaction(tr);
    end

endmodule
```

Valid Handles

If there is an error on a call to `create_transaction_stream()` or `begin_transaction()`, the handle returned will have the value zero.

Recording Transactions in SystemC

SystemC users use the SCV library's transaction recording API routines to define transactions, to start them, to end them, to create relationships between them, and to attach additional information (attributes) to them. These routines are described in the "SystemC Verification Standard Specification, Version 1.0x": please refer to it for SCV specific details.

The SCV API is a bit more involved than the Verilog or VHDL recording APIs. Specific differences for the SCV API are as follow:

- In SCV, you must create a database object that is tied to a WLF file.
- The concept of begin and end attributes is unique to SCV. In Verilog and VHDL, all attributes are recorded with a single system task: `add_attribute()`.
- Transaction handles are freed automatically in SCV.

For a full example of recorded SCV transactions with comments, see "[SCV API Code Example](#)".

Prerequisites

- Understand the material in the section entitled "[Transaction Recording Guidelines](#)" to understand the basic rules and guidelines for recording transactions.

- Be aware of the limitations for recording transactions in SCV. See “[SCV Limitations](#)”.

Procedure

1. Initialize SCV and the MTI extensions for transaction recording and debug.
2. Create a database tied to WLF.
3. Provide SCV extensions, for user-defined types used with attributes.
4. Create transaction generators.
5. Write the transactions.

Initializing SCV and Creating WLF Database Object

Before transactions can be recorded, the design must initialize the SCV library once, as part of its own initialization.

1. Enter **scv_startup()** in the design code — this initializes the SCV library.
2. Enter **scv_tr_wlf_init()** in the design code — this creates the database tied to WLF, allowing transactions to then be written to specific database objects in the code.
3. Enter database object(s) — you can create many objects, or create one and specify it as the default object. All database objects are contained the same WLF file.

Example 12-3. SCV Initialization and WLF Database Creation

Here is an example of a one-time initialization routine that sets up SCV, ties all databases to WLF, and then creates one database as the default:

```
static scv_tr_db * init_recording() {
    scv_tr_db *txdb;

    /* Initialize SCV: */
    scv_startup();

    /* Tie databases to WLF: */
    scv_tr_wlf_init();

    /* Create the new DB and make it the default: */
    txdb = new scv_tr_db("txdb");

    if (txdb != NULL)
        scv_tr_db::set_default_db(txdb);

    return txdb;
}
```

ModelSim ignores the following:

- name argument to **scv_tr_db()** — All databases are tied to the WLF file once the user calls **scv_tr_wlf_init()**.

- `sc_time_unit` argument to `scv_tr_db()` when the database is a WLF database — The time unit of the database is specified by the overall simulation time unit.

Creating Transaction Generators

If your design uses standard C and SystemC types for attributes, no preparation is needed with SCV. C/C++ and SystemC types are supported as described in [Type Support](#). However, if your design uses user-defined types — such as classes, structures, or enumerations — you must provide SCV extensions so that SCV and the ModelSim tool can extract the necessary type and composition information to record the type.

For specific details on providing SCV extensions, refer to the SystemC Verification Standard Specification, Version 1.0e.

Writing SCV Transactions

i **Tip: Important:** A space in any name (whether a database, stream, transaction, or attribute) requires the name be an escaped or extended identifier.

1. Defining a transaction stream

Before you can record a transaction, you must define the stream onto which the transaction will be written. In SCV, streams are tied to a specific database so that all transactions on them are written into that database only. Usually, the code declares the stream as a member of the module that will use it. Then, it must call the constructor, passing the stream's name and database as parameters. For example:

```
SC_MODULE (busModel)
{
    public:
        scv_tr_db *txdb;
        scv_tr_stream busStream;
        SC_CTOR (busModel) :
            txdb (init_recording()),
            busStream ( "busModel", " **TRANSACTOR** ")
        {
        }
}
```

This example code declares the database and stream objects. In the module constructor, it initializes the database by calling the setup routine (defined in [“Initializing SCV and Creating WLF Database Object”](#)). It initializes the stream object with its display name and a string indicating the stream kind. The database is presumed to be the default, though the example could have been explicit and passed "txdb" as a third parameter.

The name of the stream must be passed as a parameter. ModelSim treats it as a path name. This defines where the stream will appear in the design during debug. Each

stream lives in a design region: either the instance in which it was declared or an instance specified in the constructor parameters.

If the string is a simple name such as "busRead", ModelSim assumes the stream is to be created in the current scope, usually the instance of the module. If the string is a partial path such as "dut/bus/busRead", ModelSim will try to find parent region "dut/bus" as the home for the stream. If the string is a full path, such as "/top/dut/bus/busRead", ModelSim tries to find the exact region "/top/dut/bus". For full and partial paths, the region specified must exist or you will receive a runtime error.

If you specify a stream that already exists, returns a handle to the same stream even though the design will have two different scv_tr_stream objects.

For more specific details on writing a transaction, refer to the "SystemC Verification Standard Specification, Version 1.0e".

2. Defining a transaction kind (This step is Optional.)

In SCV, each transaction is defined by a generator object, which is a template for a transaction. The generator:

- Specifies the name of the transaction. Anonymous transactions are not allowed.
- Specifies optional begin and end attributes. Begin and end attributes are part of the generator for that kind. They are treated as part of each instance of that transaction.

First, the code must declare each generator, usually in the module in which it will be used. Any begin and end attribute types must be provided as template parameters. Then, the generator must be constructed, usually in the constructor initialization list of the parent module:

```
SC_MODULE(busModel)
{
    public:
        scv_tr_db *txdb;
        scv_tr_stream busStream;
        scv_tr_generator<busAddrAttr, busDataAttr> busRead;
        SC_CTOR(busModel) :
            txdb(init_recording()),
            busStream("busModel", "**TRANSACTION**"),
            busRead("busRead", busStream)
        {
        }
}
```

The third and fourth arguments to `scv_tr_generator::scv_tr_generator()` are the names for the begin and end attributes, which SCV allows to be NULL by default. (Any attributes you create with an empty string or NULL pointer for the name are called *anonymous attributes*. For more information on how these are treated by ModelSim, see "[Anonymous Attributes](#)".)

The example above adds the declaration of the generator "busRead" and shows how the constructor is provided with both the name (also "busRead") and the stream on which the generator will be used. The begin and end attributes are left anonymous.

3. Starting a Transaction

- a. Prepare the value for the begin attribute.
- b. Call **scv_tr_generator::begin_transaction()** with the appropriate parameters as defined in the SCV API.
- c. Optional — You can apply additional parameters to specify relationships, or to specify a begin time other than the current simulation time. For more information, see [Recording Phase Transactions](#) and [Specifying Transaction Start and End Times](#).

Example:

```
scv_tr_handle txh;  
busAddrAttr busAddr;  
busAddr._addr = 0x00FC01;  
txh = busRead.begin_transaction(busAddr);
```

In this example, only the begin attribute "busAddr" is passed to **::begin_transaction()**. Other parameters may be used to specify relationships or specify a begin time other than the current simulation time. (See ["Definition of Relationship in Transactions"](#) and ["Retroactive Recording / Start and End Times"](#).)

4. Recording Special Attributes

Special attributes are not part of the original transaction generator: they are afterthoughts. Record special attributes by:

- a. Define the attribute type.
- b. Modify a specific transaction instance through the transaction handle using the **scv_tr_handle::record_attribute()** routine.

Example:

```
if (status != BUS_OK) {  
    errorAttr err;  
    err.code = status;  
    txh.record_attribute(err);  
}
```

Nothing prevents a design from setting the same special attribute many times during the transaction. However, the ModelSim simulator records only the last value of the attribute prior to the end of the transaction.

For greater detail on recording special attributes, refer to the SystemC Verification Standard Specification, Version 1.0e.

5. Recording Phase Transactions

Phase transactions are unique to ModelSim. If recorded, they appear as transactions within their parent transaction. The SCV specification does not describe this kind of transaction, but ModelSim can record it. Any transaction may have phases, including another phase transaction. To record phase transactions:

- a. Specify **mti_phase** as the relation name in a call to **::begin_transaction()**.

You can also specify your own relation name for phases by modifying the value of the variable [ScvPhaseRelationName](#) in the *modelsim.ini* from “mti_phase” to something else, such as “child”. This variable applies to recording only; once a phase is recorded in a WLF file, it is drawn as a phase, regardless of the setting of this variable.

- b. Provide an appropriate parent transaction handle in a call to **::begin_transaction()**.

6. Specifying Transaction Start and End Times

To specify a start and/or end time for any transaction, pass the start and end times as parameters to **::begin_transaction()** and **::end_transaction()**. The time must be the current simulation time or earlier. See [“Retroactive Recording / Start and End Times”](#) and [“Start and End Times for Phase Transactions”](#).

7. Ending a Transaction

To end transactions in your SystemC code:

- a. Set the value for the end attribute.
- b. Call **scv_tr_generator::end_transaction()**, specifying any end attributes or other appropriate parameters as defined in the SCV API.
- c. Optional — You can specify an end time other than the current simulation time. For more information, see [Specifying Transaction Start and End Times](#).

8. Freeing a Transaction Handle

Transaction handles are freed automatically in SCV when the transaction handle goes out of scope or when the transaction handle is reassigned. Global or static transactions remain in memory for the duration of the simulation. See [“Transaction Handles and Memory Leaks”](#) for details.

9. Specifying Relationships

Provide:

- two valid transaction handles: one for the source, one for the target
- the <name> of the relation (that is, the relationship of the <source> to the <target>)
- Specify relation within an existing transaction using **scv_tr_handle::add_relation()** methods. For example:

```
scv_tr_handle prev_txx;  
scv_tr_handle txx;  
busAddrAttr busAddr;  
busAddr._addr = 0x00FC01;  
txx = busRead.begin_transaction(busAddr);  
txx.add_relation("successor", prev_txx);
```

- Specify relation for a new transaction by creating a relationship to a target transaction when it begins the source transaction. For example:

```
scv_tr_handle prev_txx;  
scv_tr_handle txx;  
busAddrAttr busAddr;  
busAddr._addr = 0x00FC01;  
txx = busRead.begin_transaction(busAddr, "successor", prev_txx);
```

In both these examples, the design specifies that the current transaction is a “successor” to the previous transaction.

Example 12-4. SCV API Code Example

This example is distributed with ModelSim and can be found in the *install_dir/examples/systemc/transactions/simple* directory.

```
#include <systemc.h>  
#include <scv.h>  
  
typedef scv_tr_generator<int, int> generator;  
  
SC_MODULE(tx)  
{  
    public:  
    scv_tr_db      *txdb;          /* a handle to a transaction database */  
    scv_tr_stream *stream;        /* a handle to a transaction stream */  
    generator      *gen;          /* a handle to a transaction generator */  
  
    SC_CTOR(tx)  
    {  
        SC_THREAD(initialize);  
        SC_THREAD(thread);  
    }  
  
    /* initialize transaction recording, create one new transaction */  
    /* database and one new transaction stream */  
    void initialize(void) {  
        scv_startup();  
        scv_tr_wlf_init();  
        txdb = new scv_tr_db("txdb");  
        stream = new scv_tr_stream("Stream", "*** TRANSACTOR **", txdb);  
    }  
  
    /* create one new transaction */  
    void thread(void) {  
        scv_tr_handle trh;  
  
        gen = new generator("Generator", *stream, "begin", "end");
```

```
        wait(10, SC_NS);                /* Idle period      */
        trh = gen->begin_transaction(10); /* Start a transaction */
        wait(2, SC_NS);
        trh.record_attribute("special", 12); /* Add an attribute */
        wait(2, SC_NS);
        gen->end_transaction(trh, 14);    /* End a transaction */
        wait(2, SC_NS);                /* Idle period      */
    }
};

SC_MODULE(top)
{
    public:
        tx *a;
        SC_CTOR(top)
        {
            a = new tx("tx");
        }
};

SC_MODULE_EXPORT(top);
```

SCV Limitations

You can record transactions in only one WLF file at a time. The SCV API routines allow you to create and use multiple databases, however — if the chosen database is WLF — all databases are aliased to the same WLF file. Once created, you may load multiple WLF files that contain transactions into ModelSim for viewing and debugging.

Type Support

The following types are not supported for SystemC transactions:

- bit-field and T* (pointer) native C/C++ types
- SystemC fixed point types (sc_fix, sc_fix_fast, sc_fixed, sc_fixed_fast, sc_ufix, sc_ufix_fast, sc_ufixed, sc_ufixed_fast) are not supported.

CLI Debugging Command Reference

A list of CLI commands available for debugging your transactions are as follow:

add list	examine	search
add wave	find	show
context	left	up
dataset clear	log / nolog	write list

dataset save	precision	write wave
dataset snapshot	property list	
delete	property wave	
describe	radix	
down	right	

Verilog and VHDL API System Task Reference

The ModelSim tool's available API system tasks used for recording transactions in Verilog and VHDL are:

- [create_transaction_stream](#) — creates the transaction stream.
Verilog: `$create_transaction_stream()`
VHDL: `create_transaction_stream()`
- [begin_transaction](#) — starts a transaction.
Verilog: `$begin_transaction()`
VHDL: `begin_transaction()`
- [add_attribute](#) — adds attributes to an existing transaction.
Verilog: `$add_attribute()`
VHDL: `add_attribute()`
- [add_relation](#) — records relations on an existing transaction.
Verilog: `$add_relation()`
VHDL: `add_relation()`
- [end_transaction](#) — ends the transaction.
Verilog: `$end_transaction()`
VHDL: `end_transaction()`
- [delete_transaction](#) — deletes the transaction from the database.
Verilog: `$delete_transaction()`
VHDL: `delete_transaction()`
- [free_transaction](#) — frees the transaction handle.
Verilog: `$free_transaction()`
VHDL: `free_transaction()`

add_attribute

Add an attribute to a transaction.

Syntax

- Verilog

\$add_attribute(transaction, value, attribute_name)

- VHDL

add_attribute(transaction, value, attribute_name)

Returns

Nothing

Arguments

Name	Type	Description
transaction	Verilog: integer VHDL: TrHandle	Required. Handle for the transaction to which you are adding the attribute.
value	object	Required. Verilog — object that is the value to be used for the attribute. VHDL — constant, variable, signal or generic that is the value to be used for the attribute. Valid types: integer, real, bit, boolean, bit_vector, std_logic, std_logic_vector.
attribute_name	string	Optional for Verilog. Required for VHDL. The name of the attribute to be added to the transaction. Default: The name of the variable used for the value parameter if it can be determined, “anonymous” otherwise.

Related Topics

[Recording Transactions in Verilog and VHDL](#) [Attribute Type](#)

[Multiple Uses of the Same Attribute](#) [Anonymous Attributes](#)

add_relation

Add a relation from the source transaction to the target transaction.

Syntax

- Verilog

\$add_relation(source_transaction, target_transaction, relationship_name)

- VHDL

add_relation(source_transaction, target_transaction, relationship_name)

Returns

Nothing

Arguments

Name	Type	Description
source_transaction	Verilog: integer VHDL: TrHandle	Required. Handle to the source transaction for which the relationship is to be established.
target_transaction	Verilog: integer VHDL: TrHandle	Required. Handle to the target transactions for which the relationship is to be established.
relationship_name	string	Required. The name of the relationship.

Related Topics

[Recording Transactions in Verilog and VHDL](#) [Definition of Relationship in Transactions](#)

begin_transaction

Begin a transaction on the specified stream. The transaction handle must be saved for use in other transaction API calls. `$begin_transaction()` or `begin_transaction()` is used to start all transactions. The optional fourth parameter allows you to specify a parent transaction, making the new transaction a phase transaction of the parent.

A handle returned by `$begin_transaction()` or `begin_transaction()` will be non-zero unless there is an error. The error is reported to the transcript.

Syntax

- Verilog
`$begin_transaction(stream, transaction_name, begin_time, parent_transaction)`
- VHDL
`begin_transaction(stream, transaction_name, begin_time, parent_transaction)`

Returns

Name	Type	Description
transaction	Verilog: integer VHDL: TrHandle	The handle to the newly started transaction.

Arguments

Name	Type	Description
stream	Verilog:integer VHDL: TrHandle	Required. Handle to the previously created stream.
transaction_name	string	Required. Name of the transaction to begin. Name must be a legal C identifier. White spaces must be used with escaped or extended identifiers. ¹ See Names of Streams and Substreams for further details on legal names.
begin_time	time	Optional. The absolute simulation time that the transaction will begin. Default: The current simulation time.
parent_transaction	Verilog:integer VHDL: TrHandle	Optional. Handle to an existing transaction that is the parent to the new one. The new transaction will be a phase transaction of the parent. Default: NULL for Verilog, 0 for VHDL.

1. If you are using closed kit OVM components, and your <transaction_name> contains a non-extended or escaped white space, you may receive an OVM warning message to the effect that your name is not a legal c identifier name, and that the name has been changed to a legal name. For example:
OVM_WARNING @ xxxx: reporter [ILLEGALNAME] 'transaction name' is not a legal c identifier name, changed to 'transaction_name'. Streams must be named as a legal c identifier.

Related Topics

[Recording Transactions in Verilog and VHDL](#) [Retroactive Recording / Start and End Times](#)
[The Life-cycle of a Transaction](#) [Valid Handles](#)

create_transaction_stream

Create a transaction stream that can be used to record transactions. The stream handle must be saved for use in other transaction API calls.

A handle returned by \$create_transaction_stream() or create_transaction_stream() will be non-zero unless there is an error. The error is reported to the transcript.

Syntax

- Verilog

\$create_transaction_stream(stream_name, stream_kind)

- VHDL

create_transaction_stream(stream_name, stream_kind)

Returns

Name	Type	Description
stream	Verilog:integer VHDL: TrHandle	Handle to the created stream.

Arguments

Name	Type	Description
stream_name	string	Required. The name of the stream to be created. For OVM designs, the string specified must adhere to the following format: <pre style="margin-left: 40px;">{"..",get_full_name(),".", "<your_name>"}</pre> The stream name should not match that of any object in the parent region for the stream.
stream_kind	string	Optional. The kind of stream to be created. Default is "Stream". The kind can be any string and it is not interpreted. It is stored in the WLF file.

Related Topics

- [Recording Transactions in Verilog and VHDL](#) [About Transaction Streams](#)
- [Names of Streams and Substreams](#) [Stream Logging](#)
- [Valid Handles](#)

delete_transaction

Removes a transaction from the transaction database. The transaction is not recorded in the WLF file.

Syntax

- Verilog
`$delete_transaction(transaction)`
- VHDL
`delete_transaction(transaction)`

Returns

Nothing

Arguments

Name	Type	Description
transaction	Verilog:integer VHDL: TrHandle	Required. Handle for the transaction which you are deleting.

Related Topics

[Recording Transactions in Verilog and VHDL](#) [Valid Handles](#)

end_transaction

End the specified transaction. Ending the transaction simply sets the end-time for the transaction and may be done only once. However, if free is not specified, the transaction handle is still valid for use in recording relations and attributes until a call to `$free_transaction()` or `free_transaction()`.

Syntax

- Verilog
`$end_transaction(transaction, end_time, free)`
- VHDL
`end_transaction(transaction, end_time, free)`

Returns

Nothing

Arguments

Name	Type	Description
transaction	Verilog:integer VHDL: TrHandle	Required. Handle to the name of the transaction being ended.
end_time	time	Optional. The absolute simulation time that the transaction will end. Default: The current simulation time.
free	Verilog: integer VHDL: boolean	Optional. If this argument is non-zero (Verilog) or true (VHDL), the memory allotted for this transaction will be freed. This is for convenience, and is equivalent to a call to <code>\$free_transaction()</code> or <code>free_transaction()</code> for this transaction. Use only when no more attributes will be recorded for this transaction and no relations will be made for this transaction. Default: zero (Verilog) or false (VHDL) — do not free memory.

Related Topics

[Recording Transactions in Verilog and VHDL](#) [Retroactive Recording / Start and End Times](#)
[The Life-cycle of a Transaction](#)

free_transaction

Free a transaction. This call allows the memory allotted for this transaction to be freed. The handle will no longer be valid. Attributes can no longer be recorded for the transaction. Relations can no longer be made with the transaction.

i **Tip: Important:** You must free all transaction handles in your design. This is a requirement specific to Verilog and VHDL recording. If a handle is not freed, the result is a memory leak in the simulation.

Syntax

- Verilog
`$free_transaction(transaction)`
- VHDL
`free_transaction(transaction)`

Returns

Nothing

Arguments

Name	Type	Description
transaction	Verilog:integer VHDL: TrHandle	Handle to the transaction to be freed.

Related Topics

[The Life-cycle of a Transaction](#)

Chapter 13

Verifying Designs with Questa Verification IP Library Components

Note



The functionality described in this chapter requires an additional license feature for ModelSim SE. Refer to the section "[License Feature Names](#)" in the Installation and Licensing Guide for more information or contact your Mentor Graphics sales representative.

This chapter provides a brief introduction to Questa Verification IP transactions in ModelSim, and discusses additional features of the ModelSim GUI available when simulating a model that includes Questa Verification IP(s).

What is Questa Verification IP?	685
What is a Questa Verification IP Transaction?	686
Questa Verification IP Transaction Relationships	686
Questa Verification IP Transaction Viewing in the GUI	687
Questa Verification IP Objects in the GUI	687
Arrays in Questa Verification IP	689
Viewing Questa Verification IP Transactions in the Wave Window	690
Viewing Questa Verification IP Transactions in Objects Window	696
Viewing Questa Verification IP Transactions in List Window	696
Questa Verification IP Transaction Debug	697
Debugging Using Relationships	698
Questa Verification IP Transaction Details in Transaction View Window	699

What is Questa Verification IP?

A Questa Verification IP is a model used to verify a protocol or interface. These models bridge the gap between RTL, TLM, and system-level verification by using a hierarchy of transactions to create a link between different TLM and RTL abstraction levels. Questa Verification IP's include stimulus generation, reference checking and coverage measurements and are available for popular protocols and standard interfaces as part of the Questa Verification IP Library

For more detailed information on Questa Verification IPs and the Questa Verification IP Library, including which library components are available, as well as how to load and run a model using a Questa Verification IP, refer to the "Questa Verification IP Library Data Book", available from SupportNet at <http://supportnet.mentor.com>.

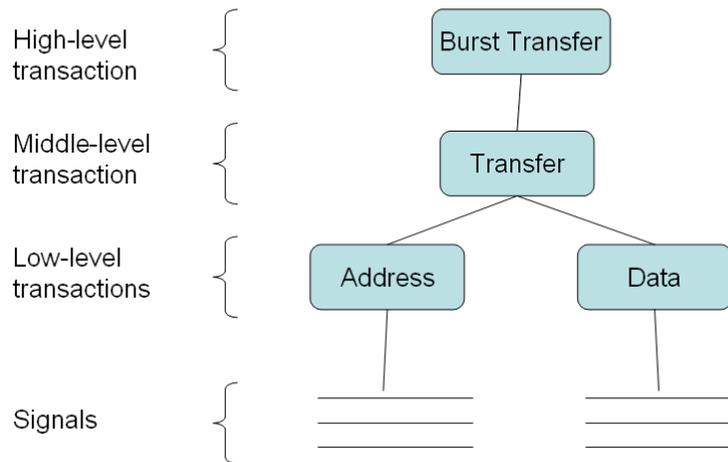
What is a Questa Verification IP Transaction?

i Tip: Important: Transactions in a Questa Verification IP are quite distinct from SystemC or SystemVerilog transactions in the ModelSim tool. In this chapter, any occurrence of the word “transaction” refers to Questa Verification IP transactions exclusively, unless otherwise specified.

A Questa Verification IP transaction is unique in that it represents communication at multiple levels of abstraction.

To illustrate, consider a hypothetical Questa Verification IP transaction shown in [Figure 13-1](#). This transaction contains a single “Transfer” transaction representing a read or a write. However, the transaction also references lower level transactions, which separately represent the address and data transfers. It is also referenced by a higher level transaction, “Burst Transfer.” Note that “Burst Transfer” can consist of multiple “Transfer” transactions. Signals are always represented at the lowest level of abstraction.

Figure 13-1. Questa Verification IP Transaction at Different Levels of Abstraction



Questa Verification IP Transaction Relationships

Where more than one level of abstraction exists, a “relationship” exists between the different levels. In the example shown in [Figure 13-1](#), a “Transfer” transaction is related to the “Address” and “Data” transactions that communicate the address and data information for that transfer. These transactions in turn are related to the individual signals which communicate the equivalent information across the bus. Questa Verification IPs maintain these relationships, allowing for simulation and debugging across the different levels of abstraction.

The term “parent” describes a related transaction at a higher level of abstraction, and “child” describes a related transaction or signal at a lower level of abstraction. Each transaction may have many related child or parent transactions. In [Figure 13-1](#), the “Transfer” transaction has two children: a “Address” transaction and a “Data” transaction. It also has one parent: “Burst Transfer” transaction. Each Burst Transfer transaction can have multiple “Transfer” transactions as children.

Related Topics

[Questa Verification IP Arrays in the Wave Window](#)

[Color and Questa Verification IP Arrays](#)

Questa Verification IP Transaction Viewing in the GUI

Before you can view Questa Verification IP objects in the ModelSim GUI, you must have loaded the design containing the library protocols. For information on how to hook up the protocols to your design, refer to the "Questa Verification IP Library Data Book" available from SupportNet at <http://supportnet.mentor.com>.

Transactions can be viewed in the following GUI windows:

- Wave window - must be logged during a simulation run (see [Viewing Questa Verification IP Transactions in the Wave Window](#))
- Objects window - visible when you select an instance of the interface in Questa Verification IP in the Structure (sim) window (see [Viewing Questa Verification IP Transactions in Objects Window](#))
- List window - must be logged during a simulation run (see [Viewing Questa Verification IP Transactions in List Window](#))

Related Topics

[Questa Verification IP Objects in the GUI](#)

[Arrays in Questa Verification IP](#)

Questa Verification IP Objects in the GUI

The display of Questa Verification IP transactions in ModelSim differs from “normal” (SystemC or SystemVerilog) transactions. Designs containing Questa Verification IP's can display transactions of many different object kinds.

As a Questa Verification IP user — someone viewing and analyzing the results from the simulation of an OVM design — you are most interested in only three kinds of objects: MVC_Transaction, MVC_Message and MVC_Stripe. All other objects, listed in the shaded

cells in [Table 13-1](#), represent internal logic and are most relevant to a Questa Verification IP developer (someone designing a Questa Verification IP library).

The MVC_Message and MVC_Transaction objects represent higher level transactions (see [Figure 13-1](#) for an example) that can be related to other MVC_Stripe, MVC_Message and MVC_Transaction objects. MVC_Stripe's are always the lowest level transaction, having a direct relationship to a set of signals.

Questa Verification IP objects can be viewed in the Wave window, where they are drawn as transaction streams or signals. [Table 13-1](#) includes a description of the various objects and how they appear.

Table 13-1. Questa Verification IP Objects

Object Kind and Icon	Definition	Appearance in GUI
MVC_Transaction 	A type of transaction representing multi-directional communication between one or more devices across an interface or bus. for example, a combined request and response.	As transaction streams. Can have sub-streams to show concurrent (overlapping or pipelined) transactions.
MVC_Message 	A type of transaction representing one-way or broadcast communication from one device to one or more devices on an interface or bus. for example, a request.	As transaction streams. Can have sub-streams to show concurrent (overlapping or pipelined) transactions.
MVC_Stripe 	A type of transaction representing a single clock cycle of activity on one or more signals or wires originating from a single device. Unlike MVC_Message or MVC_Transaction, it must always have a duration.	As transaction streams.
MVC_ExternalMethod 	A type of method or task within the Questa Verification IP that represents a call to one of the transactions from outside the Questa Verification IP.	As transaction streams.
MVC_Activity 	A method or task within the Questa Verification IP.	As transaction streams.
MVC_TimelessActivity 	A method or task within the Questa VIP that completes in zero time, zero deltas.	As transaction streams.

Table 13-1. Questa Verification IP Objects

Object Kind and Icon	Definition	Appearance in GUI
MVC_Function 	A function within the Questa VIP. Completes in zero time, zero deltas.	As transaction streams.
MVC_MapFunction 	An object within the Questa VIP. Like a function, but with two sets of parameters and forward and reverse algorithms specified.	As transaction streams.
MVC_Label 	A tag on a code statement within the Questa VIP. Can be used to monitor execution of that statement.	As transaction streams.

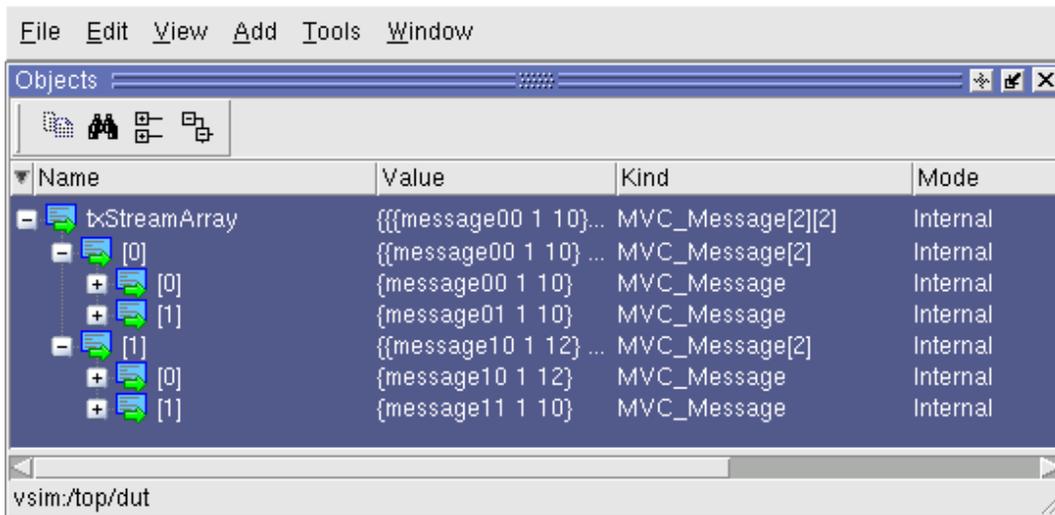
Arrays in Questa Verification IP

Questa Verification IP objects can be declared as arrays in specific protocols. Questa Verification IP arrays are created by the developer, solely for the sake of convenience, and do not have any effect on the behavior of the objects themselves. In other words, an element of an array is not aware of, nor is it affected by, any other element of the same array.

Arrays of Questa Verification IP objects appear much like other arrays in the GUI. When the array is made from a stream-like object (such as an MVC_Stripe) each leaf element of the array is a fully unique and independent stream. As such, each element can have different attributes and can have any number of active sub-streams at any time during simulation.

Questa Verification IP arrays can have multiple dimensions. For example, one component might contain a two-by-two array of MVC_Messages. This results in a single object in the GUI. Consider the following array in the objects window (as shown in [Figure 13-2](#)). Selecting the array's expand button reveals the two rows of the array. The "Kind" field for the array displays the Questa Verification IP kind ("MVC_Message") and the size of the array ("[2][2]").

Figure 13-2. Arrays in Objects Window



The level below the array (*txStreamArray*) is that of the two sub-arrays (*0* and *1*). These are the rows of the parent array, each of which has the correct name for an array element, and its kind field indicates the size of the sub-array. Expanding the sub-arrays reveals the leaf streams of the array, whose “Kind” fields do not contain any index values. Expand buttons on “leaf” streams indicate that they have sub-streams or attributes.

Related Topics

[Questa Verification IP Arrays in the Wave Window](#)
[Color and Questa Verification IP Arrays](#)

Viewing Questa Verification IP Transactions in the Wave Window

The advantage of viewing Questa Verification IP transactions in the Wave window is that it allows you to easily view the relationships between transactions and the signals/wires that comprise them.

Prerequisites

- Before you can view transaction objects in the Wave window, log the Questa Verification IP objects during a simulation run.
- General viewing instructions and conventions applicable to all transactions (SystemVerilog or SystemC as well as Questa Verification IP) can be found in the section “[Viewing Transactions in the Wave Window](#)”.

Procedure

1. Ensure the model containing the Questa Verification IP protocol is loaded. For example:

```
vsim top
```

2. Enable logging for the desired object(s). Objects must be explicitly logged for transaction viewing. Logging the object(s) results in the objects being recorded in the .wlf file, allowing them to be viewed post-simulation. To log Questa Verification IP objects, you can:

- Add transactions to the Wave window by dragging and dropping them into the window. Alternatively, you can use a [add wave](#) CLI command, such as:

```
add wave /top/interface/read_id
```

- Use the [log](#) command, such as:

```
log /top/interface/read_id
```

- By default, Questa Verification IP transactions are ignored with normal wildcard usage. To enable wildcard matching for all transactions, use the `-mvcall` switch, such as:

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

```
log -mvcall -r /*
```

- To log all OVM sequence Questa Verification IP transactions (i.e., only the transactions in the Questa Verification IP protocol stack with sequence items) use the `-mvcovm` switch:

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

```
log -mvcovm -r /*
```

3. Run simulation on a design containing transactions. For example:

```
run -all
```

4. Within the Wave window, navigate to the portion of the design containing the Questa Verification IP component or protocol.

Additional Steps for the Viewing of Completed Transactions

By default, only recognized transactions that have become used as legal protocol are logged, which may prevent the transaction instances of interest from being logged soon enough to observe an issue. To enable the logging of transaction instances that have been recognized as completed (which may later become used or deleted), please follow these additional steps:

5. Add the required transaction to the Wave window.
6. Right-click on the transaction name.

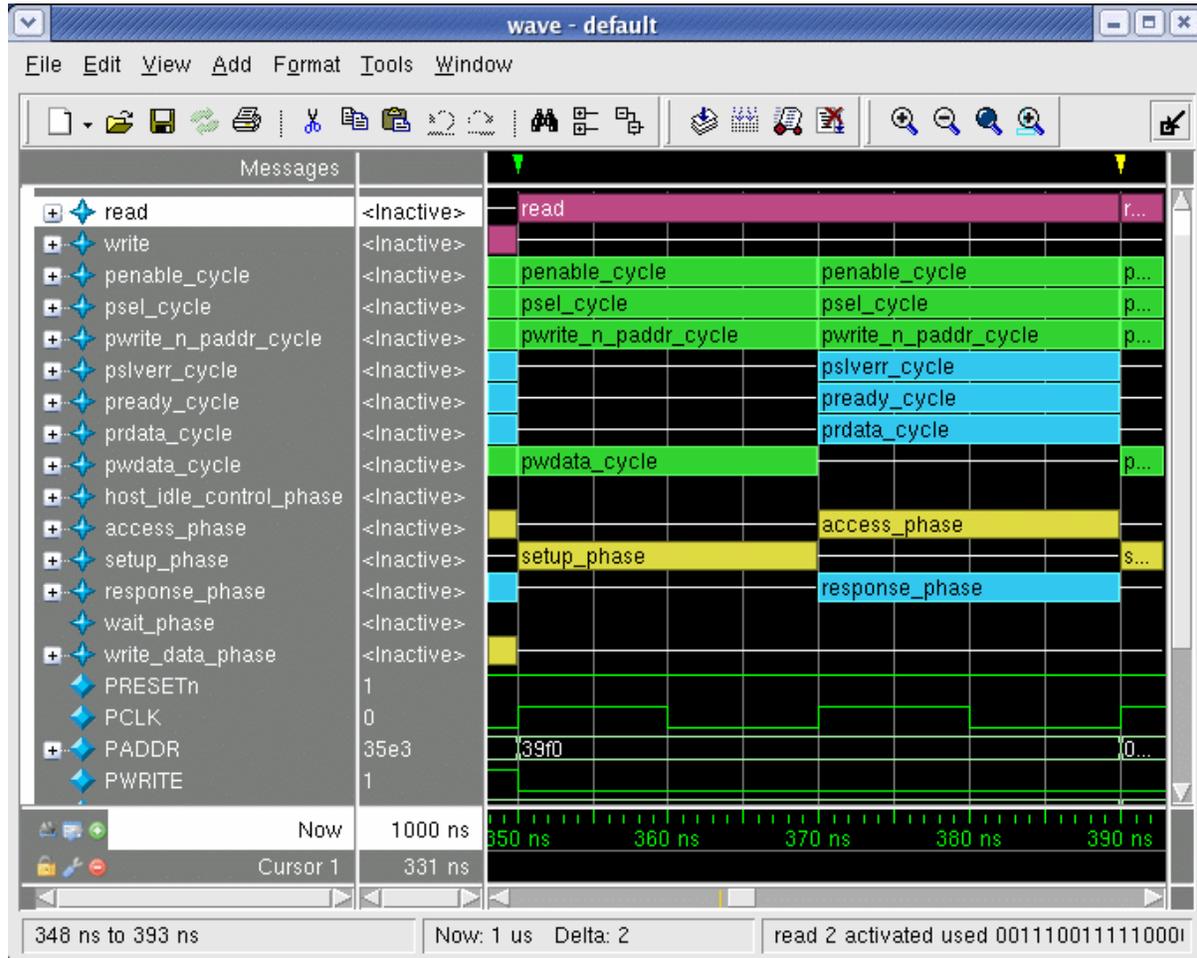
You can select any number of transaction streams to enable the logging of completed transaction instances for those additional transaction streams.

7. Select "Transaction Properties..." from the pop-up menu.
8. Select the "Questa VIP Logging" tab.
9. Click the "Deletion Logging Enabled" box.
10. Click OK.
11. Run the simulation - all completed transactions now appear on the Wave window.

Results

Transactions within Questa Verification IP components appear in the Wave window, as shown in [Table 13-3](#). For information on the colors of transactions, see ["What the Colors Mean"](#).

Figure 13-3. Questa Verification IP Transactions in Wave Window



What the Colors Mean

When viewing transactions within a Questa Verification IP, the color of the transaction object indicates what caused it to exist. The possible colors and their meaning are shown in [Table 13-2](#). These colors are used as the fill color for a transaction, or to highlight a signal.

Table 13-2. Questa Verification IP Colors and Causation

Color	Status	Causal relationships to other transaction types
	Sent	Represents unidirectional communication. Created by the Questa Verification IP as a result of a request or call to start this MVC_Message or MVC_Stripe. Can generate lower level activity (that is, cause lower level transactions or signal activity).
	Activated	Represents bidirectional communication. Created by the Questa Verification IP as a result of a request or call to start this MVC_Transaction. Can generate and recognize lower level activity (that is, allow and pick-up attribute values from lower level transactions or signal activity).
	Recognized	Created by the Questa Verification IP due to activity at a lower level of abstraction. Can recognize lower level activity.
	Generated	Created by the Questa Verification IP as a result of activity at a higher level of abstraction. Can generate lower level activity.
	Genact	Created by the Questa Verification IP as a result of a higher level MVC_Transaction. Can generate and recognize lower level activity.
	Error	Represents an error condition for that Questa Verification IP object.

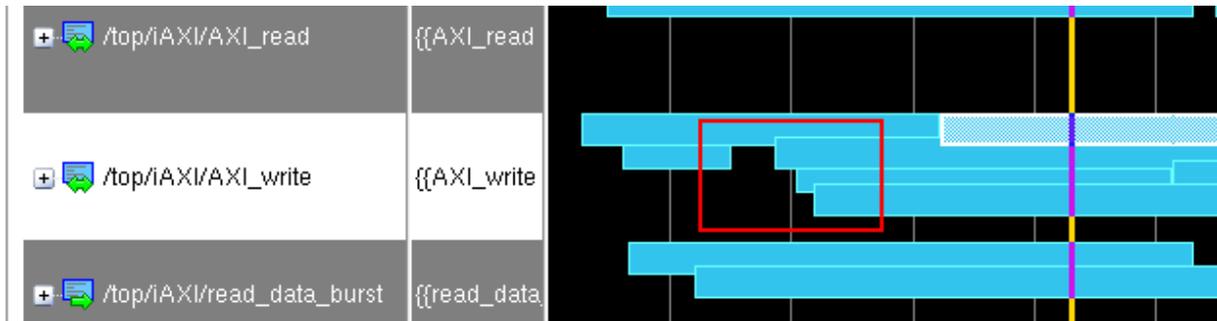
For example, in [Figure 13-3](#), the color shown in the transaction objects allows us to determine that a read transaction was started by a TLM master. This resulted in the Questa Verification IP generating a *setup_phase* transaction and a number of *xxx_cycle* messages, which in turn resulted in some pin level activity. This caused an RTL slave to issue a response which was recognized by the Questa Verification IP up into a *response_phase message* (through the *xxx_cycle* messages), and finally back into the read transaction.

This basic color scheme is modified slightly for arrays of Questa Verification IP objects. See [“Color and Questa Verification IP Arrays”](#) for more information.

Appearance of Concurrent Transactions

Multiple transactions recorded on a single stream at one time are known as concurrent transactions. They appear as overlapping in the Wave window, as shown in [Figure 13-4](#).

Figure 13-4. Concurrent Transactions Overlapping in Wave Window



Related Topics

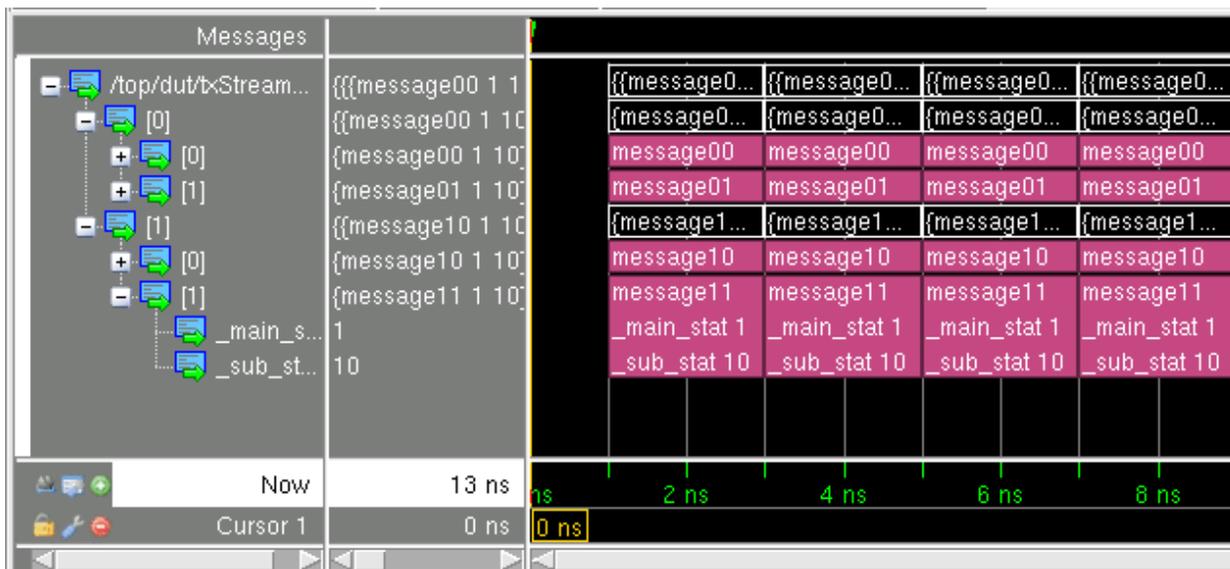
- [About Transaction Streams](#)
- [Questa Verification IP Transaction Debug](#)
- [What is a Questa Verification IP Transaction?](#)

Questa Verification IP Arrays in the Wave Window

Questa Verification IP protocols may create arrays of streams with one or more dimensions. In the wave window, these behave much like other array objects. Expanding the parent reveals the next level of detail and each sub-level name is the appropriate index value.

[Figure 13-5](#) shows the expansion of a two-by-two array of message streams. It reveals that each level of the array expands to reveal the details below, as with other arrays. The expansion of the leaf `Stream[1][1]` reveals the attributes of that stream element. See “[Color and Questa Verification IP Arrays](#)” for information on the effect that arrays have on the Questa Verification IP object color scheme.

Figure 13-5. Questa Verification IP Array (2X2) in Wave Window

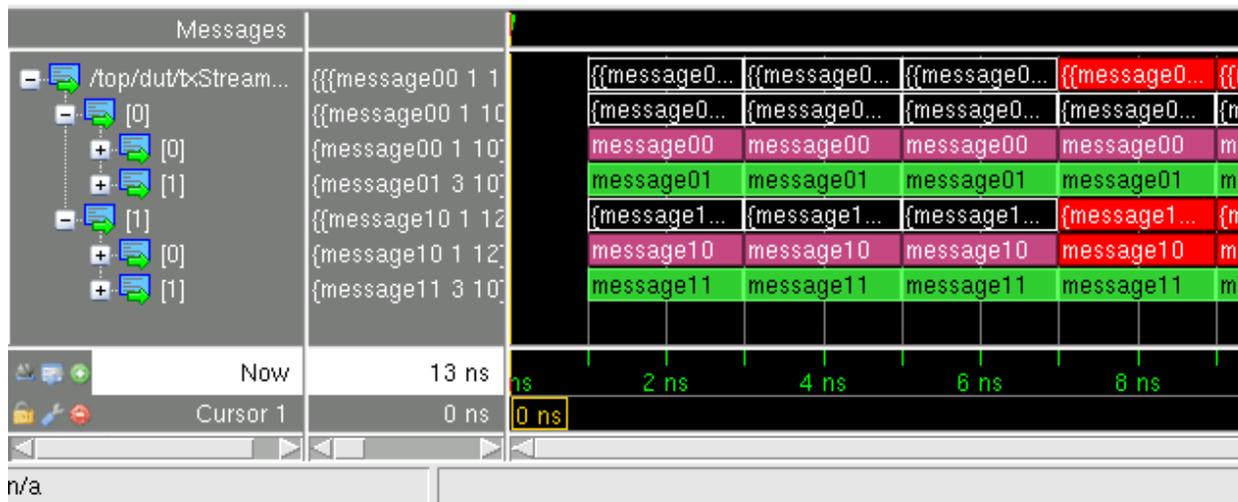


Color and Questa Verification IP Arrays

Questa Verification IP stream arrays slightly modify the generic color scheme as described in “[What the Colors Mean](#)”.

In [Figure 13-6](#), errors occur on the last three instances on stream *txStream[1][0]*. These are drawn in red, as expected. This color extends upward to the parent array *txStream[1]* and its parent, *txStream*. This enables errors to be spotted easily when the array has not been expanded.

Figure 13-6. Color of Arrays



You can see that all of the other instances on the four “leaf” streams are either activated (purple) or generated (green) (see [Table 13-2](#) for color descriptions). These colors are NOT reflected up

to the parent arrays: the parent arrays are black. This is due to the fact that it is typical to have paired elements in an array to separate outgoing and incoming traffic, and it is not feasible to mix the colors symbolizing this pairing in any straightforward way. Thus, arrays are always drawn in black, unless an error occurs one or more element.

Related Topics

[What the Colors Mean](#)
[Questa Verification IP Transaction Debug](#)

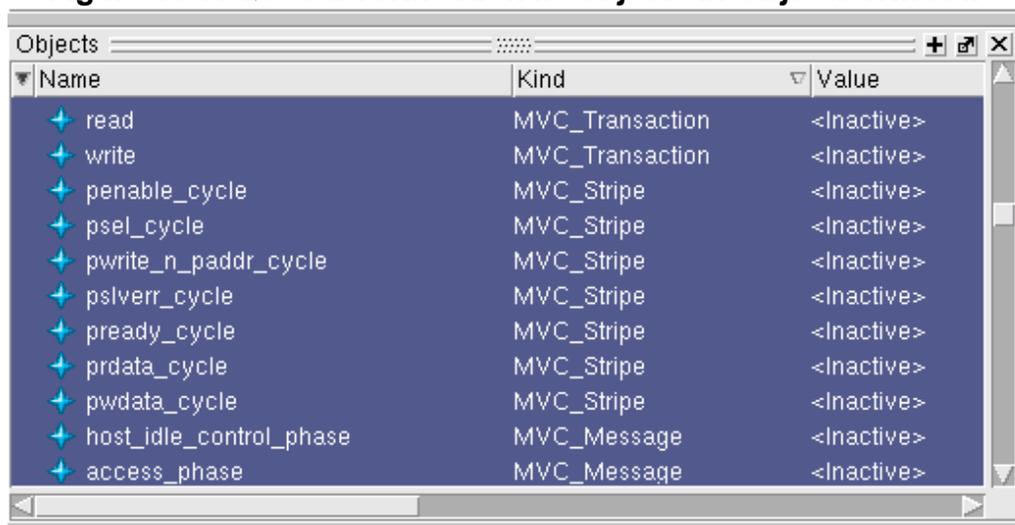
Viewing Questa Verification IP Transactions in Objects Window

Use the Objects window to browse the available objects in order to log them or add them to the Wave window.

1. Ensure the model containing the Questa Verification IP protocol is loaded. For example:
vsim top
2. With the Objects window open (**View > Objects**), select the top level SystemVerilog interface in the Questa Verification IP.

The objects in that interface appear in the Objects window, similar to those in [Figure 13-7](#).

Figure 13-7. Questa Verification IP Objects in Objects Window



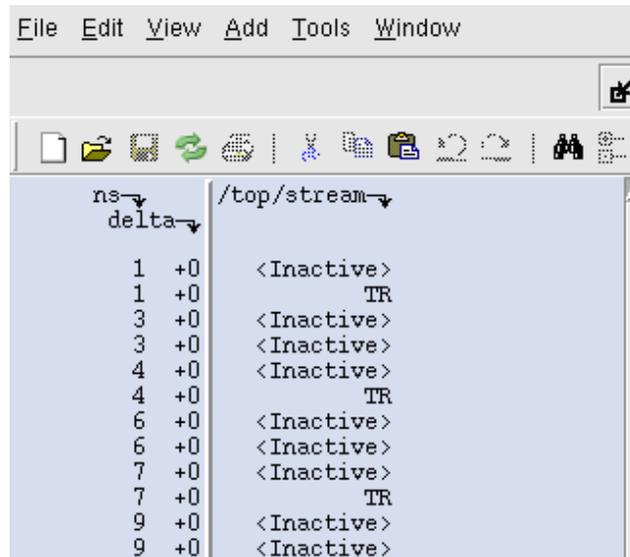
Viewing Questa Verification IP Transactions in List Window

Use the List window to view the values of all selected design elements at each time or delta advance. The list window allows for the easy creation of tabular reports for transactions.

1. Ensure the model containing the Questa Verification IP protocol is loaded. For example:
vsim top
2. With the List window open (**View > List**), select the top level SystemVerilog interface in the Questa Verification IP.

The objects in that interface appear in the List window, similar to those in [Figure 13-8](#).

Figure 13-8. Questa Verification IP Objects in List Window



When transactions are present in the list window, rows are written any time a transaction's state changes:

- when a transaction starts or ends
- when any attribute changes state

In [Figure 13-8](#) above, there is an internal attribute (not shown) changing state and causing extra rows to be drawn.

Questa Verification IP arrays display the value of every element.

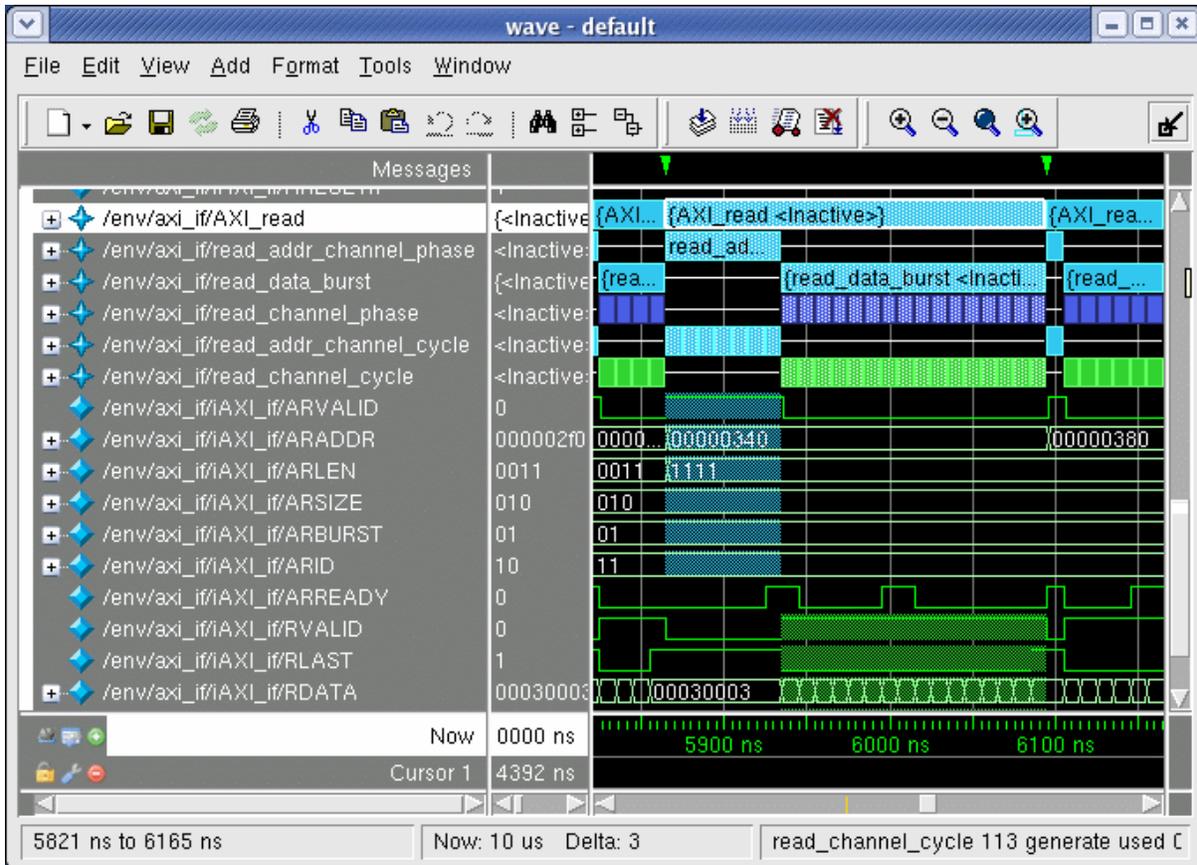
Questa Verification IP Transaction Debug

Questa Verification IP objects appear as transactions or signals in the Wave window, depending on the type of the object being displayed (see [Table 13-1](#) on page 688). The colors of transactions that appear in the Wave window are defined in [Table 13-2](#) on page 693.

Debugging Using Relationships

Relationships for Questa Verification IP transaction objects can be viewed by selecting them in the usual way, by clicking on an object in the Wave window. As well as highlighting related transactions, selecting a single instance of a Questa Verification IP object also highlights any related signal activity. Signals are highlighted for only the period of time that they are taking part in the selected transaction. You can also select a signal to highlight the transactions that are using that signal at that specific time.

Figure 13-9. Viewing Questa Verification IP Relationships



This simple way of viewing relationships between the different transaction abstraction levels and the signals allows very rapid movement between levels of abstraction when debugging. For example, on an interface that allows multiple outstanding requests, you could select the data signal at a certain point in time, and immediately see the transaction that the data is part of. This provides you with information such as the address, requesting master, burst type, and so forth, without needing to carefully trace back along the signals.

Note



IMPORTANT — For the highlighting of Transaction/Wire relationships to function properly, all Questa Verification IP transactions in the protocol hierarchy (from the top level transactions down to the stripes) must be logged to WLF (as described in the section "[Viewing Questa Verification IP Transactions in the Wave Window](#) -> Procedure").

Transaction viewing and navigation for Questa Verification IP transactions are the same as for any transaction in ModelSim. See '[Viewing Transactions in the Wave Window](#)' and "[Selecting Transactions or Streams](#)" for more information.

Questa Verification IP Transaction Details in Transaction View Window

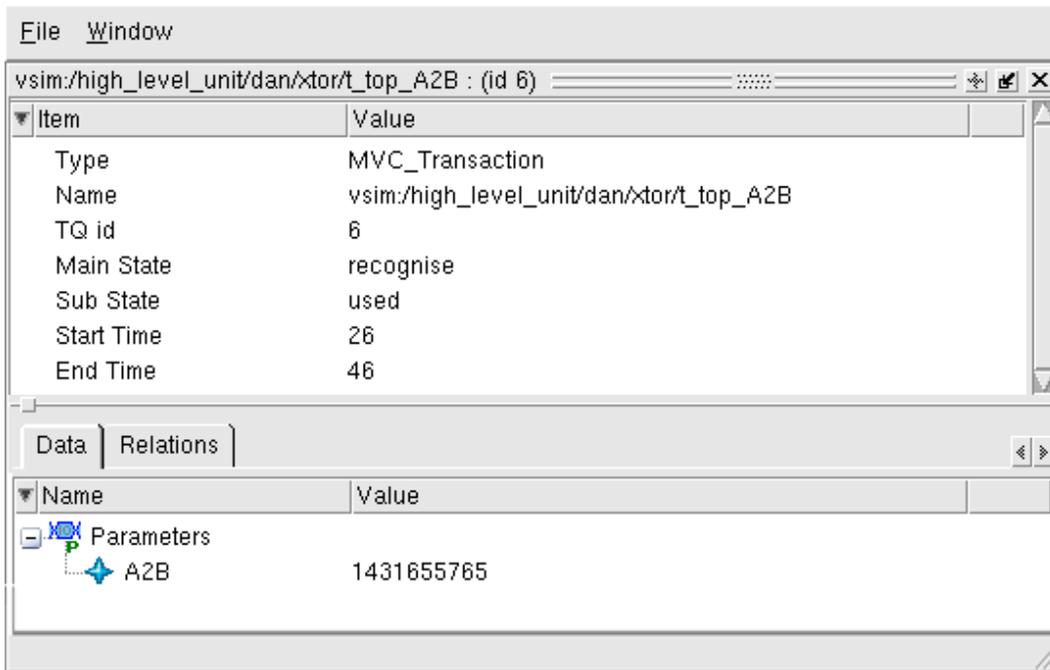
The Transaction window displays information about a selected Questa Verification IP transaction instance. The window has an upper pane, called the main pane, and a lower pane with two tabs: the Data tab and the Relations tab.

Opening the Transaction View Window

- Double-click on a Questa Verification IP transaction instance or a transaction stream in the Wave window, or
- Select a transaction instance or stream in the Wave window, and right-click > **Transaction View**

Multiple Transaction windows can be open simultaneously. The Transaction View window opens with the Data tab open in the lower pane, as shown in [Figure 13-10](#).

Figure 13-10. Transaction Window - Data Tab



Data Tab

The "Data" tab contains all Questa Verification IP parameter attributes and their values. These are presented in a tree format (in the same style as in the Objects window).

The parameter values displayed for transaction instances are:

- Type — The type of Questa Verification IP transaction instance.
- Name — The name of the transaction instance.
- TQ id— A serial number for instances on the same stream. (This is not the same as the UID, which is a unique ID assigned by the ModelSim simulator).
- Main State — Identifies how the transaction instance came into existence.
- Sub State — Identifies the phase of life the instance has reached.
- Start Time — Start time of the transaction instance.
- End Time — End time of the transaction instance.

The parameter values displayed for transaction stream are:

- Name — The name of the transaction instance.
- Count — A count of the number of instances on the stream

- Attribute data for all the transaction instances from the stream. You can save this attribute data to a file named <stream_name>.csv by selecting **File > Save**.

Relations Tab

The "Relations" tab contains a list of related instances. These are categorized by relationship type, and presented in a tree format. Each instance includes its general data (type, name, and so on — see parameters listed in “[Data Tab](#)”).

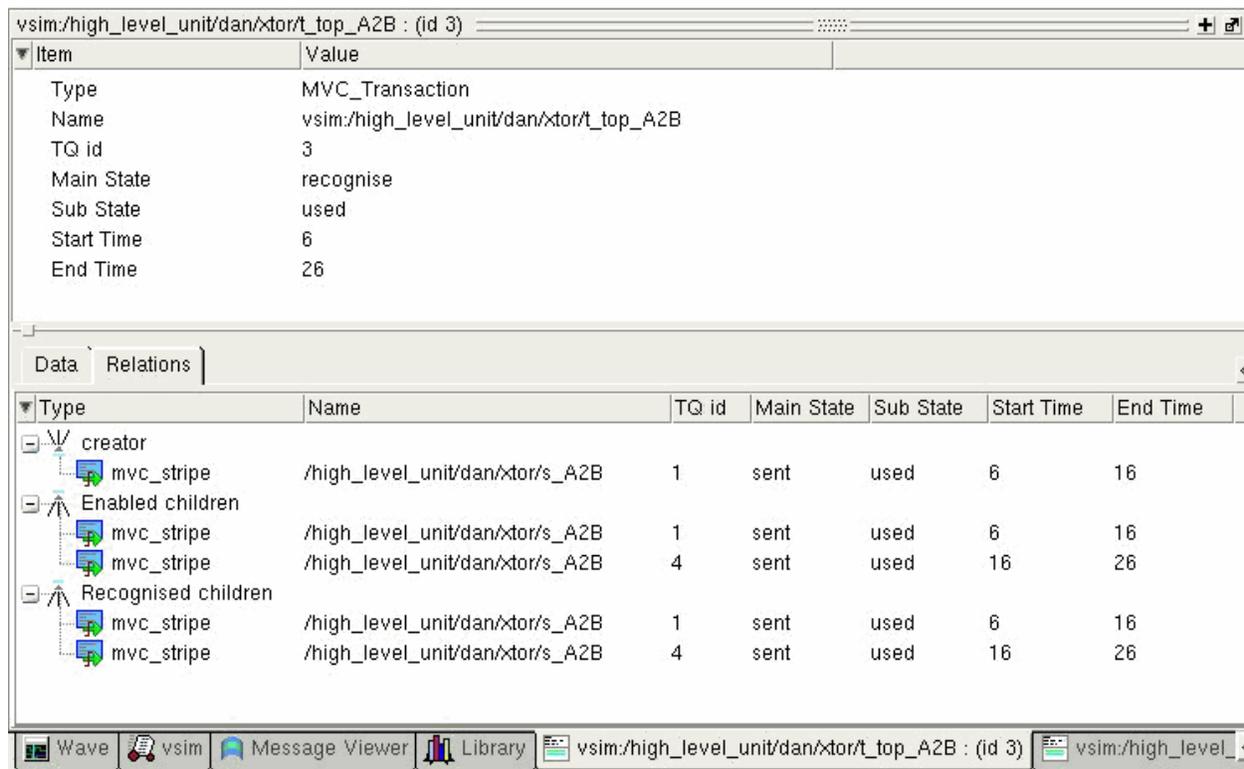
Updating Contents of the Transaction Window

To update contents of the Transaction window, including transaction name in the label of the window, with data from a new instance:

1. Select the Questa Verification IP transaction instance/stream in Wave window.
2. Drag and drop the instance into either the upper pane or lower pane of an open Transaction window.

[Figure 13-11](#) shows details for instance `/high_level_unit/dan/xtor/t_top_A2B`, and lists the relations of that instance.

Figure 13-11. Transaction Window - Relations Tab



Chapter 14

Recording Simulation Results With Datasets

This chapter describes how to save the results of a ModelSim simulation and use them in your simulation flow. In general, any recorded simulation data that has been loaded into ModelSim is called a *dataset*.

One common example of a dataset is a wave log format (WLF) file. In particular, you can save any ModelSim simulation to a wave log format (WLF) file for future viewing or comparison to a current simulation. You can also view a wave log format file during the currently running simulation.

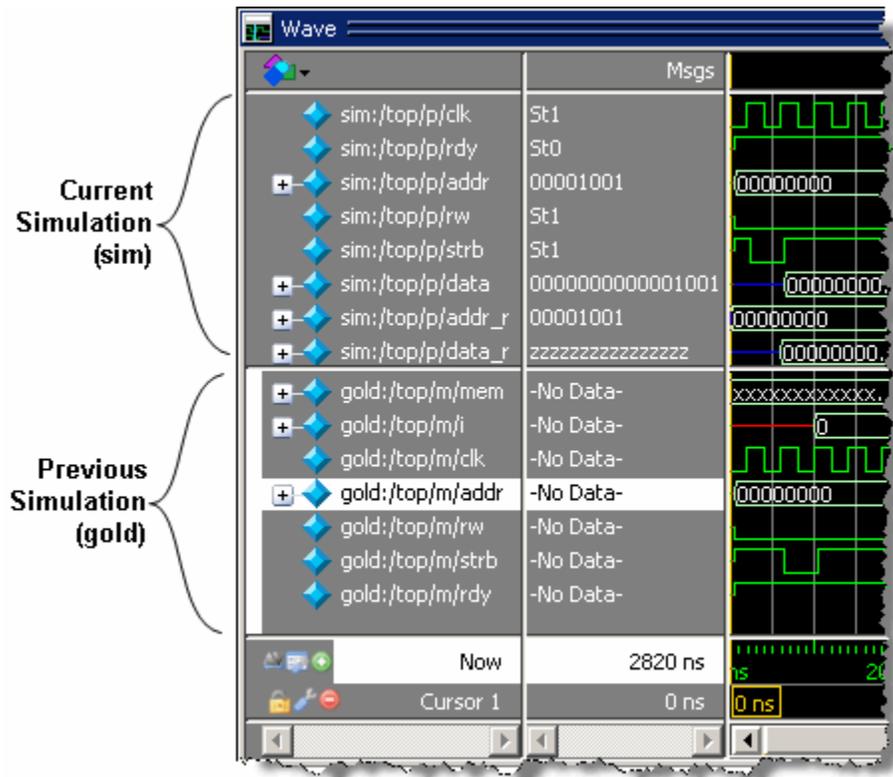
A WLF file is a recording of a simulation run that is written as an archive file in binary format and used to drive the debug windows at a later time. The files contain data from logged objects (such as signals and variables) and the design hierarchy in which the logged objects are found. You can record the entire design or choose specific objects.

A WLF file provides you with precise in-simulation and post-simulation debugging capability. You can reload any number of WLF files for viewing or comparing to the active simulation.

You can also create *virtual signals* that are simple logical combinations or functions of signals from different datasets. Each dataset has a logical name to indicate the dataset to which a command applies. This logical name is displayed as a prefix. The current, active simulation is prefixed by "sim:" WLF datasets are prefixed by the name of the WLF file by default.

[Figure 14-1](#) shows two datasets in the Wave window. The current simulation is shown in the top pane along the left side and is indicated by the "sim" prefix. A dataset from a previous simulation is shown in the bottom pane and is indicated by the "gold" prefix.

Figure 14-1. Displaying Two Datasets in the Wave Window



The simulator resolution (see [Simulator Resolution Limit \(Verilog\)](#) or [Simulator Resolution Limit for VHDL](#)) must be the same for all datasets you are comparing, including the current simulation. If you have a WLF file that is in a different resolution, you can use the `wlfman` command to change it.

Saving a Simulation to a WLF File

If you add objects to the Dataflow, Schematic, List, or Wave windows, or log objects with the `log` command, the results of each simulation run are automatically saved to a WLF file called `vsim.wlf` in the current directory. You can also log variables and values to a WLF file with the `$wlfdumpvars()` Verilog system task. If you then run a new simulation in the same directory, the `vsim.wlf` file is overwritten with the new results.

If you want to save the WLF file and not have it be overwritten, select the Structure tab and then select **File > Save**. Or, you can use the `-wlf <filename>` argument to the `vsim` command or the `dataset save` command.

Note



If you do not use **dataset save** or **dataset snapshot**, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you do not end the simulation in this manner, the WLF file will not close properly, and ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions. If you end up with a damaged WLF file, you can try to repair it using the [wlfrecover](#) command.

WLF File Parameter Overview

There are a number of WLF file parameters that you can control via the *modelsim.ini* file or a simulator argument. This section summarizes the various parameters.

Table 14-1. WLF File Parameters

Feature	modelsim.ini	modelsim.ini Default	vsim argument
WLF Cache Size ^a	WLFCacheSize = <n>	0 (no reader cache)	
WLF Collapse Mode	WLFCollapseModel = 0 1 2	1 (-wlfcollapsedelta)	-nowlfcollapse -wlfcollapsedelta -wlfcollapsetime
WLF Compression	WLFCompress = 0 1	1 (-wlfcompress)	-wlfcompress -nowlfccompress
WLF Delete on Quit ^a	WLFDeleteOnQuit = 0 1	0 (-wlfdeleteonquit)	-wlfdeleteonquit -nowlfdeleteonquit
WLF File Lock	WLFFileLock = 0 1	0 (-nowlflock)	-wlflock -nowlflock
WLF File Name	WLFFilename=<filename>	<i>vsim.wlf</i>	-wlf <filename>
WLF Index	WLFIndex 0 1	1 (-wlfindex)	
WLF Optimization ¹	WLFOptimize = 0 1	1 (-wlfopt)	-wlfopt -nowlfopt
WLF Sim Cache Size	WLFSimCacheSize = <n>	0 (no reader cache)	
WLF Size Limit	WLFSizeLimit = <n>	no limit	-wlfslim <n>
WLF Time Limit	WLFTimeLimit = <t>	no limit	-wlfylim <t>

1. These parameters can also be set using the [dataset config](#) command.

- **WLF Cache Size** — Specify the size in megabytes of the WLF reader cache. WLF reader cache size is zero by default. This feature caches blocks of the WLF file to reduce

redundant file I/O. If the cache is made smaller or disabled, least recently used data will be freed to reduce the cache to the specified size.

- WLF Collapse Mode —WLF event collapsing has three settings: disabled, delta, time:
 - When disabled, all events and event order are preserved.
 - Delta mode records an object's value at the end of a simulation delta (iteration) only. Default.
 - Time mode records an object's value at the end of a simulation time step only.
- WLF Compression — Compress the data in the WLF file.
- WLF Delete on Quit — Delete the WLF file automatically when the simulation exits. Valid for current simulation dataset (*vsim.wlf*) only.
- WLF File Lock — Control overwrite permission for the WLF file.
- WLF Filename — Specify the name of the WLF file.
- WLF Indexing — Write additional data to the WLF file to enable fast seeking to specific times. Indexing makes viewing wave data faster, however performance during optimization will be slower because indexing and optimization require significant memory and CPU resources. Disabling indexing makes viewing wave data slow unless the display is near the start of the WLF file. Disabling indexing also disables optimization of the WLF file but may provide a significant performance boost when archiving WLF files. Indexing and optimization information can be added back to the file using [wlfman optimize](#). Defaults to on.
- WLF Optimization — Write additional data to the WLF file to improve draw performance at large zoom ranges. Optimization results in approximately 15% larger WLF files.
- WLFSimCacheSize — Specify the size in megabytes of the WLF reader cache for the current simulation dataset only. This makes it easier to set different sizes for the WLF reader cache used during simulation and those used during post-simulation debug. If WLFSimCacheSize is not specified, the WLFCacheSize settings will be used.
- WLF Size Limit — Limit the size of a WLF file to <n> megabytes by truncating from the front of the file as necessary.
- WLF Time Limit — Limit the size of a WLF file to <t> time by truncating from the front of the file as necessary.

Limiting the WLF File Size

The WLF file size can be limited with the [WLFSizeLimit](#) simulation control variable in the *modelsim.ini* file or with the `-wlfslim` switch for the `vsim` command. Either method specifies the number of megabytes for WLF file recording. A WLF file contains event, header, and symbol portions. The size restriction is placed on the event portion only. When ModelSim exits, the

entire header and symbol portion of the WLF file is written. Consequently, the resulting file will be larger than the size specified with `-wlfslim`. If used in conjunction with `-wlftlim`, the more restrictive of the limits takes precedence.

The WLF file can be limited by time with the [WLFTimeLimit](#) simulation control variable in the *modelsim.ini* file or with the `-wlftlim` switch for the `vsim` command. Either method specifies the duration of simulation time for WLF file recording. The duration specified should be an integer of simulation time at the current resolution; however, you can specify a different resolution if you place curly braces around the specification. For example,

```
vsim -wlftlim {5000 ns}
```

sets the duration at 5000 nanoseconds regardless of the current simulator resolution.

The time range begins at the current simulation time and moves back in simulation time for the specified duration. In the example above, the last 5000ns of the current simulation is written to the WLF file.

If used in conjunction with `-wlfslim`, the more restrictive of the limits will take effect.

The `-wlfslim` and `-wlftlim` switches were designed to help users limit WLF file sizes for long or heavily logged simulations. When small values are used for these switches, the values may be overridden by the internal granularity limits of the WLF file format. The WLF file saves data in a record-like format. The start of the record (checkpoint) contains the values and is followed by transition data. This continues until the next checkpoint is written. When the WLF file is limited with the `-wlfslim` and `-wlftlim` switches, only whole records are truncated. So if, for example, you are were logging only a couple of signals and the amount of data is so small there is only one record in the WLF file, the record cannot be truncated; and the data for the entire run is saved in the WLF file.

Multithreading on Linux Platforms

Multithreading enables the logging of information on a secondary processor while the simulation and other tasks are performed on the primary processor. Multithreading is on by default on multi-core or multi-processor Linux platforms when you specify `vsim -wlfopt`.

You can turn this option off with the `vsim -nowlftopt` switch, which you may want to do if you are performing several simulations with logging at the same time. You can also control this behavior with the [WLFUseThreads](#) variable in the *modelsim.ini* file.

Note



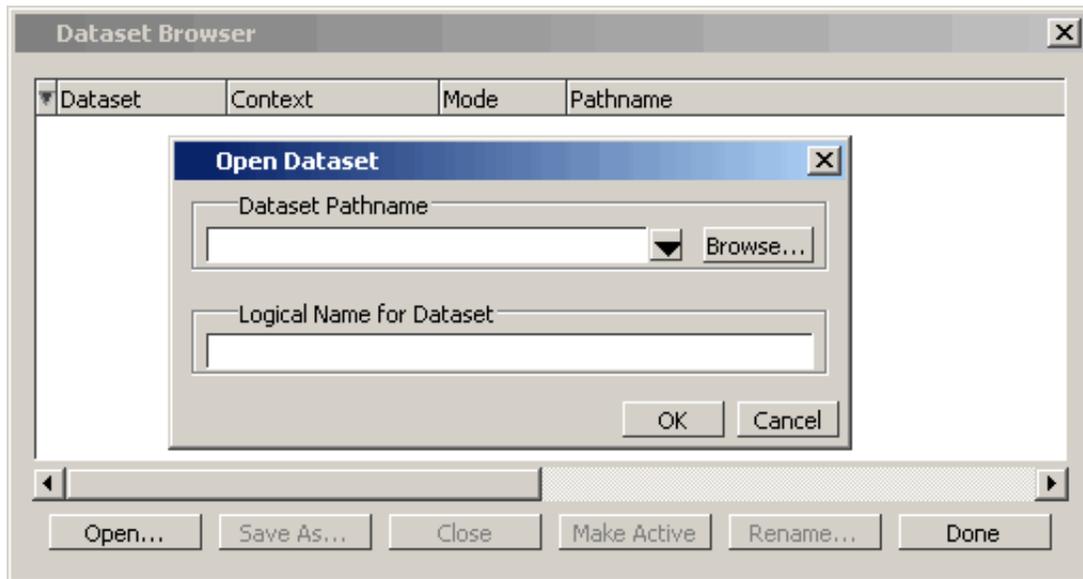
If there is only one processor available the behavior is disabled. The behavior is not available on a Windows system.

Opening Datasets

To open a dataset, do one of the following:

- Select **File > Open** to open the Open File dialog and set the “Files of type” field to Log Files (*.wlf). Then select the .wlf file you want and click the Open button.
- Select **File > Datasets** to open the Dataset Browser; then click the Open button to open the Open Dataset dialog (Figure 14-2).

Figure 14-2. Open Dataset Dialog Box



- Use the [dataset open](#) command to open either a saved dataset or to view a running simulation dataset: *vsim.wlf*. Running simulation datasets are automatically updated.

The Open Dataset dialog includes the following options:

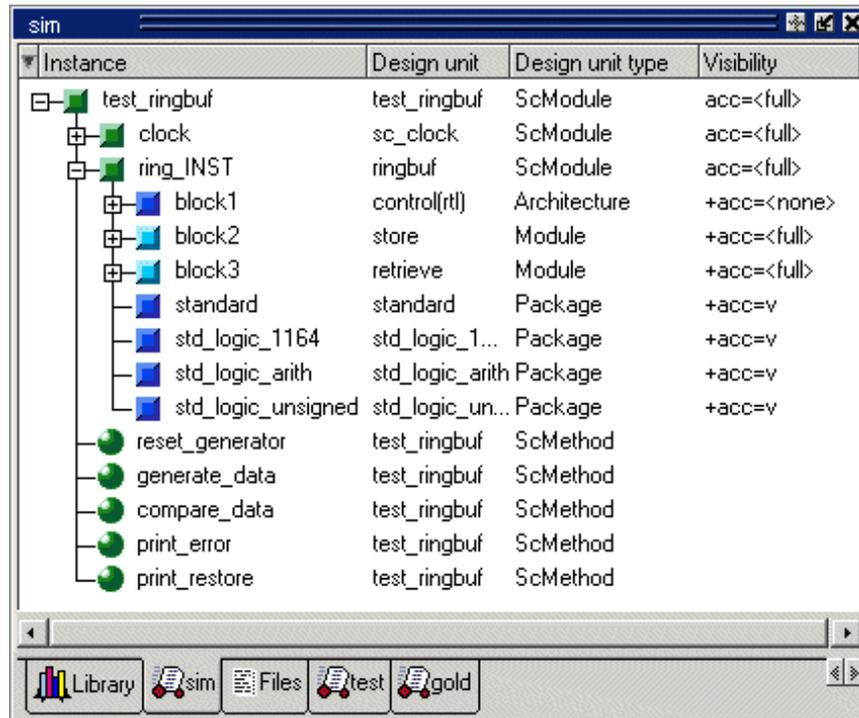
- **Dataset Pathname** — Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset** — This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing Dataset Structure

Each dataset you open creates a structure tab in the Main window. The tab is labeled with the name of the dataset and displays a hierarchy of the design units in that dataset.

The graphic below shows three structure tabs: one for the active simulation (*sim*) and one each for two datasets (*test* and *gold*).

Figure 14-3. Structure Tabs



If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking the arrow icons at the bottom right-hand corner of the window.

Structure Tab Columns

Table 14-2 lists the columns displayed in each structure tab by default.

Table 14-2. Structure Tab Columns

Column name	Description
Instance	the name of the instance
Design unit	the name of the design unit
Design unit type	the type (for example, Module, Entity, and so forth) of the design unit
Visibility	the current visibility of the object as it relates to design optimization; see Design Object Visibility for Designs with PLI for more information

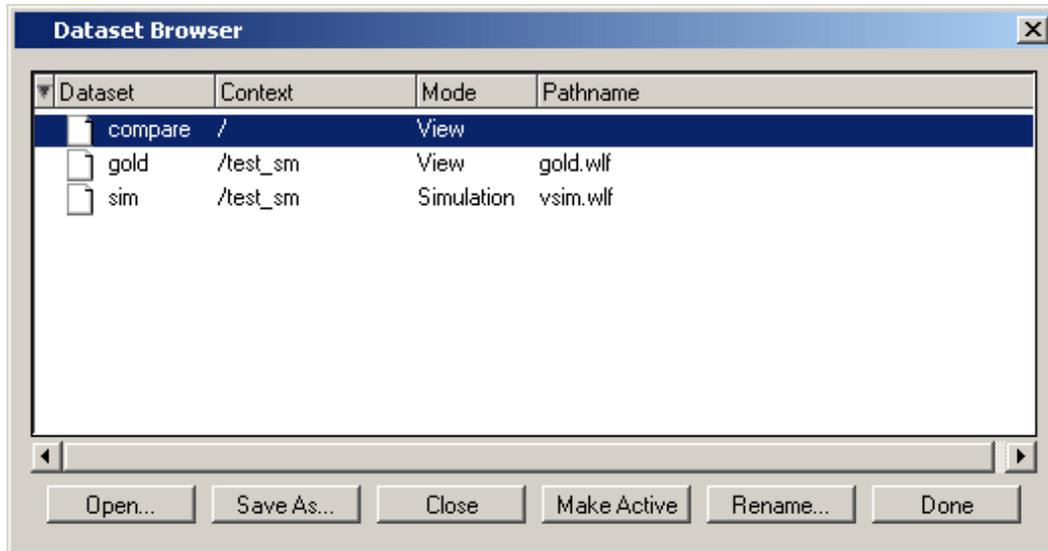
Aside from the columns listed above, there are numerous columns related to code coverage that can be displayed in structure tabs. You can hide or show columns by right-clicking a column name and selecting the name on the list.

Managing Multiple Datasets

Managing Multiple Datasets in the GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **File > Datasets**.

Figure 14-4. The Dataset Browser



Command Line

You can open multiple datasets when the simulator is invoked by specifying more than one **vsim -view <filename>** option. By default the dataset prefix will be the filename of the WLF file. You can specify a different dataset name as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

-view <dataset>=<filename>

For example:

vsim -view foo=vsim.wlf

ModelSim designates one of the datasets to be the active dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's Structure window, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the [environment](#) command to change the active dataset.

Design regions and signal names can be fully specified over multiple WLF files by using the dataset name as a prefix in the path. For example:

sim:/top/alu/out

view:/top/alu/out

golden:.top.alu.out

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting:

- List Window active: List > List Preferences; Window Properties tab > Dataset Prefix pane
- Wave Window active: Wave > Wave Preferences; Display tab > Dataset Prefix Display pane

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the [environment](#) command, 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.

You can lock the Objects window to a specific context of a dataset. Being locked to a dataset means that the pane updates only when the content of that dataset changes. If locked to both a dataset and a context (such as test: /top/foo), the pane will update only when that specific context changes. You specify the dataset to which the pane is locked by selecting **File > Environment**.

Restricting the Dataset Prefix Display

You can turn dataset prefix viewing on or off by setting the value of a preference variable called DisplayDatasetPrefix. Setting the variable value to 1 displays the prefix, setting it to 0 does not. It is set to 1 by default. To change the value of this variable, do the following:

1. Choose Tools > Edit Preferences... from the main menu.
2. In the Preferences dialog box, click the By Name tab.
3. Scroll to find the Preference Item labeled Main and click [+] to expand the listing of preference variables.
4. Select the DisplayDatasetPrefix variable then click the Change Value... button.
5. In the Change Preference Value dialog box, type a value of 0 or 1, where

- 0 = turns off prefix display
 - 1 = turns on prefix display (default)
6. Click OK; click OK.

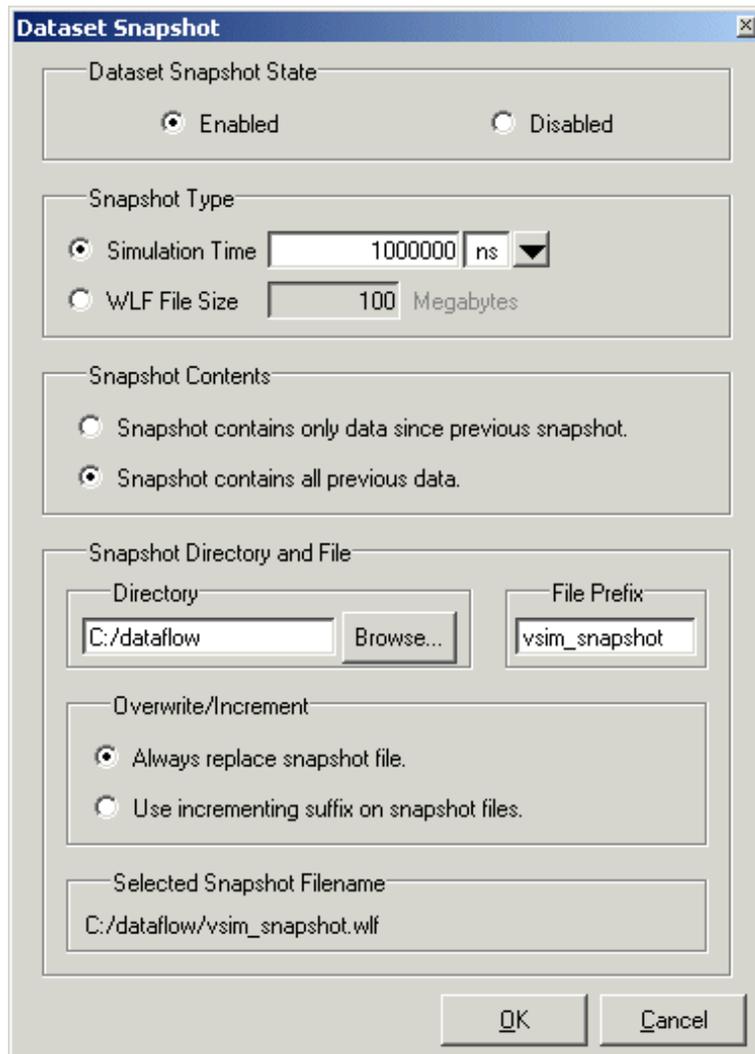
Additionally, you can prevent display of the dataset prefix by using the environment `-nodataset` command to view a dataset. To enable display of the prefix, use the [environment](#) `-dataset` command (note that you do not need to specify this command argument if the `DisplayDatasetPrefix` variable is set to 1). These arguments of the environment command override the value of the `DisplayDatasetPrefix` variable.

Saving at Intervals with Dataset Snapshot

Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic "snapshots" of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Once you have logged the appropriate objects, select **Tools > Dataset Snapshot** (Wave window).

Figure 14-5. Dataset Snapshot Dialog



Collapsing Time and Delta Steps

By default ModelSim collapses delta steps. This means each logged signal that has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. The event order in the WLF file matches the order of the first events of each signal.

You can configure how ModelSim collapses time and delta steps using arguments to the `vsim` command or by setting the `WLFCollapseMode` variable in the `modelsim.ini` file. The table below summarizes the arguments and how they affect event recording.

Table 14-3. vsim Arguments for Collapsing Time and Delta Steps

vsim argument	effect	modelsim.ini setting
-nowlfcollapse	All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.	WLFCollapseMode = 0
-wlfcollapsedelta	Each logged signal which has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. Default.	WLFCollapseMode = 1
-wlfcollapsesttime	Same as delta collapsing but at the timestep granularity.	WLFCollapseMode = 2

When a run completes that includes single stepping or hitting a breakpoint, all events are flushed to the WLF file regardless of the time collapse mode. It's possible that single stepping through part of a simulation may yield a slightly different WLF file than just running over that piece of code. If particular detail is required in debugging, you should disable time collapsing.

Virtual Objects

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the ModelSim simulation kernel. ModelSim supports the following kinds of virtual objects:

- [Virtual Signals](#)
- [Virtual Functions](#)
- [Virtual Regions](#)
- [Virtual Types](#)

Virtual objects are indicated by an orange diamond as illustrated by *Bus1* in [Figure 14-6](#):

The definitions of virtuals can be saved to a macro file using the [virtual save](#) command. By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

Implicit and Explicit Virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Objects pane 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 Objects, Wave, and List windows and accessed by the [examine](#) command, but cannot be set by the [force](#) command.

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"

You can also use virtual functions to convert signal types and map signal values.

The result type of a virtual function 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.

To create a virtual function, use the [virtual function](#) command.

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Objects,

Wave, or List window. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual Regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

To create and attach a virtual region, use the [virtual region](#) command.

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.

To create a virtual type, use the [virtual type](#) command.

Chapter 15

Waveform Analysis

When your simulation finishes, you typically use the Wave window to analyze the graphical display of waveforms to assess and debug your design. However, you can also look at waveform data in a textual format in the List window.

To analyze waveforms in ModelSim, follow these steps:

1. Compile your files.
2. Load your design.
3. Add objects to the Wave or List window.

```
add wave <object_name>
add list <object_name>
```

4. Run the design.

Objects You Can View

The list below identifies the types of objects can be viewed in the Wave or List window. Refer to the section “[Using the WildcardFilter Preference Variable](#)” for information on controlling the information that is added to the Wave window when using wild cards.

- **VHDL objects** — (indicated by dark blue diamond in the Wave window)
signals, aliases, process variables, and shared variables
- **Verilog and SystemVerilog objects** — (indicated by light blue diamond in the Wave window)
nets, registers, variables, named events, and classes
- **SystemC objects** — (indicated by a green diamond in the Wave window)
primitive channels and ports
- **Virtual objects** — (indicated by an orange diamond in the Wave window)
virtual signals, buses, and functions, see; [Virtual Objects](#) for more information
- **Comparisons** — (indicated by a yellow triangle)
comparison regions and comparison signals; see [Waveform Compare](#) for more information

- **Assertions** — (indicated by a triangle in the Wave window; light-blue for SystemVerilog and magenta for PSL)
PSL and SystemVerilog assertions
- **Cover Directives** — (indicated by a chevron in the Wave window; light-blue for SystemVerilog and magenta for PSL)
PSL and SystemVerilog cover directives

Wave Window Overview

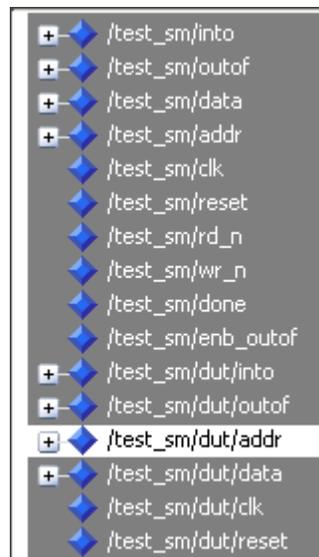
The Wave window opens in the Main window as shown [Figure 15-1](#). The window can be undocked from the main window by clicking the Undock button in the window header.

When the Wave window is docked in the Main window, all menus and icons that were in the undocked Wave window move into the Main window menu bar and toolbar.

Wave Window Panes

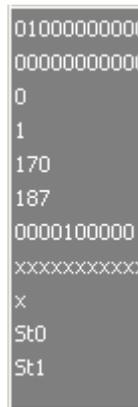
The Wave window is divided into a number of window panes. The Object Pathnames Pane displays object paths.

Figure 15-1. Wave Window Object Pathnames Pane



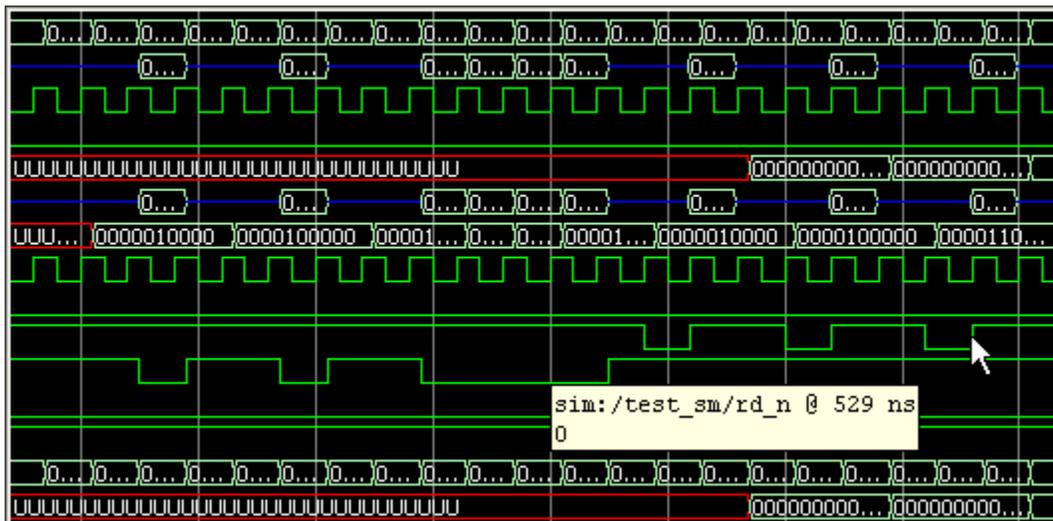
The Object Values Pane displays the value of each object in the pathnames pane at the time of the selected cursor.

Figure 15-2. Wave Window Object Values Pane



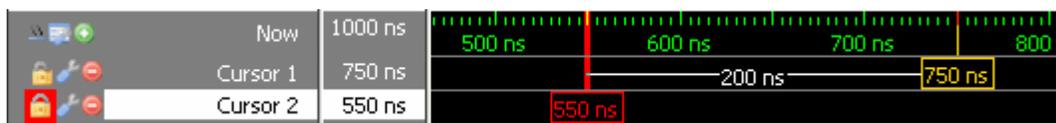
The Waveform Pane displays the object waveforms over the time of the simulation.

Figure 15-3. Wave Window Waveform Pane



The Cursor Pane displays cursor names, cursor values and the cursor locations on the timeline. This pane also includes a toolbox that gives you quick access to cursor and timeline features and configurations.

Figure 15-4. Wave Window Cursor Pane



All of these panes can be resized by clicking and dragging the bar between any two panes.

In addition to these panes, the Wave window also contains a Messages bar at the top of the window. The Messages bar contains indicators pointing to the times at which a message was output from the simulator.

Adding Objects with Mouse Actions

- Drag and drop objects into the Wave or List window from the Structure, Processes, Memory, Objects, Source, or Locals windows. When objects are dragged into the Wave window, the add wave command is reflected in the Transcript window.
- Drag objects from the Wave window to the List window and vice versa.
- Select the objects in the first window, then drop them into the Wave window. Depending on what you select, all objects or any portion of the design can be added.
- Place the cursor over an individual object or selected objects in the Objects or Locals windows, then click the middle mouse button to place the object(s) in the Wave window.

Adding Objects with Menu Selections

- **Add > window** — Add objects to the Wave window, List window, or Log file.
- **Add Selected to Window Button** — Add objects to the Wave, Dataflow, Schematic, List, and Watch windows.

You can also add objects using right-click popup menus. For example, if you want to add all signals in a design to the Wave window you can do one of the following:

- Right-click a design unit in a Structure (sim) window and select **Add > To Wave > All Items in Design** from the popup context menu.
- Right-click anywhere in the Objects window and select **Add > To Wave > Signals in Design** from the popup context menu.

Adding Objects with a Command

Use the [add list](#) or [add wave](#) commands to add objects from the command line. For example:

```
VSIM> add wave /proc/a
```

Adds signal */proc/a* to the Wave window.

```
VSIM> add list *
```

Adds all the objects in the current region to the List window.

```
VSIM> add wave -r /*
```

Adds all objects in the design to the Wave window.

Refer to the section “[Using the WildcardFilter Preference Variable](#)” for information on controlling the information that is added to the Wave window when using wild cards.

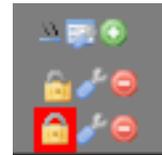
Adding Objects with a Window Format File

Select **File > Load** and specify a previously saved format file. See [Saving the Window Format](#) for details on how to create a format file.

Working with Cursors

Cursors mark simulation time in the Wave window. When ModelSim first draws the Wave window, it places one cursor at time zero. Clicking anywhere in the waveform display brings the nearest cursor to the mouse location. You can use cursors to find transitions, a rising or falling edge, and to measure time intervals.

The [Cursor and Timeline Toolbox](#) on the left side of the cursor pane gives you quick access to cursor and timeline features.



[Table 15-1](#) summarizes common cursor actions you can perform with the icons in the toolbox, or with menu selections.

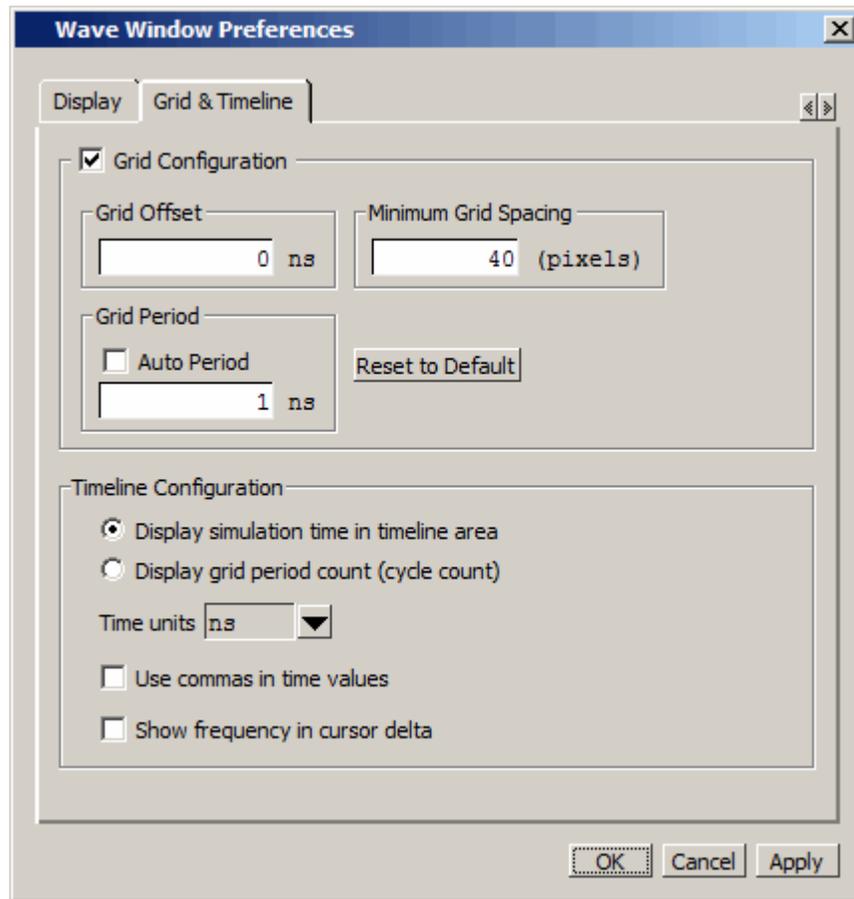
Table 15-1. Actions for Cursors

Icon	Action	Menu path or command (Wave window docked)	Menu path or command (Wave window undocked)
	Toggle leaf names <-> full names	Wave > Wave Preferences > Display Tab	Tools > Wave Preferences > Display Tab
	Edit grid and timeline properties	Wave > Wave Preferences > Grid and Timeline Tab	Tools > Wave Preferences > Grid and Timeline Tab
	Add cursor	Add > To Wave > Cursor	Add > Cursor
	Edit cursor	Wave > Edit Cursor	Edit > Edit Cursor
	Delete cursor	Wave > Delete Cursor	Edit > Delete Cursor
	Lock cursor	Wave > Edit Cursor	Edit > Edit Cursor
NA	Select a cursor	Wave > Cursors	View > Cursors
NA	Zoom In on Active Cursor	Wave > Zoom > Zoom Cursor	View > Zoom > Zoom Cursor

The **Toggle leaf names <-> full names** icon allows you to switch from displaying full pathnames (the default) in the Pathnames Pane to displaying leaf or short names. You can also control the number of path elements in the Wave Window Preferences dialog. Refer to [Hiding/Showing Path Hierarchy](#).

The **Edit grid and timeline properties** icon opens the Wave Window Properties dialog to the Grid & Timeline tab ([Figure 15-7](#)).

Figure 15-7. Grid and Timeline Properties



- The Grid Configuration selections allow you to set grid offset, minimum grid spacing, and grid period. You can also reset these grid configuration settings to their default values.
- The Timeline Configuration selections give you a user-definable time scale. You can display simulation time on a timeline or a clock cycle count. If you select Display simulation time in timeline area, use the Time Units dropdown list to select one of the following as the timeline unit:

fs, ps, ns, us, ms, sec, min, hr

Note



The time unit displayed in the Wave window does not affect the simulation time that is currently defined.

The current configuration is saved with the wave format file so you can restore it later.

- The **Show frequency in cursor delta** box causes the timeline to display the difference (delta) between adjacent cursors as frequency. By default, the timeline displays the delta between adjacent cursors as time.

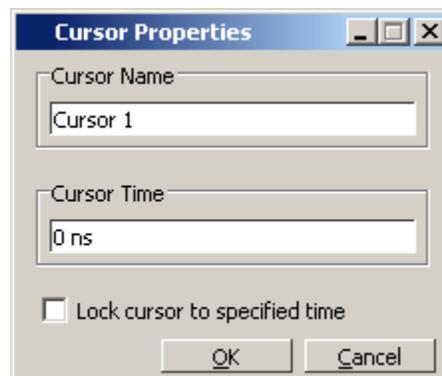
Adding Cursors

To add cursors when the Wave window is active you can:

- click the Insert Cursor icon
- choose **Add > To Wave > Cursor** from the menu bar
- press the “A” key while the mouse pointer is located in the cursor pane
- right click in the cursor pane and select **New Cursor @ <time> ns** to place a new cursor at a specific time.

Each added cursor is given a default cursor name (Cursor 2, Cursor 3, and so forth) which can be changed by simply right-clicking the cursor name, then typing in a new name, or by clicking the **Edit this cursor** icon. The Edit this cursor icon will open the Cursor Properties dialog (Figure 15-8), where you assign a cursor name and time. You can also lock the cursor to the specified time.

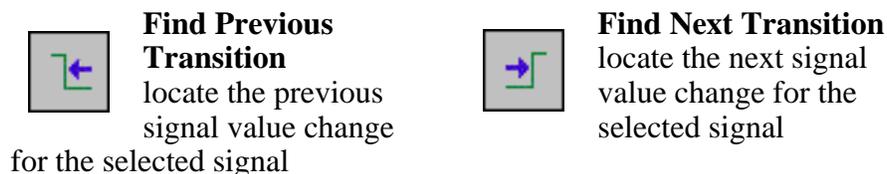
Figure 15-8. Cursor Properties Dialog Box



Jumping to a Signal Transition

You can move the active (selected) cursor to the next or previous transition on the selected signal using these two toolbar icons shown in Figure 15-9.

Figure 15-9. Find Previous and Next Transition Icons



These actions will not work on locked cursors.

Measuring Time with Cursors in the Wave Window

ModelSim uses cursors to measure time in the Wave window. Cursors extend a vertical line over the waveform display and identify a specific simulation time.

When the Wave window is first drawn it contains two cursors — the **Now** cursor, and **Cursor 1** (Figure 15-10).

Figure 15-10. Original Names of Wave Window Cursors



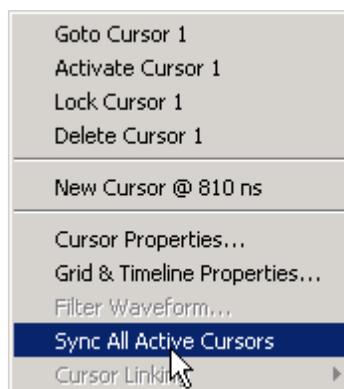
The **Now** cursor is always locked to the current simulation time and it is not manifested as a graphical object (vertical cursor bar) in the Wave window.

Cursor 1 is located at time zero. Clicking anywhere in the waveform display moves the **Cursor 1** vertical cursor bar to the mouse location and makes this cursor the selected cursor. The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines.

Syncing All Active Cursors

You can synchronize the active cursors within all open Wave windows and the Wave viewers in the Dataflow and Schematic windows. Simply right-click the time value of the active cursor in any window and select Sync All Active Cursors from the popup menu (Figure 15-11).

Figure 15-11. Sync All Active Cursors

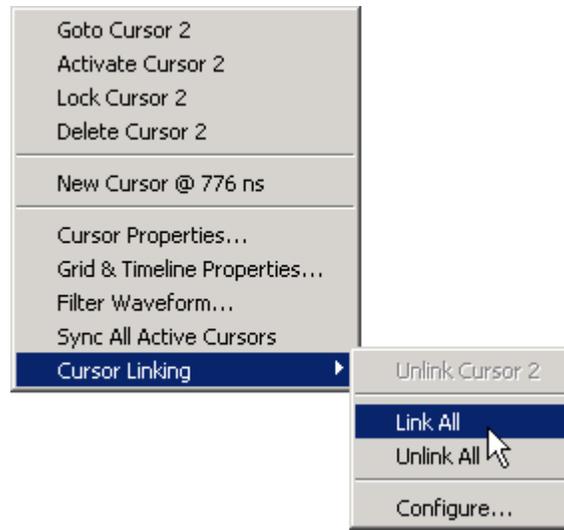


When all active cursors are synced, moving a cursor in one window will automatically move the active cursors in all opened Wave windows to the same time location. This option is also available by selecting **Wave > Cursors > Sync All Active Cursors** in the menu bar when a Wave window is active.

Linking Cursors

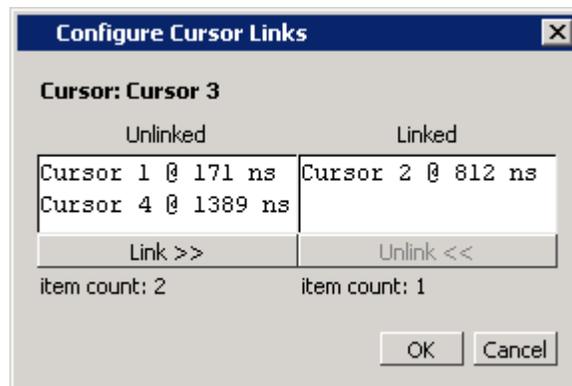
Cursors within the Wave window can be linked together, allowing you to move two or more cursors together across the simulation timeline. You simply click one of the linked cursors and drag it left or right on the timeline. The other linked cursors will move by the same amount of time. You can link all displayed cursors by right-clicking the time value of any cursor in the timeline, as shown in [Figure 15-12](#), and selecting **Cursor Linking > Link All**.

Figure 15-12. Cursor Linking Menu



You can link and unlink selected cursors by selecting the time value of any cursor and selecting **Cursor Linking > Configure** to open the **Configure Cursor Links** dialog ([Figure 15-13](#)).

Figure 15-13. Configure Cursor Links Dialog



Understanding Cursor Behavior

The following list describes how cursors behave when you click in various panes of the Wave window:

- If you click in the waveform pane, the closest unlocked cursor to the mouse position is selected and then moved to the mouse position.
- Clicking in a horizontal track in the cursor pane selects that cursor and moves it to the mouse position.
- Cursors snap to the nearest waveform edge to the left if you click or drag a cursor along the selected waveform to within ten pixels of a waveform edge. You can set the snap distance in the Display tab of the Window Preferences dialog. Select **Tools > Options > Wave Preferences** when the Wave window is docked in the Main window MDI frame. Select **Tools > Window Preferences** when the Wave window is a stand-alone, undocked window.
- You can position a cursor without snapping by dragging a cursor in the cursor pane below the waveforms.

Shortcuts for Working with Cursors

There are a number of useful keyboard and mouse shortcuts related to the actions listed above:

- Select a cursor by clicking the cursor name.
- Jump to a hidden cursor (one that is out of view) by double-clicking the cursor name.
- Name a cursor by right-clicking the cursor name and entering a new value. Press <Enter> on your keyboard after you have typed the new name.
- Move a locked cursor by holding down the <shift> key and then clicking-and-dragging the cursor.
- Move a cursor to a particular time by right-clicking the cursor value and typing the value to which you want to scroll. Press <Enter> on your keyboard after you have typed the new value.

Setting Time Markers in the List Window

Time markers in the List window are similar to cursors in the Wave window. Time markers tag lines in the data table so you can quickly jump back to that time. Markers are indicated by a thin box surrounding the marked line.

The expanded time function makes these intermediate values visible in the Wave window. Expanded time shows the actual order in which objects change values and shows all transitions of each object within a given time step.

Expanded Time Terminology

- **Simulation Time** — the basic time step of the simulation. The final value of each object at each simulation time is what is displayed by default in the Wave window.
- **Delta Time** — the time intervals or steps taken to evaluate the design without advancing simulation time. Object values at each delta time step are viewed in the List window or by using the `-delta` argument of the `examine` command. Refer to [Delta Delays](#) for more information.
- **Event Time** — the time intervals that show each object value change as a separate event and that shows the relative order in which these changes occur

During a simulation, events on different objects in a design occur in a particular order or sequence. Typically, this order is not important and only the final value of each object for each simulation time step is important. However, in situations like debugging glitches on clocked objects or race conditions, the order of events is important. Unlike simulation time steps and delta time steps, only one object can have a single value change at any one event time. Object values and the exact order which they change can be saved in the `.wlf` file.

- **Expanded Time** — the Wave window feature that expands single simulation time steps to make them wider, allowing you to see object values at the end of each delta cycle or at each event time within the simulation time.
- **Expand** — causes the normal simulation time view in the Wave window to show additional detailed information about when events occurred during a simulation.
- **Collapse** — hides the additional detailed information in the Wave window about when events occurred during a simulation.

Recording Expanded Time Information

You can use the `vsim` command, or the `WLFCollapseMode` variable in the `modelsim.ini` file, to control recording of expanded time information in the `.wlf` file.

Table 15-3. Recording Delta and Event Time Information

vsim command argument	modelsim.ini setting	effect
<code>-nowlfcollapse</code>	<code>WLFCollapseMode = 0</code>	All events for each logged signal are recorded to the <code>.wlf</code> file in the exact order they occur in the simulation.

Table 15-3. Recording Delta and Event Time Information

vsim command argument	modelsim.ini setting	effect
-wlfcollapsedelta	WLFCollapseMode = 1 (Default)	Each logged signal that has events during a simulation delta has its final value recorded in the <i>.wlf</i> file when the delta has expired.
-wlfcollapsetime	WLFCollapseMode = 2	Similar to delta collapsing but at the simulation time step granularity.

Recording Delta Time

Delta time information is recorded in the *.wlf* file using the **-wlfcollapsedelta** argument of **vsim** or by setting the `WLFCollapseMode` *modelsim.ini* variable to 1. This is the default behavior.

Recording Event Time

To save multiple value changes of an object during a single time step or single delta cycle, use the **-nowlfcollapse** argument with **vsim**, or set `WLFCollapseMode` to 0. Unlike delta times (which are explicitly saved in the *.wlf* file), event time information exists implicitly in the *.wlf* file. That is, the order in which events occur in the simulation is the same order in which they are logged to the *.wlf* file, but explicit event time values are not logged.

Choosing Not to Record Delta or Event Time

You can choose not to record event time or delta time information to the *.wlf* file by using the **-wlfcollapsetime** argument with **vsim**, or by setting `WLFCollapseMode` to 2. This will prevent detailed debugging but may reduce the size of the *.wlf* file and speed up the simulation.

Viewing Expanded Time Information in the Wave Window

Expanded time information is displayed in the Wave window toolbar, the right portion of the Messages bar, the Waveform pane, the time axis portion of the Cursor pane, and the Waveform pane horizontal scroll bar as described below.

- **Expanded Time Toolbar** — The Expanded Time toolbar can (optionally) be displayed in the toolbar area of the undocked Wave window or the toolbar area of the Main window when the Wave window is docked. It contains three exclusive toggle buttons for selecting the Expanded Time mode (see [Toolbar Selections for Expanded Time Modes](#)) and four buttons for expanding and collapsing simulation time.
- **Messages Bar** — The right portion of the Messages Bar is scaled horizontally to align properly with the Waveform pane and the time axis portion of the Cursor pane.

- **Waveform Pane Horizontal Scroll Bar** — The position and size of the thumb in the Waveform pane horizontal scroll bar is adjusted to correctly reflect the current state of the Waveform pane and the time axis portion of the Cursor pane.
- **Waveform Pane and the Time Axis Portion of the Cursor Pane** — By default, the Expanded Time is off and simulation time is collapsed for the entire time range in the Waveform pane. When the Delta Time mode is selected (see [Recording Delta Time](#)), simulation time remains collapsed for the entire time range in the Waveform pane. A red dot is displayed in the middle of all waveforms at any simulation time where multiple value changes were logged for that object.

Figure 15-15 illustrates the appearance of the Waveform pane when viewing collapsed event time or delta time. It shows a simulation with three signals, s1, s2, and s3. The red dots indicate multiple transitions for s1 and s2 at simulation time 3ns.

Figure 15-15. Waveform Pane with Collapsed Event and Delta Time

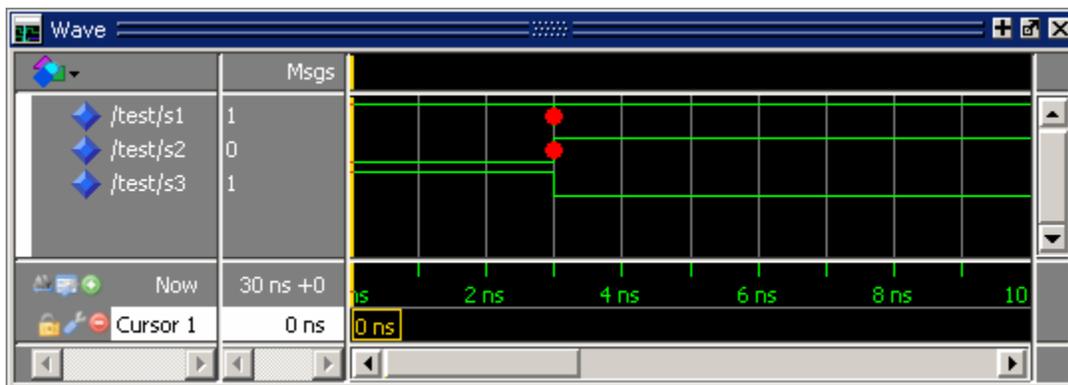
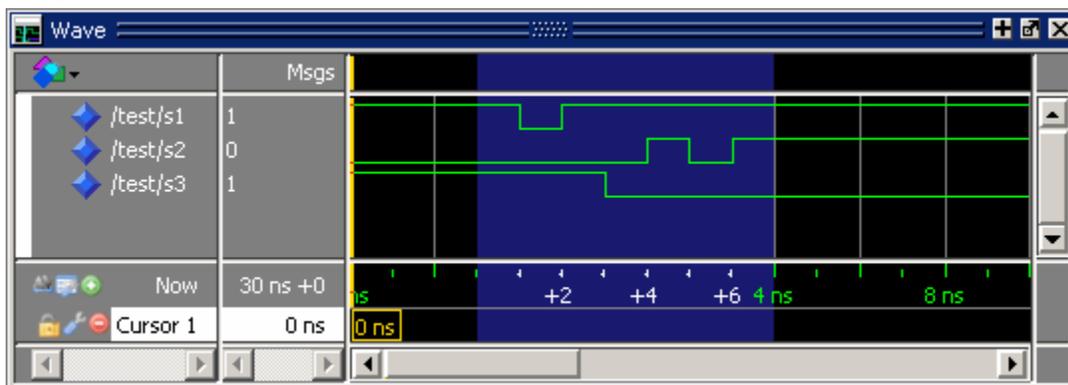


Figure 15-16 shows the Waveform pane and the timescale from the Cursors pane after expanding simulation time at time 3ns. The background color is blue for expanded sections in Delta Time mode and green for expanded sections in Event Time mode.

Figure 15-16. Waveform Pane with Expanded Time at a Specific Time



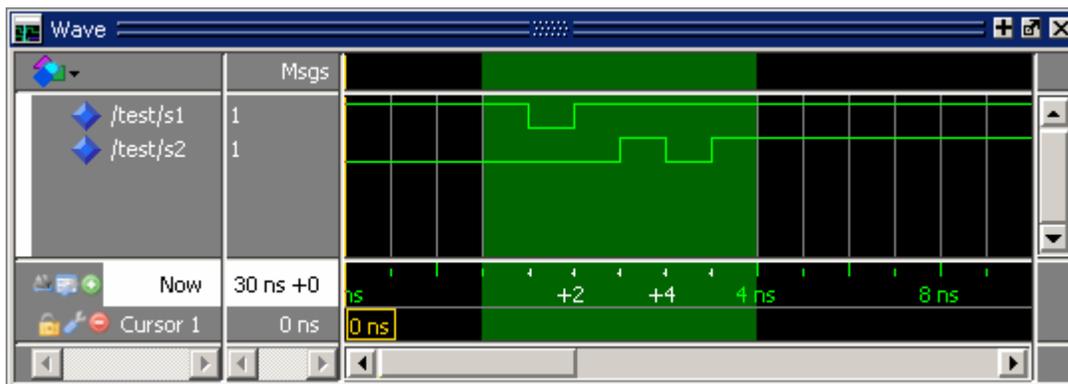
In Delta Time mode, more than one object may have an event at the same delta time step. The labels on the time axis in the expanded section indicate the delta time steps within the given simulation time.

In Event Time mode, only one object may have an event at a given event time. The exception to this is for objects that are treated atomically in the simulator and logged atomically. The individual bits of a SystemC vector, for example, could change at the same event time.

Labels on the time axis in the expanded section indicate the order of events from all of the objects added to the Wave window. If an object that had an event at a particular time but it is not in the viewable area of the Waveform panes, then there will appear to be no events at that time.

Depending on which objects have been added to the Wave window, a specific event may happen at a different event time. For example, if *s3* shown in [Figure 15-16](#), had not been added to the Wave window, the result would be as shown in [Figure 15-17](#).

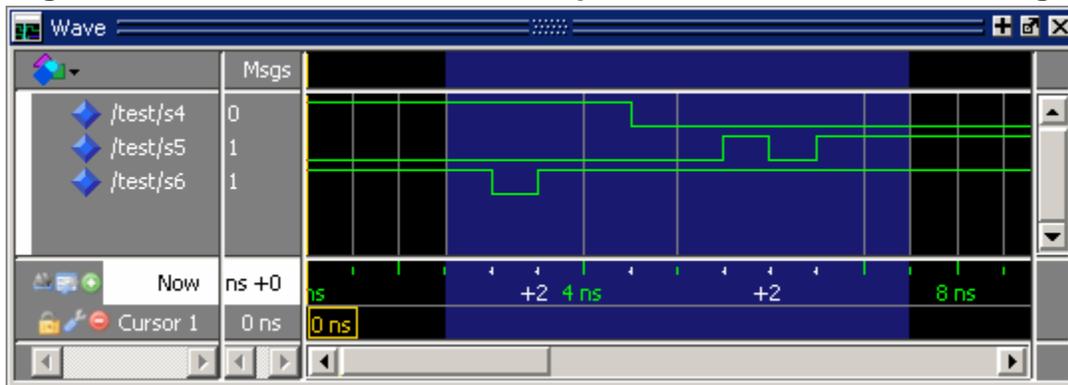
Figure 15-17. Waveform Pane with Event Not Logged



Now the first event on *s2* occurs at event time $3\text{ns} + 2$ instead of event time $3\text{ns} + 3$. If *s3* had been added to the Wave window (whether shown in the viewable part of the window or not) but was not visible, the event on *s2* would still be at $3\text{ns} + 3$, with no event visible at $3\text{ns} + 2$.

[Figure 15-18](#) shows an example of expanded time over the range from 3ns to 5ns . The expanded time range displays delta times as indicated by the blue background color. (If Event Time mode is selected, a green background is displayed.)

Figure 15-18. Waveform Pane with Expanded Time Over a Time Range



When scrolling horizontally, expanded sections remain expanded until you collapse them, even when scrolled out of the visible area. The left or right edges of the Waveform pane are viewed in either expanded or collapsed sections.

Expanded event order or delta time sections appear in all panes when multiple Waveform panes exist for a Wave window. When multiple Wave windows are used, sections of expanded event or delta time are specific to the Wave window where they were created.

For expanded event order time sections when multiple datasets are loaded, the event order time of an event will indicate the order of that event relative to all other events for objects added to that Wave window for that object's dataset only. That means, for example, that signal sim:s1 and gold:s2 could both have events at time 1ns+3.

Note



The order of events for a given design will differ for optimized versus unoptimized simulations, and between different versions of ModelSim. The order of events will be consistent between the Wave window and the List window for a given simulation of a particular design, but the event numbering may differ. See [Expanded Time Viewing in the List Window](#).

You may display any number of disjoint expanded times or expanded ranges of times.

Customizing the Expanded Time Wave Window Display

As noted above, the Wave window background color is blue instead of black for expanded sections in Delta Time mode and green for expanded sections in Event Time mode.

The background colors for sections of expanded event time are changed as follows:

1. Select **Tools > Edit Preferences** from the menus. This opens the Preferences dialog.
2. Select the By Name tab.
3. Scroll down to the Wave selection and click the plus sign (+) for Wave.

4. Change the values of the Wave Window variables `waveDeltaBackground` and `waveEventBackground`.

Selecting the Expanded Time Display Mode

There are three Wave window expanded time display modes: Event Time mode, Delta Time mode, and Expanded Time off. These display modes are initiated by menu selections, toolbar selections, or via the command line.

Menu Selections for Expanded Time Display Modes

Table 15-4 shows the menu selections for initiating expanded time display modes.

Table 15-4. Menu Selections for Expanded Time Display Modes

action	menu selection with Wave window docked or undocked
select Delta Time mode	docked: Wave > Expanded Time > Delta Time Mode undocked: View > Expanded Time > Delta Time Mode
select Event Time mode	docked: Wave > Expanded Time > Event Time Mode undocked: View > Expanded Time > Event Time Mode
disable Expanded Time	docked: Wave > Expanded Time > Expanded Time Off undocked: View > Expanded Time > Expanded Time Off

Select Delta Time Mode or Event Time Mode from the appropriate menu according to Table 15-4 to have expanded simulation time in the Wave window show delta time steps or event time steps respectively. Select Expanded Time Off for standard behavior (which is the default).

Toolbar Selections for Expanded Time Modes

There are three exclusive toggle buttons in the [Wave Expand Time Toolbar](#) for selecting the time mode used to display expanded simulation time in the Wave window.

- The "Expanded Time Deltas Mode" button displays delta time steps.
- The "Expanded Time Events Mode" button displays event time steps.
- The "Expanded Time Off" button turns off the expanded time display in the Wave window.

Clicking any one of these buttons on toggles the other buttons off. This serves as an immediate visual indication about which of the three modes is currently being used. Choosing one of these modes from the menu bar or command line also results in the appropriate resetting of these three buttons. The "Expanded Time Off" button is selected by default.

In addition, there are four buttons in the [Wave Expand Time Toolbar](#) for expanding and collapsing simulation time.

- The “Expand All Time” button expands simulation time over the entire simulation time range, from time 0 to the current simulation time.
- The “Expand Time At Active Cursor” button expands simulation time at the simulation time of the active cursor.
- The “Collapse All Time” button collapses simulation time over entire simulation time range.
- The “Collapse Time At Active Cursor” button collapses simulation time at the simulation time of the active cursor.

Command Selection of Expanded Time Mode

The command syntax for selecting the time mode used to display objects in the Wave window is:

```
wave expand mode [-window <win>] none | deltas | events
```

Use the wave expand mode command to select which mode is used to display expanded time in the wave window. This command also results in the appropriate resetting of the three toolbar buttons.

Switching Between Time Modes

If one or more simulation time steps have already been expanded to view event time or delta time, then toggling the Time mode by any means will cause all of those time steps to be redisplayed in the newly selected mode.

Expanding and Collapsing Simulation Time

Simulation time may be expanded to view delta time steps or event time steps at a single simulation time or over a range of simulation times. Simulation time may be collapsed to hide delta time steps or event time steps at a single simulation time or over a range of simulation times. You can expand or collapse the simulation time with menu selections, toolbar selections, via commands, or with the mouse cursor.

- Expanding/Collapsing Simulation Time with Menu Selections — Select **Wave > Expanded Time** when the Wave window is docked, and **View > Expanded Time** when the Wave window is undocked. You can expand/collapse over the full simulation time range, over a specified time range, or at the time of the active cursor,.
- Expanding/Collapsing Simulation Time with Toolbar Selections — There are four buttons in the toolbar for expanding and collapsing simulation time in the Wave window: Expand Full, Expand Cursor, Collapse Full, and Collapse Cursor.

- Expanding/Collapsing Simulation Time with Commands — There are six commands for expanding and collapsing simulation time in the Wave window.

wave expand all

wave expand range

wave expand cursor

wave collapse all

wave collapse range

wave collapse cursor

These commands have the same behavior as the corresponding menu and toolbar selections. If valid times are not specified, for **wave expand range** or **wave collapse range**, no action is taken. These commands effect all Waveform panes in the Wave window to which the command applies.

Expanded Time Viewing in the List Window

Event time may be shown in the List window in the same manner as delta time by using the **-delta events** option with the **configure list** command.

When the List window displays event times, the event time is relative to events on other signals also displayed in the List window. This may be misleading, as it may not correspond to event times displayed in the Wave window for the same events if different signals are added to the Wave and List windows.

The **write list** command (when used after the **configure list -delta events** command) writes a list file in tabular format with a line for every event. Please note that this is different from the **write list -events** command, which writes a non-tabular file using a print-on-change format.

The following examples illustrate the appearance of the List window and the corresponding text file written with the **write list** command after various options for the **configure list -delta** command are used.

Figure 15-19 shows the appearance of the List window after the **configure list -delta none** command is used. It corresponds to the file resulting from the **write list** command. No column is shown for deltas or events.

Figure 15-19. List Window After configure list -delta none Option is Used

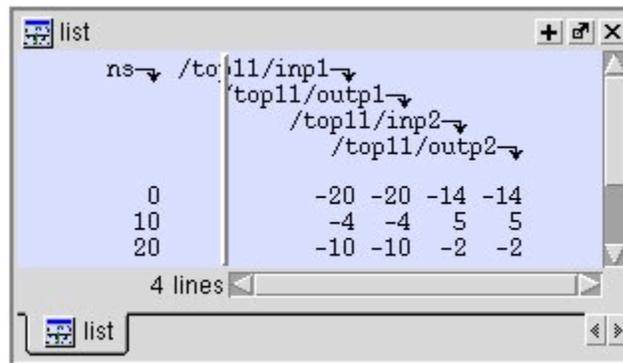


Figure 15-20 shows the appearance of the List window after the `configure list -delta collapse` command is used. It corresponds to the file resulting from the `write list` command. There is a column for delta time and only the final delta value and the final value for each signal for each simulation time step (at which any events have occurred) is shown.

Figure 15-20. List Window After configure list -delta collapse Option is Used

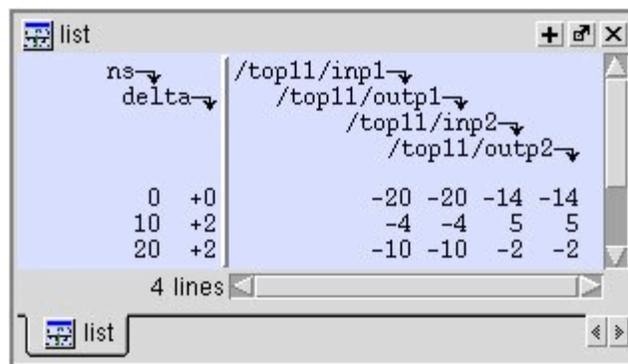


Figure 15-21 shows the appearance of the List window after the `configure list -delta all` option is used. It corresponds to the file resulting from the `write list` command. There is a column for delta time, and each delta time step value is shown on a separate line along with the final value for each signal for that delta time step.

Figure 15-21. List Window After write list -delta all Option is Used

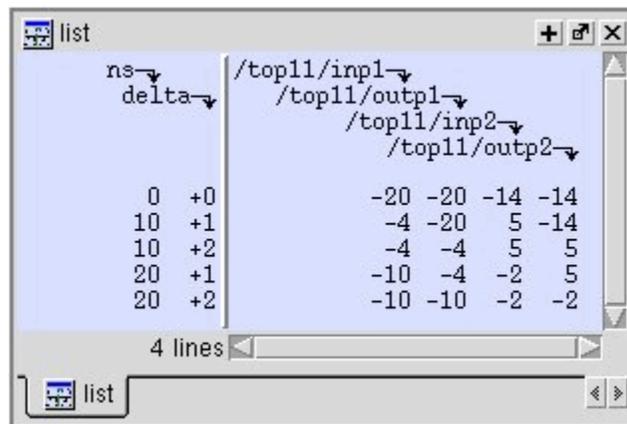
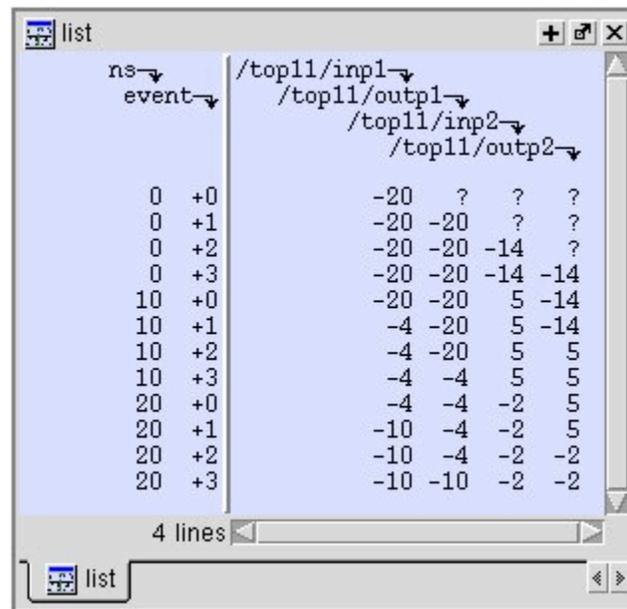


Figure 15-22 shows the appearance of the List window after the `configure list -delta events` command is used. It corresponds to the file resulting from the `write list` command. There is a column for event time, and each event time step value is shown on a separate line along with the final value for each signal for that event time step. Since each event corresponds to a new event time step, only one signal will change values between two consecutive lines.

Figure 15-22. List Window After write list -event Option is Used



Zooming the Wave Window Display

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

Zooming with the Menu, Toolbar and Mouse

You can access Zoom commands in any of the following ways:

- From the **Wave > Zoom** menu selections in the Main window when the Wave window is docked
- From the **View** menu in the Wave window when the Wave window is undocked
- Right-clicking in the waveform pane of the Wave window

These zoom buttons are available on the toolbar:



Zoom In 2x

zoom in by a factor of two from the current view



Zoom In on Active Cursor

centers the active cursor in the waveform display and zooms in



Zoom Mode

change mouse pointer to zoom mode; see below



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

To zoom with the mouse, first enter zoom mode by selecting **View > Zoom > Mouse Mode > Zoom Mode**. The left mouse button then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

Also note the following about zooming with the mouse:

- The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

- You can enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.
- With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

To zoom with your the scroll-wheel of your mouse, hold down the ctrl key at the same time to scroll in and out. The waveform pane will zoom in and out, centering on your mouse cursor

Saving Zoom Range and Scroll Position with Bookmarks

Bookmarks 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 Objects with a Window Format File](#)) and are restored when the format file is read.

Managing Bookmarks

The table below summarizes actions you can take with bookmarks.

Table 15-5. Actions for Bookmarks

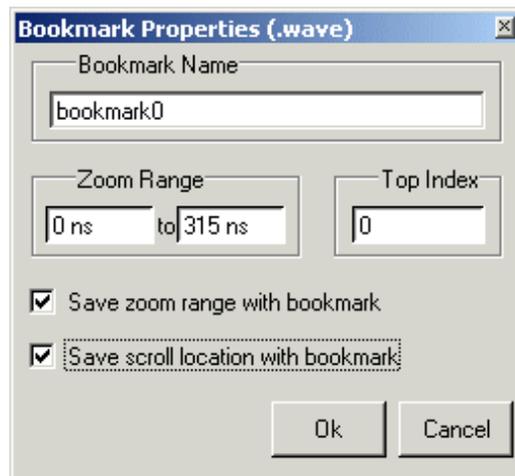
Action	Menu commands (Wave window docked)	Menu commands (Wave window undocked)	Command
Add bookmark	Add > To Wave > Bookmark	Add > Bookmark	bookmark add wave
View bookmark	Wave > Bookmarks > <bookmark_name>	View > Bookmarks > <bookmark_name>	bookmark goto wave
Delete bookmark	Wave > Bookmarks > Bookmarks > <select bookmark then Delete>	View > Bookmarks > Bookmarks > <select bookmark then Delete>	bookmark delete wave

Adding Bookmarks

To add a bookmark, follow these steps:

1. Zoom the Wave window as you see fit using one of the techniques discussed in [Zooming the Wave Window Display](#).
2. If the Wave window is docked, select **Add > Wave > Bookmark**. If the Wave window is undocked, select **Add > Bookmark**.

Figure 15-23. Bookmark Properties Dialog



3. Give the bookmark a name and click OK.

Editing Bookmarks

Once a bookmark exists, you can change its properties by selecting **Wave > Bookmarks > Bookmarks** if the Wave window is docked; or by selecting **Tools > Bookmarks** if the Wave window is undocked.

Searching in the Wave and List Windows

The Wave and List windows provide two methods for locating objects:

1. Finding signal names:
 - o Select **Edit > Find**
 - o click the **Find** toolbar button (binoculars icon)
 - o use the [find](#) command

The first two of these options will open a Find mode toolbar at the bottom of the Wave or List window. By default, the “Search For” option is set to “Name.” For more information, see [Using the Find and Filter Functions](#).

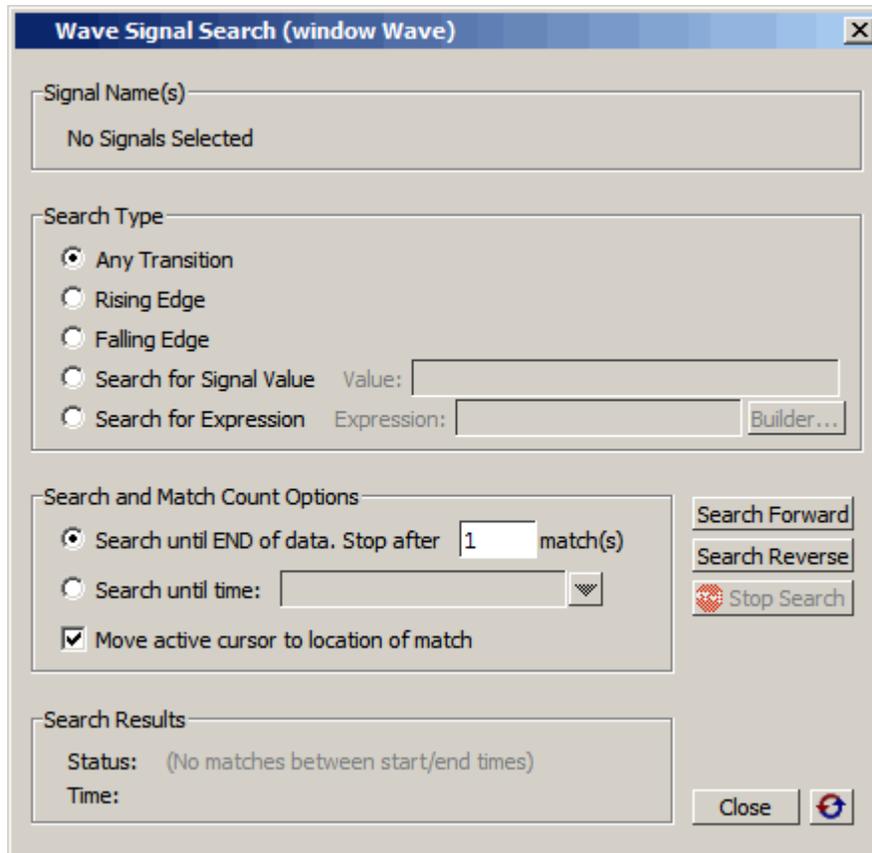
2. Search for values or transitions:
 - o Select **Edit > Signal Search**
 - o click the **Find** toolbar button (binoculars icon) and select **Search For > Value** from the Find toolbar that appears at the bottom of the Wave or List window.
 - o use the [search](#) command

Wave window searches can be stopped by clicking the “Stop Wave Drawing” or “Break” toolbar buttons.

Searching for Values or Transitions

The search command lets you search for transitions or values on selected signals. When you select **Edit > Signal Search**, the Signal Search dialog (Figure 15-24) appears.

Figure 15-24. Wave Signal Search Dialog



One option of note is **Search for Expression**. The expression can involve more than one signal but is limited to signals currently in the window. Expressions can include constants, variables, and DO files. Refer to [Expression Syntax](#) for more information.

Any search terms or settings you enter are saved from one search to the next in the current simulation. To clear the search settings during debugging click the Reset To Initial Settings button. The search terms and settings are cleared when you close ModelSim.

Note

If your signal values are displayed in binary radix, refer to [Searching for Binary Signal Values in the GUI](#) for details on how signal values are mapped between a binary radix and std_logic.

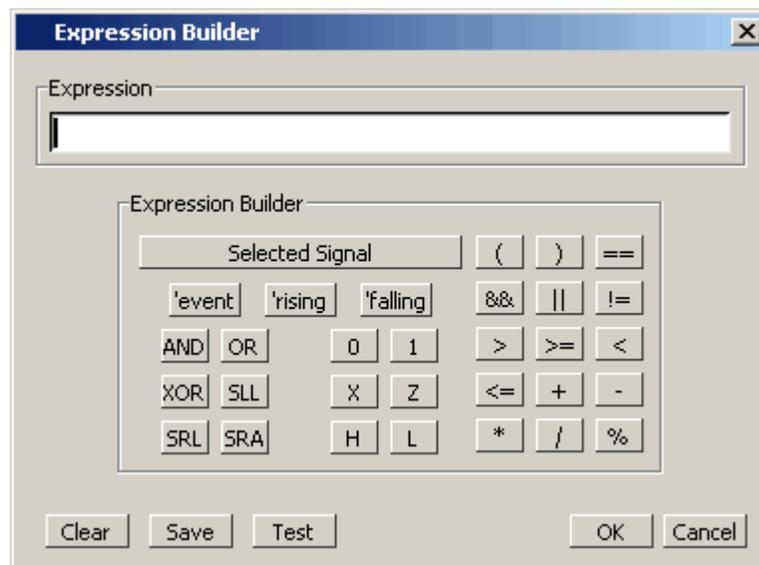
Using the Expression Builder for Expression Searches

The Expression Builder is a feature of the Wave and List Signal Search dialog boxes and the List trigger properties dialog box. You can use it to create a search expression that follows the [GUI_expression_format](#).

To display the Expression Builder dialog box, do the following:

1. Choose **Edit > Signal Search...** from the main menu. This displays the Wave Signal Search dialog box.
2. Select **Search for Expression**.
3. Click the **Builder** button. This displays the Expression Builder dialog box shown in [Figure 15-25](#)

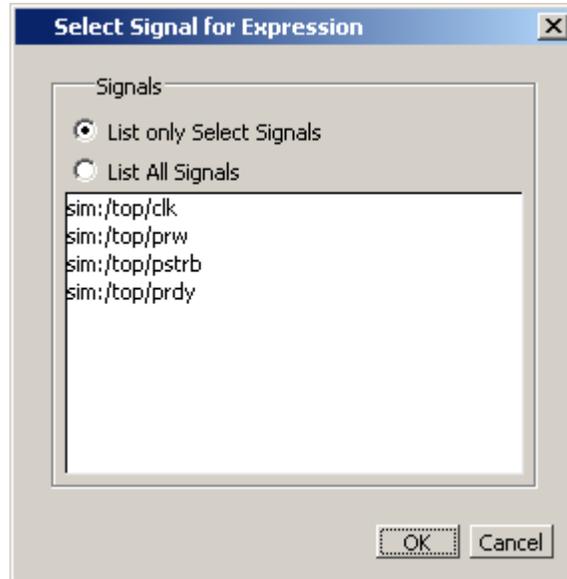
Figure 15-25. Expression Builder Dialog Box



You click the buttons in the Expression Builder dialog box to create a GUI expression. Each button generates a corresponding element of [Expression Syntax](#) and is displayed in the Expression field. In addition, you can use the **Selected Signal** button to create an expression from signals you select from the associated Wave or List window.

For example, instead of typing in a signal name, you can select signals in a Wave or List window and then click **Selected Signal** in the Expression Builder. This displays the Select Signal for Expression dialog box shown in [Figure 15-26](#).

Figure 15-26. Selecting Signals for Expression Builder



Note that the buttons in this dialog box allow you to determine the display of signals you want to put into an expression:

- List only Select Signals — list only those signals that are currently selected in the parent window.
- List All Signals — list all signals currently available in the parent window.

Once you have selected the signals you want displayed in the Expression Builder, click OK.

Saving an Expression to a Tcl Variable

Clicking the **Save** button will save the expression to a Tcl variable. Once saved this variable can be used in place of the expression. For example, say you save an expression to the variable "foo." Here are some operations you could do with the saved variable:

- Read the value of *foo* with the set command:

```
set foo
```
- Put \$foo in the Expression: entry box for the Search for Expression selection.
- Issue a searchlog command using foo:

```
searchlog -expr $foo 0
```

Searching for when a Signal Reaches a Particular Value

Select the signal in the Wave window and click **Insert Selected Signal** and **==**. Then, click the value buttons or type a value.

Evaluating Only on Clock Edges

Click the **&&** button to AND this condition with the rest of the expression. Then select the clock in the Wave window and click **Insert Selected Signal** and **'rising**. You can also select the falling edge or both edges.

Operators

Other buttons will add operators of various kinds (see [Expression Syntax](#)), or you can type them in.

Filtering the Wave Window Display

The Wave window includes a filtering function that allows you to filter the display to show only the desired signals and waveforms. To activate the filtering function:

1. Select **Edit > Find** in the menu bar (with the Wave window active) or click the **Find** icon in the [Standard Toolbar](#). This opens a “Find” toolbar at the bottom of the Wave window. 
2. Click the binoculars icon in the Find field to open a popup menu and select **Contains**. This enables the filtering function.

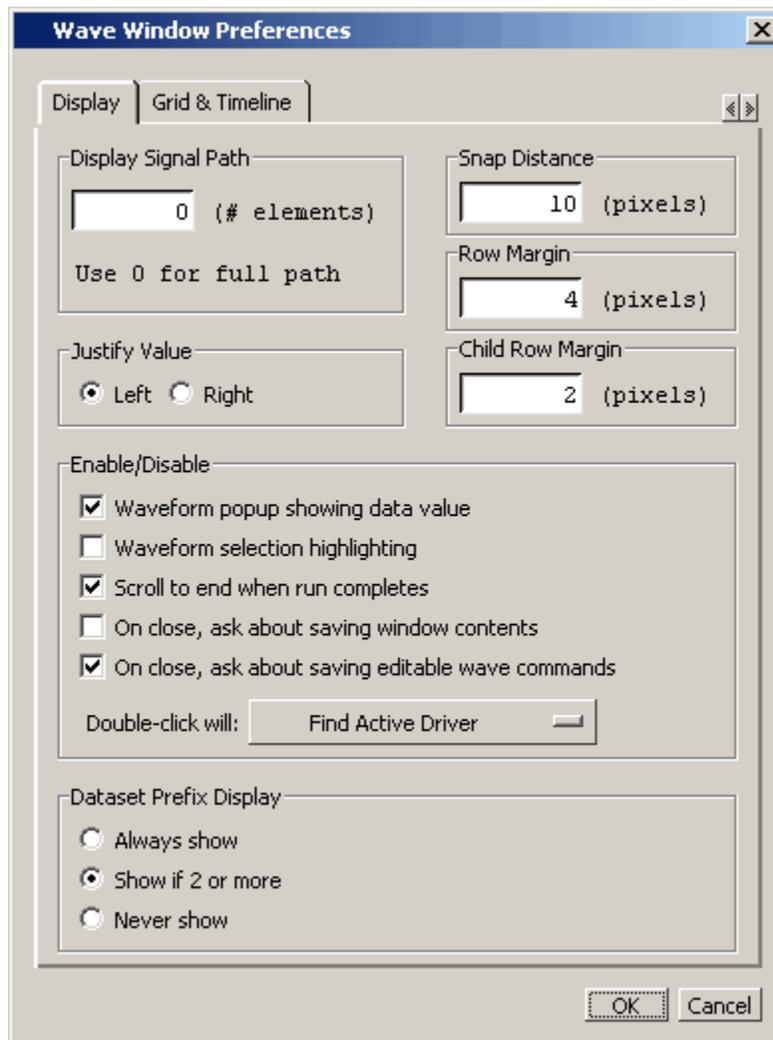
For more information, see [Using the Find and Filter Functions](#).

Formatting the Wave Window

Setting Wave Window Display Preferences

You can set Wave window display preferences by selecting **Wave > Wave Preferences** (when the window is docked) or **Tools > Window Preferences** (when the window is undocked). These commands open the Wave Window Preferences dialog ([Figure 15-27](#)).

Figure 15-27. Display Tab of the Wave Window Preferences Dialog Box



Hiding/Showing Path Hierarchy

You can set how many elements of the object path display by changing the Display Signal Path value in the Wave Window Preferences dialog (Figure 15-27). Zero specifies the full path, 1 specifies the leaf name, and any other positive number specifies the number of path elements to be displayed.

Double-Click Behavior in the Wave Window

You can set the default behavior for double-clicking a signal in the wave window. In the Display Tab of the Wave Window Preferences dialog box, select the Display tab, choose the Enable/Disable pane, click on the Find Active Driver button and choose one of the following from the popup menu:

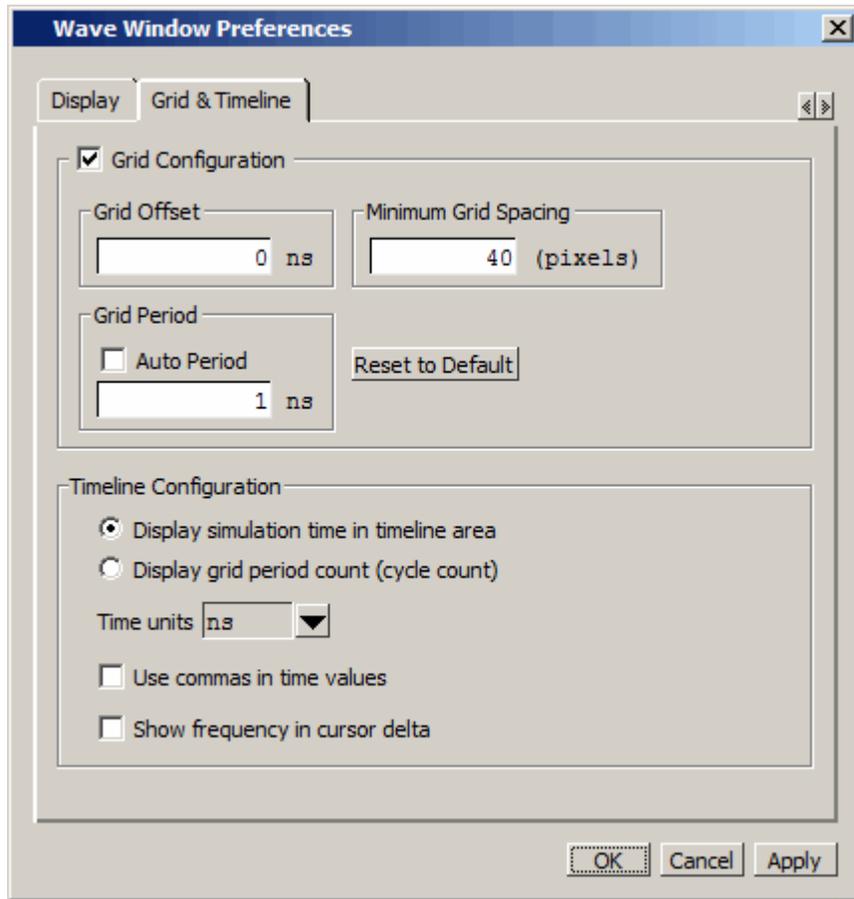
1. Do Nothing — Double-clicking on a wave form signal does nothing.

2. Show Drivers in Schematic — Double-clicking on a signal in the wave window traces the event for the specified signal and time back to the process causing the event. The results of the trace are placed in a Schematic Window that includes a waveform viewer below. Refer to [Tracing to the Immediate Driving Process](#) for more information about tracing immediate drivers and events.
3. Show Drivers in Dataflow — Double-clicking on a signal in the wave window traces the event for the specified signal and time back to the process causing the event. The results of the trace are placed in a Dataflow Window that includes a waveform viewer below.
4. Find Active Driver — Double-clicking on a signal in the wave window traces the event for the specified signal and time back to the process causing the event. The source file containing the line of code is opened and the driving signal code is highlighted. Refer to [Tracing to the First Sequential Process](#) for more information about finding the active driver.

Setting the Timeline to Count Clock Cycles

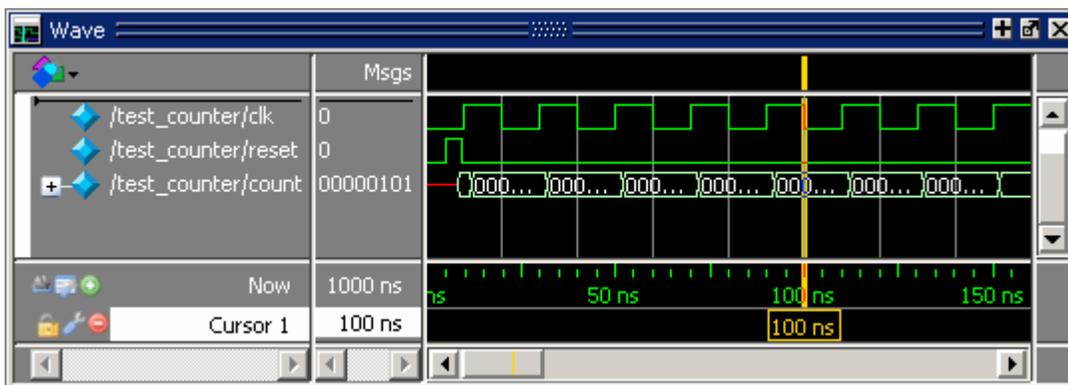
You can set the timeline of the Wave window to count clock cycles rather than elapsed time. If the Wave window is docked, open the Wave Window Preferences dialog by selecting **Wave > Wave Preferences** from the Main window menus. If the Wave window is undocked, select **Tools > Window Preferences** from the Wave window menus. This opens the Wave Window Preferences dialog. In the dialog, select the Grid & Timeline tab ([Figure 15-28](#)).

Figure 15-28. Grid and Timeline Tab of Wave Window Preferences Dialog Box



Enter the period of your clock in the Grid Period field and select “Display grid period count (cycle count).” The timeline will now show the number of clock cycles, as shown in Figure 15-29.

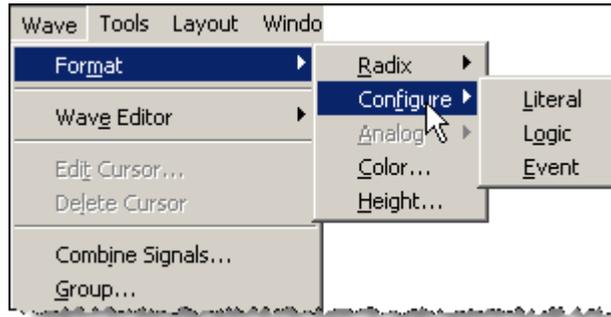
Figure 15-29. Clock Cycles in Timeline of Wave Window



Formatting Objects in the Wave Window

You can adjust various object properties to create the view you find most useful. Select one or more objects in the Wave window pathnames pane and then select **Wave > Format** from the menu bar (Figure 15-30).

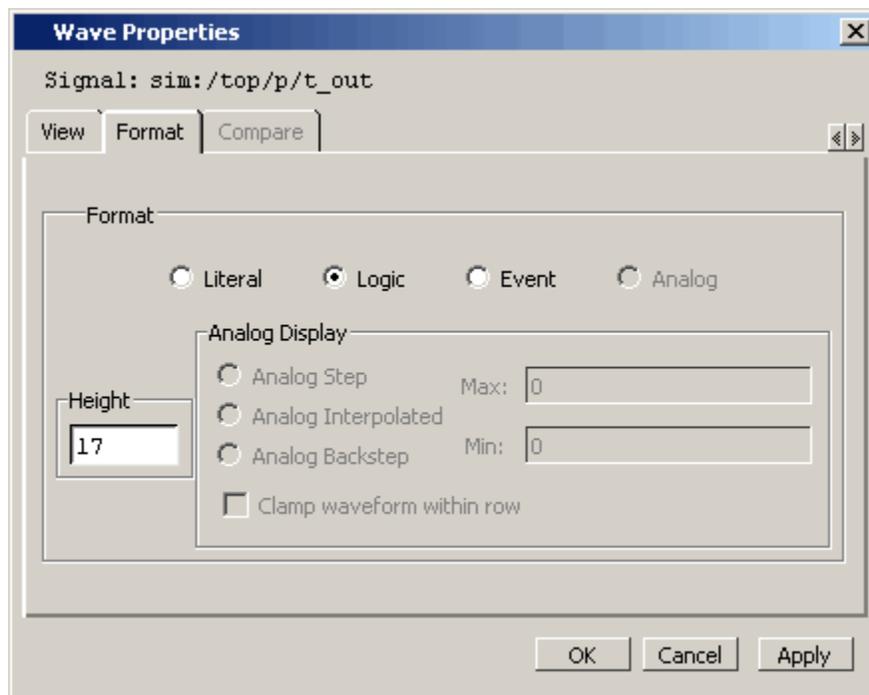
Figure 15-30. Wave Format Menu Selections



Or, you can right-click the selected object(s) and select **Format** from the popup menu.

If you right-click the and selected object(s) and select **Properties** from the popup menu, you can use the Format tab of the Wave Properties dialog to format selected objects (Figure 15-31).

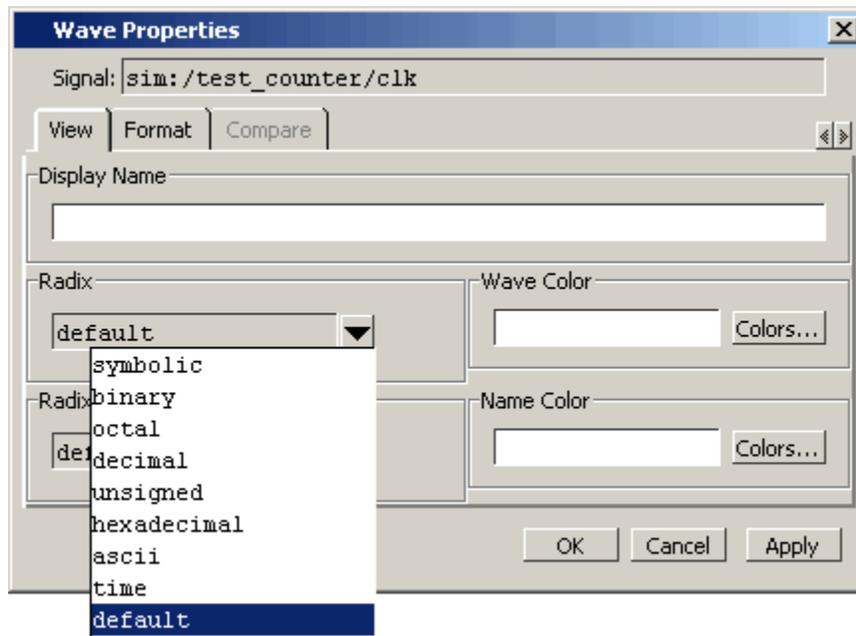
Figure 15-31. Format Tab of Wave Properties Dialog



Changing Radix (base) for the Wave Window

One common adjustment is changing the radix (base) of selected objects in the Wave window. When you right-click a selected object, or objects, and select **Properties** from the popup menu, the Wave Properties dialog appears. You can change the radix of the selected object(s) in the View tab (Figure 15-32).

Figure 15-32. Changing Signal Radix



The default radix is symbolic, which means that for an enumerated type, the value pane lists the actual values of the enumerated type of that object. For the other radices - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Note



When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

Aside from the Wave Properties dialog, there are three other ways to change the radix:

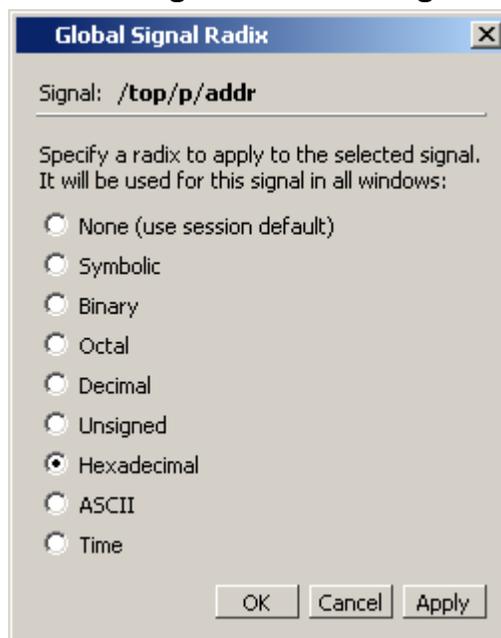
- Change the default radix for all objects in the current simulation using **Simulate > Runtime Options** (Main window menu).
- Change the default radix for the current simulation using the [radix](#) command.
- Change the default radix permanently by editing the [DefaultRadix](#) variable in the *modelsim.ini* file.

Setting the Global Signal Radix for Selected Objects

The Global Signal Radix feature allows you to change the radix for a selected object or objects in the Wave window and in every other window where the object appears.

1. Select an object or objects in the Wave window.
2. Right-click to open a popup menu.
3. Select **Radix > Global Signal Radix** from the popup menu. This opens the Global Signal Radix dialog, where you can set the radix for the Wave window and other windows where the selected object(s) appears.

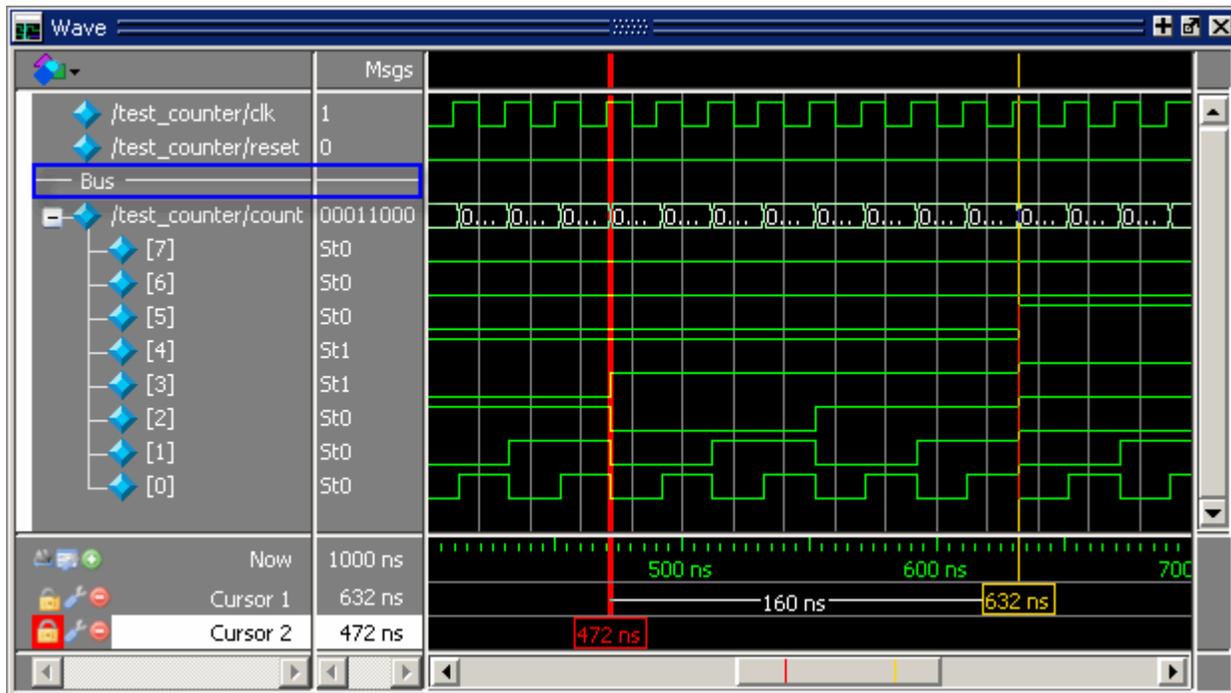
Figure 15-33. Global Signal Radix Dialog in Wave Window



Dividing the Wave Window

Dividers serve as a visual aid for debugging, allowing you to separate signals and waveforms for easier viewing. In the graphic below, a bus is separated from the two signals above it with a divider called "Bus."

Figure 15-34. Separate Signals with Wave Window Dividers



To insert a divider, follow these steps:

1. Select the signal above which you want to place the divider.
2. If the Wave pane is docked, select **Add > To Wave > Divider** from the Main window menu bar. If the Wave window stands alone, undocked from the Main window, select **Add > Divider** from the Wave window menu bar.
3. Specify the divider name in the Wave Divider Properties dialog. The default name is New Divider. Unnamed dividers are permitted. Simply delete "New Divider" in the Divider Name field to create an unnamed divider.
4. Specify the divider height (default height is 17 pixels) and then click OK.

You can also insert dividers with the **-divider** argument to the [add wave](#) command.

Working with Dividers

The table below summarizes several actions you can take with dividers:

Table 15-6. Actions for Dividers

Action	Method
Move a divider	Click-and-drag the divider to the desired location
Change a divider's name or size	Right-click the divider and select Divider Properties

Table 15-6. Actions for Dividers (cont.)

Action	Method
Delete a divider	Right-click the divider and select Delete

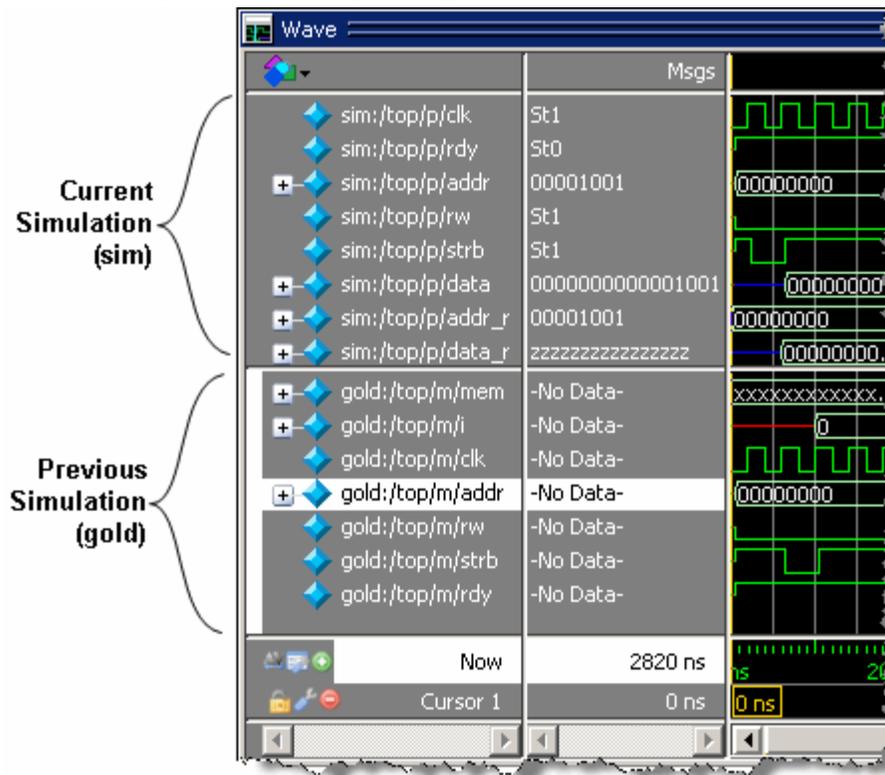
Splitting Wave Window Panes

The pathnames, values, and waveform panes of the Wave window display can be split to accommodate signals from one or more datasets. For more information on viewing multiple simulations, see [Recording Simulation Results With Datasets](#).

To split the window, select **Add > Window Pane**.

In the illustration below, the top split shows the current active simulation with the prefix "sim," and the bottom split shows a second dataset with the prefix "gold."

Figure 15-35. Splitting Wave Window Panes



The Active Split

The active split is denoted with a solid white bar to the left of the signal names. The active split becomes the target for objects added to the Wave window.

Wave Groups

You can create a wave group to collect arbitrary groups of items in the Wave window. Wave groups have the following characteristics:

- A wave group may contain 0, 1, or many items.
- You can add or remove items from groups either by using a command or by dragging and dropping.
- You can drag a group around the Wave window or to another Wave window.
- You can nest multiple wave groups, either from the command line or by dragging and dropping. Nested groups are saved or restored from a wave.do format file, restart and checkpoint/restore.

Creating a Wave Group

There are three ways to create a wave group:

- [Grouping Signals through Menu Selection](#)
- [Grouping Signals with the add wave Command](#)
- [Grouping Signals with a Keyboard Shortcut](#)

Grouping Signals through Menu Selection

If you've already added some signals to the Wave window, you can create a group of signals using the following procedure.

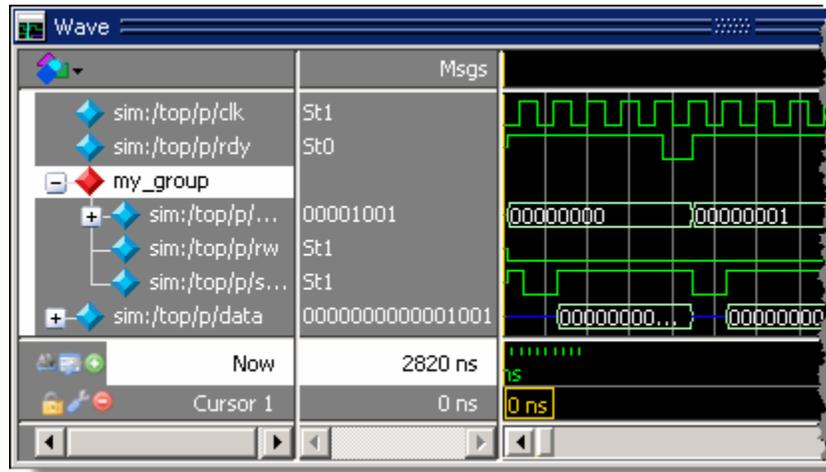
Procedure

1. Select a set of signals in the Wave window.
2. Select the **Wave > Group** menu item.
The Wave Group Create dialog appears.
3. Complete the Wave Group Create dialog box:
 - **Group Name** — specify a name for the group. This name is used in the wave window.
 - **Group Height** — specify an integer, in pixels, for the height of the space used for the group label.
4. Ok

Results

The selected signals become a group denoted by a red diamond in the Wave window pathnames pane (Figure 15-36), with the name specified in the dialog box.

Figure 15-36. Wave Groups Denoted by Red Diamond



Grouping Signals with the add wave Command

Add grouped signals to the Wave window from the command line use the following procedure.

Procedure

1. Determine the names of the signals you want to add and the name you want to assign to the group.
2. From the command line, use the `add wave` and the `-group` argument.

Examples

- Create a group named *mygroup* containing three items:

```
add wave -group mygroup sig1 sig2 sig3
```

- Create an empty group named *mygroup*:

```
add wave -group mygroup
```

Grouping Signals with a Keyboard Shortcut

If you've already added some signals to the Wave window, you can create a group of signals using the following procedure.

Procedure

1. Select the signals you want to group.

2. Ctrl-g

Results

The selected signals become a group with a name that references the dataset and common region, for example: sim:/top/p.

If you use Ctrl-g to group any other signals, they will be placed into any existing group for their region, rather than creating a new group of only those signals.

Deleting or Ungrouping a Wave Group

If a wave group is selected and cut or deleted the entire group and all its contents will be removed from the Wave window. Likewise, the [delete](#) wave command will remove the entire group if the group name is specified.

If a wave group is selected and the **Wave > Ungroup** menu item is selected the group will be removed and all of its contents will remain in the Wave window in existing order.

Adding Items to an Existing Wave Group

There are three ways to add items to an existing wave group.

1. Using the drag and drop capability to move items outside of the group or from other windows into the group. The insertion indicator will show the position the item will be dropped into the group. If the cursor is moved over the lower portion of the group item name a box will be drawn around the group name indicating the item will be dropped into the last position in the group.
2. After selecting an insertion point within a group, place the cursor over the object to be inserted into the group, then click the middle mouse button.
3. After selecting an insertion point within a group, select multiple objects to be inserted into the group, then click the **Add Selected to Window** button in the **Standard Toolbar**.
4. The cut/copy/paste functions may be used to paste items into a group.
5. Use the **add wave -group** command.

The following example adds two more signals to an existing group called *mygroup*.

```
add wave -group mygroup sig4 sig5
```

Removing Items from an Existing Wave Group

You can use any of the following methods to remove an item from a wave group.

1. Use the drag and drop capability to move an item outside of the group.
2. Use menu or icon selections to cut or delete an item or items from the group.
3. Use the [delete](#) wave command to specify a signal to be removed from the group.

Note

The delete wave command removes all occurrences of a specified name from the Wave window, not just an occurrence within a group.

Miscellaneous Wave Group Features

Dragging a wave group from the Wave window to the List window will result in all of the items within the group being added to the List window.

Dragging a group from the Wave window to the Transcript window will result in a list of all of the items within the group being added to the existing command line, if any.

Composite Signals or Buses

You can create a composite signal or bus from arbitrary groups of items in the Wave window. Composite signals have the following characteristics:

- Composite signals may contain 0, 1, or many items.
- You can drag a group around the Wave window or to another Wave window.

Creating Composite Signals through Menu Selection

If you've already added some signals to the Wave window, you can create a composite signal using the following procedure.

To create a new composite signal or bus from one or more signals:

1. Select signals to combine:
 - Shift-click on signal pathnames to select a contiguous set of signals, records, and/or busses.
 - Control-click on individual signal, record, and/or bus pathnames.
2. Select **Wave > Combine Signals**
3. Complete the Combine Selected Signals dialog box.
 - Name — Specify the name of the new combined signal or bus.
 - Order to combine selected items — Specify the order of the signals within the new combined signal.

- Top down— (default) Signals ordered from the top as selected in the Wave window.
- Bottom Up — Signals ordered from the bottom as selected in the Wave window.
- Order of Result Indexes — Specify the order of the indexes in the combined signal.
- Ascending — Bits indexed [0 : n] starting with the top signal in the bus.
- Descending — (default) Bits indexed [n : 0] starting with the top signal in the bus.
- Remove selected signals after combining — Saves the selected signals in the combined signal only.
- Reverse bit order of bus items in result — Reverses the bit order of busses that are included in the new combined signal.
- Flatten Arrays — (default) Moves elements of arrays to be elements of the new combined signal. If arrays are not flattened the array itself will be an element of the new combined signal.
- Flatten Records — Moves fields of selected records and signals to be elements of the new combined signal. If records are not flattened the record itself will be an element of the new combined signal.

For more information, refer to [Virtual Signals](#).

Related Topics

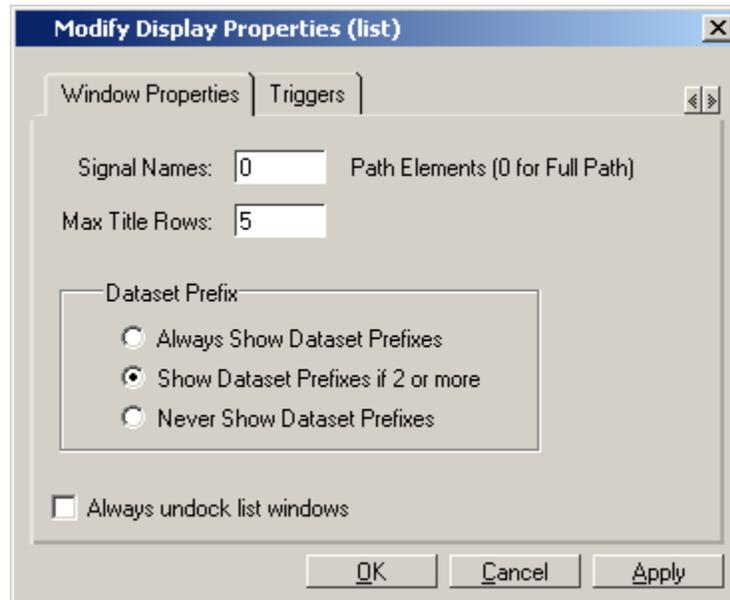
- [virtual signal](#)
- [“Virtual Objects”](#)
- [“Creating a Virtual Signal”](#)
- [“Concatenation of Signals or Subelements”](#)

Formatting the List Window

Setting List Window Display Properties

Before you add objects to the List window, you can set the window’s display properties. To change when and how a signal is displayed in the List window, select **Tools > List Preferences** from the List window menu bar (when the window is undocked).

Figure 15-37. Modifying List Window Display Properties



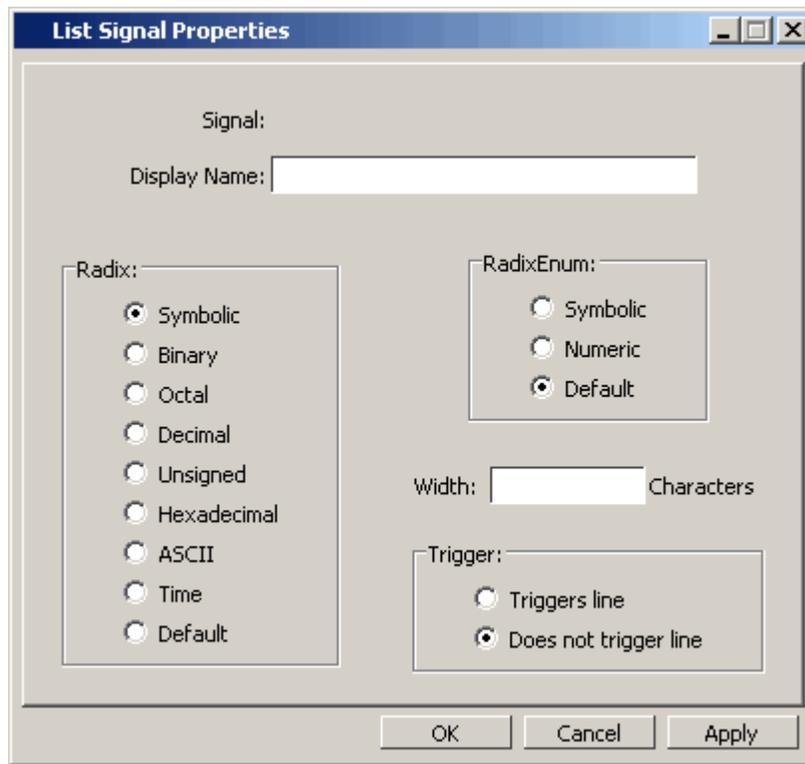
Formatting Objects in the List Window

You can adjust various properties of objects to create the view you find most useful. Select one or more objects and then select **View > Signal Properties** from the List window menu bar (when the window is undocked).

Changing Radix (base) for the List Window

One common adjustment you can make to the List window display is to change the radix (base) of an object. To do this, choose **View > Properties** from the main menu, which displays the List Signal Properties dialog box. [Figure 15-38](#) shows the list of radix types you can select in this dialog box.

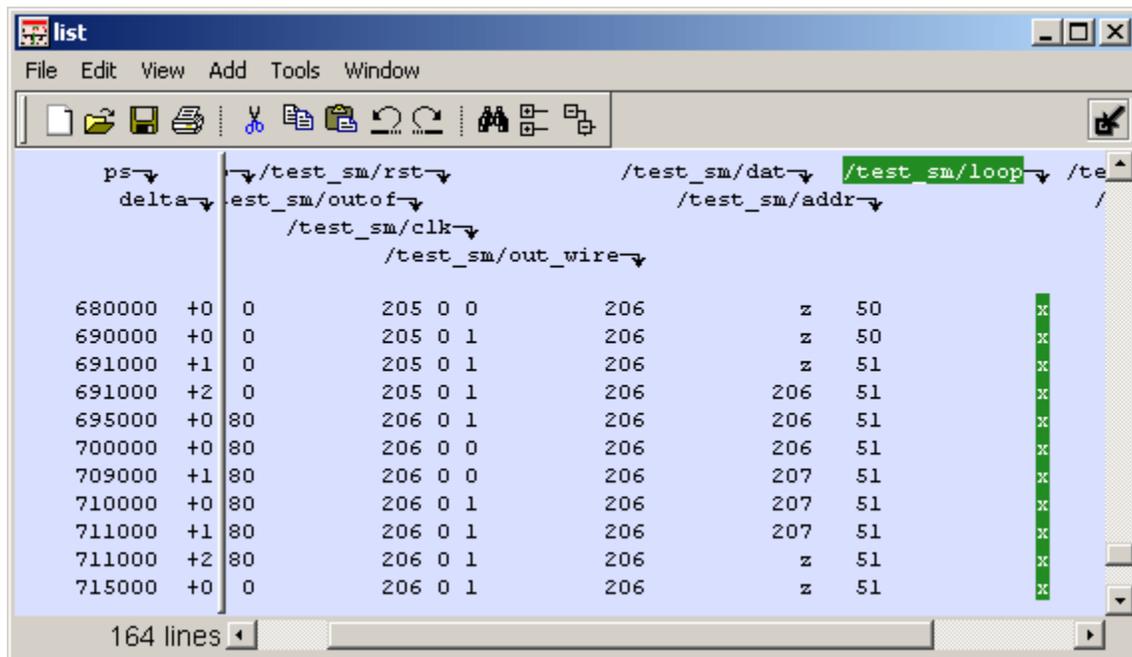
Figure 15-38. List Signal Properties Dialog



The default radix type is symbolic, which means that for an enumerated type, the window lists the actual values of the enumerated type of that object. For the other radix types (binary, octal, decimal, unsigned, hexadecimal, ASCII, time), the object value is converted to an appropriate representation in that radix.

Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image in the section [List Window Overview](#) (with symbolic values).

Figure 15-39. Changing the Radix in the List Window



In addition to the List Signal Properties dialog box, you can also change the radix:

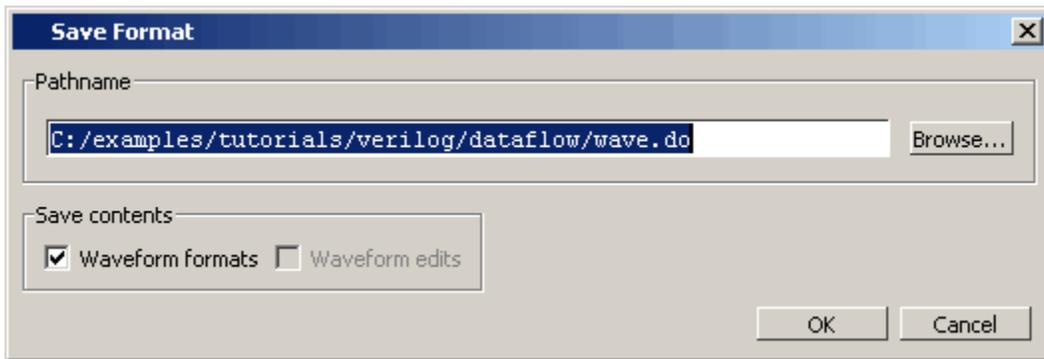
- Change the default radix for the current simulation using **Simulate > Runtime Options** (Main window)
- Change the default radix for the current simulation using the `radix` command.
- Change the default radix permanently by editing the `DefaultRadix` variable in the `modelsim.ini` file.

Saving the Window Format

By default, all Wave and List window information is lost once you close the windows. If you want to restore the windows to a previously configured layout, you must save a window format file. Follow these steps:

1. Add the objects you want to the Wave or List window.
2. Edit and format the objects to create the view you want.
3. Save the format to a file by selecting **File > Save**. This opens the Save Format dialog box (Figure 15-40), where you can save waveform formats in a `.do` file.

Figure 15-40. Save Format Dialog



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

- Invoke the `do` command from the command line:

```
VSIM> do <my_format_file>
```
- Select **File > Load**.

Note



Window format files are design-specific. Use them only with the design you were simulating when they were created.

In addition, you can use the `write format restart` command to create a single `.do` file that will recreate all debug windows and breakpoints (see [Saving and Restoring Breakpoints](#)) when invoked with the `do` command in subsequent simulation runs. The syntax is:

```
write format restart <filename>
```

If the `ShutdownFile` `modelsim.ini` variable is set to this `.do` filename, it will call the `write format restart` command upon exit.

Exporting Waveforms from the Wave window

This section describes ways to save or print information from the Wave window.

Exporting the Wave Window as a Bitmap Image

You can export the current view of the Wave window to a Bitmap (`.bmp`) image by selecting the **File > Export > Image** menu item and completing the Save Image dialog box.

The saved bitmap image only contains the current view; it does not contain any signals not visible in the current scroll region.

Note that you should not select a new window in the GUI until the export has completed, otherwise your image will contain information about the newly selected window.

Printing the Wave Window to a Postscript File

You can export the contents of the Wave window to a Postscript (.ps) or Extended Postscript file by selecting the **File > Print Postscript** menu item and completing the Write Postscript dialog box.

The Write Postscript dialog box allows you to control the amount of information exported.

- Signal Selection — allows you to select which signals are exported
- Time Range — allows you to select the time range for the given signals.

Note that the output is a simplified black and white representation of the wave window.

You can also perform this action with the `write wave` command.

Printing the Wave Window on the Windows Platform

You can print the contents of the Wave window to a networked printer by selecting the File > Print menu item and completing the Print dialog box.

The Print dialog box allows you to control the amount of information exported.

- Signal Selection — allows you to select which signals are exported
- Time Range — allows you to select the time range for the given signals.

Note that the output is a simplified black and white representation of the wave window.

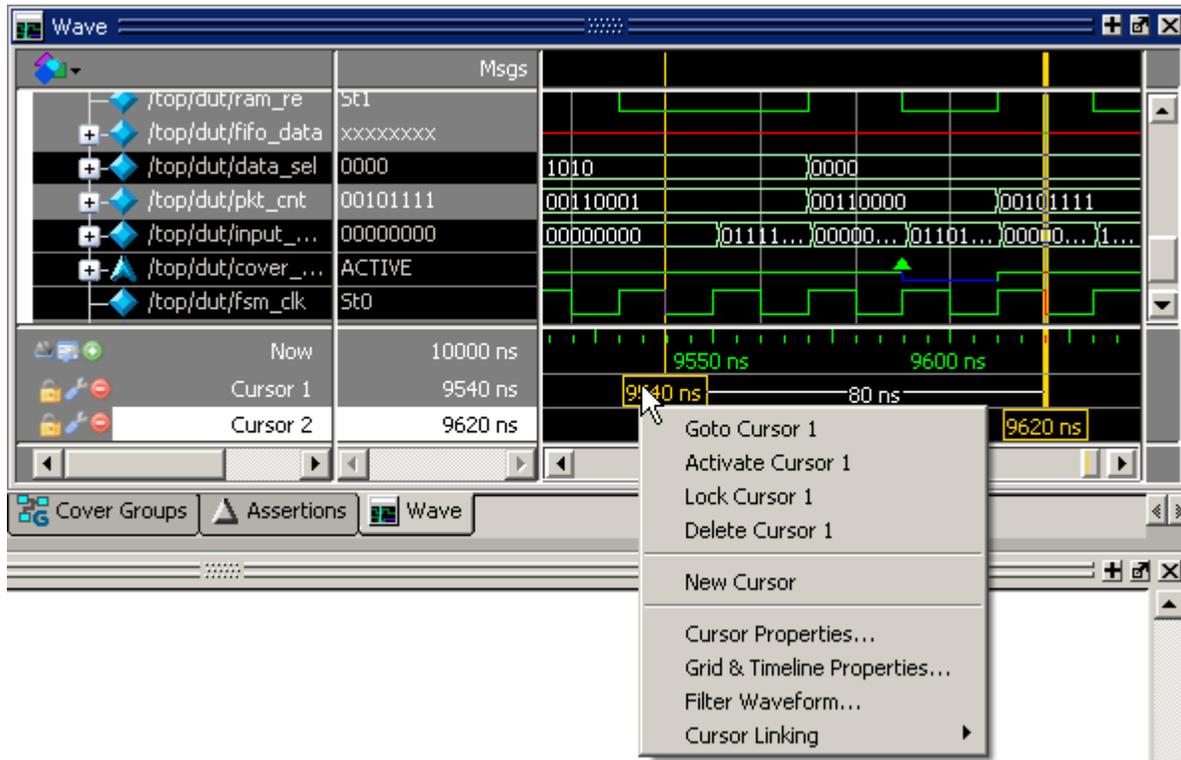
Saving Waveforms Between Two Cursors

You can choose one or more objects or signals in the waveform pane and save a section of the generated waveforms to a separate WLF file for later viewing. Saving selected portions of the waveform pane allows you to create a smaller dataset file.

The following steps refer to [Figure 15-41](#).

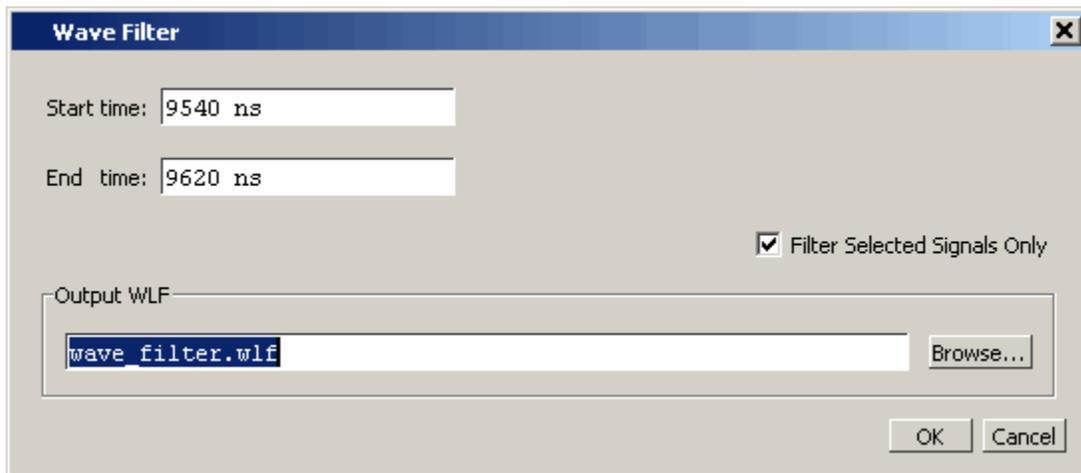
1. Place the first cursor (Cursor 1 in [Figure 15-41](#)) at one end of the portion of simulation time you want to save.
2. Click the **Insert Cursor** icon to insert a second cursor (Cursor 2). 
3. Move Cursor 2 to the other end of the portion of time you want to save. Cursor 2 is now the active cursor, indicated by a bold yellow line and a highlighted name.
4. Right-click the time indicator of the inactive cursor (Cursor 1) to open a drop menu.

Figure 15-41. Waveform Save Between Cursors



5. Select **Filter Waveform** to open the **Wave Filter** dialog box. (Figure 15-42)

Figure 15-42. Wave Filter Dialog



6. Select **Filter Selected Signals Only** to save selected objects or signals. Leaving this checkbox blank will save data for all waveforms displayed in the Wave window between the specified start and end time.

7. Enter a name for the file using the *.wlf* extension. Do not use *vsim.wlf* since it is the default name for the simulation dataset and will be overwritten when you end your simulation.

Viewing Saved Waveforms

1. Open the saved *.wlf* file by selecting **File > Open** to open the Open File dialog and set the “Files of type” field to Log Files (*.wlf). Then select the *.wlf* file you want and click the Open button. Refer to [Opening Datasets](#) for more information.
2. Select the top instance in the Structure window
3. Select **Add > To Wave > All Items in Region and Below**.
4. Scroll to the simulation time that was saved. ([Figure 15-43](#))

Figure 15-43. Wave Filter Dataset



Working With Multiple Cursors

You can save a portion of your waveforms in a simulation that has multiple cursors set. The new dataset will start and end at the times indicated by the two cursors chosen, even if the time span includes another cursor.

Saving List Window Data to a File

Select **File > Write List** in the List window to save the data in one of these formats:

- **Tabular** — writes a text file that looks like the window listing

ns	delta	/a	/b	/cin	/sum	/cout
0	+0	X	X	U	X	U
0	+1	0	1	0	X	U
2	+0	0	1	0	X	U

- **Events** — writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI** — writes a file in standard TSSI format; see also, the [write tssi](#) command.

```
0 000000000000000010?????????
2 000000000000000010?????????1?
3 000000000000000010???????010
4 0000000000000000100000000010
100 000000010000000010000000010
```

You can also save List window output using the [write list](#) command.

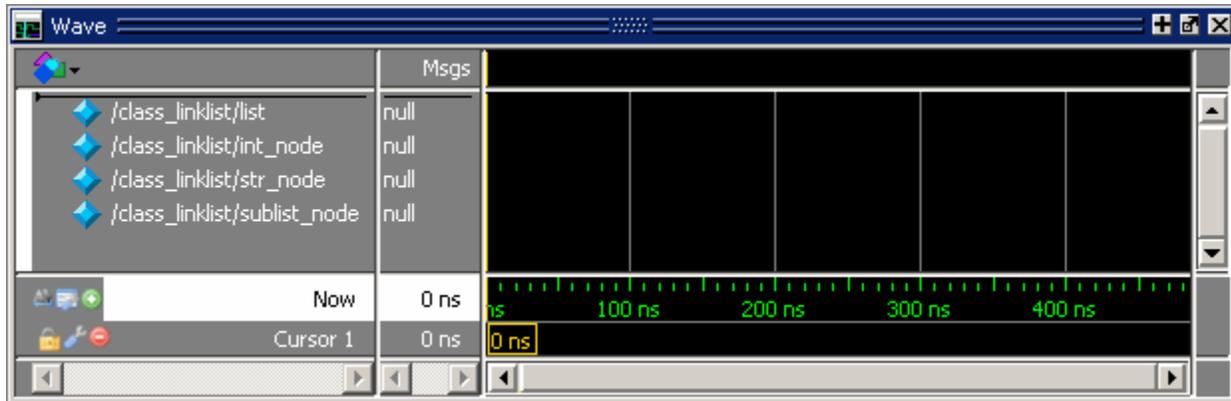
Viewing SystemVerilog Class Objects

The suggested workflow for viewing SystemVerilog class objects in the Wave window is as follows.

1. Select a module in a Workspace structure view that contains the class variables you want to see. All class variables associated with that module are displayed in the Objects window. (Open the Objects window by selecting **View > Objects** from the menus or use the [view objects](#) command.)
2. You can place class objects in the Wave window by doing any one of the following:
 - Drag a class variable from the Objects window and drop it into the Wave window,
 - Place the cursor over the class variable, then click the middle mouse button.
 - Right-click the class variable in the Objects window and select **Add > To Wave** from the popup context window.
 - **Select multiple objects, click and hold the Add Selected to Window button in the Standard toolbar**, then select the position of the placement; the top of the wave window, the end of the wave window, or above the anchor location.
 - Use the [add wave](#) command at the command prompt. For example:
add wave int_mode

SystemVerilog objects are denoted in the Wave window by a light blue diamond, as shown in Figure 15-44.

Figure 15-44. Class Objects in the Wave Window



- Items in the pathnames column are class variables.
- Items in the values columns are symbolic representations of class objects that refer to a class instance. These class references may have a “null” value - which means that they do not yet refer to any class instance - or they may be denoted with the “@” symbol, which does refer to a class instance. The “@” indicates that the value is a pointer (or a handle) to an object rather than the value of the object itself. The number that appears after the “@” is a unique reference number. This is needed because class objects are not named by the class variable that stores the handle to the object.

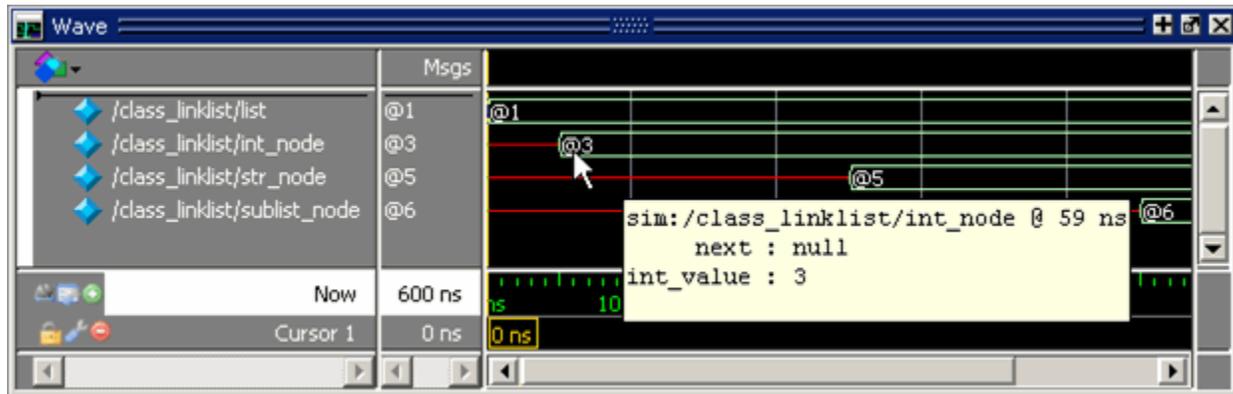
When the simulation is run, the “waveforms” for class objects appear as shown in Figure 15-45.

Figure 15-45. Class Waveforms



You can hover the mouse over any class waveform to display information about the class variable (Figure 15-46).

Figure 15-46. Class Information Popup



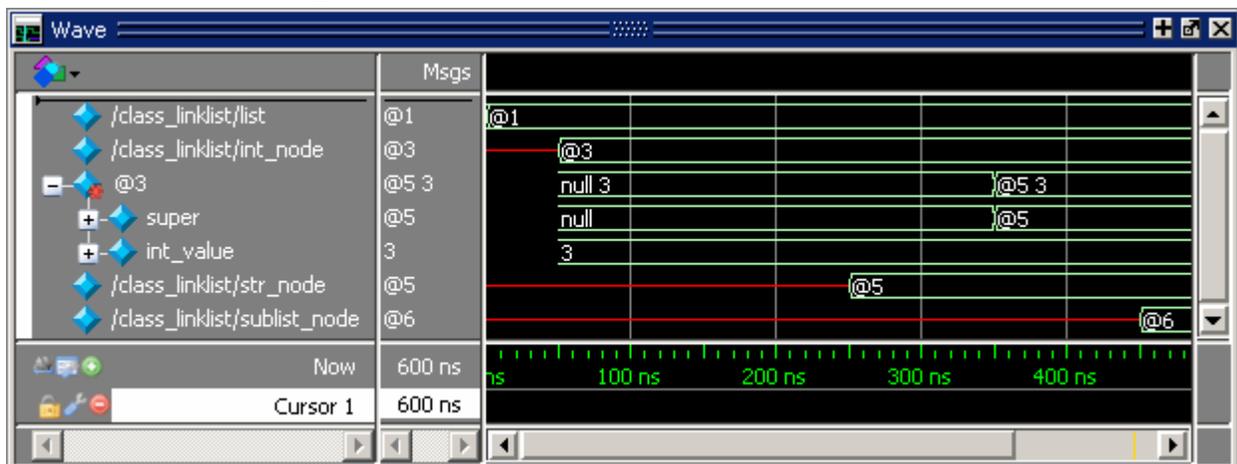
To display the waveforms for the actual class instances, do either of the following:

- Right-click the class “waveform” and select **Add Wave** from the popup context menu.
- Use the add wave command with a class reference name.

add wave @3

The class instances are denoted by a light blue diamond with a red asterisk in the pathnames column. In Figure 15-47, the class object referred to by “@3” is expanded to show the integer value for the `int_value` instance.

Figure 15-47. Waveforms for Class Instances



Combining Objects into Buses

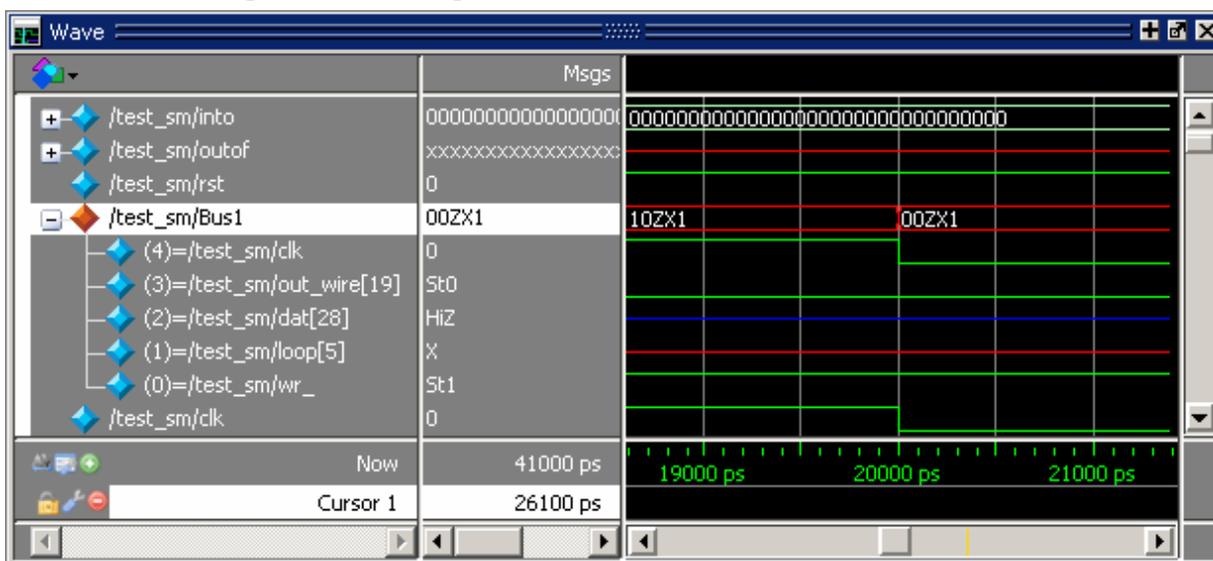
You can combine signals in the Wave or List window into buses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. A virtual compare signal (the result of a comparison simulation) is not supported for combination with any other signal.

To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave or List window and then choose **Tools > Combine Signals** from the menu bar. A virtual signal that is the result of a comparison simulation is not supported for combining with any other signal.
- Use the [virtual signal](#) command at the Main window command prompt.

In the illustration below, four 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 value of the bus is made up of the values of its component signals, arranged in a specific order.

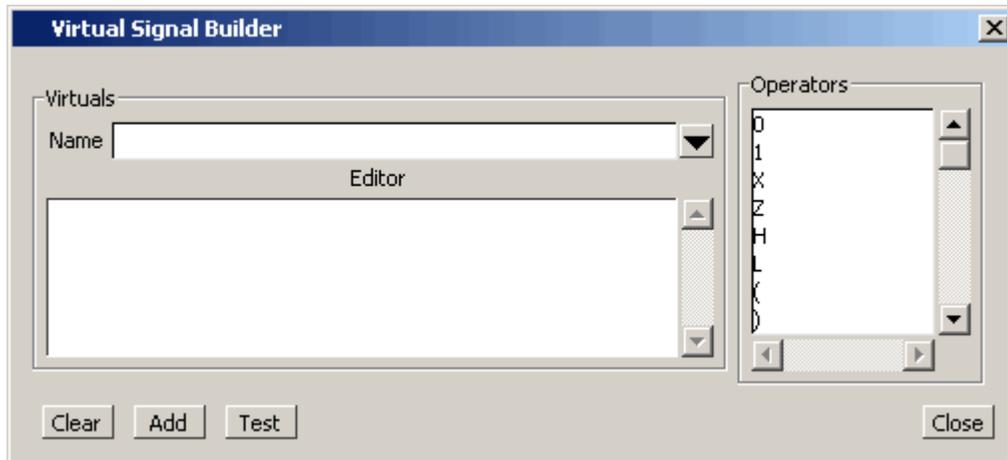
Figure 15-48. Signals Combined to Create Virtual Bus



Creating a Virtual Signal

The Wave window allows you to build Virtual Signals with the Virtual Signal Builder. The Virtual Signal Builder is accessed by selecting **Wave > Virtual Builder** when the Wave window is docked or selecting **Tools > Virtual Builder** when the Wave window is undocked.

Figure 15-49. Virtual Expression Builder



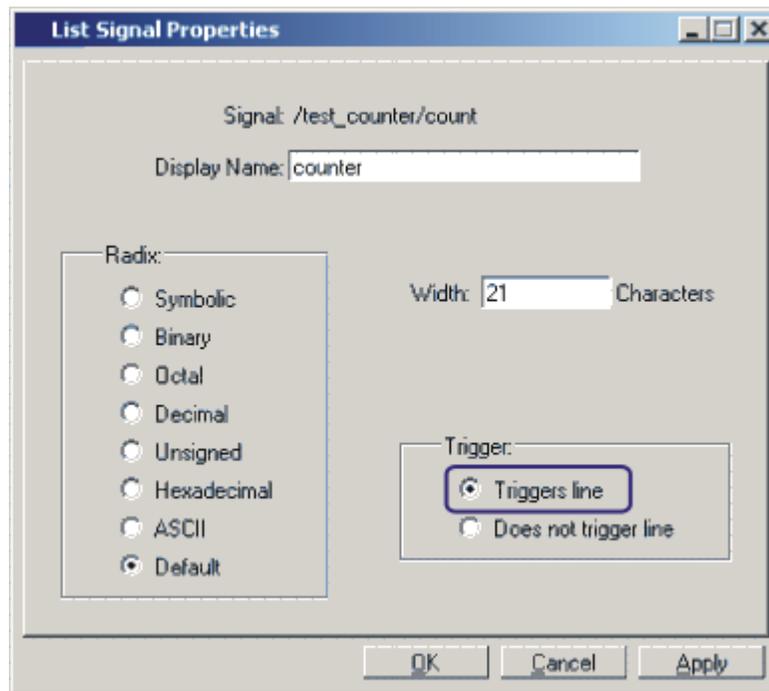
- The Name field allows you to enter the name of the new virtual signal.
- The Editor field is simply a regular text box. You can write directly to it, copy and paste or drag a signal from the Objects window, Locals window or the Wave window and drop it in the Editor field.
- The Operators field allows you to select from a list of operators. Simply double-click an operator to add it to the Editor field.
- The Clear button will clear the Editor field.
- The Add button will add the virtual signal to the wave window
- The Test button will test your virtual signal for proper operation.

Configuring New Line Triggering in the List Window

New line triggering refers to what events cause a new line of data to be added to the List window. By default ModelSim adds a new line for any signal change including deltas within a single unit of time resolution.

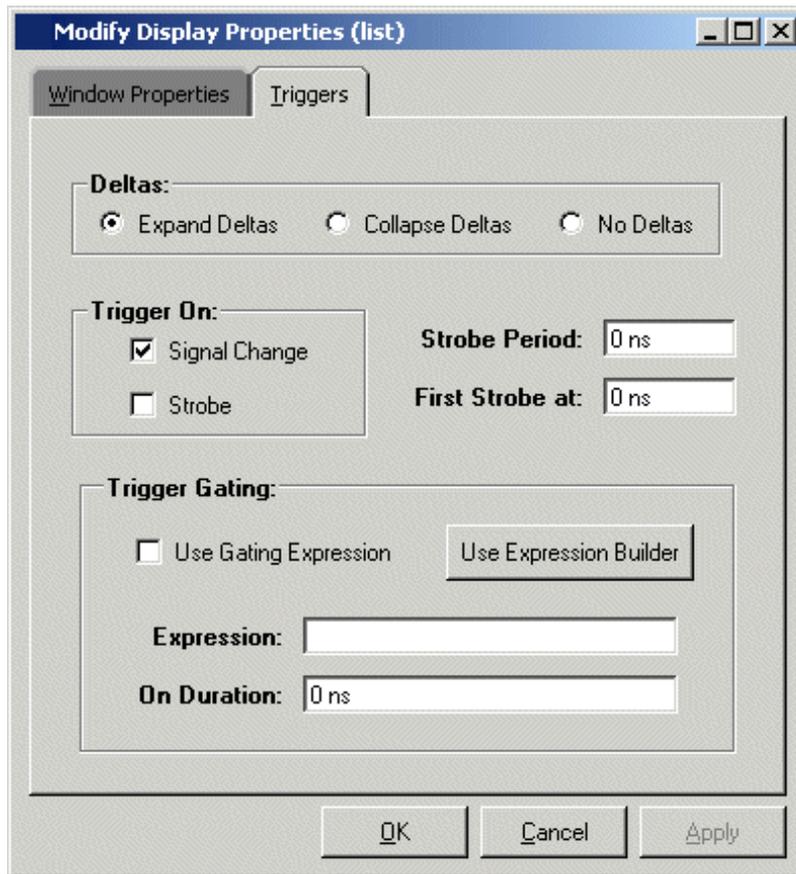
You can set new line triggering on a signal-by-signal basis or for the whole simulation. To set for a single signal, select **View > Signal Properties** from the List window menu bar (when the window is undocked) and select the **Triggers line** setting. Individual signal settings override global settings.

Figure 15-50. Line Triggering in the List Window



To modify new line triggering for the whole simulation, select **Tools > List Preferences** from the List window menu bar (when the window is undocked), or use the [configure](#) command. When you select **Tools > List Preferences**, the Modify Display Properties dialog appears:

Figure 15-51. Setting Trigger Properties



The following table summarizes the triggering options:

Table 15-7. Triggering Options

Option	Description
Deltas	Choose between displaying all deltas (Expand Deltas), displaying the value at the final delta (Collapse Delta). You can also hide the delta column all together (No Delta), however this will display the value at the final delta.
Strobe trigger	Specify an interval at which you want to trigger data display
Trigger gating	Use a gating expression to control triggering; see Using Gating Expressions to Control Triggering for more details

Using Gating Expressions to Control Triggering

Trigger gating controls the display of data based on an expression. Triggering is enabled once the gating expression evaluates to true. This setup behaves much like a hardware signal analyzer that starts recording data on a specified setup of address bits and clock edges.

Here are some points about gating expressions:

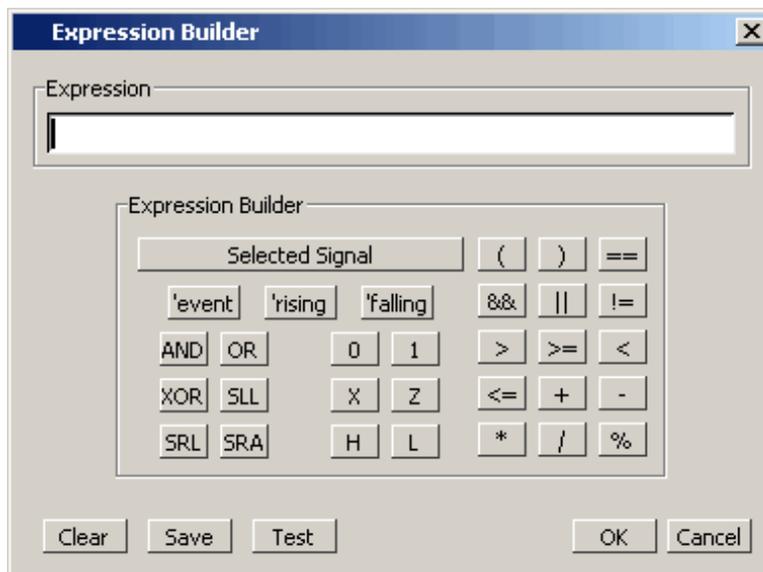
- Gating expressions affect the display of data but not acquisition of the data.
- The expression is evaluated when the List window would normally have displayed a row of data (given the other trigger settings).
- The duration determines for how long triggering stays enabled after the gating expression returns to false (0). The default of 0 duration will enable triggering only while the expression is true (1). The duration is expressed in x number of default timescale units.
- Gating is level-sensitive rather than edge-triggered.

Trigger Gating Example Using the Expression Builder

This example shows how to create a gating expression with the ModelSim Expression Builder. Here is the procedure:

1. Select **Tools > Window Preferences** from the List window menu bar (when the window is undocked) and select the Triggers tab.
2. Click the **Use Expression Builder** button.

Figure 15-52. Trigger Gating Using Expression Builder



3. Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window.
4. Click **Insert Selected Signal** and then '**rising**' in the Expression Builder.
5. Click OK to close the Expression Builder.

You should see the name of the signal plus "rising" added to the Expression entry box of the Modify Display Properties dialog box.

6. Click **OK** to close the dialog.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**.

Trigger Gating Example Using Commands

The following commands show the gating portion of a trigger configuration statement:

```
configure list -usegating 1
configure list -gateduration 100
configure list -gateexpr {/test_delta/iom_dd'rising}
```

See the [configure](#) command for more details.

Sampling Signals at a Clock Change

You easily can sample signals at a clock change using the [add list](#) command with the **-notrigger** argument. The **-notrigger** argument disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

1. Turn off the List window triggering on the clock signal, and then define a repeating strobe for the List window.
2. Define a "gating expression" for the List window that requires the clock to be in a specified state. See above.

Miscellaneous Tasks

Examining Waveform Values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See [Setting Wave Window Display Preferences](#).
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse. This method works in the List window as well.

Displaying Drivers of the Selected Waveform

You can display the drivers of a signal selected in the Wave window in either the Dataflow or the Schematic window. To select which window will display the specified signal, you must first set the [Double-Click Behavior in the Wave Window](#).

You can display the signal in one of three ways:

- Select a waveform and click the Show Drivers button on the toolbar. 
- Right-click a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see [Setting Wave Window Display Preferences](#))

This operation opens the Dataflow or Schematic window and displays the drivers of the signal selected in the Wave window. A Wave pane also opens in the Dataflow or Schematic window to show the selected signal with a cursor at the selected time. The Dataflow or Schematic window shows the signal(s) values at the Wave pane cursor position.

Sorting a Group of Objects in the Wave Window

Select **View > Sort** to sort the objects in the pathname and values panes.

Creating and Managing Breakpoints

ModelSim supports both signal (that is, when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

Note



When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. See [Preserving Object Visibility for Debugging Purposes](#) and [Design Object Visibility for Designs with PLI](#).

Breakpoints within SystemC portions of the design can only be set using [File-Line Breakpoints](#).

Signal Breakpoints

Signal breakpoints (“when” conditions) instruct ModelSim to perform actions when the specified conditions are met. For example, you can break on a signal value or at a specific simulator time (see the [when](#) command for additional details). When a breakpoint is hit, a message in the Main window transcript identifies the signal that caused the breakpoint.

Setting Signal Breakpoints with the when Command

Use the [when](#) command to set a signal breakpoint from the VSIM> prompt. For example,

```
when {errorFlag = '1' OR $now = 2 ms} {stop}
```

adds 2 ms to the simulation time at which the “when” statement is first evaluated, then stops. The white space between the value and time unit is required for the time unit to be understood by the simulator. See the [when](#) command in the Command Reference for more examples.

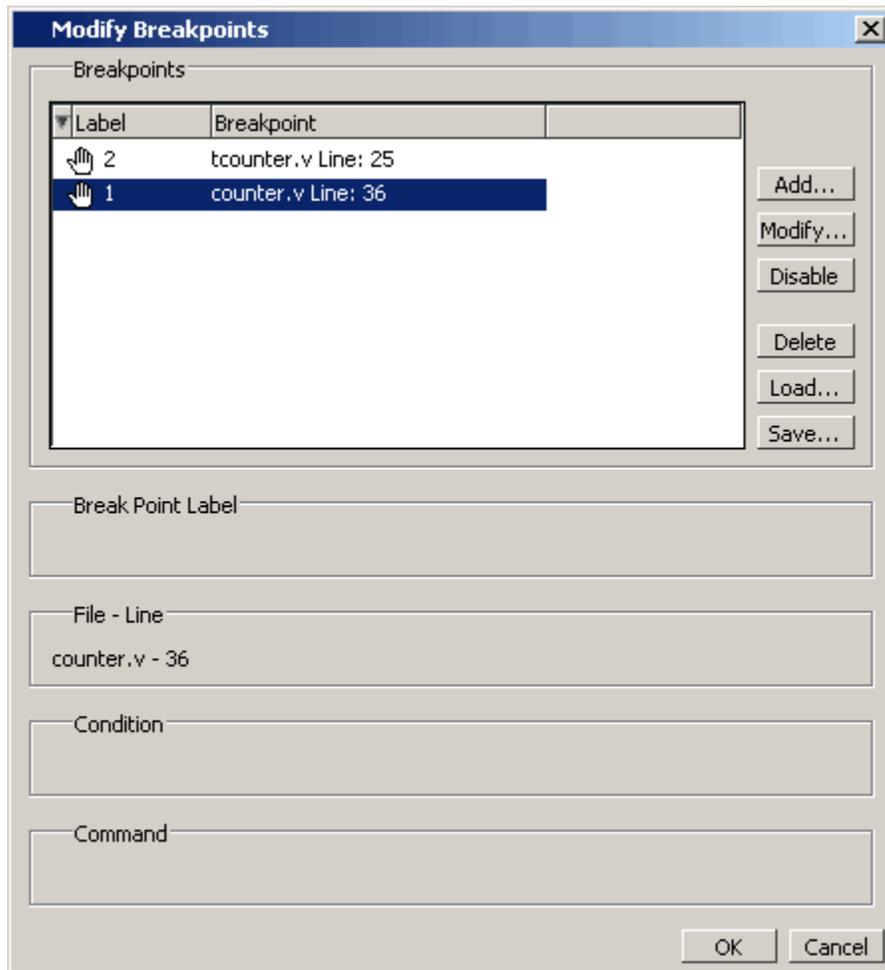
Setting Signal Breakpoints with the GUI

Signal breakpoints are most easily set in the [Objects Window](#) and the Wave window. Right-click a signal and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Modify Breakpoints** dialog accessible by selecting **Tools > Breakpoints** from the Main menu bar.

Modifying Signal Breakpoints

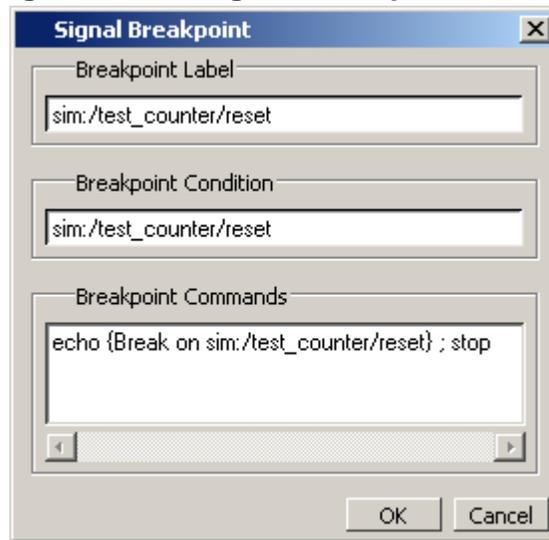
You can modify signal breakpoints by selecting **Tools > Breakpoints** from the Main menus. This will open the Modify Breakpoints dialog ([Figure 15-53](#)), which displays a list of all breakpoints in the design.

Figure 15-53. Modifying the Breakpoints Dialog



When you select a signal breakpoint from the list and click the Modify button, the Signal Breakpoint dialog (Figure 15-54) opens, allowing you to modify the breakpoint.

Figure 15-54. Signal Breakpoint Dialog



File-Line Breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops and the Source window opens to show the line with the breakpoint. You can change this behavior by editing the `PrefSource(OpenOnBreak)` variable. See [Simulator GUI Preferences](#) for details on setting preference variables.

Since C Debug is invoked when you set a breakpoint within a SystemC module, your C Debug settings must be in place prior to setting a breakpoint. See [Setting Up C Debug](#) for more information. Once invoked, C Debug can be exited using the C Debug menu.

Setting File-Line Breakpoints Using the bp Command

Use the `bp` command to set a file-line breakpoint from the `VSIM>` prompt. For example:

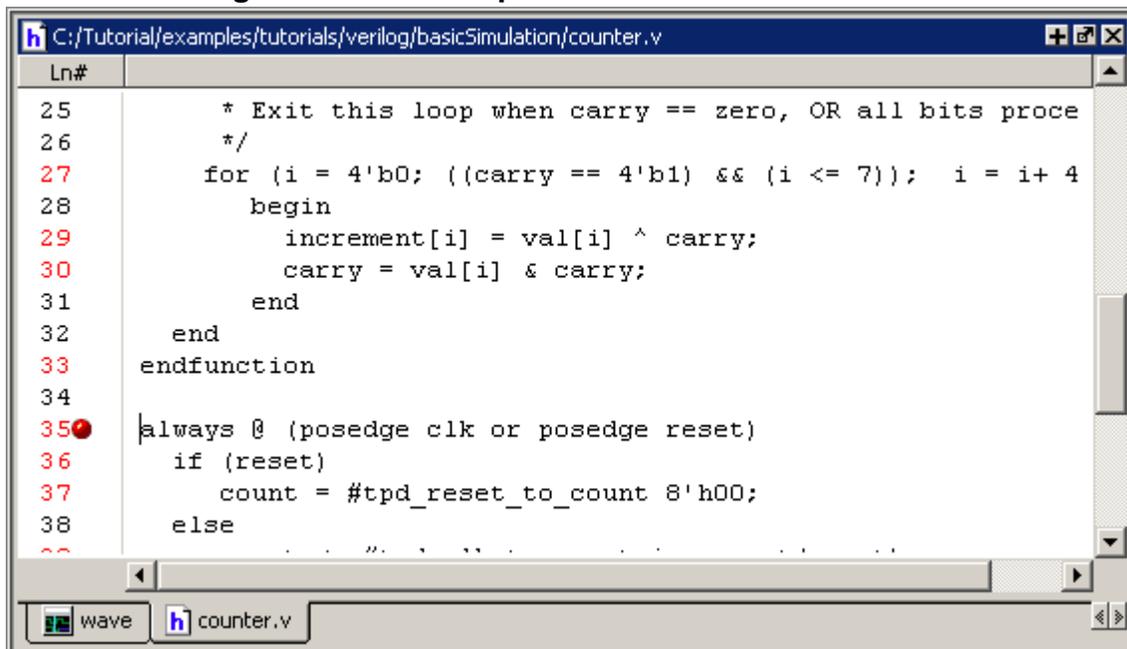
```
bp top.vhd 147
```

sets a breakpoint in the source file *top.vhd* at line 147.

Setting File-Line Breakpoints Using the GUI

File-line breakpoints are most easily set using your mouse in the [Source Window](#). Position your mouse cursor in the line number column next to a red line number (which indicates an executable line) and click the left mouse button. A red ball denoting a breakpoint will appear ([Figure 15-55](#)).

Figure 15-55. Breakpoints in the Source Window



The breakpoints are toggles. Click the left mouse button on the red breakpoint marker to disable the breakpoint. A disabled breakpoint will appear as a black ball. Click the marker again to enable it.

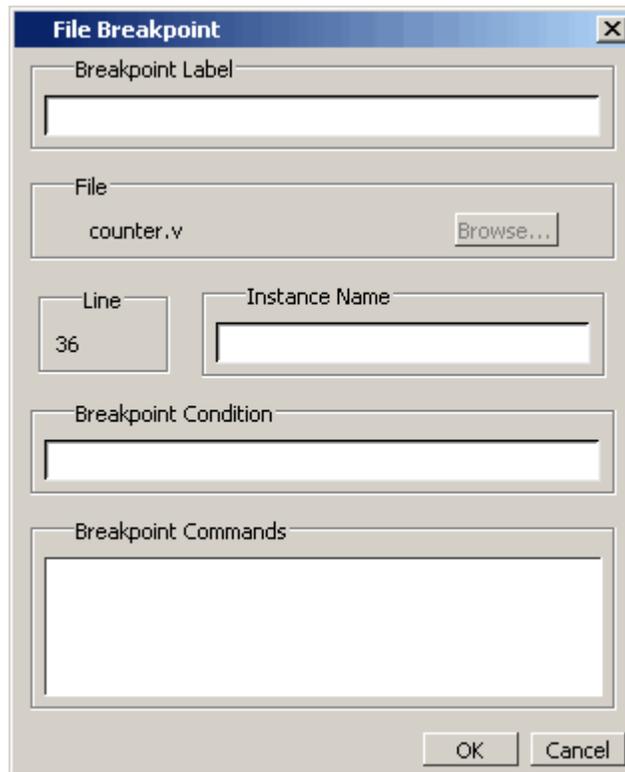
Right-click the breakpoint marker to open a context menu that allows you to **Enable/Disable**, **Remove**, or **Edit** the breakpoint. create the colored diamond; click again to disable or enable the breakpoint.

Modifying a File-Line Breakpoint

You can modify a file-line breakpoint by selecting **Tools > Breakpoints** from the Main menus. This will open the Modify Breakpoints dialog (Figure 15-53), which displays a list of all breakpoints in the design.

When you select a file-line breakpoint from the list and click the Modify button, the File Breakpoint dialog (Figure 15-56) opens, allowing you to modify the breakpoint.

Figure 15-56. File Breakpoint Dialog Box



Saving and Restoring Breakpoints

The [write format](#) restart command creates a single *.do* file that will recreate all debug windows, all file/line breakpoints, and all signal breakpoints created using the [when](#) command. The syntax is:

write format restart <filename>

If the [ShutdownFile](#) *modelsim.ini* variable is set to this *.do* filename, it will call the [write format](#) restart command upon exit.

The file created is primarily a list of [add list](#), [add wave](#), and [configure](#) commands, though a few other commands are included. This file may be invoked with the [do](#) command to recreate the window format on a subsequent simulation run.

Waveform Compare

The ModelSim Waveform Compare feature allows you to compare simulation runs. Differences encountered in the comparison are summarized and listed in the Main window transcript and are shown in the Wave and List windows. In addition, you can write a list of the differences to a file using the [compare info](#) command.

The basic steps for running a comparison are as follows:

1. Run one simulation and save the dataset. For more information on saving datasets, see [Saving a Simulation to a WLF File](#).
2. Run a second simulation.
3. Setup and run a comparison.
4. Analyze the differences in the Wave or List window.

Mixed-Language Waveform Compare Support

Mixed-language compares are supported as listed in the following table:

Table 15-8. Mixed-Language Waveform Compares

Language	Compares
C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg

The number of elements must match for vectors; specific indexes are ignored.

Three Options for Setting up a Comparison

There are three options for setting up a comparison:

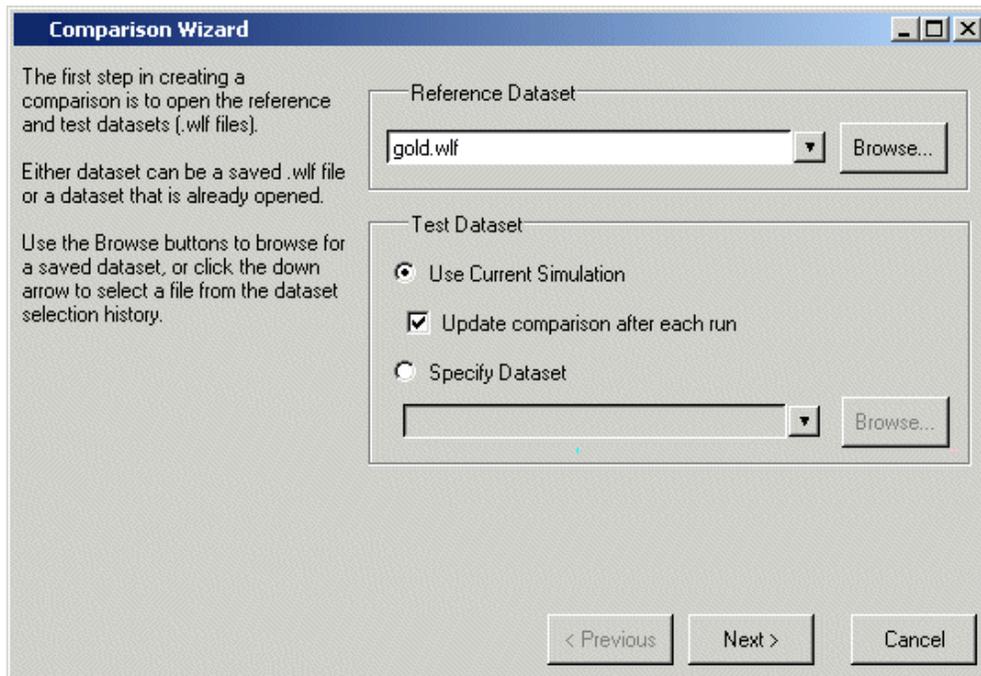
- Comparison Wizard – A series of dialogs that "walk" you through the process
- Comparison commands – Use a series of **compare** commands
- GUI – Use various dialogs to "manually" configure the comparison

Comparison Wizard

The simplest method for setting up a comparison is using the Wizard. The wizard is a series of dialogs that walks you through the process. To start the Wizard, select **Tools > Waveform Compare > Comparison Wizard** from either the Wave or Main window.

The graphic below shows the first dialog in the Wizard. As you can see from this example, the dialogs include instructions on the left-hand side.

Figure 15-57. Waveform Comparison Wizard



Comparison Graphic Interface

You can also set up a comparison via the GUI without using the Wizard. The steps of this process are described further in [Setting Up a Comparison with the GUI](#).

Comparison Commands

There are numerous commands that give you complete control over a comparison. These commands can be entered in the Transcript window or run via a DO file. The commands are detailed in the Reference Manual, but the following example shows the basic sequence:

```
compare start gold vsim  
compare add /*  
compare run
```

This example command sequence assumes that the *gold.wlf* reference dataset is loaded with the current simulation, the *vsim.wlf* dataset. The [compare start](#) command instructs ModelSim to compare the reference *gold.wlf* dataset against the current simulation. The **compare add /*** command instructs ModelSim to compare all signals in the *gold.wlf* reference dataset against all signals in the *vsim.wlf* dataset. The [compare run](#) command runs the comparison.

Comparing Signals with Different Names

You can use the [compare add](#) command to specify a comparison between two signals with different names.

Setting Up a Comparison with the GUI

To setup a comparison with the GUI, follow these steps:

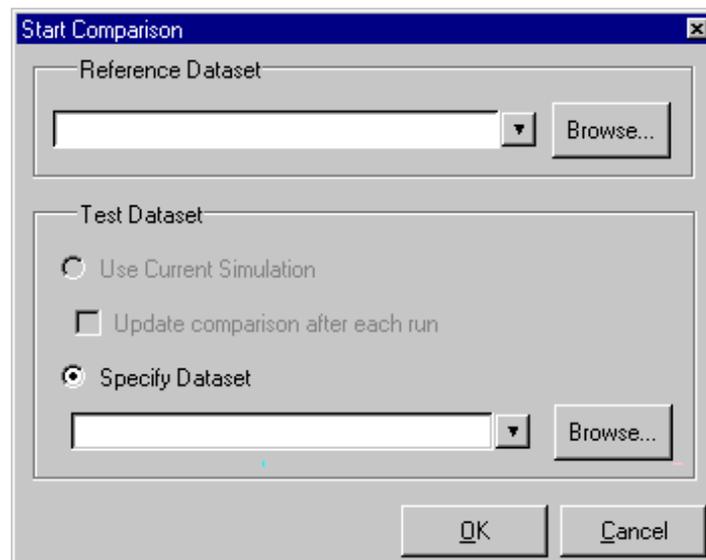
1. Initiate the comparison by specifying the reference and test datasets. See [Starting a Waveform Comparison](#) for details.
2. Add objects to the comparison. See [Adding Signals, Regions, and Clocks](#) for details.
3. Specify the comparison method. See [Specifying the Comparison Method](#) for details.
4. Configure comparison options. See [Setting Compare Options](#) for details.
5. Run the comparison by selecting Tools > Waveform Compare > Run Comparison.
6. View the results. See [Viewing Differences in the Wave Window](#), [Viewing Differences in the List Window](#), and [Viewing Differences in Textual Format](#) for details.

Waveform Compare is initiated from either the Main or Wave window by selecting **Tools > Waveform Compare > Start Comparison**.

Starting a Waveform Comparison

Select **Tools > Waveform Compare > Start Comparison** to initiate the comparison. The Start Comparison dialog box allows you define the Reference and Test datasets.

Figure 15-58. Start Comparison Dialog



Reference Dataset

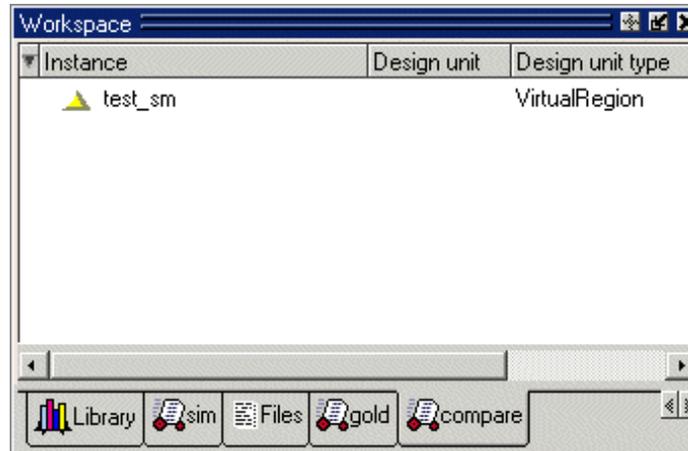
The Reference Dataset is the *.wlf* file to which the test dataset will be compared. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

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.

Once you click **OK** in the Start Comparison dialog box, ModelSim adds a Compare tab to the Main window.

Figure 15-59. Compare Tab in the Workspace Pane



After adding the signals, regions, and/or clocks you want to use in the comparison (see [Adding Signals, Regions, and Clocks](#)), you will be able to drag compare objects from this tab into the Wave and List windows.

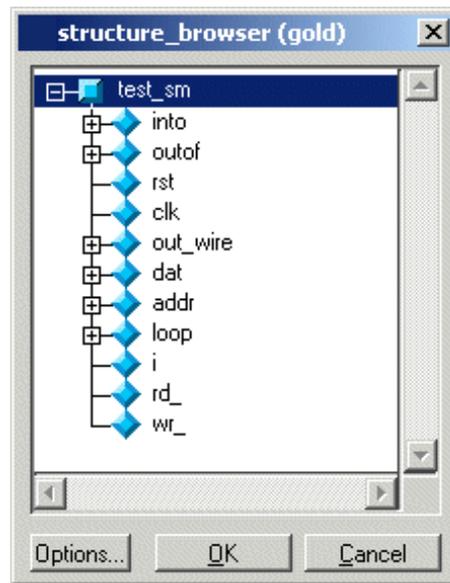
Adding Signals, Regions, and Clocks

To designate the signals, regions, or clocks to be used in the comparison, click **Tools > Waveform Compare > Add**.

Adding Signals

Clicking **Tools > Waveform Compare > Add > Compare by Signal** in the Wave window opens the structure_browser window, where you can specify signals to be used in the comparison.

Figure 15-60. Structure Browser



Adding Regions

Rather than comparing individual signals, you can also compare entire regions of your design. Select **Tools > Waveform Compare > Add > Compare by Region** to open the Add Comparison by Region dialog.

Figure 15-61. Add Comparison by Region Dialog



Adding Clocks

You add clocks when you want to perform a clocked comparison. See [Specifying the Comparison Method](#) for details.

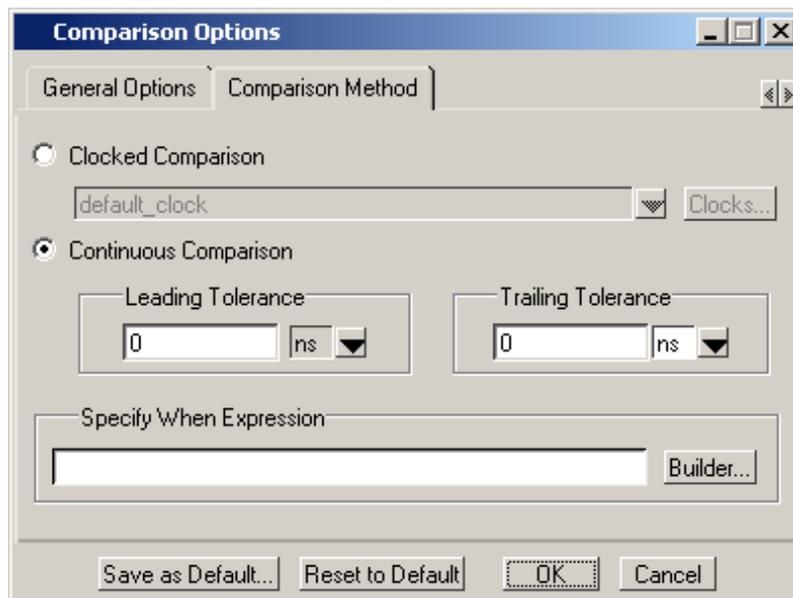
Specifying the Comparison Method

The Waveform Compare feature provides two comparison methods:

- Continuous comparison — Test signals are compared to reference signals at each transition of the reference. Timing differences between the test and reference signals are shown with rectangular red markers in the Wave window and yellow markers in the List window.
- Clocked comparisons — Signals are compared only at or just after an edge on some signal. In this mode, you define one or more clocks. The test signal is compared to a reference signal and both are sampled relative to the defined clock. The clock can be defined as the rising or falling edge (or either edge) of a particular signal plus a user-specified delay. The design need not have any events occurring at the specified clock time. Differences between test signals and the clock are highlighted with red diamonds in the Wave window.

To specify the comparison method, select **Tools > Waveform Compare > Options** and select the Comparison Method tab.

Figure 15-62. Comparison Methods Tab



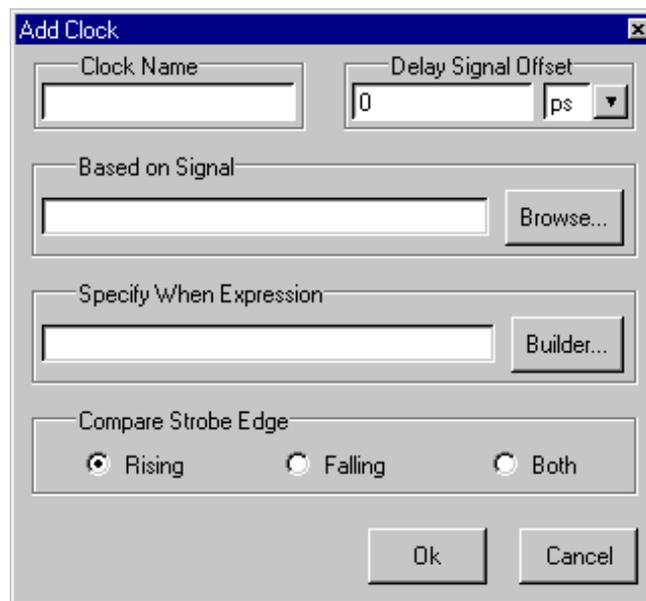
Continuous Comparison

Continuous comparisons are the default. You have the option of specifying leading and trailing tolerances and a when expression that must evaluate to "true" or 1 at the signal edge for the comparison to become effective.

Clocked Comparison

To specify a clocked comparison you must define a clock in the Add Clock dialog. You can access this dialog via the Clocks button in the Comparison Method tab or by selecting **Tools > Waveform Compare > Add > Clocks**.

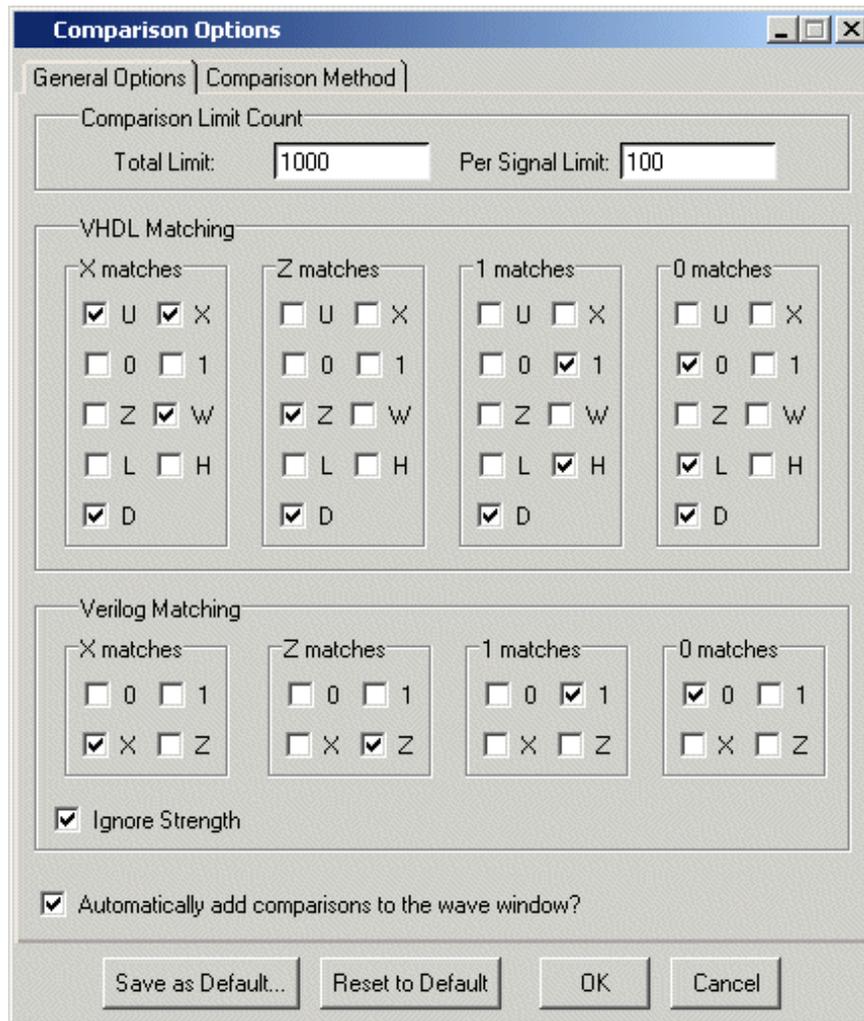
Figure 15-63. Adding a Clock for a Clocked Comparison



Setting Compare Options

There are a few "global" options that you can set for a comparison. Select **Tools > Waveform Compare > Options**.

Figure 15-64. Waveform Comparison Options

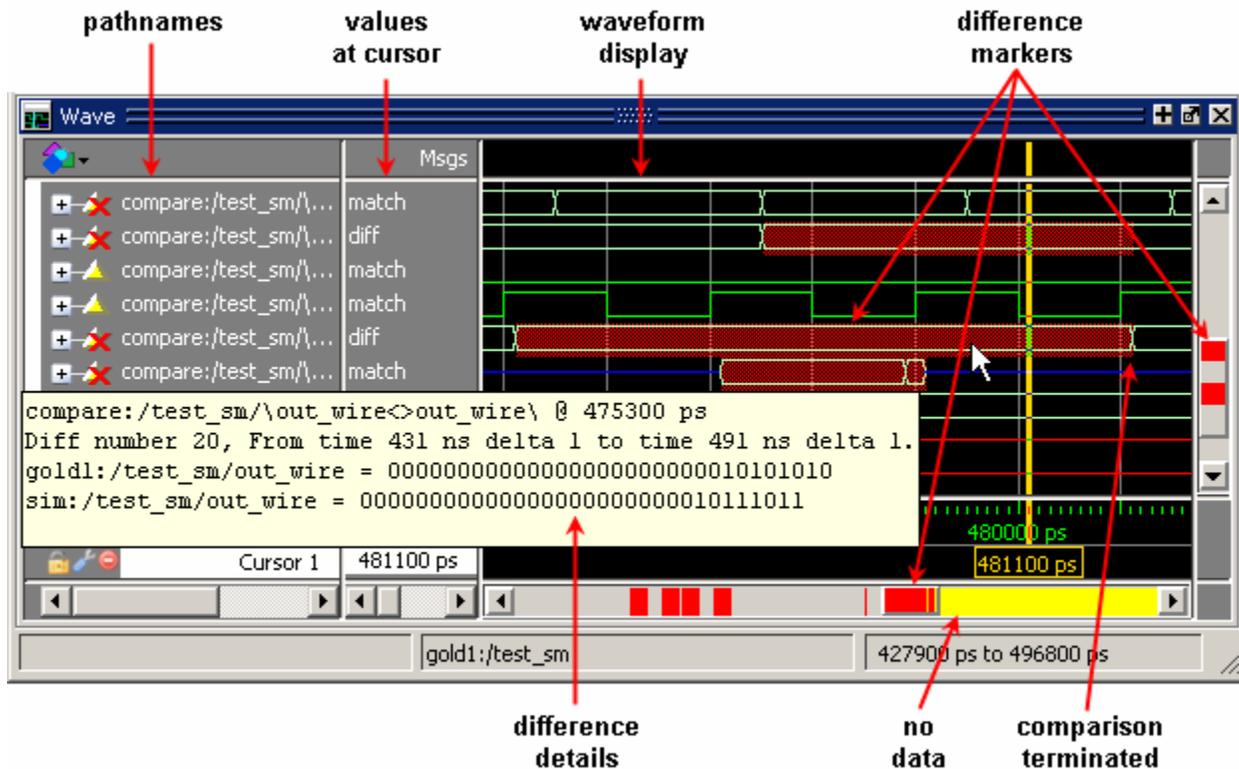


Options in this dialog include setting the maximum number of differences allowed before the comparison terminates, specifying signal value matching rules, and saving or resetting the defaults.

Viewing Differences in the Wave Window

The Wave window provides a graphic display of comparison results. Pathnames of all test signals included in the comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.

Figure 15-65. Viewing Waveform Differences in the Wave Window



The names of the comparison objects take the form:

```
<path>/\refSignalName<>testSignalName\
```

If you compare two signals from different regions, the signal names include the uncommon part of the path.

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.

If the total number of differences between test and reference signals exceeds the maximum difference limit, a yellow marker appears in the horizontal scroll bar, showing where waveform comparison was terminated and no data was collected. You can set the difference limit in the [Waveform Comparison Options](#) dialog box or with the [compare options](#) or [compare start](#) commands.

The values column of the Wave window displays the words "match", "diff", or "No Data" 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. “No Data” indicates that the cursor is placed in an area where comparison of test and reference signals stopped.

In comparisons of signals with multiple bits, you can display them in "buswise" or "bitwise" format. Buswise format lists the busses under the compare object whereas bitwise format lists each individual bit under the compare object. To select one format or the other, click your right mouse button on the plus sign ('+') next to a compare object.

Annotating Differences

You can tag differences with textual notes that are included in the difference details popup and comparison reports. Click a difference with the right mouse button, and select **Annotate Diff.** Or, use the [compare annotate](#) command.

Compare Icons

The Wave window includes six comparison icons that let you quickly jump between differences. From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.



These buttons cycle through differences on all signals. To view differences for just the selected signal, press <tab> and <shift - tab> on your keyboard.

Note



If you have differences on individual bits of a bus, the compare icons will stop on those differences but <tab> and <shift - tab> will not.

The compare icons cycle through comparison objects in all open Wave windows. If you have two Wave windows displayed, each containing different comparison objects, the compare icons will cycle through the differences displayed in both windows.

Viewing Differences in the List Window

Compare objects can be displayed in the List window too. Differences are highlighted with a yellow background. Tabbing on selected columns moves the selection to the next difference (actually difference edge). Shift-tabbing moves the selection backwards.

Figure 15-66. Waveform Differences in the List Window

ns	delta	compare: /top/\clk<>clk\	compare: /top/\paddr<>paddr\	compare: /top/\prw<>prw\	compare: /top/\pstrb<>pstrb\	compare: /top/\prdy<>prdy\
1980	+0	1 1 0 0 1 1 1 1	00001001 00001001	00000000000001001		
1985	+0	1 1 0 0 1 1 1 1	00001001 00001001	00000000000001001		
1990	+0	1 1 0 0 1 1 0 0	00001001 00001001	00000000000001001		
2000	+0	0 0 0 0 1 1 0 0	00001001 00001001	00000000000001001		
2020	+0	1 1 0 0 1 1 0 0	00001001 00001001	00000000000001001		
2025	+0	1 1 1 0 0 1 1 1	00000000 00001001	ZZZZZZZZZZZZZZZZ		
2035	+0	1 1 1 1 0 0 1 1	00000000 00000000	ZZZZZZZZZZZZZZZZ		
2040	+0	0 0 1 1 0 0 1 1	00000000 00000000	ZZZZZZZZZZZZZZZZ		
2060	+0	1 1 1 1 0 0 1 1	00000000 00000000	ZZZZZZZZZZZZZZZZ		
2065	+0	1 1 1 1 1 1 0 0	00000000 00000000	0000000000000000		
2080	+0	0 0 1 1 1 1 0 0	00000000 00000000	0000000000000000		
2100	+0	1 1 1 1 1 1 0 0	00000000 00000000	0000000000000000		
2105	+0	1 1 1 1 0 0 1 1	00000001 00000000	ZZZZZZZZZZZZZZZZ		
2120	+0	0 0 1 1 0 0 1 1	00000001 00000000	ZZZZZZZZZZZZZZZZ		
2140	+0	1 1 1 1 0 0 1 1	00000001 00000000	ZZZZZZZZZZZZZZZZ		
2145	+0	1 1 1 1 1 1 0 0	00000001 00000000	0000000000000001		
2160	+0	0 0 1 1 1 1 0 0	00000001 00000000	0000000000000001		
2180	+0	1 1 1 1 1 1 0 0	00000001 00000000	0000000000000001		

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.

Viewing Differences in Textual Format

You can also view text output of the differences either in the Transcript pane of the Main window or in a saved file. To view them in the transcript, select **Tools > Waveform Compare > Differences > Show**. To save them to a text file, select **Tools > Waveform Compare > Differences > Write Report**.

Saving and Reloading Comparison Results

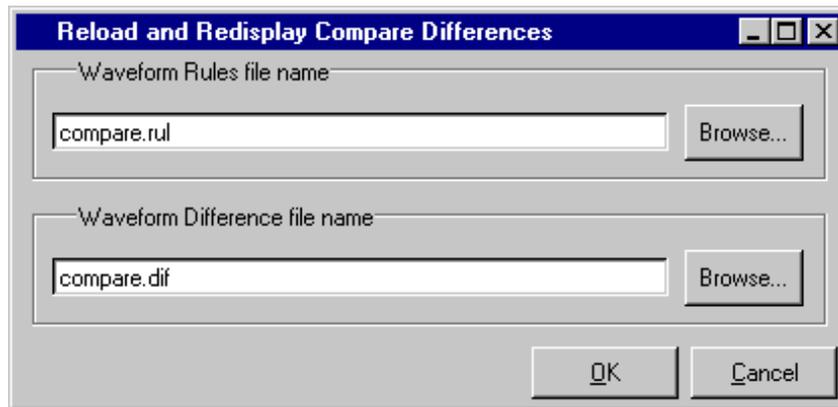
To save comparison results for future use, you must save both the comparison setup rules and the comparison differences.

To save the rules, select **Tools > Waveform Compare > Rules > Save**. This file will contain all rules for reproducing the comparison. The default file name is "*compare.rul*."

To save the differences, select **Tools > Waveform Compare > Differences > Save**. The default file name is "*compare.dif*."

To reload the comparison results at a later time, select **Tools > Waveform Compare > Reload** and specify the rules and difference files.

Figure 15-67. Reloading and Redisplaying Compare Differences



Comparing Hierarchical and Flattened Designs

If you are comparing a hierarchical RTL design simulation against a flattened synthesized design simulation, you may have different hierarchies, different signal names, and the buses may be broken down into one-bit signals in the gate-level design. All of these differences can be handled by ModelSim's Waveform Compare feature.

- If the test design is hierarchical but the hierarchy is different from the hierarchy of the reference design, you can use the `compare add` command 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 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 to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

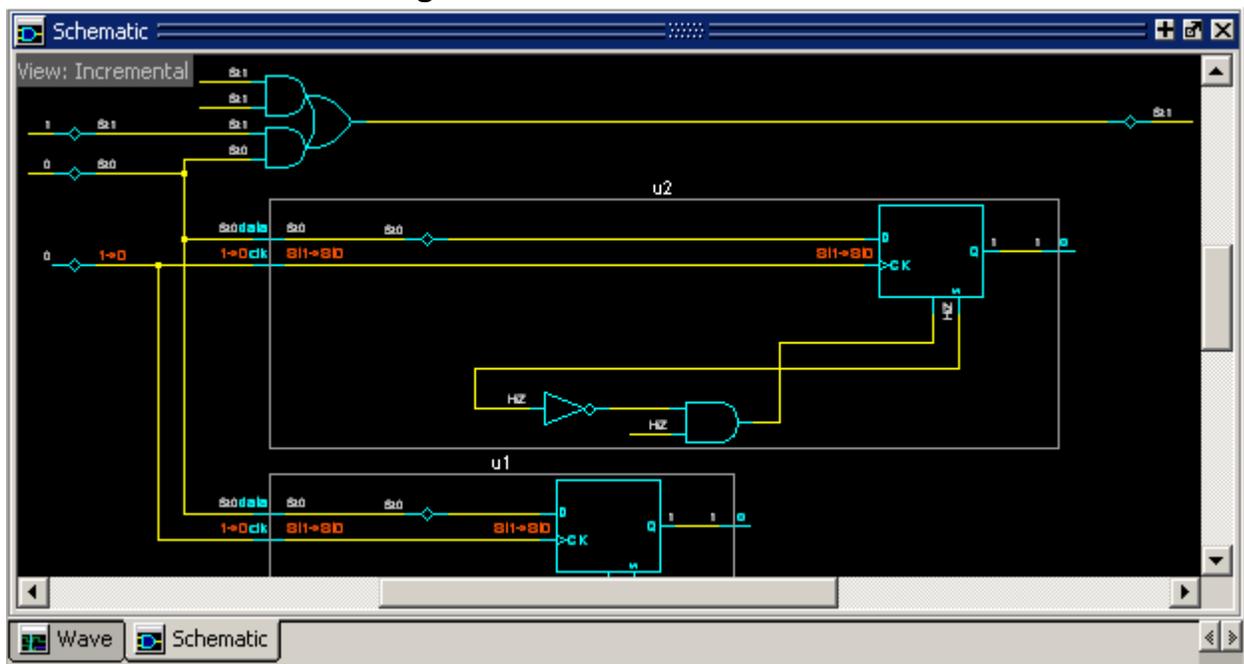
If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Compare feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Compare will let you know.

Chapter 16

Schematic Window

The Schematic window provides an implementation view of your design, allowing you to see design structure, connectivity, and hierarchy without consulting the RTL. It allows you to explore the “physical” connectivity of your design; to trace events that propagate through the design; and to identify the cause of unexpected outputs.

Figure 16-1. Schematic Window



The Schematic window displays both synthesizable and non-synthesizable parts of your design. For the synthesizable parts, the Schematic window will:

- Show connectivity between components and separate data paths from control paths;
- Identify clock and event triggers;
- Separate combinational (Mux, Gates , Tristates) and sequential logic (Flops);
- Infer RAM/ROM blocks.

In addition, integrated features like Causality Traceback and fan-in/fan-out trace help you explore and debug the synthesizable parts of your design.

Non-synthesizable constructs are enclosed in black boxes in the Schematic window display, and connectivity with surrounding context is maintained.

Schematic Window Usage Flow

The Schematic window can be used to debug the design during simulation, or to perform post-simulation debugging. To enable the full debug capabilities of the Schematic window, you must create a debug database at design load time, before elaboration. The database specifies the combinatorial and sequential elements of your design. Then, the data generated during a simulation run is logged into the database for immediate debugging or post-sim debugging.

1. Create a library for your work

vlib <library_name>

2. Compile your design

vlog/vcom <design_name>

3. Optimize your design

vopt +acc <design_name> **-o** <optimized_design_name> **-debugdb**

The **+acc** switch enables full visibility into the design for debugging purposes. The **-o** option is required for naming the optimized design object. And the **-debugdb** switch collects combinatorial and sequential logic data into the work library.

4. Load the design

vsim -debugdb <optimized_design_name>

The **-debugdb** switch creates a debug database, with the default file name *vsim.dbg*, in the current working directory. This database contains annotated schematic connectivity information.

5. Log simulation data:

log -r /*

It is advisable to log the entire design. This will provide the historic values of the events of interest plus its drivers. However, to reduce overhead, you may log only the regions of interest.

You may use the **log** command to simply save the simulation data to the *.wlf* file; or, use the **add wave** command to log the simulation data to the *.wlf* file *and* display simulation results as waveforms in the Wave window.

6. Run the simulation.
7. Debug your design using the Schematic window.
8. Quit the simulation.

Note

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

Post Simulation Schematic Debug Flow

The post simulation debug flow for Schematic analysis is most commonly used when performing simulations of large designs in simulation farms, where simulation results are gathered over extended periods and saved for analysis at a later date.

1. Start ModelSim by typing **vsim** at a UNIX shell prompt; or double-click a ModelSim icon in Windows.
2. Select **File > Change Directory** and change to the directory where the post-simulation debug database resides.
3. Recall the post-simulation debug database with the following:

```
vsim -view <db_pathname.wlf>
```

ModelSim opens the *.wlf* dataset and its associated debug database (*.dbg* file with the same basename), if it can be found. If ModelSim cannot find *db_pathname.dbg*, it will attempt to open *vsim.dbg*.

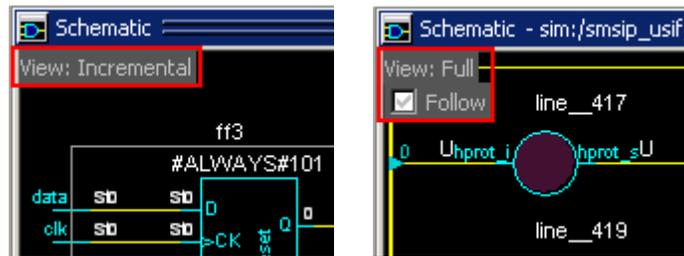
Two Schematic Views

The Schematic window provides two views of the design — a Full View, which is a structural overview of the design hierarchy; and an Incremental View, which uses Click-and-Sprout actions to incrementally add to the selected net's fanout.

- The Full View provides the connectivity information between various components of a module/architecture. The lowest granularity is process/always blocks. The Follow mode (Figure 16-2) allows the Full View to synchronize with the Incremental View.
- The Incremental View displays the logical gate equivalent of the RTL portion of the design, making it easier to understand the intent of the design. It allows you to start by displaying only a net or block, and then double-click the net to sprout drivers and readers. It is ideal for design debugging, allowing you to explore design connectivity by tracing signal readers/drivers to determine where and why signals change values at various times.

The “View” indicator is displayed in the top left corner of the window (Figure 16-2). You can toggle back and forth between views by simply clicking this “View” indicator.

Figure 16-2. Schematic View Indicator



Common Tasks for Schematic Debugging

Common tasks for current and post-simulation Schematic debugging include:

- Adding Objects to the Incremental View
- Display a Structural Overview in the Full View
- Exploring the Connectivity of the Design
- Folding and Unfolding Instances in the Incremental View
- Exploring Designs with the Embedded Wave Viewer
- Tracing Events in the Incremental View
- Tracing the Source of an Unknown State (StX)
- Finding Objects by Name in the Schematic Window

Adding Objects to the Incremental View

You can use any of the following methods to add objects to the Schematic window's Incremental View:

- Drag and drop objects from other windows. Both nets and instances may be dragged and dropped. Dragging an instance will result in the addition of all nets connected to that instance.
- Use the **Add > To Schematic** menu options:
 - **Selected Signals**— Display selected signals
 - **Signals in Region**— Display all signals from the current region.
 - **Signals in Design**— Clear the window and display all signals from the entire design.
- Select the object(s) you want placed in the Schematic Window, then click-and-hold the [Add Selected to Window Button](#) in the **Standard** toolbar and select **Add to Schematic**. 

- Use the `add schematic` command.

When you view regions or entire nets, the window initially displays only the drivers of the added objects. You can easily view readers as well by selecting an object, right-clicking the object, then selecting **Expand Net To > Readers** from the popup menu.

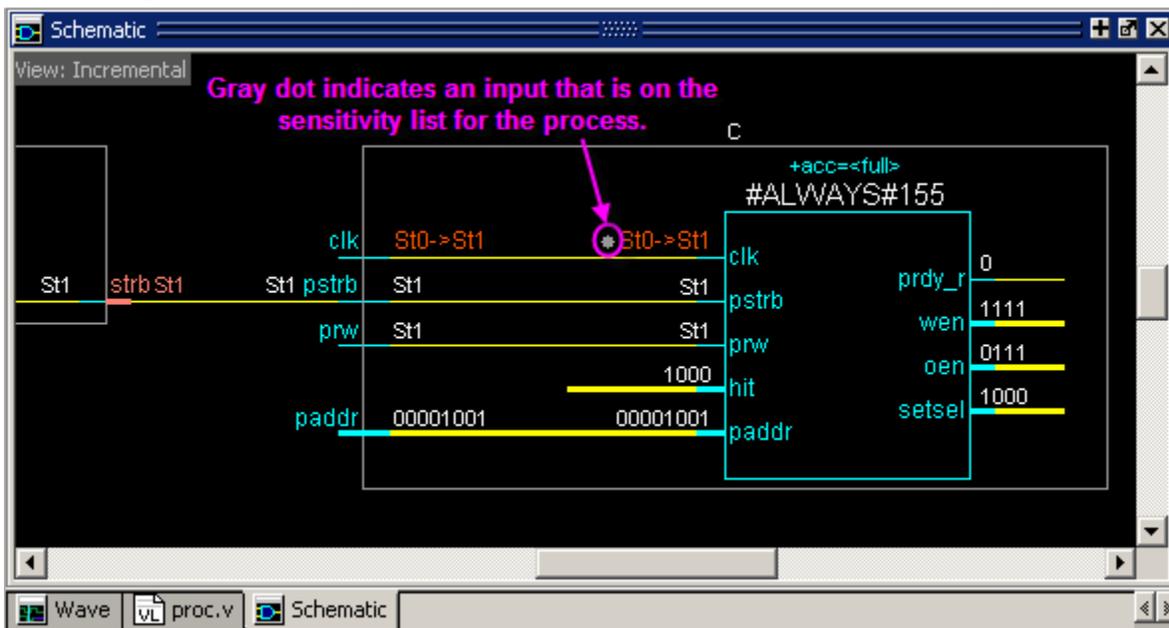
The Incremental view provides automatic indication of input signals that are included in the process sensitivity list. [Figure 16-3](#) shows the gray dot next to the state of the input `clk` signal for the `#ALWAYS#155` process. This indicates that the `clk` signal is in the sensitivity list for the process and will trigger process execution. Inputs without gray dots are read by the process but will not trigger process execution, and are not in the sensitivity list (will not change the output by themselves).

Note



Gray dots are only shown on the signals in the sensitivity list of a process that did not synthesize down to gate components. Gates will not have the grey dots because the behavior of their inputs is clearly defined.

Figure 16-3. Gray Dot Indicates Input in Process Sensitivity List



Display a Structural Overview in the Full View

To display a structural overview of a region of the design in the Schematic window's Full View simply click the Follow box in the Schematic View indicator ([Figure 16-2](#)). When the Follow box is checked, any design unit you select in the Structure window is displayed in the Full View. If you then select a specific signal in the Objects windows, the selected signal is highlighted in the Full View. In other words, the Full View follows the selections you make in

other windows that are dynamically connected to the Schematic window. It allows you to quickly find specific signals within the overall design schematic.

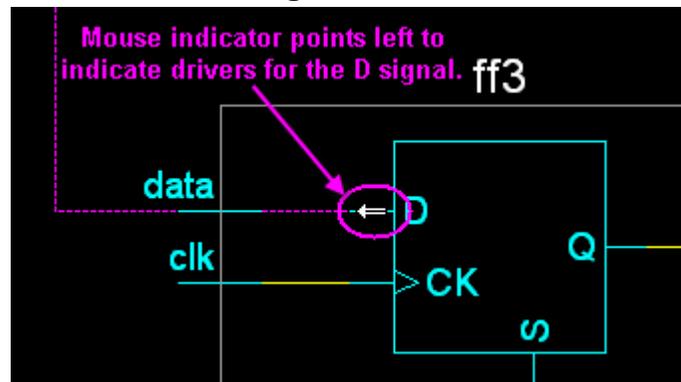
Exploring the Connectivity of the Design

A primary use of the Incremental view is to explore the “physical” connectivity of your design. When you mouse over any signal it changes from a solid to a dashed line, giving you a quick visual indicator of where you are in your design and how design elements are connected.

You can explore connectivity further by expanding the view from process to process. This allows you to see the drivers and readers of a particular signal, net, or register.

You can expand the view of your design using menu selections or your mouse. When you hover the mouse over a signal pin, the mouse cursor will change to a right-pointing or left-pointing arrow. If the arrow points to the right, you can double-click the pin to expand the net’s fanout to its readers. If the arrow points left, you can double-click the pin to expand the net’s fanout to its drivers (Figure 16-4).

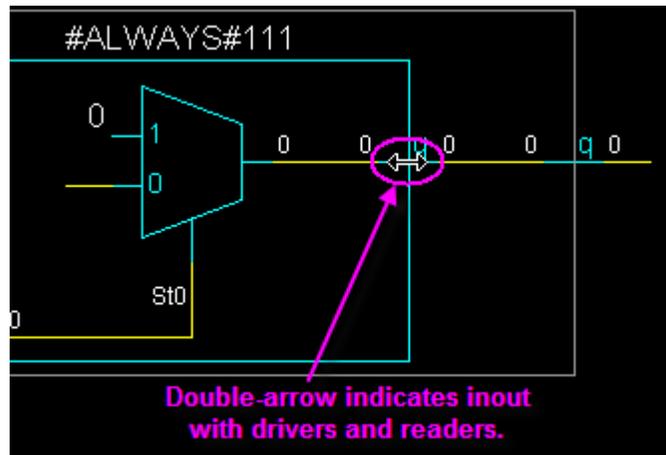
Figure 16-4. Left-Pointing Mouse Arrow Indicates Drivers



You can change the default click-and-sprout expansion mode from a double-click of the left mouse button to a single click by pressing the C shortcut key. (See [How do I Use Keyboard Shortcuts?](#))

A double-headed arrow that points in both directions indicates an inout signal pin, with drivers and readers (Figure 16-5).

Figure 16-5. Double-Headed Arrow Indicates Inout with Drivers and Readers



To expand with the mouse, simply double-click a signal pin. Depending on the specific pin you double-click, the view will expand to show the driving process and interconnecting nets, the reading process and interconnecting nets, or both.

Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons in the first column of [Table 16-1](#); or, right-click the selected item and make the menu selection described in the second column of [Table 16-1](#).

Table 16-1. Icon and Menu Selections for Exploring Design Connectivity

	Expand net to all drivers display driver(s) of the selected signal, net, or register	Expand Net To > Drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Expand Net To > Drivers & Readers
	Expand net to all readers display reader(s) of the selected signal, net, or register	Expand Net To > Readers

As you expand the view, the layout of the design may adjust to show the connectivity more clearly. For example, the location of an input signal may shift from the bottom to the top of a process.

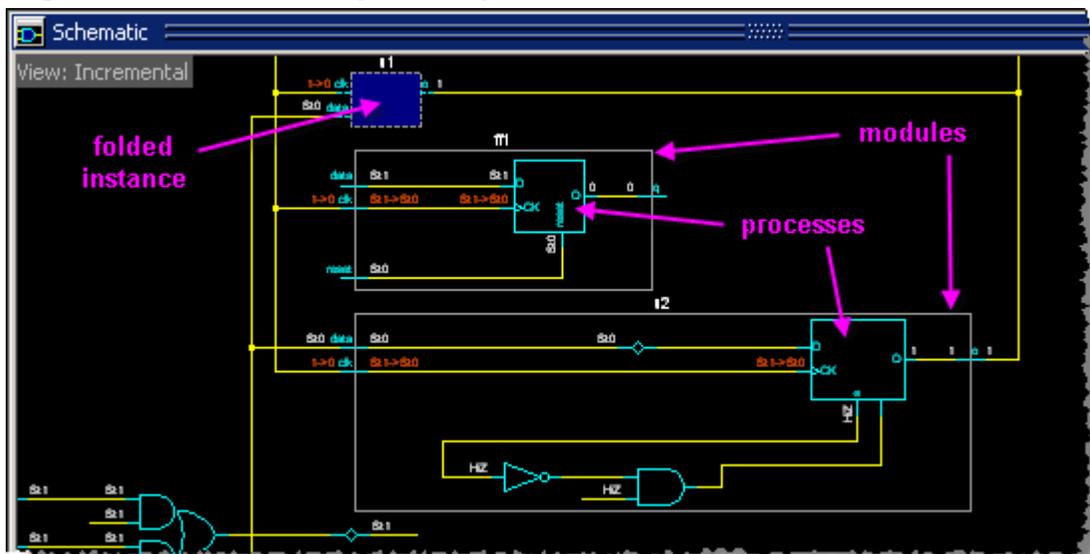
In addition, you may use the Regenerate button in the [Schematic Toolbar](#) to automatically clear and redraw either the Incremental or the Full view in order to better display schematic information. For example, if you turn on signal values, some values for the pins of adjacent processes may overlap. Click the Regenerate button to automatically redraw the schematic so values do not overlap.

Viewing Design Information in the Incremental View

The Schematic window provides the following features for getting design information from the Incremental view:

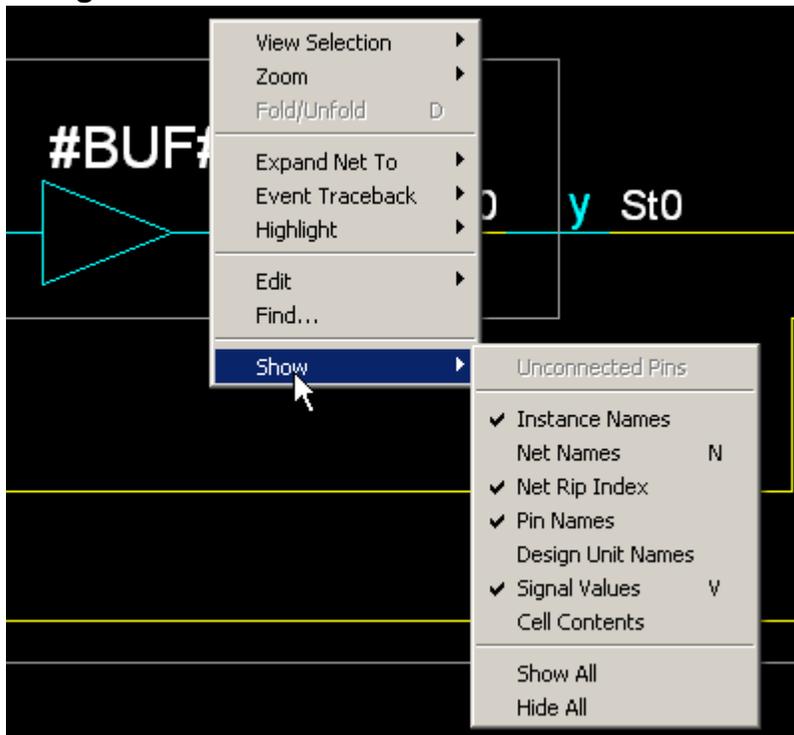
- The Incremental view displays design primitives – logic gates, buffers, fifos, muxs, etc. – as commonly recognized symbols for easy identification.
- Colors help you identify different design elements. For example, light gray boxes denote VHDL architectures and Verilog modules. Blue boxes denote processes (Figure 16-6). Solid blue boxes with dashed white borders denote folded instances (see [Folding and Unfolding Instances in the Incremental View](#)).

Figure 16-6. Colors Help Identify Architectures, Modules, and Processes



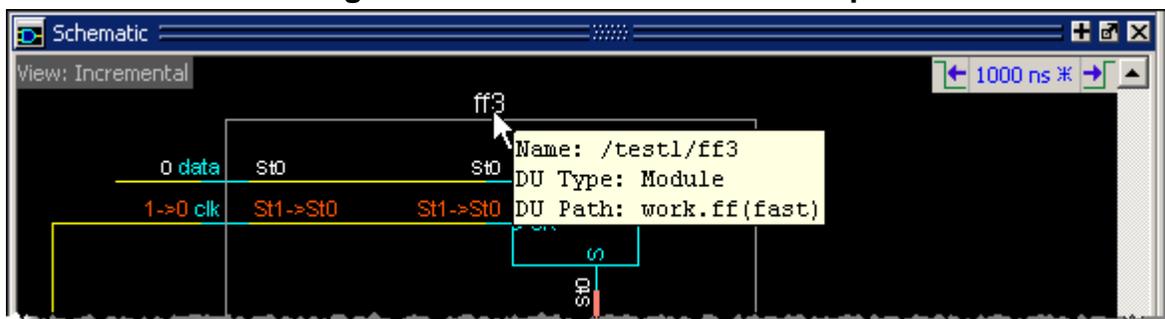
- Customize the annotation of the Incremental view with the **Schematic > Show** menu selection, or right-clicking the Incremental view and selecting **Show** to open the annotation options (Figure 16-7). By default, all displayed signal values are for the current active time, as displayed in the Active Time label.

Figure 16-7. Show Incremental View Annotation



- Hovering the mouse cursor over a design object opens a tooltip (text popup box) that displays design object information for the specific object type. For example, the tooltip for a module displays the module name, design unit type, and design unit path as shown in Figure 16-8.

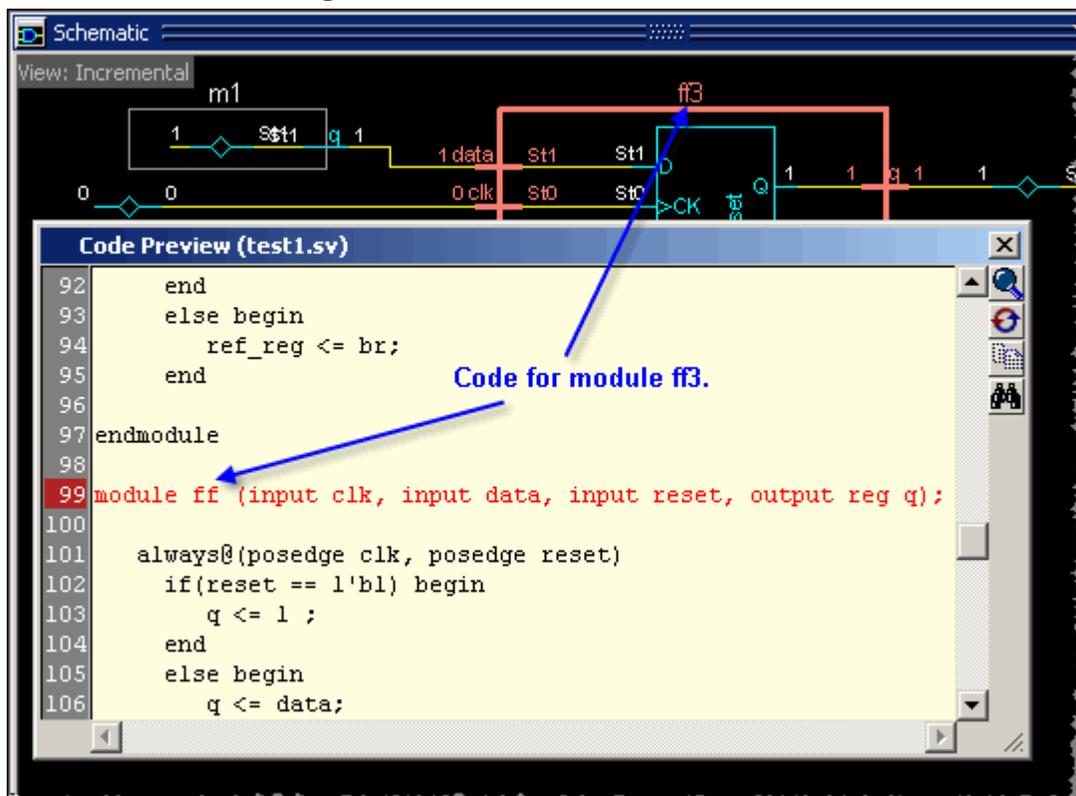
Figure 16-8. Hover Mouse for Tooltip



The tooltip for a signal net displays the net name and its value at the active time.

- Double-click any object in the Incremental view to view its source code in a Code Preview window. The code for the selected object is highlighted (Figure 16-9).

Figure 16-9. Code Preview Window



The Code Preview window includes a four-button toolbar that provides the options shown in [Table 16-2](#)

Table 16-2. Code Preview Toolbar Buttons

	View in Source Editor — Opens the code in an annotated source code window where the code can be edited.
	Recenter on Target Line — Recenters the highlighted code so it appears in the center of the Code Preview.
	Copy Selection — Copies the selected code so it can be pasted into another point in the code, or to a text editor.
	Find — Opens a Find toolbar at the bottom of the Code Preview window, allowing you to search for a signal, net, register or instance by name. See Finding Objects by Name in the Schematic Window .

Limiting the Display of Readers

Some nets (such as a clock) in a design can have many readers. This can cause the display to draw numerous processes that you do not want to see when expanding the selected signal, net, or register. The schematic display tests for the number of readers to be drawn and compares that

number to a limit that you set in Schematic Preferences (**Tools > Edit Preferences > By Name tab > Schematic > outputquerylimit**). The default value of this limit is 100 (if you set **outputquerylimit** to 0, the test is not done). If this limit is exceeded, a dialog box asks whether you want all readers to be drawn. If you choose No, then no readers are displayed.

Note

This limit does not affect the display of drivers.

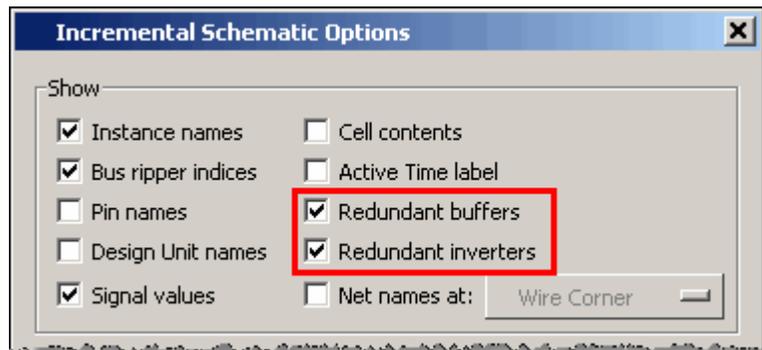
Controlling the Display of Redundant Buffers and Inverters

The Schematic window automatically traces a signal through buffers and inverters. This can cause chains of redundant buffers or inverters to be displayed in the Schematic window. You can collapse these chains of buffers or inverters to make the design displayed in the Schematic window more compact.

Incremental View

To change the display of redundant buffers and inverters in the Incremental view: with the Incremental Schematic window active, select **Schematic > Preferences** to open the Incremental Schematic Options dialog. The default setting is to display both redundant buffers and redundant inverters (Figure 16-10).

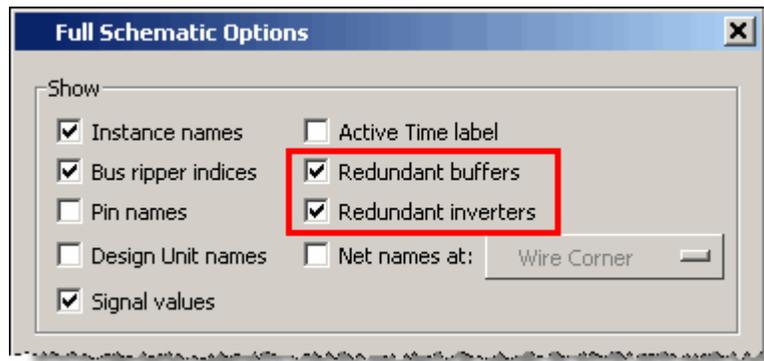
Figure 16-10. Redundant Buffers and Inverters in the Incremental Schematic View



Full View

To change the display of redundant buffers and inverters in the Full view: with the Full Schematic window active, select **Schematic > Preferences** to open the Full Schematic Options dialog. The default setting is to display both redundant buffers and redundant inverters (Figure 16-11).

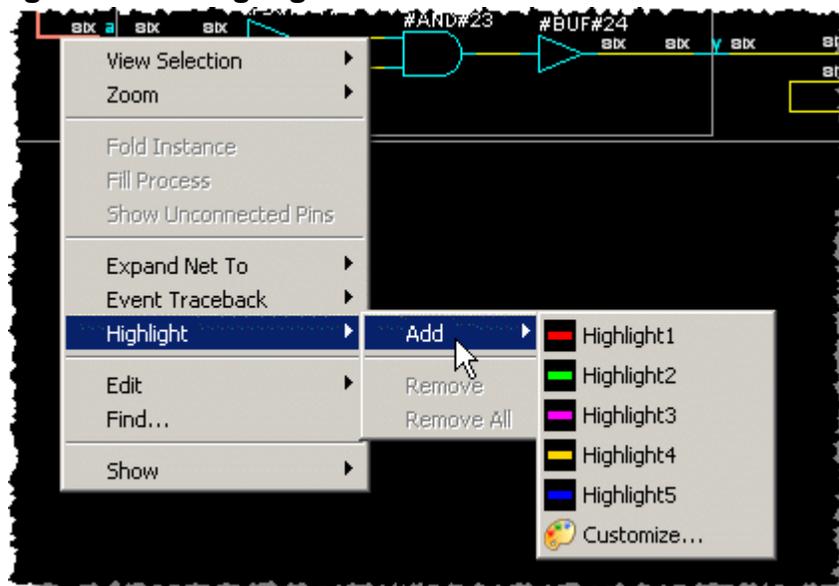
Figure 16-11. Redundant Buffers and Inverters in the Full Schematic View



Tracking Your Path Through the Design with Highlighting

You can highlight any selected trace with any color of your choice by right-clicking Schematic window and selecting **Highlight > Add** from the popup menu (Figure 16-12).

Figure 16-12. Highlight Selected Trace with Custom Color



You can then choose from one of five pre-defined colors, or **Customize** to choose from the palette in the Preferences dialog box.

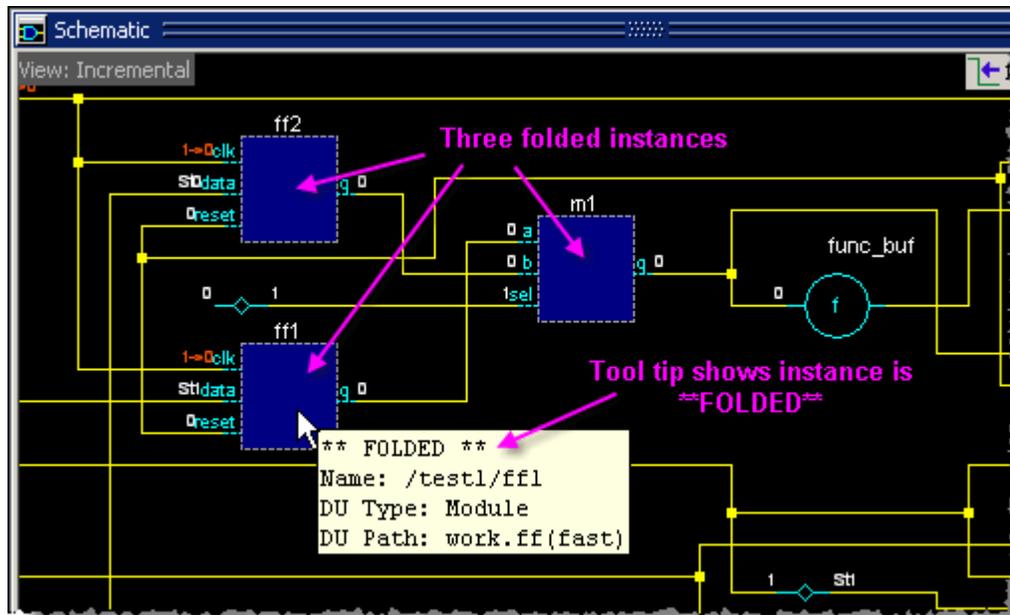
You can clear this highlighting using the **Schematic > Highlight > Remove** menu selection or by clicking the **Remove All Highlights** icon in the toolbar. If you click and hold the **Remove All Highlights** icon a dropdown menu appears, allowing you to remove the selected highlights



Folding and Unfolding Instances in the Incremental View

The Fold/Unfold feature reduces schematic clutter and allows you to focus on the surrounding logic of interest. Contents of complex instances are folded (hidden) inside a box to maximize screen space and improve the readability of the schematic. Folded instances are indicated by dark blue squares with dashed light gray borders. When you hover the mouse cursor over a folded instance, the tooltip (text box popup) will show that it is ****FOLDED**** (Figure 16-13).

Figure 16-13. Folded Instances



To unfold an instance and display its contents, do either of the following:

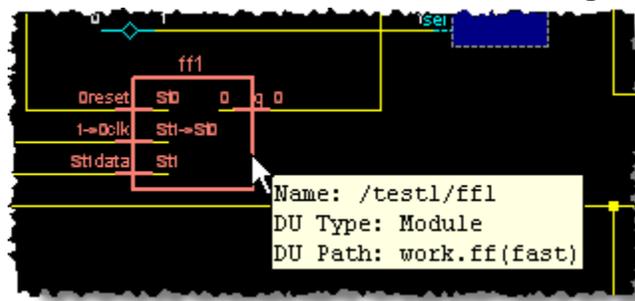
- Double-click the folded instance.
- Click the folded instance to select it, then right-click and select **Unfold Instance** from the popup menu.

To fold an instance, do either of the following:

- Ctrl + double-click the instance.
- Click the instance to select it, then right-click and select **Fold Instance** from the popup menu.

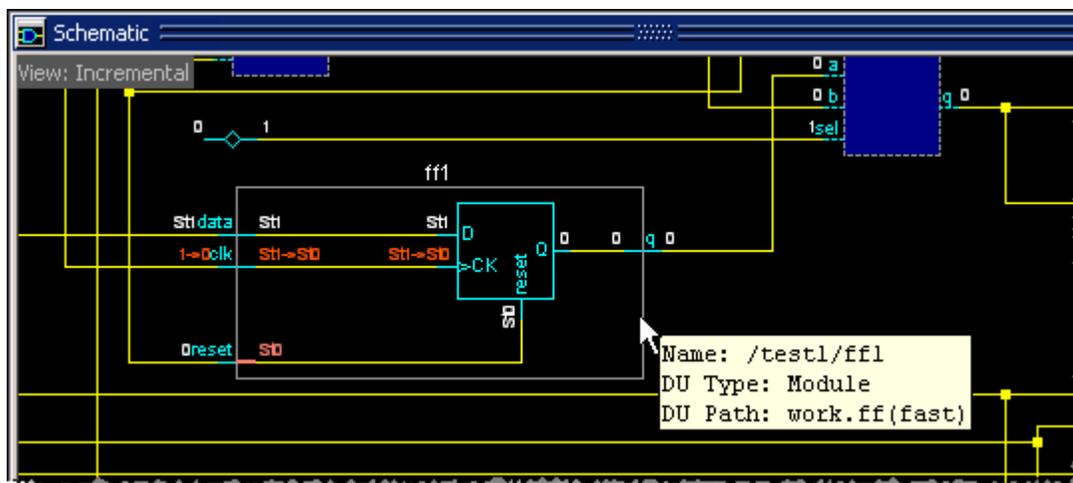
If you have not traced any signals into a folded instance (for example, if you simply dragged an instance into the incremental view) and then you unfold it, this action will only make the instance box transparent — you will not see the contents (Figure 16-14).

Figure 16-14. Unfolded Instance Not Showing Contents



However, you can double-click any input/output pin to trace the drivers/readers and cause the connected gates and internal instances to appear (Figure 16-15).

Figure 16-15. Unfolded Instance with All Contents Displayed



Exploring Designs with the Embedded Wave Viewer

Another way of exploring your design is to use the embedded wave viewer for the Incremental view. This viewer closely resembles, in appearance and operation, the Wave window (see [Waveform Analysis](#) for more information).

To open the wave viewer, use the **Schematic > Show Wave** menu selection when the Incremental view is active, or simply click the **Show Wave** toolbar button.



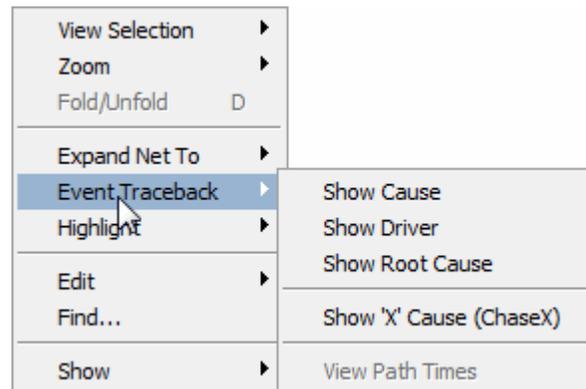
When wave viewer is first displayed, the visible zoom range is set to match that of the last active Wave window, if one exists. Additionally, the wave viewer's moveable cursor (Cursor 1) is automatically positioned to the location of the active cursor in the last active Wave window.

When you select an instance or process in the schematic, all signals attached to that instance or process are added to the wave viewer. In [Figure 16-16](#), the #ALWAYS#35 process is selected and the wave viewer displays 3 inputs, 1 output, and an inout bus. See [Tracing Events in the Incremental View](#) for another example of using the embedded wave viewer.

- trace an event to its immediate driving process – Show Driver
- trace an event to its root cause – Show Root Cause

These options are available when you right-click anywhere in the Incremental View and select Event Traceback from the popup menu (Figure 16-17).

Figure 16-17. Event Traceback Options



The event trace begins at the current “active time,” which is set a number of different ways:

- with the selected cursor in the Wave window
- with the selected cursor in the Schematic window’s embedded Wave viewer
- or with the Active Time label in the Source or Schematic windows.

Figure 16-18 shows the Active Time label in the upper right corner of the Incremental view. (This label is displayed by default. If you want to turn it off, select **Schematic > Preferences** to open the Incremental Schematic Options dialog and uncheck the “Active Time Label” box.)

Figure 16-18. Active Time Label in Incremental View

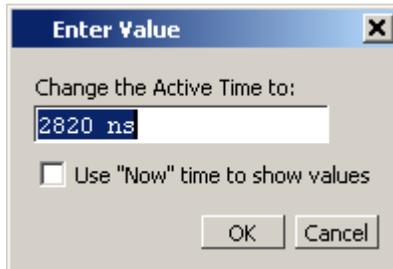


The Active Time label includes a minimize/maximize button that allows you to hide or display the label.

When a signal or net is selected, you can jump to the previous or next transition of that signal, with respect to the active time, by clicking the Find Previous/Next Transition buttons.

To change the Active Time, simply click the label and type in the time you want to examine in the Enter Value dialog box (Figure 16-19). The dialog includes a check box that allows you to switch to Now time (the time the simulation ended) or Active time (if “Now” is displayed in the Active Time label).

Figure 16-19. Enter Active Time Value

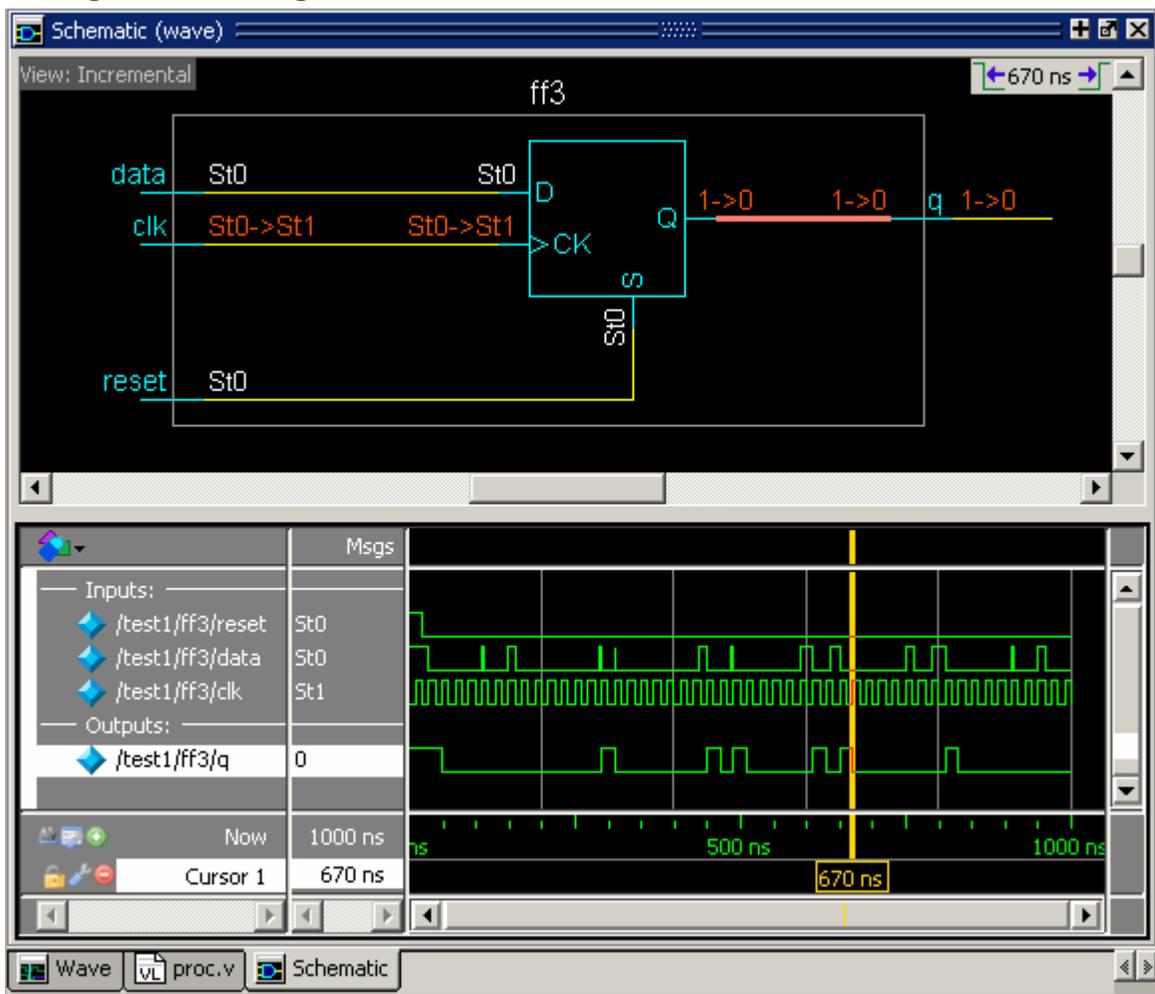


See [Active Time Label](#) for details.

The recommended work flow for initiating an event trace from the Incremental view is as follows:

1. Add a process or signal of interest into the Incremental view (if adding a signal, include its driving process).
2. Open the embedded wave viewer by clicking the Show Wave toolbar button.
3. In the Incremental view, click the process of interest so that all signals attached to the selected process will appear in the embedded wave viewer. 
4. In the wave viewer, select a signal and place a cursor at an event of interest. In [Figure 16-20](#), signal *q* of the fifo module *ff3* is selected and a cursor is placed on the transition at 670 ns.

Figure 16-20. Signals for Selected Process in Embedded Wave Viewer

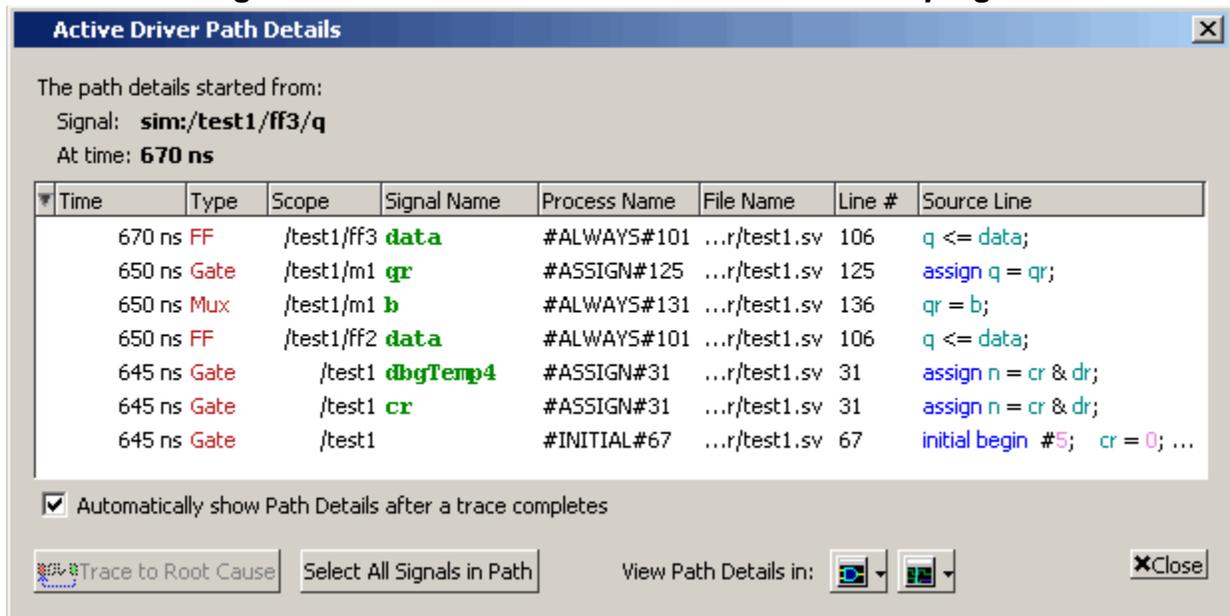


5. Right-click and select **Event Traceback**, then one of the three traceback options, from the popup menu.

A Source window opens with the cause of the event highlighted.

In addition, an Active Driver Path Details window opens to show information about the processes that caused the event (Figure 16-21).

Figure 16-21. Active Driver Path Details for the *q* Signal



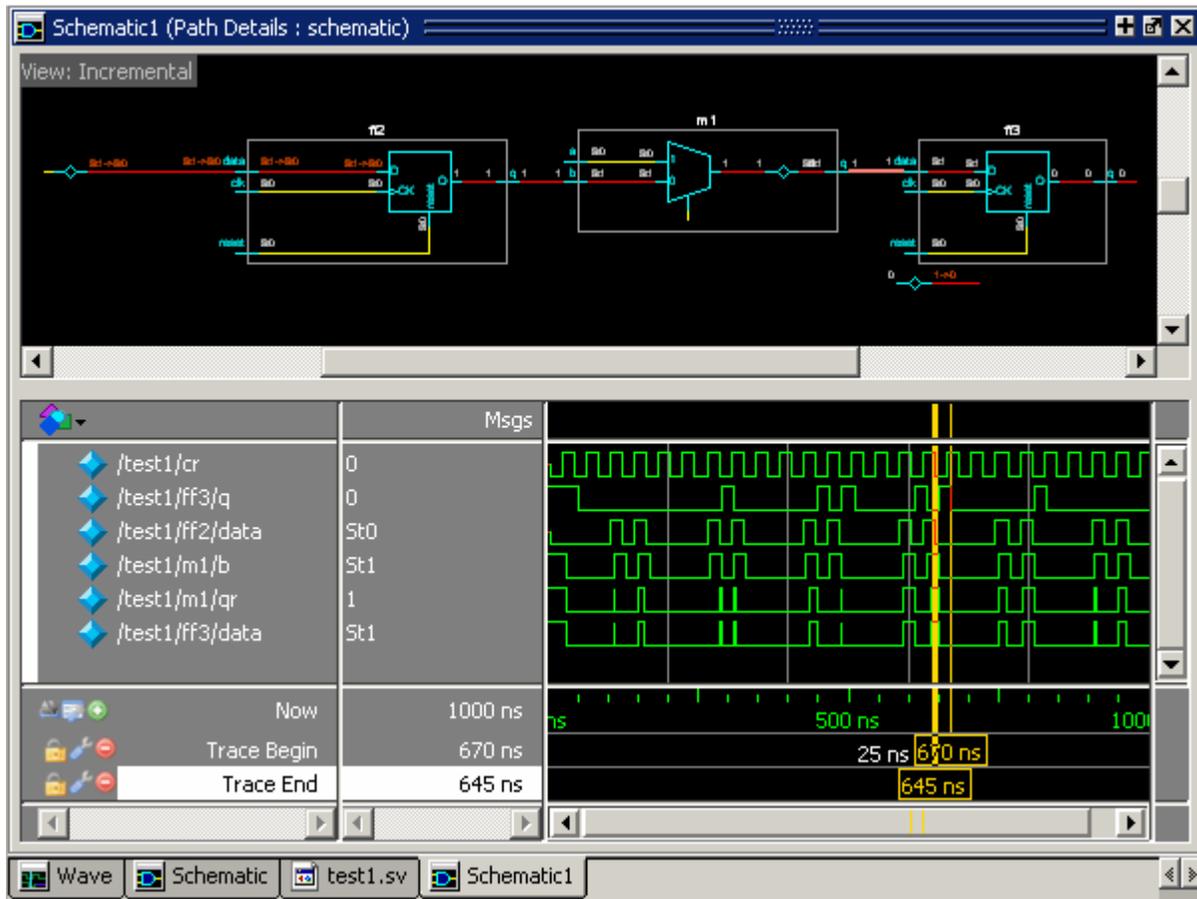
If you want to see the path details in the Schematic window, click the Schematic Window button at the bottom of the Active Driver Path Details Window.

Figure 16-22. Click Schematic Window Button to View Path Details



This will open a Schematic window with the title **Schematic (Path Details)**. In Figure 16-23, the *q* signal event at 670 ns is traced to its root cause. All signals in the path to the root cause are displayed in the wave viewer, and the path through the schematic is highlighted in red. The wave viewer also displays two new cursors, labeled **Trace Begin** and **Trace End** to designate where the event trace started and ended.

Figure 16-23. Path to Root Cause



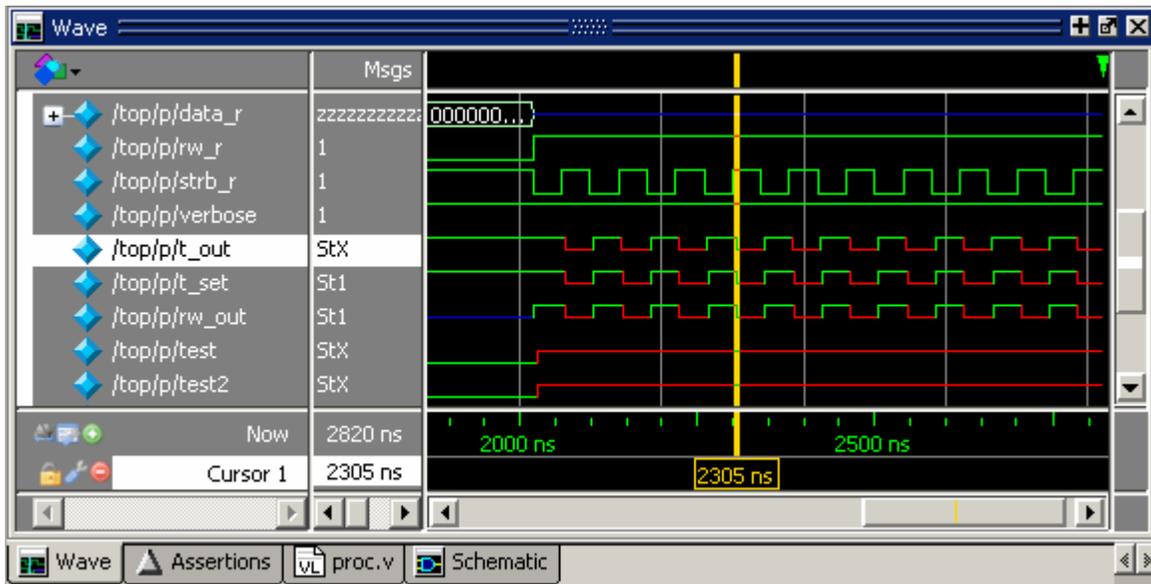
For more details about event tracing see [Using Causality Traceback](#).

Tracing the Source of an Unknown State (StX)

You can use the Causality Traceback feature of the Schematic window to trace an unknown state (StX) back to its source. (Refer to [Using Causality Traceback](#) chapter for additional details.)

Unknown values are indicated by red lines in the Wave window ([Figure 16-24](#)) and in the Wave Viewer pane of the Schematic window.

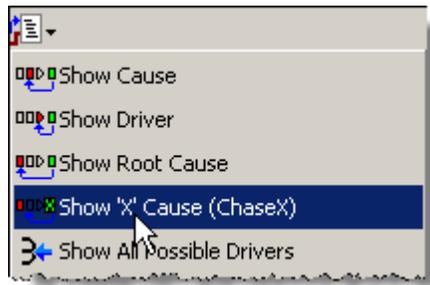
Figure 16-24. Unknown States Shown as Red Lines in Wave Window



The procedure for tracing to the source of an unknown state in the Schematic window is as follows:

1. Optimize your design with `+acc` (for debugging visibility) and with `-debugdb` (to save combinatorial and sequential logic events to the working library).
2. Load your design with `vsim -debugdb` to create a database (`vsim.dbg`) from the combinatorial and sequential logic event data.
3. Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /*** will log all signals in the design).
4. Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.
5. Put a Wave window cursor on the time at which the signal value is unknown (`StX`). In [Figure 16-24](#), Cursor 1 at time 2305 shows an unknown state on signal `t_out`.
6. Add the signal of interest to the Schematic window. You can drag and drop it from the Objects window, use the **Add Selected to Window** toolbar button, or the **Add > to Schematic > Selected Signals** menu selection,
7. In the Schematic window, make sure the signal of interest is selected.
8. Click and hold the Event Traceback menu button to open the menu ([Figure 16-25](#)), then select **Show 'X' Cause (ChaseX)**.

Figure 16-25. Event Traceback Menu



Finding Objects by Name in the Schematic Window

Select **Edit > Find** from the menu bar, or click the Find icon in the toolbar, to search for signal, net, or register names or an instance of a component. The Find toolbar opens at the bottom of the Schematic window (Figure 16-26).

Figure 16-26. Find Toolbar for Schematic Window



With the Find toolbar you can limit the search by type to instances or signal nets. You may do hierarchical searching from the design root (when you check “Search from Top”) or from the current context. The **Zoom to** selection zooms in to the item you enter in the **Find** field. The **Match case** selection enforces case-sensitive matching of your entry. And you can select **Exact (whole word)** to find an item that exactly matches the entry you type in the **Find** field.

The **Find All Matches in Current Schematic** button allows you to find and highlight all occurrences of the item in the **Find** field. If the **Zoom to** box is checked, the view changes so all selected items are viewable. If **Zoom to** is not checked, then no change is made to the zoom or scroll state.

Schematic Concepts

This section provides an introduction to the following important Schematic concepts:

- [Symbol Mapping](#)
- [Schematic Window Graphic Interface Reference](#)

Symbol Mapping

The Schematic window has built-in mappings for all Verilog primitive gates (for example, AND, OR, and so forth). You can also map VHDL entities and Verilog/SystemVerilog modules that represent a cell definition, or processes, to built-in gate symbols.

The mappings are saved in a file where the default filename is *schematic.bsm* (*.bsm* stands for "Built-in Symbol Map") The Schematic window looks in the current working directory and inside each library referenced by the design for the file. It will read all files found. You can also manually load a *.bsm* file by selecting **Schematic > Schematic Preferences > Load Built in Symbol Map**.

The *schematic.bsm* file contains comments and name pairs, one comment or name per line. Use the following Backus-Naur Format naming syntax:

Syntax

```
<bsm_line> ::= <comment> | <statement>
<comment> ::= "#" <text> <EOL>
<statement> ::= <name_pattern> <gate>
<name_pattern> ::= [<library_name> "."] <du_name> ["(" <specialization> ")"]
    ["," <process_name>]
<gate> ::= "BUF"|"BUFIF0"|"BUFIF1"|"INV"|"INVIF0"|"INVIF1"|"AND"|"NAND"|"
    "NOR"|"OR"|"XNOR"|"XOR"|"PULLDOWN"|"PULLUP"|"NMOS"|"PMOS"|"CM
    OS"|"TRAN"|"TRANIF0"|"TRANIF1"
```

For example:

```
org(only),p1 OR
andg(only),p1 AND
mylib,andg.p1 AND
norg,p2 NOR
```

Entities and modules representing cells are mapped the same way:

```
AND1 AND
# A 2-input and gate
AND2 AND
mylib,andg.p1 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

Note



Note that for primitive gate symbols, pin mapping is automatic. When you map a module/entity, it must be defined as a cell via ``celldefine` in Verilog.

The default filename is *schematic.bsm* (.bsm stands for "Built-in Symbol Map"). The Schematic window looks in the current working directory and inside each library referenced by the design for the file *schematic.bsm*. It will read all files found. You can also manually load a .bsm file by selecting **Schematic > Symbol Library > Load Built in Symbol Map**.

Note



The Schematic window will search for mapping files named *dataflow.bsm* first, then *schematic.bsm* in order to maintain backwards compatibility with designs simulated with older versions of ModelSim.

User-Defined Symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview™ widget Symlib format. The symbol definitions are saved in the *schematic.sym* file.

The formal BNF format for the *schematic.sym* file format is:

Syntax

```
<sym_line> ::= <comment> | <statement>
<comment> ::= "#" <text> <EOL>
<statement> ::= "symbol" <name_pattern> "*" "DEF" <definition>
<name_pattern> ::= [<library_name> "."] <du_name> [("(" <specialization> ")"]
                [", "<process_name>]
<gate> ::= "port" | "portBus" | "permute" | "attrdsp" | "pinattrdsp" | "arc" | "path" | "fpath"
          | "text" | "place" | "boxcolor"
```

Note



The port names in the definition must match the port names in the entity or module definition or mapping will not occur.

The Schematic window will search the current working directory, and inside each library referenced by the design, for the file *schematic.sym*. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \  
  port a in -loc -12 -15 0 -15 \  
  pinattrdsp @name -cl 2 -15 8 \  
  port b in -loc -12 15 0 15 \  
  pinattrdsp @name -cl 2 15 8 \  
  port cin in -loc 20 -40 20 -28 \  
  pinattrdsp @name -uc 19 -26 8 \  
  port cout out -loc 20 40 20 28 \  
  pinattrdsp @name -lc 19 26 8 \  
  port sum out -loc 63 0 51 0 \  
  pinattrdsp @name -cr 49 0 8 \  
  path 10 0 0 7 \  
  path 0 7 0 35 \  
  path 0 35 51 17 \  
  path 51 17 51 -17 \  
  path 51 -17 0 -35 \  
  path 0 -35 0 -7 \  
  path 0 -7 10 0
```

Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Schematic > Schematic Preferences > Create Symlib Index** (Schematic window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index. If you save the file as *schematic.sym* the Schematic window will automatically load the file. You can also manually load a *.sym* file by selecting **Schematic > Schematic Preferences > Load Symlib Library**.

Note

When you map a process to a gate symbol, it is best to name the process statement within your HDL source code, and use that name in the *.bsm* or *.sym* file. If you reference a default name that contains line numbers, you will need to edit the *.bsm* and/or *.sym* file every time you add or subtract lines in your HDL source.

Note

The Schematic window will search for mapping files named *dataflow.sym* first, then *schematic.sym* in order to maintain backwards compatibility with designs simulated with older versions of ModelSim.

Schematic Window Graphic Interface Reference

This section answers several common questions about using the Schematic window's graphic user interface:

- [What Can I View in the Schematic Window?](#)

- [How is the Schematic Window Linked to Other Windows?](#)
- [How Can I Print and Save the Display?](#)
- [How do I Configure Window Options?](#)
- [How do I Zoom and Pan the Display?](#)
- [How do I Use Keyboard Shortcuts?](#)

What Can I View in the Schematic Window?

The Schematic window displays:

- processes
- signals, nets, and registers
- interconnects

The window has built-in mappings for all Verilog primitive gates (i.e. AND, OR, and so forth). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See [Symbol Mapping](#) for details.

You cannot view SystemC objects in the Schematic window; however, you can view HDL regions from mixed designs that include SystemC.

How is the Schematic Window Linked to Other Windows?

The Schematic window is dynamically linked to other debugging windows and panes as described in [Table 16-3](#).

Table 16-3. Schematic Window Links to Other Windows and Panes

Window	Link
Main Window	select a signal or process in the Schematic window, and the structure tab updates if that object is in a different design unit
Processes Window	select a process in either window, and that process is highlighted in the other
Objects Window	select a design object in either window, and that object is highlighted in the other
Wave Window	trace through the design in the Schematic window, and the associated signals are added to the Wave window along with Trace Begin and Trace End cursors
	move a cursor in the Wave window, and the values update in the Schematic window

Table 16-3. Schematic Window Links to Other Windows and Panes (cont.)

Window	Link
Source Window	double-click an object in the Schematic window to open a Code Preview; use Event Traceback in the Schematic window to go directly to the source code for the cause of the event

How Can I Print and Save the Display?

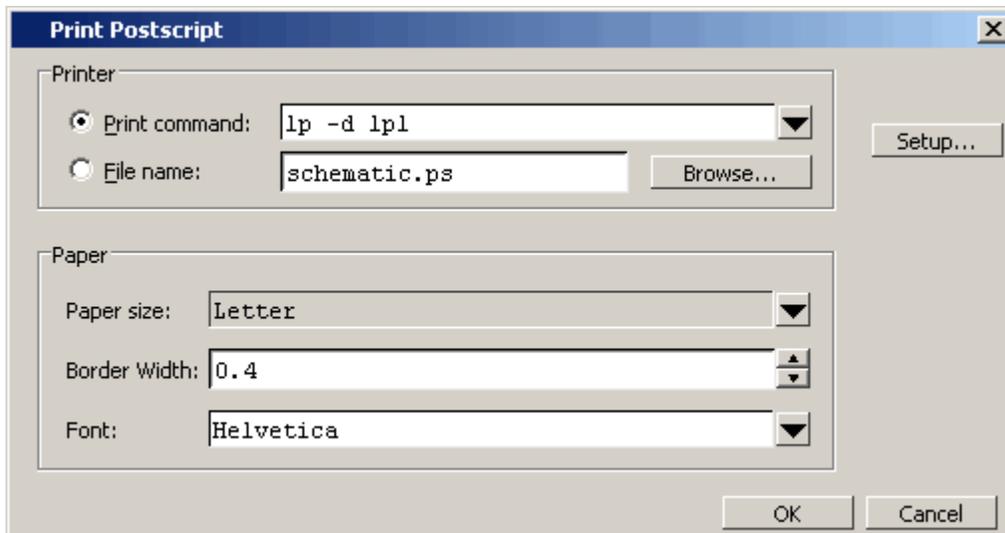
You can print the Schematic window display from a saved *.eps* file in UNIX, or by simple menu selections in Windows. The Schematic Page Setup dialog allows you to configure the display for printing.

You can also export the Schematic display as a bitmap image (*.bmp* file) by selecting **File > Export > Image** from the Main menu. This is useful for emailing or embedding into documents.

Saving a .eps File and Printing the Schematic Display from UNIX

With the Schematic window active, select **File > Print Postscript** to setup and print the Schematic display in UNIX, or save the waveform as an *.eps* file on any platform (Figure 16-27).

Figure 16-27. The Print Postscript Dialog



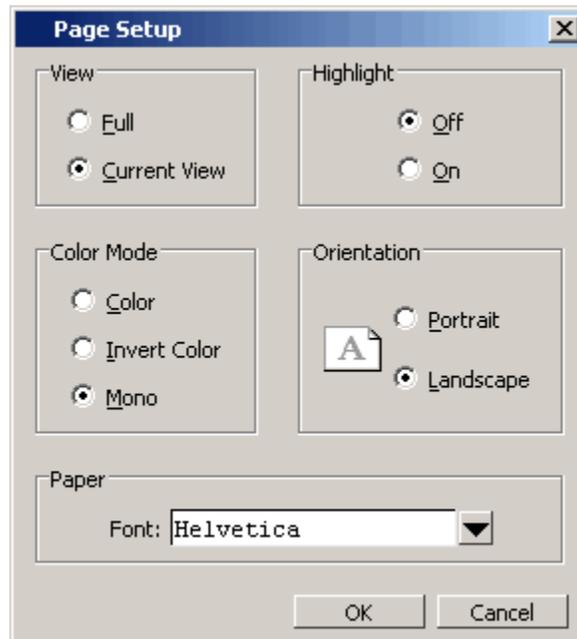
Printing from the Schematic Display on Windows Platforms

With the Schematic window active, select **File > Print** to print the Schematic display or to save the display to a file.

Configuring Page Setup

With the Schematic window active, select **File > Page setup** to open the Schematic Page Setup dialog (Figure 16-28). You can also open this dialog by clicking the Setup button in the Print Postscript dialog (Figure 16-27). This dialog allows you to configure page view, highlight, color mode, orientation, and paper options.

Figure 16-28. The Schematic Page Setup Dialog

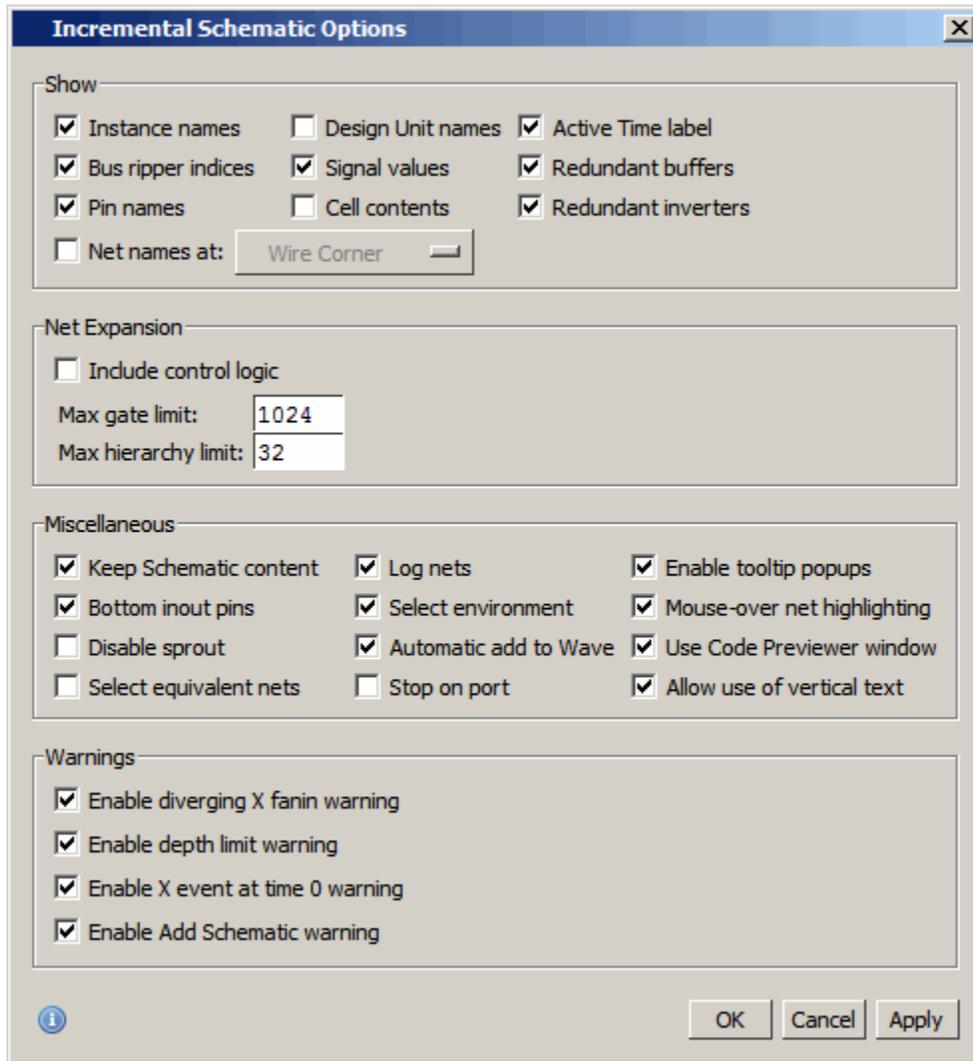


How do I Configure Window Options?

Different schematic display options are available depending on whether you are using the Incremental view or the Full view. The **Schematic > Preferences** menu selection allows you to configure several options that determine how the Incremental and Full views behave. Any changes made to the schematic display options are saved for future simulation and debugging sessions.

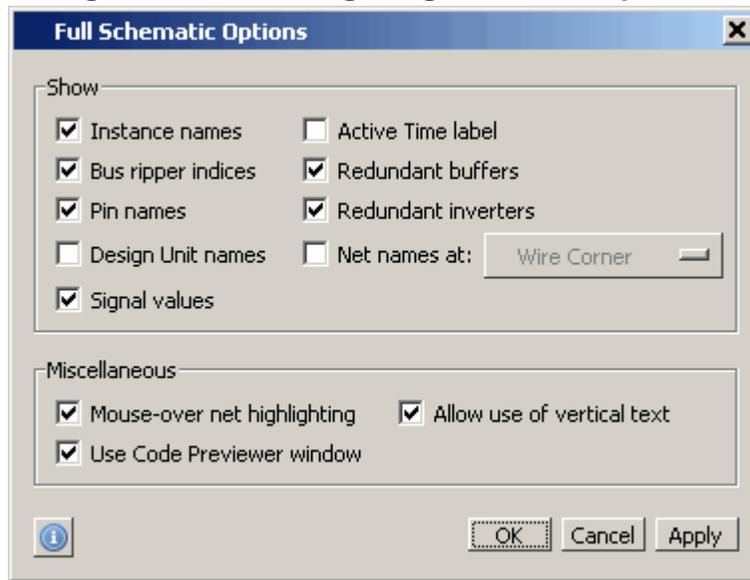
Incremental view options are shown in Figure 16-29. For a description of what each option does, click the “Show Help Text Pane” button (the ‘i’ button) in the bottom left corner of the dialog, or refer to [Controlling the Data Displayed in the Schematic Window](#).

Figure 16-29. Configuring Incremental View Options



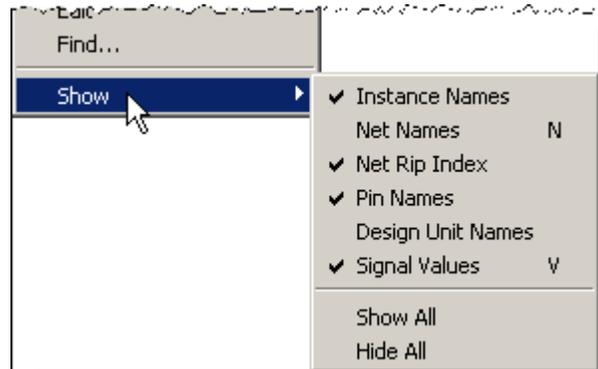
Full view options are shown in [Figure 16-30](#).

Figure 16-30. Configuring Full View Options



You may also right-click in either the Incremental or Full view to select Show from the popup menu, which gives you the display selections shown in [Figure 16-31](#).

Figure 16-31. Display Options in Right-Click Menu



Net Names and Signal Values can be toggled on and off with the N and V keys on your keyboard, respectively. (See [How do I Use Keyboard Shortcuts?](#)) By default, displayed signal values are for the current active time.

How do I Zoom and Pan the Display?

The Schematic window offers tools for zooming and panning the display.

These zoom buttons are available from the Zoom toolbar:



Zoom In

zoom in by a factor of two from the current view



Zoom Out

zoom out by a factor of two from current view



Zoom Full

zoom out to view the entire schematic

To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **Schematic > Zoom** and then use the left mouse button.

Zooming with the Mouse

Four zoom options are possible by pressing and holding the middle mouse button and dragging in different directions. If you change the mouse mode to “Zoom Mode” (Schematic > Mouse Mode > Zoom Mode), these click-and-drag options are done with the left mouse button:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected
- Up-Left: Zoom Full

Panning with the Mouse

You can pan with the mouse in two ways:

- enter Pan Mode by selecting **Schematic > Mouse Mode > Pan** and then drag with the left mouse button to move the design
- hold down the <Ctrl> key and drag with the middle mouse button to move the design.

How do I Use Keyboard Shortcuts?

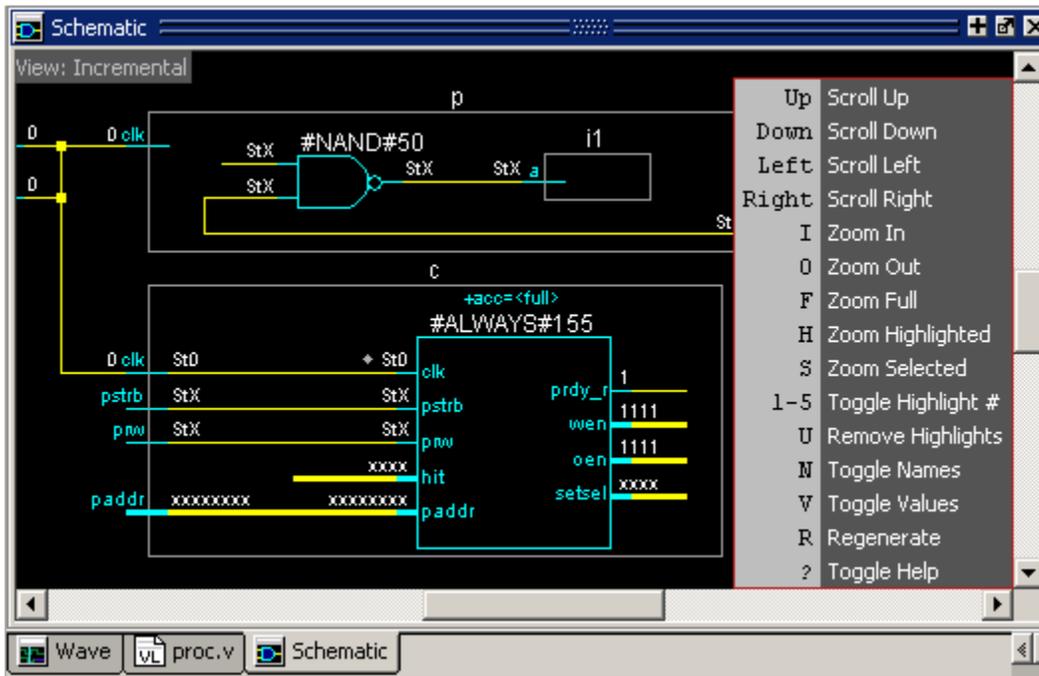
Table 16-4 provides a complete list of keyboard shortcuts you can use with the Schematic window.

Table 16-4. Keyboard Shortcuts

Key stroke	Action
Up arrow	Scroll up
Down arrow	Scroll down
Left Arrow	Scroll left
Right arrow	Scroll right
Ctrl + arrow key	Scroll by larger amount
Shift + arrow key	Scroll to edge of display
i	Zoom in
o	Zoom out
f	Zoom full
h	Zoom into highlighted selection
s	Zoom selected
1-5	Toggle highlight number
u	Remove all highlights
g	Toggle gray mode - remove all color except highlights
n	Toggle names on or off
v	Toggle signal values on or off
r	Regenerate display
?	Toggle keyboard shortcut table on or off

When the Schematic window is selected, you can display a list of keyboard shortcuts by pressing the '?' key on your keyboard.

Figure 16-32. Keyboard Shortcut Table in the Schematic Window



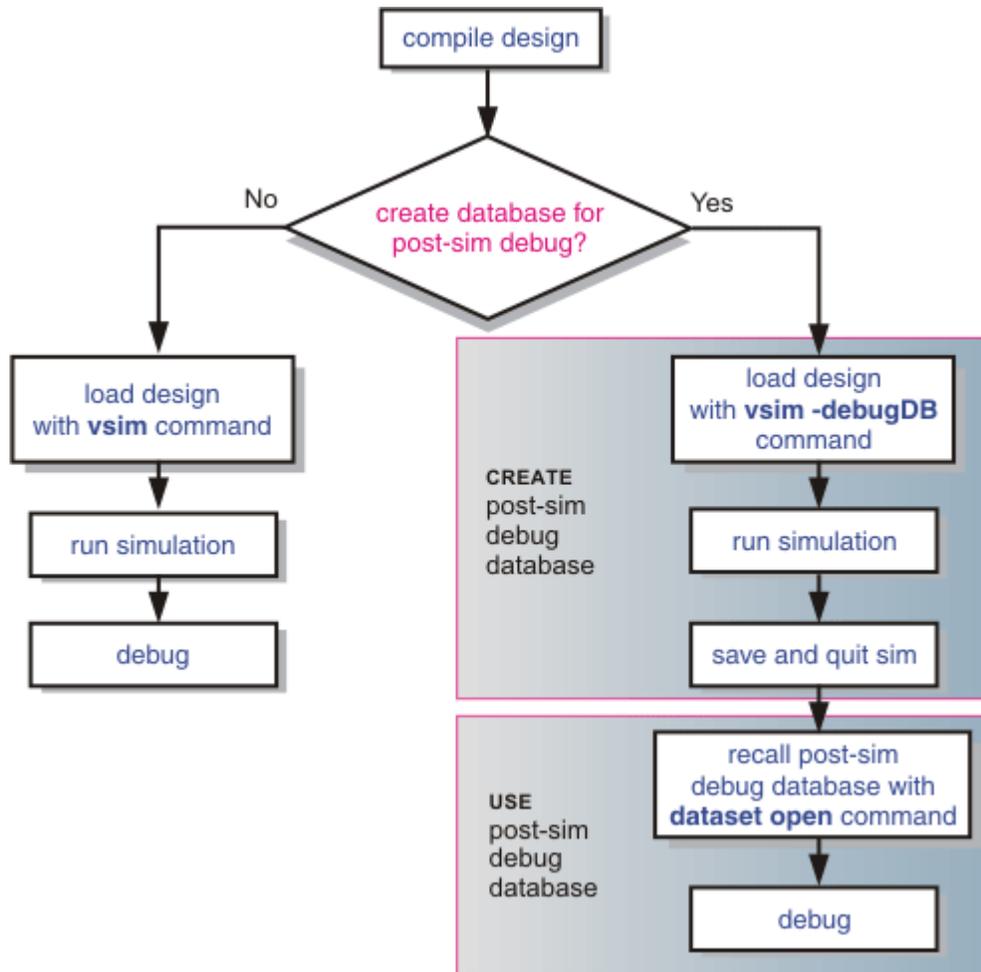
Toggle the list closed by pressing the ‘?’ key again or simply click the list.

Dataflow Usage Flow

The Dataflow window can be used to debug the design currently being simulated, or to perform post-simulation debugging of a design. ModelSim is able to create a database for use with post-simulation debugging. The database is created at design load time, immediately after elaboration, and used later.

Figure 17-2 illustrates the current and post-sim usage flows for Dataflow debugging.

Figure 17-2. Dataflow Debugging Usage Flow



Post-Simulation Debug Flow Details

The post-sim debug flow for Dataflow analysis is most commonly used when performing simulations of large designs in simulation farms, where simulation results are gathered over extended periods and saved for analysis at a later date. In general, the process consists of two steps: creating the database and then using it. The details of each step are as follows:

Create the Post-Sim Debug Database

1. Compile the design using the `vlog` and/or `vcom` commands.
2. Load the design with the following commands:

```
vsim -debugdb=<db_pathname.dbg> -wlf <db_pathname.wlf> <design_name>  
add log -r /*
```

Specify the post-simulation database file name with the `-debugdb=<db_pathname>` argument to the `vsim` command. If a database pathname is not specified, ModelSim creates a database with the file name `vsim.dbg` in the current working directory. This database contains dataflow connectivity information.

Specify the dataset that will contain the database with `-wlf <db_pathname>`. If a dataset name is not specified, the default name will be `vsim.wlf`.

The debug database and the dataset that contains it should have the same base name (`db_pathname`).

The `add log -r /*` command instructs ModelSim to save all signal values generated when the simulation is run.

3. Run the simulation.
4. Quit the simulation.

The `-debugdb=<db_pathname>` argument to the `vsim` command only needs to be used once after any structural changes to a design. After that, you can reuse the `vsim.dbg` file along with updated waveform files (`vsim.wlf`) to perform post simulation debug.

A structural change is any change that adds or removes nets or instances in the design, or changes any port/net associations. This also includes processes and primitive instances. Changes to behavioral code are not considered structural changes. ModelSim does not automatically detect structural changes. This must be done by the user.

Use the Post-Simulation Debug Database

1. Start ModelSim by typing `vsim` at a UNIX shell prompt; or double-click a ModelSim icon in Windows.
2. Select **File > Change Directory** and change to the directory where the post-simulation debug database resides.
3. Recall the post-simulation debug database with the following:

```
dataset open <db_pathname.wlf>
```

ModelSim opens the `.wlf` dataset and its associated debug database (`.dbg` file with the same basename), if it can be found. If ModelSim cannot find `db_pathname.dbg`, it will attempt to open `vsim.dbg`.

Common Tasks for Dataflow Debugging

Common tasks for current and post-simulation Dataflow debugging include:

- [Adding Objects to the Dataflow Window](#)
- [Exploring the Connectivity of the Design](#)
- [Exploring Designs with the Embedded Wave Viewer](#)
- [Tracing Events](#)
- [Tracing the Source of an Unknown State \(StX\)](#)
- [Finding Objects by Name in the Dataflow Window](#)

Adding Objects to the Dataflow Window

You can use any of the following methods to add objects to the Dataflow window:

- Drag and drop objects from other windows.
- Use the **Add > To Dataflow** menu options.
- Select the objects you want placed in the Dataflow Window, then click-and-hold the [Add Selected to Window Button](#) in the **Standard** toolbar and select **Add to Dataflow**.
- Use the [add dataflow](#) command.

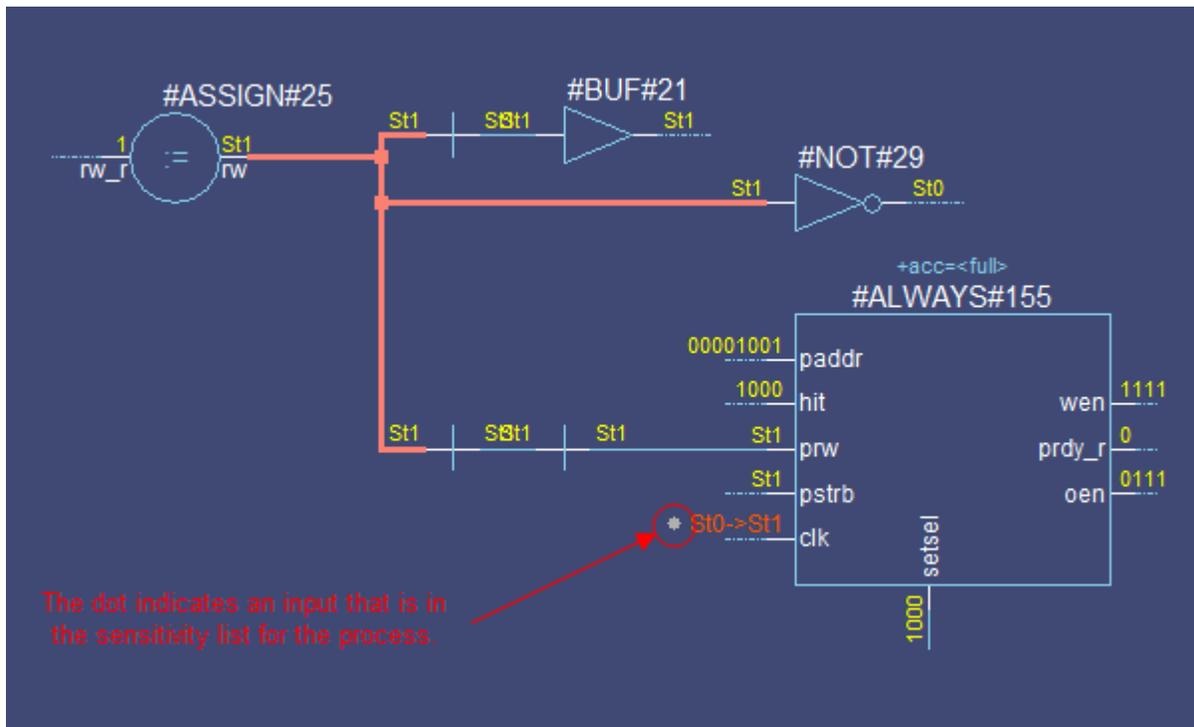
The **Add > To Dataflow** menu offers four commands that will add objects to the window:

- **View region** — clear the window and display all signals from the current region
- **Add region** — display all signals from the current region without first clearing the window
- **View all nets** — clear the window and display all signals from the entire design
- **Add ports** — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added objects. You can view readers as well by right-clicking a selected object, then selecting **Expand net to readers** from the right-click popup menu.

The Dataflow window provides automatic indication of input signals that are included in the process sensitivity list. In [Figure 17-3](#), the dot next to the state of the input *clk* signal for the #ALWAYS#155 process. This dot indicates that the *clk* signal is in the sensitivity list for the process and will trigger process execution. Inputs without dots are read by the process but will not trigger process execution, and are not in the sensitivity list (will not change the output by themselves).

Figure 17-3. Dot Indicates Input in Process Sensitivity Lis



Exploring the Connectivity of the Design

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

You can expand the view of your design using menu commands or your mouse. To expand with the mouse, simply double click a signal, register, or process. Depending on the specific object you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or drop down menu commands described in [Table 17-1](#).

Table 17-1. Icon and Menu Selections for Exploring Design Connectivity

	<p>Expand net to all drivers display driver(s) of the selected signal, net, or register</p>	<p>Right-click in the Dataflow window > Expand Net to Drivers</p>
	<p>Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register</p>	<p>Right-click in the Dataflow window > Expand Net</p>

Table 17-1. Icon and Menu Selections for Exploring Design Connectivity (cont.)

	Expand net to all readers display reader(s) of the selected signal, net, or register	Right-click in the Dataflow window > Expand Net to Readers
---	--	--

As you expand the view, the layout of the design may adjust to show the connectivity more clearly. For example, the location of an input signal may shift from the bottom to the top of a process.

Controlling the Display of Readers and Nets

Some nets (such as a clock) in a design can have many readers. This can cause the display to draw numerous processes that you may not want to see when expanding the selected signal, net, or register. By default, nets with undisplayed readers or drivers are represented by a dashed line. If all the readers and drivers for a net are shown, the net will appear as a solid line. To draw the undisplayed readers or drivers, double-click on the dashed line.

Limiting the Display of Readers

The Dataflow Window limits the number of readers that are added to the display when you click the Expand Net to Readers button. By default, the limit is 10 readers, but you can change this limit with the "sproutlimit" Dataflow preference as follows:

1. Open the Preferences dialog box by selecting **Tools > Edit Preferences**.
2. Click the By Name tab.
3. Click the '+' sign next to "Dataflow" to see the list of Dataflow preference items.
4. Select "sproutlimit" from the list and click the **Change Value** button.
5. Change the value and click the OK button to close the Change Dataflow Preference Value dialog box.
6. Click OK to close the Preferences dialog box and apply the changes.

The sprout limit is designed to improve performance with high fanout nets such as clock signals. Each subsequent click of the Expand Net to Readers button adds the sprout limit of readers until all readers are displayed.

Note  This limit does not affect the display of drivers.

Limiting the Display of Readers and Drivers

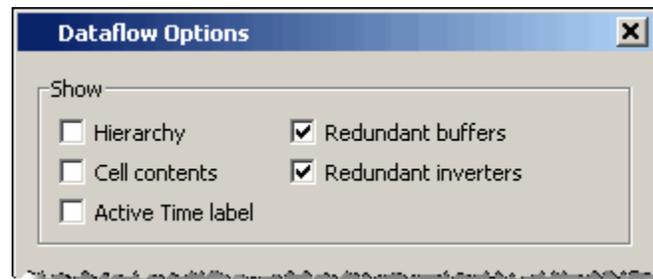
To restrict the expansion of readers and/or drivers to the hierarchical boundary of a selected signal select **Dataflow > Dataflow Options** to open the **Dataflow Options** dialog box then check **Stop on port** in the **Miscellaneous** field.

Controlling the Display of Redundant Buffers and Inverters

The Dataflow window automatically traces a signal through buffers and inverters. This can cause chains of redundant buffers or inverters to be displayed in the Dataflow window. You can collapse these chains of buffers or inverters to make the design displayed in the Dataflow window more compact.

To change the display of redundant buffers and inverters: select **Dataflow > Dataflow Preferences > Options** to open the Dataflow Options dialog. The default setting is to display both redundant buffers and redundant inverters. (Figure 17-4)

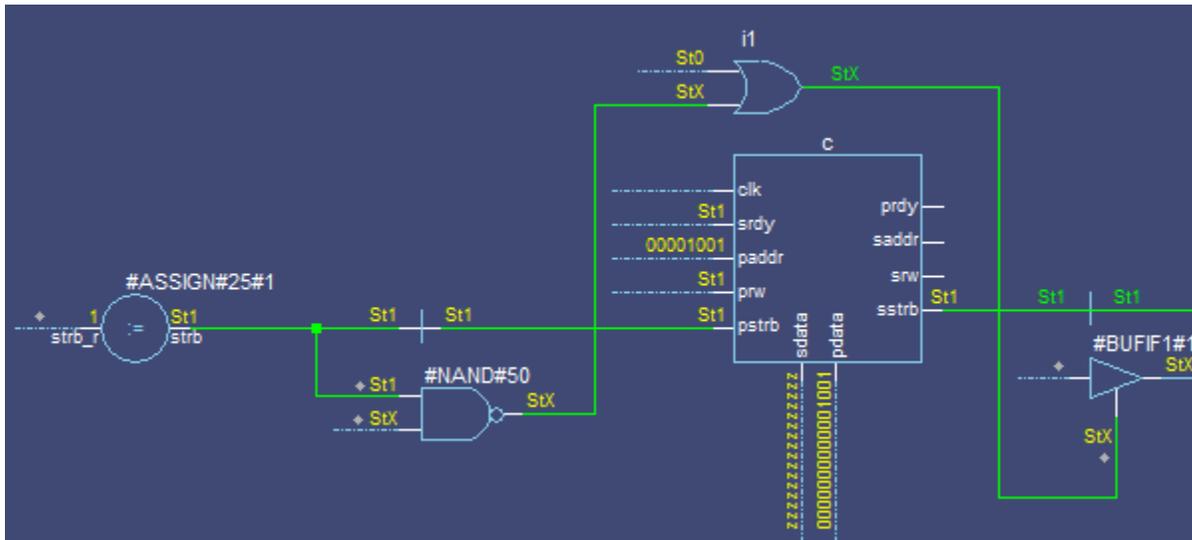
Figure 17-4. Controlling Display of Redundant Buffers and Inverters



Tracking Your Path Through the Design

You can quickly traverse through many components in your design. To help mark your path, the objects that you have expanded are highlighted in green.

Figure 17-5. Green Highlighting Shows Your Path Through the Design

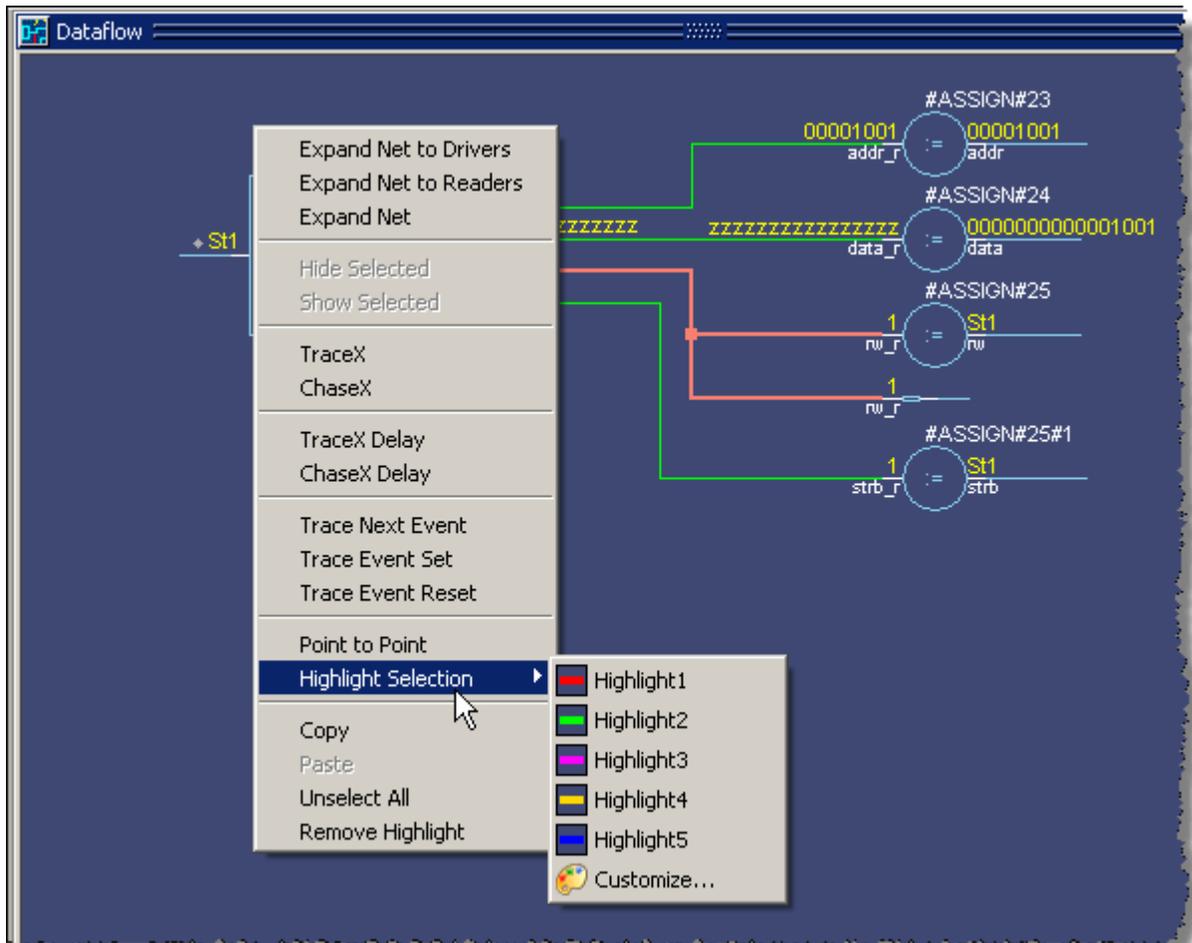


You can clear this highlighting using the **Dataflow > Remove Highlight** menu selection or by clicking the **Remove All Highlights** icon in the toolbar. If you click and hold the **Remove All Highlights** icon a dropdown menu appears, allowing you to remove only selected highlights.



You can also highlight the selected trace with any color of your choice by right-clicking Dataflow window and selecting Highlight Selection from the popup menu (Figure 17-6).

Figure 17-6. Highlight Selected Trace with Custom Color



You can then choose from one of five pre-defined colors, or **Customize** to choose from the palette in the Preferences dialog box.

Exploring Designs with the Embedded Wave Viewer

Another way of exploring your design is to use the Dataflow window's embedded wave viewer. This viewer closely resembles, in appearance and operation, the stand-alone Wave window (see [Waveform Analysis](#) for more information).

The wave viewer is opened using the **Dataflow > Show Wave** menu selection or by clicking the **Show Wave** icon.

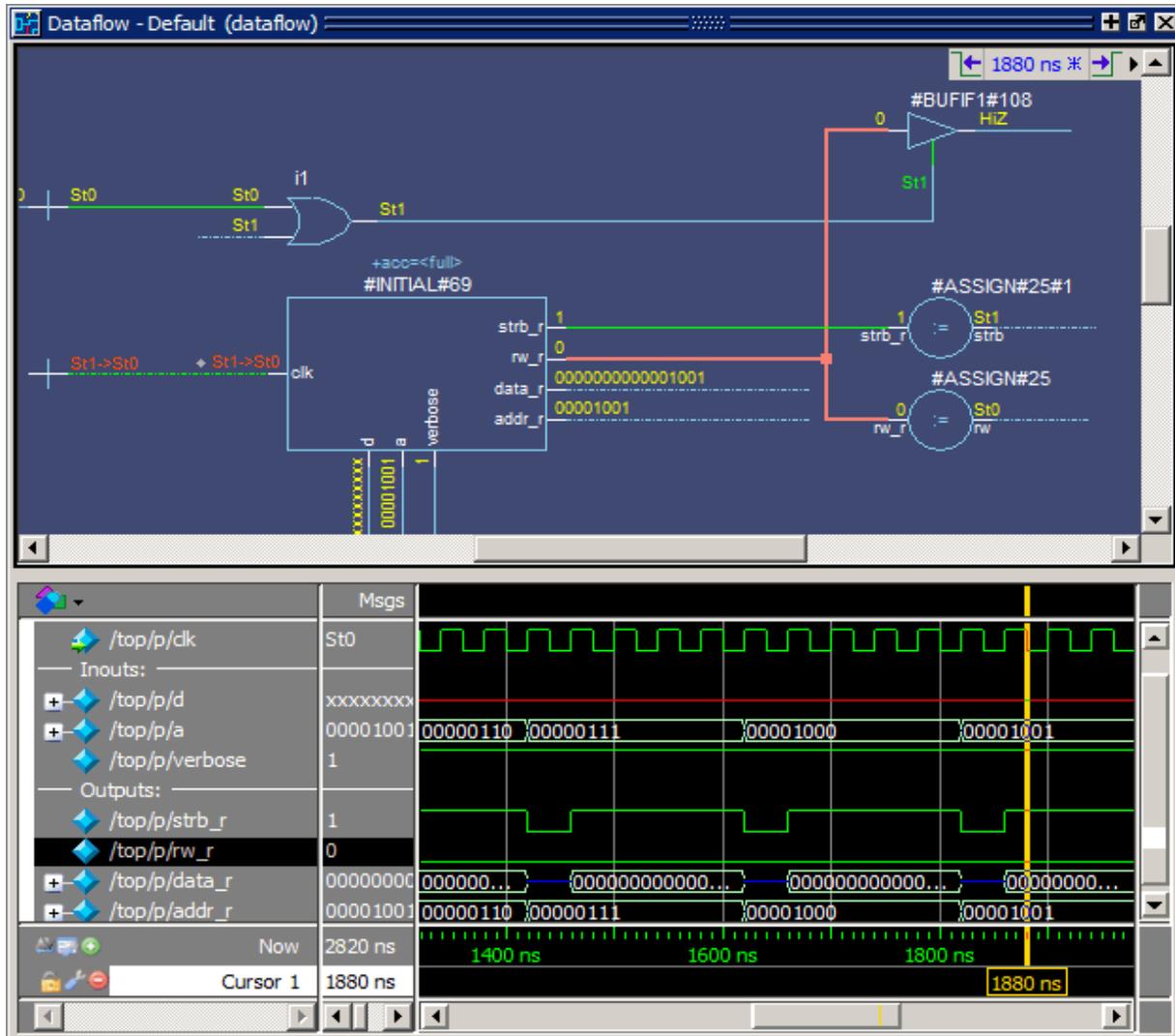


When wave viewer is first displayed, the visible zoom range is set to match that of the last active Wave window, if one exists. Additionally, the wave viewer's moveable cursor (Cursor 1) is automatically positioned to the location of the active cursor in the last active Wave window. The Active Time label in the upper right of the Dataflow window automatically displays the

time of the currently active cursor. Refer to [Active Time Label](#) for information about working with the Active Time label.

One common scenario is to place signals in the wave viewer and the Dataflow panes, run the design for some amount of time, and then use time cursors to investigate value changes. In other words, as you place and move cursors in the wave viewer pane (see [Measuring Time with Cursors in the Wave Window](#) for details), the signal values update in the Dataflow window.

Figure 17-7. Wave Viewer Displays Inputs and Outputs of Selected Process



Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See [Tracing Events](#) for another example of using the embedded wave viewer.

Tracing Events

You can use the Dataflow window to trace an event to the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see [Exploring Designs with the Embedded Wave Viewer](#) for more details). First, you identify an output of interest in the dataflow pane, then use time cursors in the wave viewer pane to identify events that contribute to the output.

The process for tracing events is as follows:

1. Log all signals before starting the simulation (**add log -r /***).
2. After running a simulation for some period of time, open the Dataflow window and the wave viewer pane.
3. Add a process or signal of interest into the dataflow pane (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.
4. Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.

5. Right-click and select **Trace Next Event**. 

A second cursor is added at the most recent input event.

6. Keep selecting **Trace Next Event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave viewer pane.

7. Right-click and select **Trace Event Set**. 

The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.

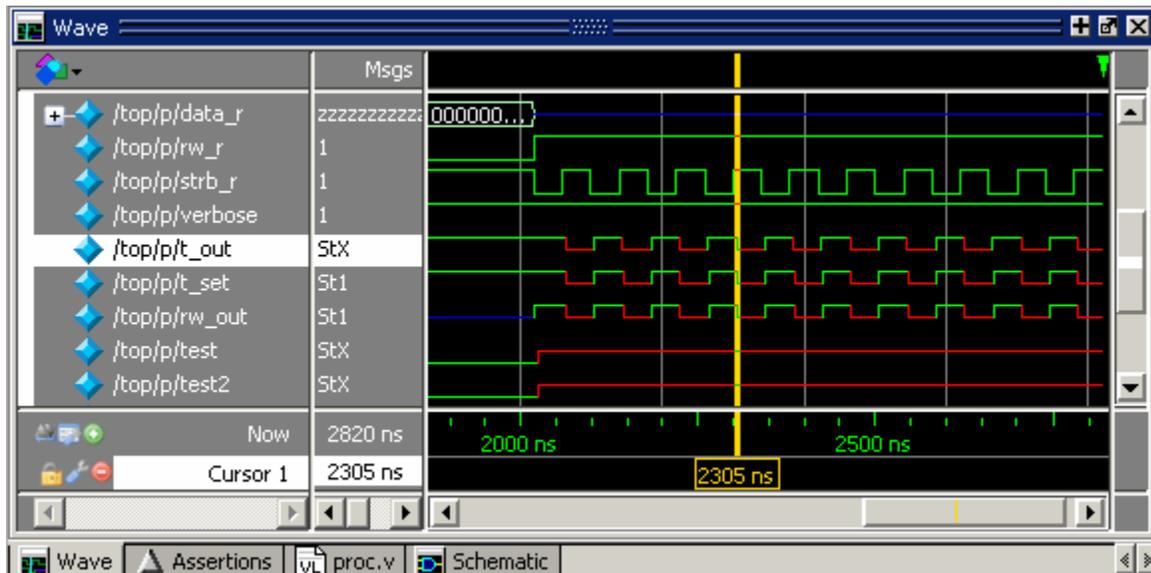
8. To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, right-click and select **Trace Event Reset**.

Tracing the Source of an Unknown State (StX)

Another useful Dataflow window debugging tool is the ability to trace an unknown state (StX) back to its source. Unknown values are indicated by red lines in the Wave window ([Figure 17-8](#)) and in the wave viewer pane of the Dataflow window.

Figure 17-8. Unknown States Shown as Red Lines in Wave Window



The procedure for tracing to the source of an unknown state in the Dataflow window is as follows:

1. Load your design.
2. Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /*** will log all signals in the design).
3. Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.
4. Put a Wave window cursor on the time at which the signal value is unknown (StX). In [Figure 17-8](#), Cursor 1 at time 2305 shows an unknown state on signal *t_out*.
5. Add the signal of interest to the Dataflow window by doing one of the following:
 - o Select the signal in the Wave Window, select **Add Selected to Window** in the Standard toolbar > **Add to Dataflow**.
 - o right-click the signal in the Objects window and select **Add > To Dataflow > Selected Signals** from the popup menu,
 - o select the signal in the Objects window and select **Add > To Dataflow > Selected Items** from the menu bar.
6. In the Dataflow window, make sure the signal of interest is selected.
7. Trace to the source of the unknown by doing one of the following:
 - o If the Dataflow window is docked, make one of the following menu selections:
Tools > Trace > TraceX,
Tools > Trace > TraceX Delay,

Tools > Trace > ChaseX, or
Tools > Trace > ChaseX Delay.

- If the Dataflow window is undocked, make one of the following menu selections:
Trace > TraceX,
Trace > TraceX Delay,
Trace > ChaseX, or
Trace > ChaseX Delay.

These commands behave as follows:

- **TraceX / TraceX Delay**— **TraceX** steps back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with back annotated delays.
- **ChaseX / ChaseX Delay** — **ChaseX** jumps through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in time to the point where the output value transitions to X. **ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with back annotated delays.

Finding Objects by Name in the Dataflow Window

Select **Edit > Find** from the menu bar, or click the Find icon in the toolbar, to search for signal, net, or register names or an instance of a component. This opens the search toolbar at the bottom of the Dataflow window.



With the search toolbar you can limit the search by type to instances or signals. You select **Exact** to find an item that exactly matches the entry you've typed in the **Find** field. The **Match case** selection will enforce case-sensitive matching of your entry. And the **Zoom to** selection will zoom in to the item in the **Find** field.

The **Find All** button allows you to find and highlight all occurrences of the item in the **Find** field. If **Zoom to** is checked, the view will change such that all selected items are viewable. If **Zoom to** is not selected, then no change is made to zoom or scroll state.

Automatically Tracing All Paths Between Two Nets

This behavior is referred to as point-to-point tracing. It allows you to visualize all paths connecting two different nets in your dataflow.

Prerequisites

- This feature is available during a live simulation, not when performing post-simulation debugging.

Procedure

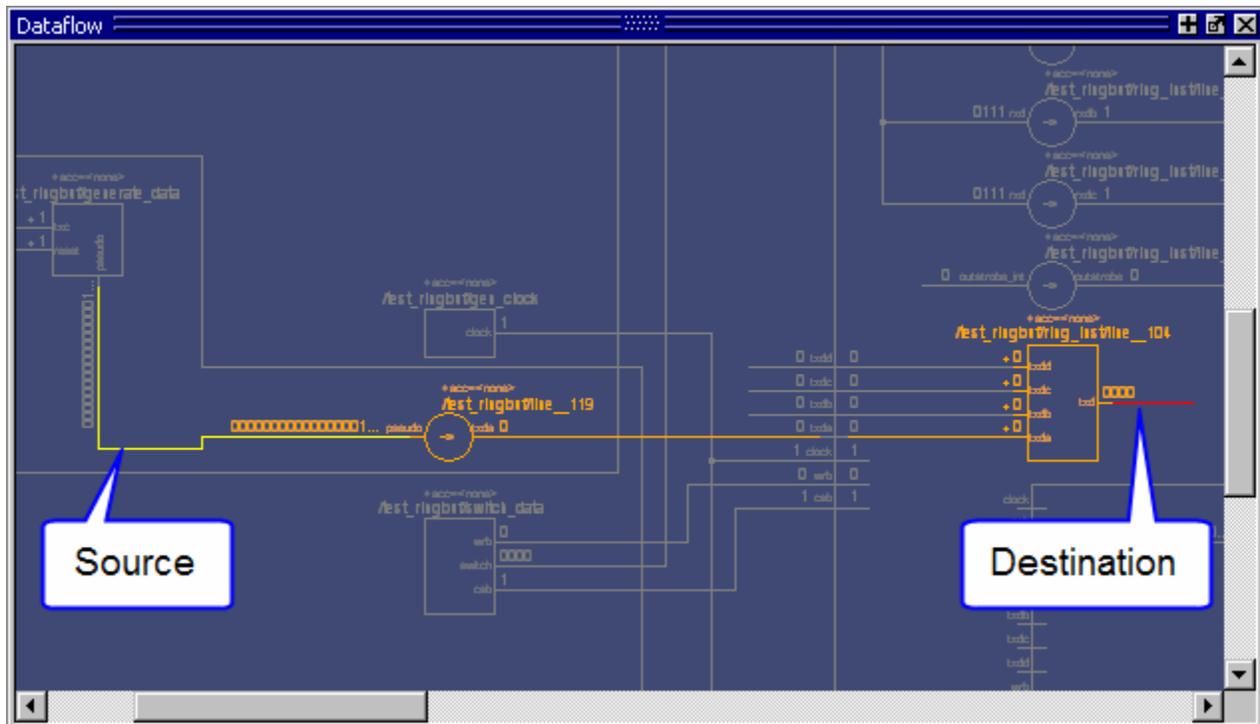
1. Select Source — Click on the net to be your source
2. Select Destination — Shift-click on the net to be your destination
3. Run point-to-point tracing — Right-click in the Dataflow window and select **Point to Point**.

Results

After beginning the point-to-point tracing, the Dataflow window highlights your design as shown in Figure 17-9:

1. All objects become gray
2. The source net becomes yellow
3. The destination net becomes red
4. All intermediate processes and nets become orange.

Figure 17-9. Dataflow: Point-to-Point Tracing



Related Tasks

- Change the limit of highlighted processes — There is a limit of 400 processes that will be highlighted.
 - a. **Tools > Edit Preferences**

- b. By Name tab
- c. **Dataflow > p2plimit** option
- Remove the point-to-point tracing
 - a. Right-click in the Dataflow window
 - b. Erase Highlights
- Perform point-to-point tracing from the command line
 - a. Determine the names of the nets
 - b. Use the `add dataflow` command with the `-connect` switch, for example:

```
add data -connect /test_ringbuf/pseudo /test_ringbuf/ring_inst/txd
```

where `/test_ringbuf/pseudo` is the source net and `/test_ringbuf/ring_inst/txd` is the destination net.

Dataflow Concepts

This section provides an introduction to the following important Dataflow concepts:

- [Symbol Mapping](#)
- [Current vs. Post-Simulation Command Output](#)

Symbol Mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (for example, AND, OR, and so forth). You can also map VHDL entities and Verilog/SystemVerilog modules that represent a cell definition, or processes, to built-in gate symbols.

The mappings are saved in a file where the default filename is `dataflow.bsm` (`.bsm` stands for "Built-in Symbol Map"). The Dataflow window looks in the current working directory and inside each library referenced by the design for the file. It will read all files found. You can also manually load a `.bsm` file by selecting **Dataflow > Dataflow Preferences > Load Built in Symbol Map**.

The `dataflow.bsm` file contains comments and name pairs, one comment or name per line. Use the following Backus-Naur Format naming syntax:

Syntax

```
<bsm_line> ::= <comment> | <statement>  
  <comment> ::= "#" <text> <EOL>  
  <statement> ::= <name_pattern> <gate>
```

```
<name_pattern> ::= [<library_name> "."] <du_name> ["(" <specialization> ")"]  
    [", "<process_name>]  
  
<gate> ::= "BUF"|"BUFIF0"|"BUFIF1"|"INV"|"INVIF0"|"INVIF1"|"AND"|"NAND" |  
    "NOR"|"OR"|"XNOR"|"XOR"|"PULLDOWN"|"PULLUP"|"NMOS"|"PMOS"|"CM  
    OS"|"TRAN"|"TRANIF0"|"TRANIF1"
```

For example:

```
org(only),p1 OR  
andg(only),p1 AND  
mylib,andg.p1 AND  
norg,p2 NOR
```

Entities and modules representing cells are mapped the same way:

```
AND1 AND  
# A 2-input and gate  
AND2 AND  
mylib,andg.p1 AND  
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

User-Defined Symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview™ widget Symlib format. The symbol definitions are saved in the *dataflow.sym* file.

The formal BNF format for the *dataflow.sym* file format is:

Syntax

```
<sym_line> ::= <comment> | <statement>  
  
<comment> ::= "#" <text> <EOL>  
  
<statement> ::= "symbol" <name_pattern> "*" "DEF" <definition>  
  
<name_pattern> ::= [<library_name> "."] <du_name> ["(" <specialization> ")"]  
    [", "<process_name>]  
  
<gate> ::= "port" | "portBus" | "permute" | "attrdsp" | "pinattrdsp" | "arc" | "path" | "fpath"  
    | "text" | "place" | "boxcolor"
```

Note



The port names in the definition must match the port names in the entity or module definition or mapping will not occur.

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file *dataflow.sym*. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \  
  port a in -loc -12 -15 0 -15 \  
  pinattrdsp @name -cl 2 -15 8 \  
  port b in -loc -12 15 0 15 \  
  pinattrdsp @name -cl 2 15 8 \  
  port cin in -loc 20 -40 20 -28 \  
  pinattrdsp @name -uc 19 -26 8 \  
  port cout out -loc 20 40 20 28 \  
  pinattrdsp @name -lc 19 26 8 \  
  port sum out -loc 63 0 51 0 \  
  pinattrdsp @name -cr 49 0 8 \  
  path 10 0 0 7 \  
  path 0 7 0 35 \  
  path 0 35 51 17 \  
  path 51 17 51 -17 \  
  path 51 -17 0 -35 \  
  path 0 -35 0 -7 \  
  path 0 -7 10 0
```

Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Dataflow > Dataflow Preferences > Create Symlib Index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index. If you save the file as *dataflow.sym* the Dataflow window will automatically load the file. You can also manually load a *.sym* file by selecting **Dataflow > Dataflow Preferences > Load Symlib Library**.

Note



When you map a process to a gate symbol, it is best to name the process statement within your HDL source code, and use that name in the *.bsm* or *.sym* file. If you reference a default name that contains line numbers, you will need to edit the *.bsm* and/or *.sym* file every time you add or subtract lines in your HDL source.

Current vs. Post-Simulation Command Output

ModelSim includes [drivers](#) and [readers](#) commands that can be invoked from the command line to provide information about signals displayed in the Dataflow window. In live simulation mode, the [drivers](#) and [readers](#) commands will provide both topological information and signal values. In post-simulation mode, however, these commands will provide only topological information. Driver and reader values are not saved in the post-simulation debug database.

Dataflow Window Graphic Interface Reference

This section answers several common questions about using the Dataflow window's graphic user interface:

- [What Can I View in the Dataflow Window?](#)
- [How is the Dataflow Window Linked to Other Windows?](#)
- [How Can I Print and Save the Display?](#)
- [How Do I Configure Window Options?](#)

What Can I View in the Dataflow Window?

The Dataflow window displays:

- processes
- signals, nets, and registers
- interconnects

The window has built-in mappings for all Verilog primitive gates (for example, AND, OR, and so forth). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See [Symbol Mapping](#) for details.

You cannot view SystemC objects in the Dataflow window; however, you can view HDL regions from mixed designs that include SystemC.

How is the Dataflow Window Linked to Other Windows?

The Dataflow window is dynamically linked to other debugging windows and panes as described in [Table 17-2](#).

Table 17-2. Dataflow Window Links to Other Windows and Panes

Window	Link
Main Window	select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit
Processes Window	select a process in either window, and that process is highlighted in the other
Objects Window	select a design object in either window, and that object is highlighted in the other

Table 17-2. Dataflow Window Links to Other Windows and Panes (cont.)

Window	Link
Wave Window	trace through the design in the Dataflow window, and the associated signals are added to the Wave window
	move a cursor in the Wave window, and the values update in the Dataflow window
Source Window	select an object in the Dataflow window, and the Source window updates if that object is in a different source file

How Can I Print and Save the Display?

You can print the Dataflow window display from a saved *.eps* file in UNIX, or by simple menu selections in Windows. The Page Setup dialog allows you to configure the display for printing.

Saving a *.eps* File and Printing the Dataflow Display from UNIX

With the Dataflow window active, select **File > Print Postscript** to setup and print the Dataflow display in UNIX, or save the waveform as an *.eps* file on any platform (Figure 17-10).

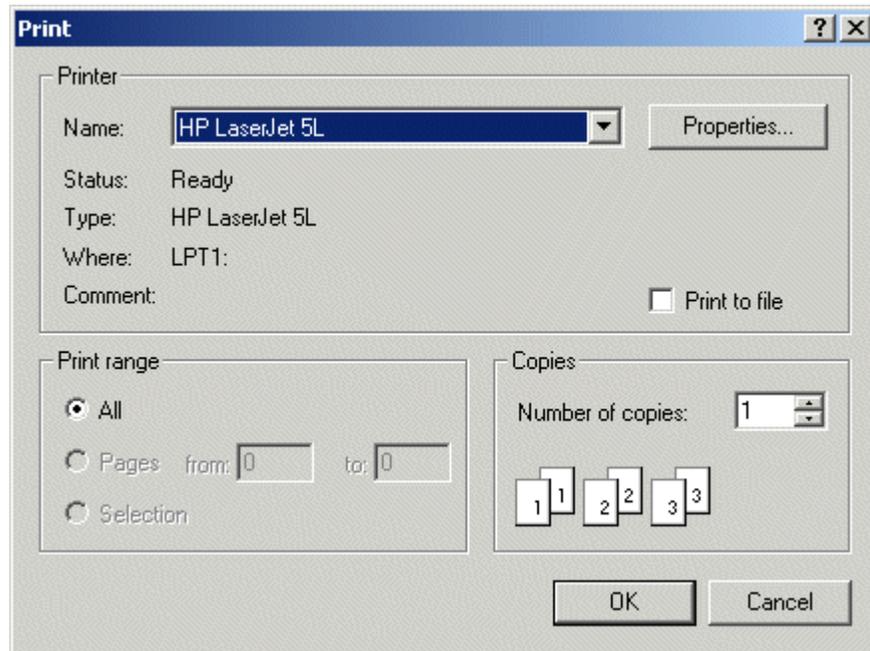
Figure 17-10. The Print Postscript Dialog



Printing from the Dataflow Display on Windows Platforms

With the Dataflow window active, select **File > Print** to print the Dataflow display or to save the display to a file (Figure 17-11).

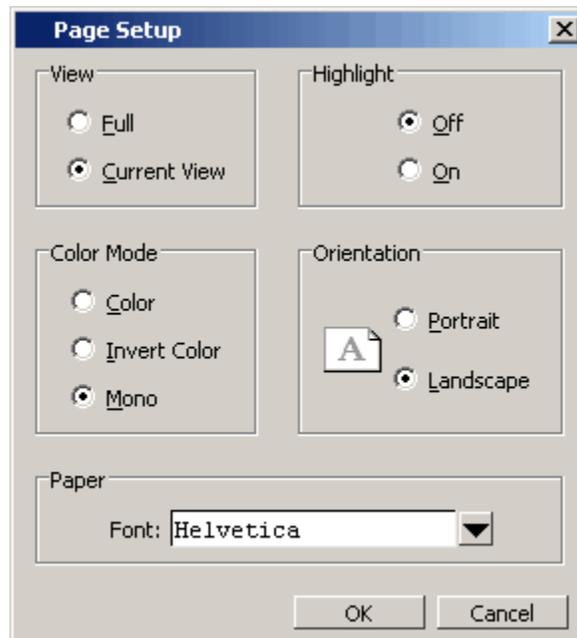
Figure 17-11. The Print Dialog



Configure Page Setup

With the Dataflow window active, select **File > Page setup** to open the Page Setup dialog (Figure 17-12). You can also open this dialog by clicking the Setup button in the Print Postscript dialog (Figure 17-10). This dialog allows you to configure page view, highlight, color mode, orientation, and paper options.

Figure 17-12. The Page Setup Dialog

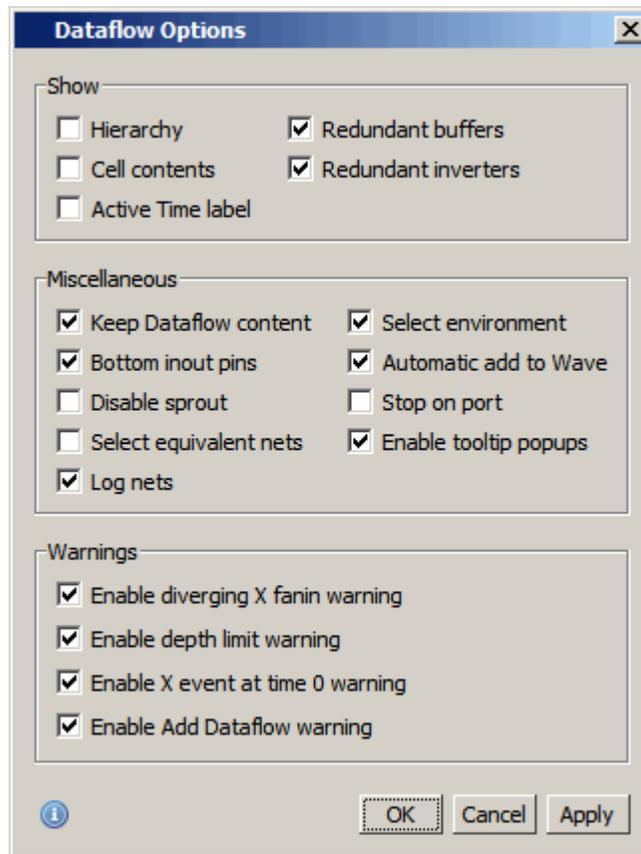


How Do I Configure Window Options?

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **DataFlow > Dataflow Preferences > Options** to open the Dataflow Options dialog box.

Figure 17-13. Configuring Dataflow Options



Chapter 18

Source Window

This chapter discusses the uses of the Source Window for editing, debugging, causality tracing, and code coverage.

Creating and Editing Source Files	851
Data and Objects in the Source Window	858
Breakpoints	867
Source Window Bookmarks	875
Setting Source Window Preferences	875

Creating and Editing Source Files

You can create and edit VHDL, Verilog, SystemVerilog, SystemC, macro (.do), and text files in the Source window. Language specific templates are available to help you write code.

Creating New Files	851
Opening Existing Files	852
Editing Files	853
Language Templates	853
Saving Files	856
Searching for Code in the Source Window	856
Navigating Through Your Design	857

Creating New Files

You can create new files by selecting **File > New > Source**, then selecting one of the following items:

- VHDL — Opens a new file with the file extension *.vhd*
- Verilog — Opens a new file with the file extension *.v*
- SystemC — Opens a new file with the file extension *.cpp*
- SystemVerilog — Opens a new file with the file extension *.sv*
- DO — Opens a new macro file with the file extension *.do*
- Other — Opens a new text file.

Opening Existing Files

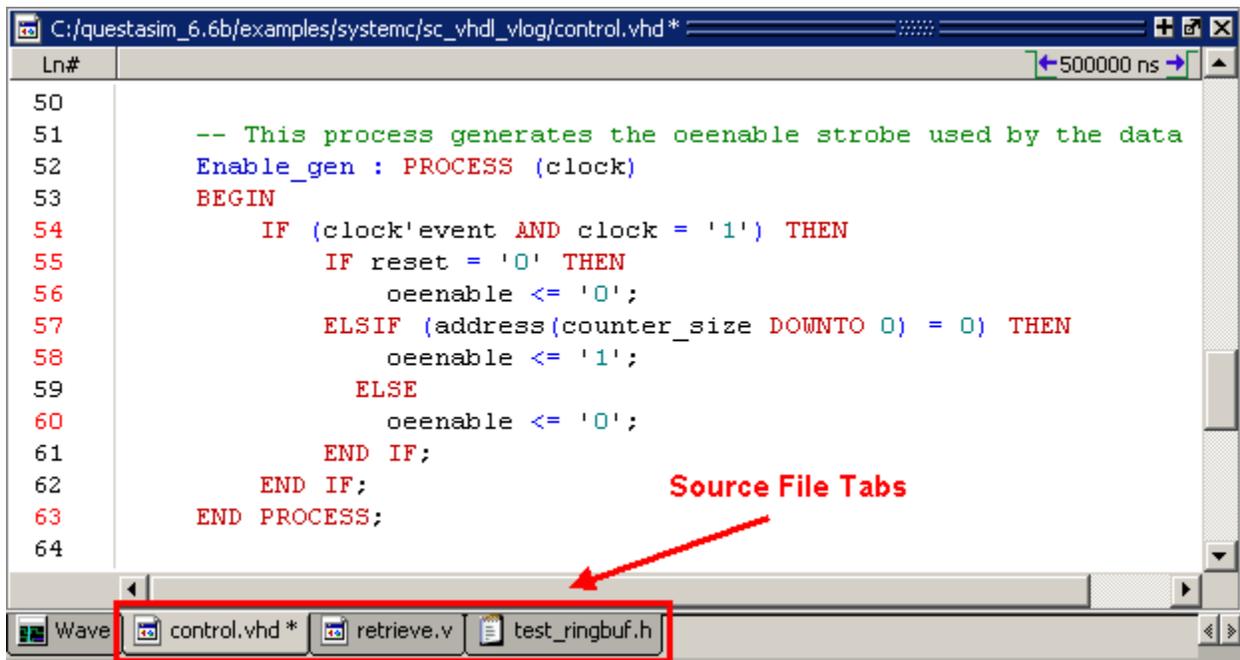
You can open files for editing in the following ways:

- Select **File > Open** then select the file from the **Open File** dialog box.
- Select the **Open** icon in the **Standard** Toolbar then select the file from the **Open File** dialog box.
- Double-click objects in the Ranked, Call Tree, Design Unit, Structure, Objects, and other windows. For example, if you double-click an item in the Objects window or in the Structure window (**sim**), the underlying source file for the object will open in the Source window, the indicator scrolls to the line where the object is defined, and the line is bookmarked.
- Selecting **View Source** from context menus in the Message Viewer, Assertions, Files, Structure, and other windows.
- Enter the `edit <filename>` command to open an existing file.

Displaying Multiple Source Files

By default each file you open or create is marked by a window tab, as shown in [Figure 18-1](#). Unsaved edits are indicated with an asterisk that follows the file name in the tab.

Figure 18-1. Displaying Multiple Source Files



Editing Files

Changing File Permissions

If your file is protected you must create a copy of your file or change file permissions in order to keep any edits you have made.

To change file permissions from the Source window:

Procedure

1. Right-click in the Source window
2. Select (uncheck) **Read Only**.

To change this default behavior, set the **PrefSource(ReadOnly)** preference variable to 0. Refer to [Simulator GUI Preferences](#) for details on setting preference variables.

Language Templates

The ModelSim Language Template is an interactive tool used for the creation and editing of source code in VHDL, Verilog, SystemVerilog, and SystemC. The templates provide you with the basic design elements, wizards, menus, and dialogs that produce code for new designs, test benches, language constructs, logic blocks, and so on.

Note



The language templates are not intended to replace a thorough knowledge of writing code. They are intended as an interactive reference for creating small sections of code. If you are unfamiliar with a particular language, you should attend a training class or consult one of the many available books.

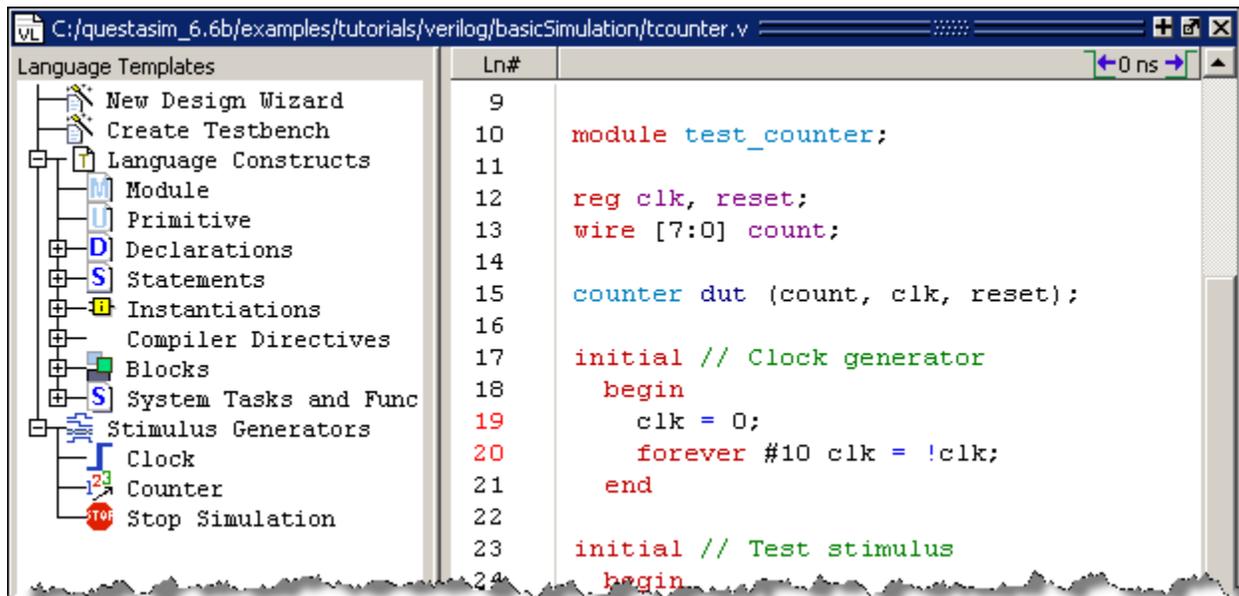
Opening a Language Template

ModelSim opens a Language Template specific to the language you are using in a separate window pane for an existing or new source file.

Procedure

1. Either open an existing file or create a new file.
2. With the source window active, select **Source > Show Language Templates**. ([Figure 18-2](#)).

Figure 18-2. Language Templates



VHDL, Verilog, and SystemVerilog language templates display the following options:

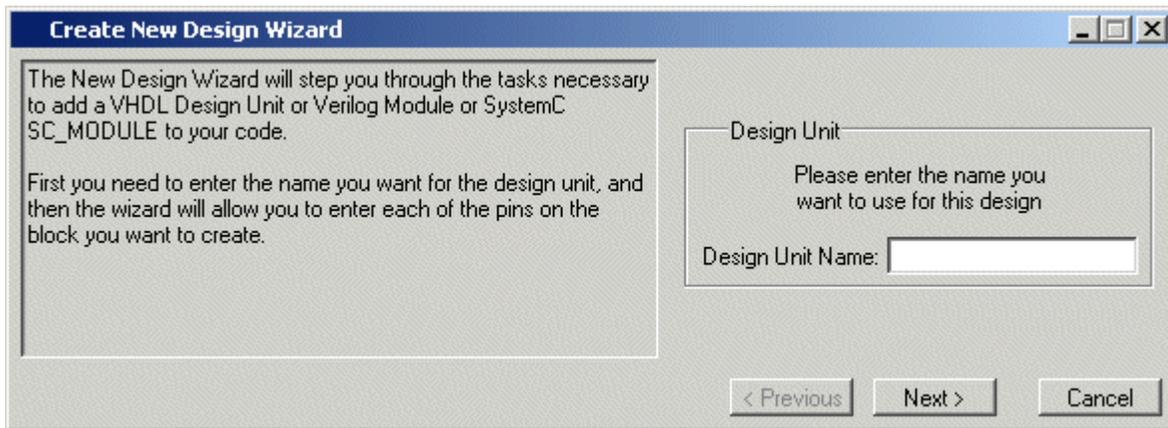
- a. New Design Wizard — Opens the Create New Design Wizard dialog. (Figure 18-3)
The New Design Wizard will step you through the tasks necessary to add a VHDL Design Unit, or Verilog Module to your code.
- b. Create Testbench — Opens the Create Testbench Wizard dialog.
The Create Testbench Wizard allows you to create a testbench for a previously compiled design unit in your library. It generates code that instantiates your design unit and wires it up inside a top-level design unit. You can add stimulus to your testbench at a later time.
- c. Language Constructs — Menu driven code templates you can use in your design.
Includes Modules, Primitives, Declarations, Statements and so on.
- d. Stimulus Generators — Provides three interactive wizards:
 - Create Clock Wizard
Steps you through the tasks necessary to add a clock generator to your code. It allows you to control a number of clock generation variables.
 - Create Count Wizard
Helps you make a counter. You can specify various parameters for the counter. For example, rising/falling edge triggered, reset active high or low, and so on.
 - Create Simulation Stop Wizard

The simulation time at which you wish to end your simulation run. This adds code that will stop the simulator at a specified time.

The SystemC language template displays the following options:

- a. New Design Wizard — Opens the Create New Design Wizard dialog for SystemC source files. (Figure 18-3)
- b. Language Constructs — A list of code templates you can use in your design.

Figure 18-3. Create New Design Wizard



Working With Language Templates

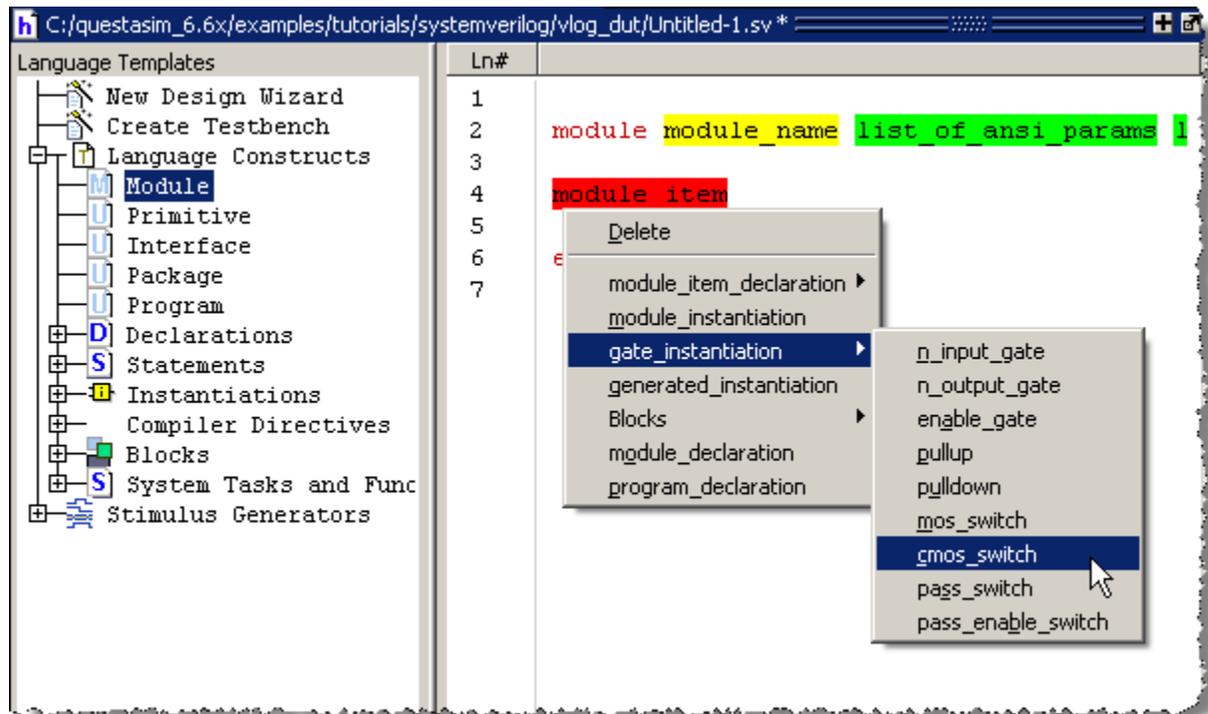
Double click an item in the Language Template pane to place pre-defined code elements into your source document. Figure 18-4 shows a module statement inserted from the SystemVerilog template.

You must right-click the highlighted text to enter new values or make choices from a drop-down menu. Items remain highlighted until the place holding text is replaced with user specified information or a choice is made.

The highlighting indicates the following type of information must be entered:

- Yellow — Requires user supplied data or string. For example, *module_name* in Figure 18-4 must be replaced with the name of the module.
- Green — Opens a drop-down context menu. Selections open more green and yellow highlighted options.
- Red — Opens a drop-down context menu. Selections here can affect multiple code lines. The example below shows the menu that appears when you double-click *module_item* then select *gate_instantiation*.

Figure 18-4. Language Template Context Menus



Saving Files

You can open the Save As dialog box in the following ways:

- Select **File > Save**.
- Click the Save icon in the Standard Toolbar. 
- Press **Ctrl-S** when the Source window is active.

Searching for Code in the Source Window

The Source window includes a Find function that allows you to search for specific code.

Searching for One Instance of a String

Procedure

1. Make the Source window the active window by clicking anywhere in the window
2. Select **Edit > Find** from the Main menu or press **Ctrl-F**. The Search bar is added to the bottom of the Source Window.
3. Enter your search string, then press Enter

The cursor jumps to the first instance of the search string and highlights it. Pressing the Enter key advances the search to the next instance of the string and so on through the source document.

You can also search for the original declaration of an object, signal, parameter, and so on, by double clicking on the object in many windows, including the Structure, Objects, and List windows. Refer to [Hyperlinked Text in the Source Window](#) for information.

Searching for All Instances of a String

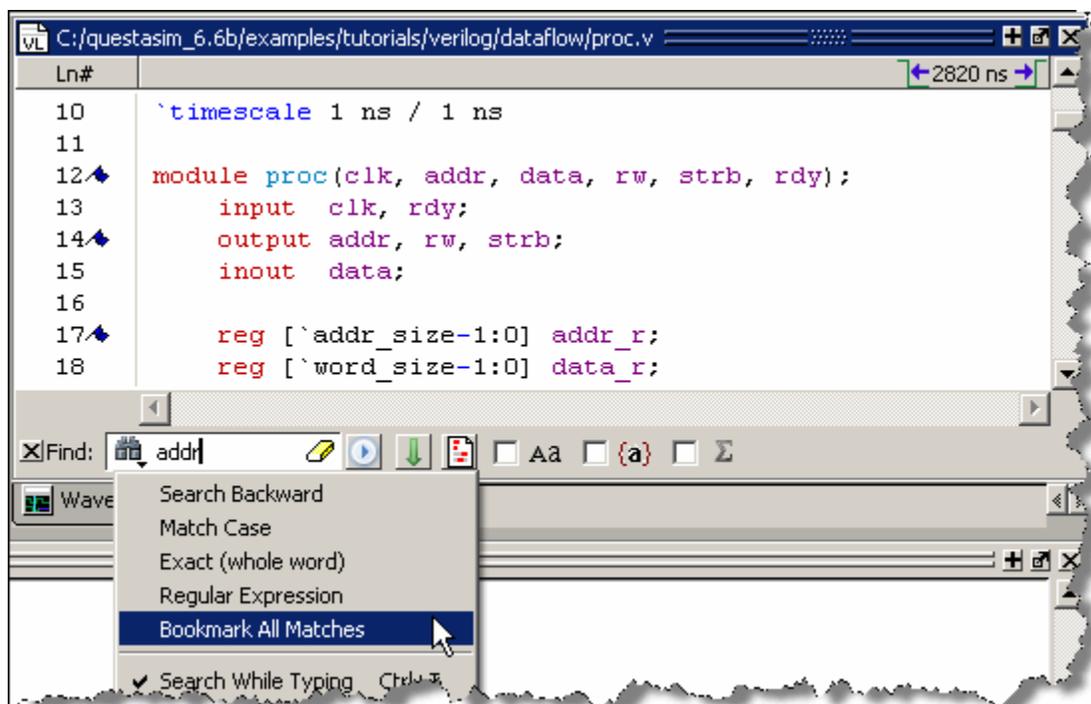
You can search for and bookmark every instance of a search string making it easier to track specific objects throughout a source file.

Procedure

1. Enter the search term in the search field.
2. Select the Find Options drop menu and select **Bookmark All Matches**.



Figure 18-5. Bookmark All Instances of a Search



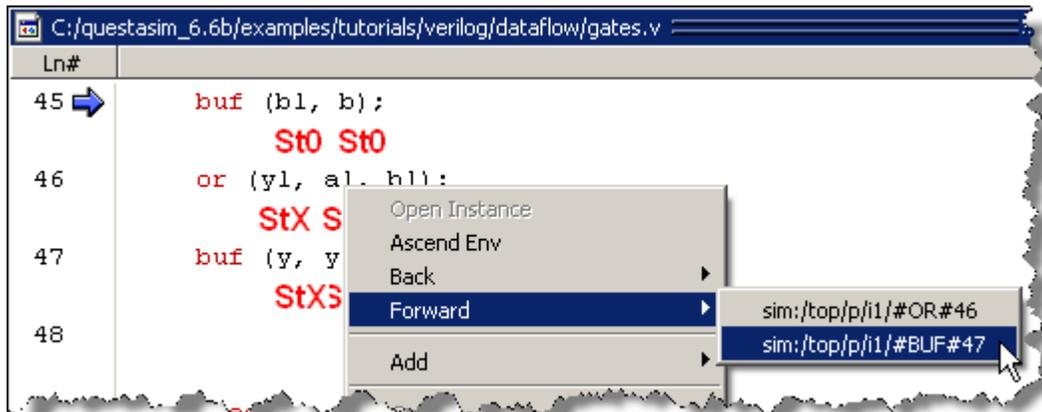
Navigating Through Your Design

When debugging your design from within the GUI, ModelSim keeps a log of all areas of the design environment you have examined or opened. This functionality allows you to easily navigate your design for debugging purposes by logging where you have been within the design hierarchy, similar to the functionality in most web browsers.

Procedure

To easily move through the history, select then right-click an instance name in a source document. This opens a drop down menu (refer to [Figure 18-6](#)) with the following options for navigating your design:

Figure 18-6. Setting Context from Source Files



- **Open Instance** — changes your context to the instance you have selected within the source file. This is not available if you have not placed your cursor in, or highlighted the name of, an instance within your source file.

If any ambiguities exist, most likely due to generate statements, this option opens a dialog box allowing you to choose from all available instances.

- **Ascend Env** — changes your context to the next level up within the design. This is not available if you are at the top-level of your design.
- **Back/Forward** — allows you to change to previously selected contexts. This is not available if you have not changed your context.

The Open Instance option is essentially executing an [environment](#) command to change your context, therefore any time you use this command manually at the command prompt, that information is also saved for use with the Back/Forward options.

Data and Objects in the Source Window

The Source window allows you to display the current value of objects, trace connectivity information, and display coverage data during a simulation run.

Determining Object Values and Descriptions.	859
Displaying Object Values with Source Annotation	859
Setting Simulation Time in the Source Window	860
Source Window Debugging and Textual Connectivity	861
Dragging Source Window Objects Into Other Windows	863

Highlighted Text in the Source Window	864
Hyperlinked Text in the Source Window	865
Code Coverage Data in the Source Window	865

Determining Object Values and Descriptions

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

- Select an object, then right-click and select **Examine** or **Describe** from the context menu
- Pause over an object with your mouse pointer to see an examine popup

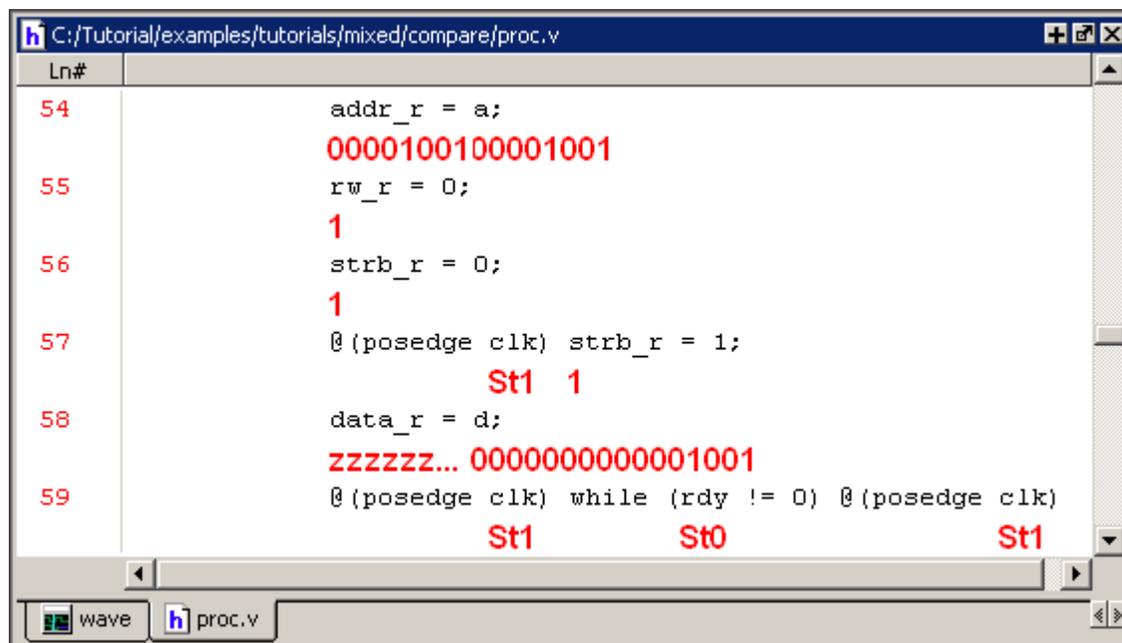
You can select **Source > Examine Now** or **Source > Examine Current Cursor** to choose at what simulation time the object is examined or described.

You can also invoke the [examine](#) and/or [describe](#) commands on the command line or in a macro.

Displaying Object Values with Source Annotation

With source annotation you can interactively debug your design by analyzing your source files in addition to using the Wave and Objects windows. Source annotation displays simulation values, including transitions, for each signal in your source file. [Figure 18-7](#) shows an example of source annotation, where the values are shown in bold red text and placed under the signals.

Figure 18-7. Source Annotation Example



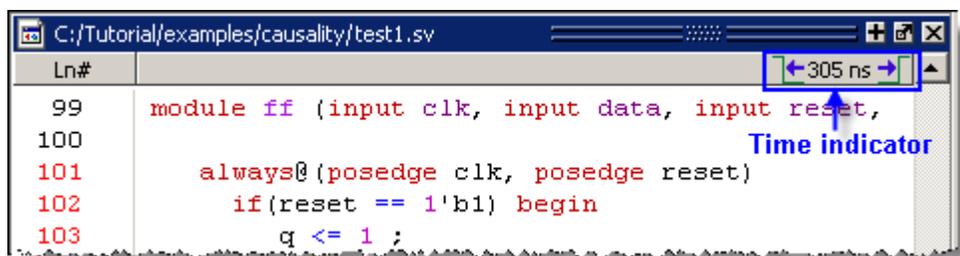
Turn on source annotation by selecting **Source > Show Source Annotation** or by right-clicking a source file and selecting **Show Source Annotation**. Note that transitions are displayed only for those signals that you have logged.

You can highlight a specific signal in the Wave window by double-clicking on an annotation value in the source file.

Setting Simulation Time in the Source Window

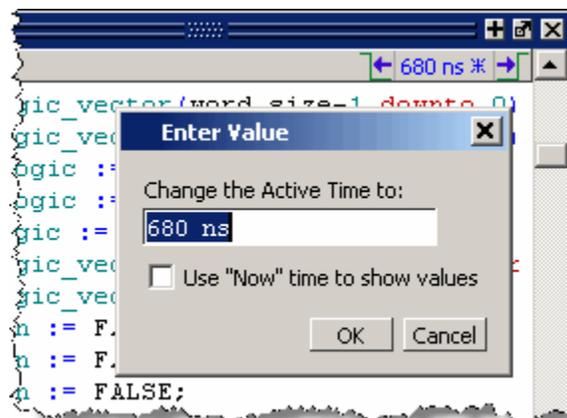
The Source window includes a time indicator in the top right corner (Figure 18-8) that displays the current simulation time, the time of the active cursor in the Wave window, or a user-designated time.

Figure 18-8. Time Indicator in Source Window



1. Click the time indicator to open the **Enter Value** dialog box (Figure 18-9).
2. Change the value to the starting time you want for the causality trace.
3. Click the **OK** button.

Figure 18-9. Enter an Event Time Value



To analyze the values at a given time of the simulation you can:

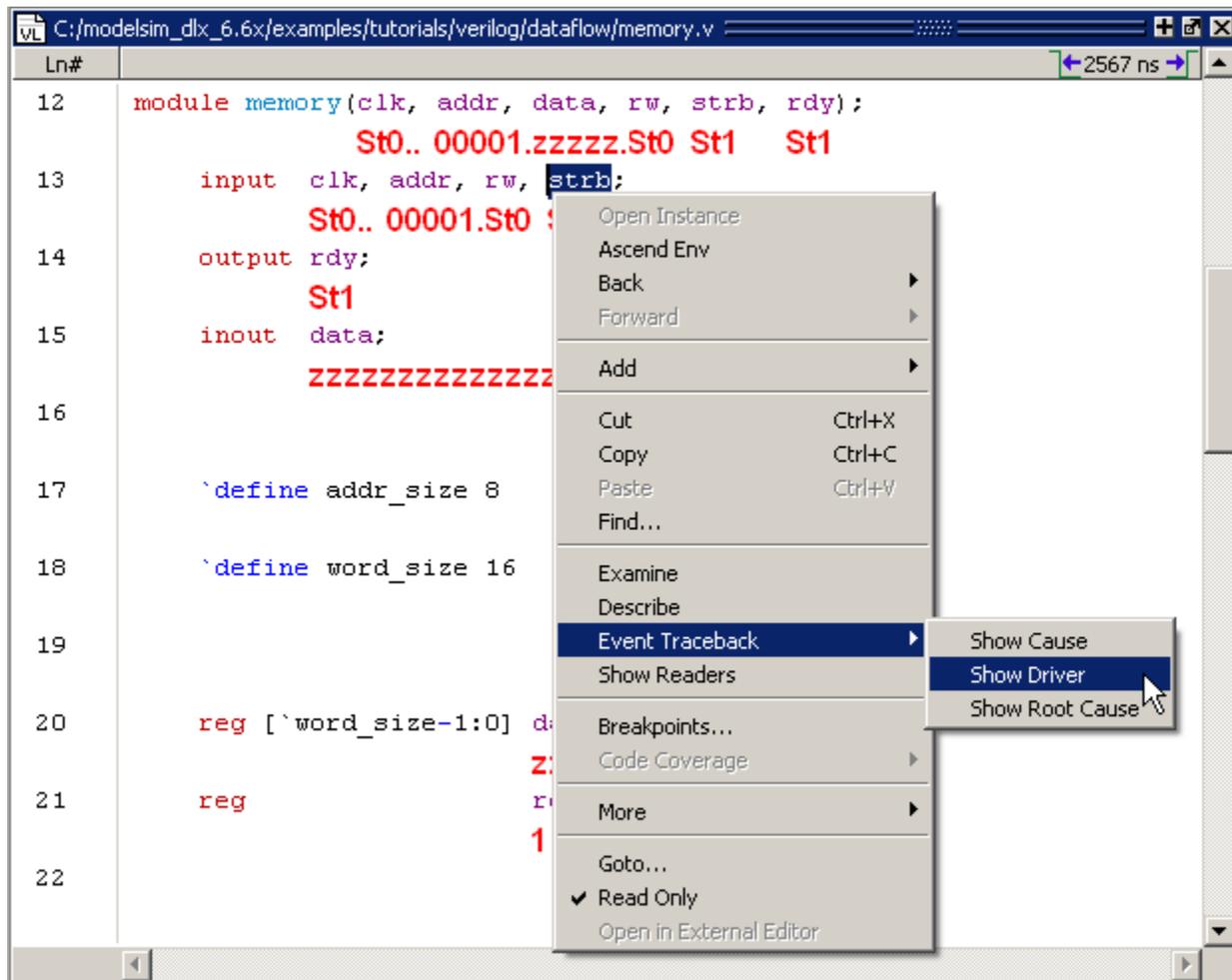
- Show the signal values at the current simulation time by selecting **Source > Examine Now**. This is the default behavior. The window automatically updates the values as you perform a run or a single-step action.
- Show the signal values at current cursor position in the Wave window by selecting **Source > Examine Current Cursor**.

Source Window Debugging and Textual Connectivity

The Source window contains textual connectivity information for the time specified in the time indicator (refer to [Setting Simulation Time in the Source Window](#)). You can explore the connectivity of your design through the source code. This feature is especially useful when used with source annotation turned on.

When you double-click an instance name in the Structure (sim) window, a Source window will open at the appropriate instance. You can then access textual connectivity information in the Source window by right-clicking any signal. This opens a popup menu that gives you the choices shown in [Figure 18-10](#).

Figure 18-10. Popup Menu Choices for Textual Dataflow Information

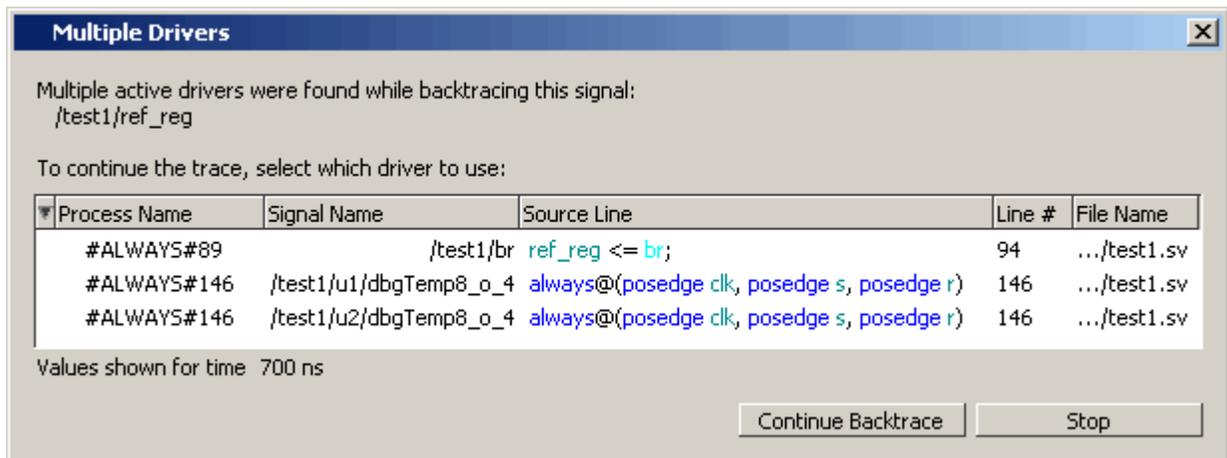


- The **Event Traceback > Show Driver** selection causes the Source window to jump to the source code defining the driver of the selected signal. If the Driver is in a different Source file, that file will open in a new Source window and the driver code will be highlighted. You can also jump to the driver of a signal by double-clicking the signal.

If there is more than one driver for the signal, a Multiple Drivers dialog will open showing all driving processes (Figure 18-11).

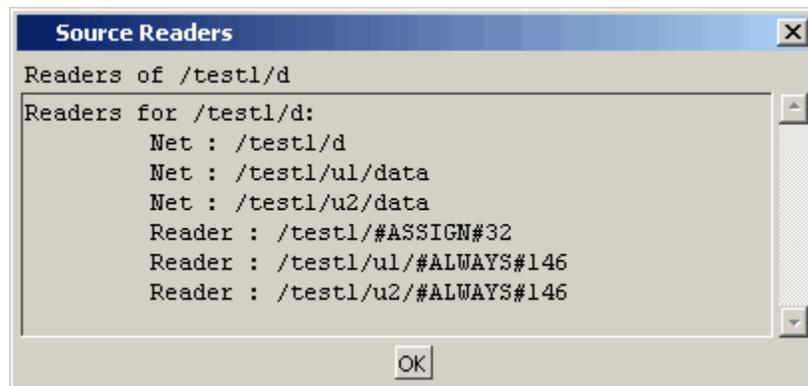
Select any driver to open the code for that driver.

Figure 18-11. Window Shows all Driving Processes



- The **Show Readers** selection opens the Source Readers window. If there is more than one reader for the signal, all will be displayed (Figure 18-12).

Figure 18-12. Source Readers Dialog Displays All Signal Readers



When the trace is complete, the Active Driver Path Details window displays all signals in the causality path, the Objects window highlights all signals in the path, and the Source window jumps to the assignment code that caused the event and highlights the code.

Limitations

The textual dataflow functions of the Source window only work for pure HDL. They will not work for SystemC or for complex data types like SystemVerilog classes.

Dragging Source Window Objects Into Other Windows

ModelSim allows you to drag and drop objects from the Source window to the Wave and List windows. Double-click an object to highlight it, then drag the object to the Wave or List window. To place a group of objects into the Wave and List windows, drag and drop any section of highlighted code.

Highlighted Text in the Source Window

The Source window can display text that is highlighted as a result of various conditions or operations, such as the following:

- Double-clicking an error message in the transcript shown during compilation
- Using **Event Traceback > Show Driver**
- Coverage-related operations

In these cases, the relevant text in the source code is shown with a persistent highlighting. To remove this highlighted display, choose **More > Clear Highlights** from the right-click popup menu of the Source window. You can also perform this action by selecting **Source > More > Clear Highlights** from the Main menu.

Note



Clear Highlights does not affect text that you have selected with the mouse cursor.

Example

To produce a compile error that displays highlighted text in the Source window, do the following:

1. Choose **Compile > Compile Options**
2. In the Compiler Options dialog box, click either the VHDL tab or the Verilog & System Verilog tab.
3. Enable Show source lines with errors and click OK.
4. Open a design file and create a known compile error (such as changing the word “entity” to “entry” or “module” to “nodule”).
5. Choose **Compile > Compile** and then complete the Compile Source Files dialog box to finish compiling the file.
6. When the compile error appears in the Transcript window, double-click on it.
7. The source window is opened (if needed), and the text containing the error is highlighted.
8. To remove the highlighting, choose **Source > More > Clear Highlights**.

Hyperlinked Text in the Source Window

The Source window supports hyperlinked navigation. When you double-click hyperlinked text the selection jumps from the usage of an object to its declaration and highlights the declaration. Hyperlinked text is indicated by a mouse cursor change from an arrow pointer icon to a pointing finger icon: 

Double-clicking hyperlinked text does one of the following:

- Jump from the usage of a signal, parameter, macro, or a variable to its declaration.
- Jump from a module declaration to its instantiation, and vice versa.
- Navigate back and forth between visited source files.

Procedure

Turn hyperlinked text on or off in the Source window:

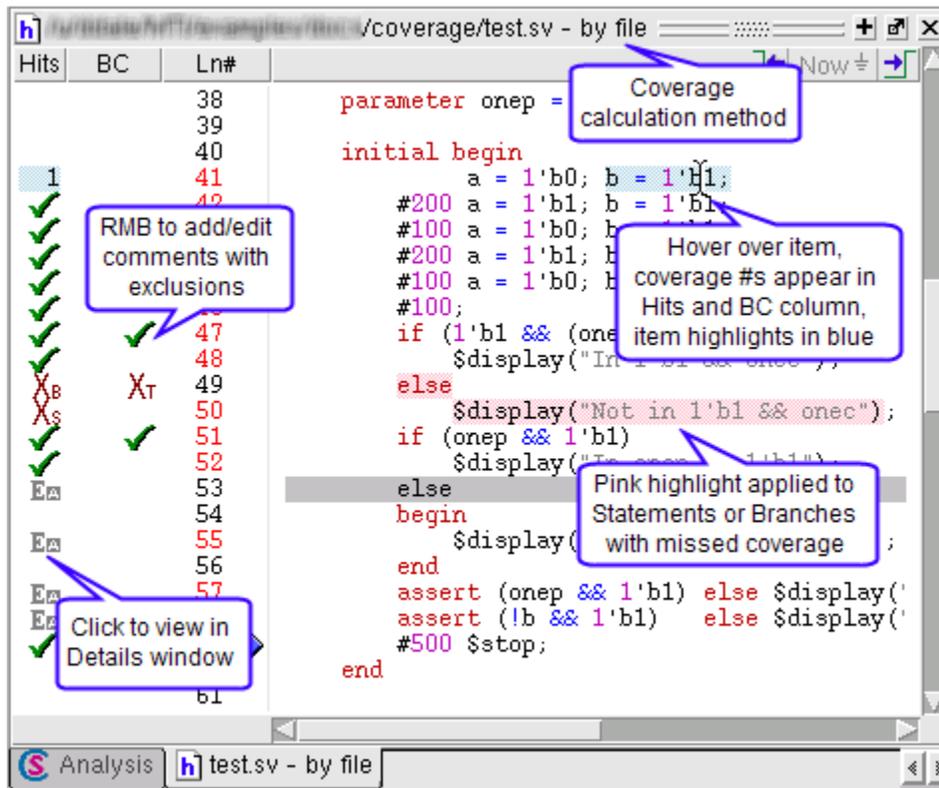
1. Make sure the Source window is the active window.
2. Select **Source > Hyperlinks**.

To change hyperlinks to display as underlined text set **prefMain(HyperLinkingUnderline)** to 1 (select **Tools > Edit Preferences**, By Name tab, and expand the Main Object).

Code Coverage Data in the Source Window

The Source window includes two columns for code coverage statistics – the Hits column and the BC (Branch Coverage) column. These columns provide an immediate visual indication about how your source code is executing. The code coverage indicators are check marks, Xs and Es, the complete variety of which are described in [Source Window Code Coverage Indicator Icons](#).

Figure 18-13. Coverage in Source Window



To see more information about any coverage item, click on the indicator icon, or in the Hits or BC column for the line of interest. This brings up detailed coverage information for that line in the Coverage Details window.

For example, when you select an expression in the Missed Expressions window, and you click in the column of a line containing an expression, the associated truth tables appear in the Coverage Details window. Each line in the truth table is one of the possible combinations for the expression. The expression is considered to be covered (gets a green check mark) only if the entire truth table is covered.

When you hover over statements, conditions or branches in the Source window, the Hits and BC columns display the coverage numbers for that line of code. For example, in Figure 18-13, the blue line shows that the expression (a && b) was hit 5 times and that the branch (if) was evaluated as true once (1t) and false four times (4f). The value in the Hits column shows the total coverage for all items in the UDP table (as shown in the Coverage Details window when you click the specific line in the hits column).

Coverage data presented in the Source window is either calculated “by file” or “by instance”, as indicated just after the source file name. If coverage numbers are mismatched between Missed <coverage_type> window and the Source window, check to make sure that both are being calculated the same — either “by file” or “by instance”.

To display only numbers in Hits and BC columns, select **Tools > Code Coverage > Show Coverage Numbers**.

When the source window is active, you can skip to "missed lines" three ways:

- select **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** from the menu bar
- click the Previous zero hits and Next zero hits icons on the toolbar
- press Shift-Tab (previous miss) or Tab (next miss)

Controlling Coverage Data Display

The **Tools > Code Coverage** menu contains several commands for controlling coverage data display in a Source window.

- **Hide/Show coverage data** — Toggles the *Hits* column off and on.
- **Hide/Show branch coverage** — Toggles the *BC* column off and on.
- **Hide/Show coverage numbers** — Displays the number of executions in the *Hits* and *BC* columns rather than check marks and Xs. When multiple statements occur on a single line an ellipsis ("...") replaces the Hits number. In such cases, hover the cursor over each statement to highlight it and display the number of executions for that statement.
- **Show coverage By Instance** — Displays only the number of executions for the currently selected instance in the Main window workspace.

Breakpoints

You can set a breakpoint on an executable file, file-line number, signal, signal value, or condition in a source file. When the simulation hits a breakpoint, the simulator stops, the Source window opens, and a blue arrow marks the line of code where the simulation stopped. You can change this behavior by editing the **PrefSource(OpenOnBreak)** variable. Refer to [Simulator GUI Preferences](#) for more information on setting preference variables.

Note



When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. Refer to [Preserving Object Visibility for Debugging Purposes](#) and [Design Object Visibility for Designs with PLI](#).

You can set breakpoints in the following ways:

Setting Individual Breakpoints in a Source File 868

Setting Breakpoints with the bp Command	868
Setting SystemC Breakpoints	869
Editing Breakpoints	869
Saving and Restoring Breakpoints	871
Setting Conditional Breakpoints.....	872

Setting Individual Breakpoints in a Source File

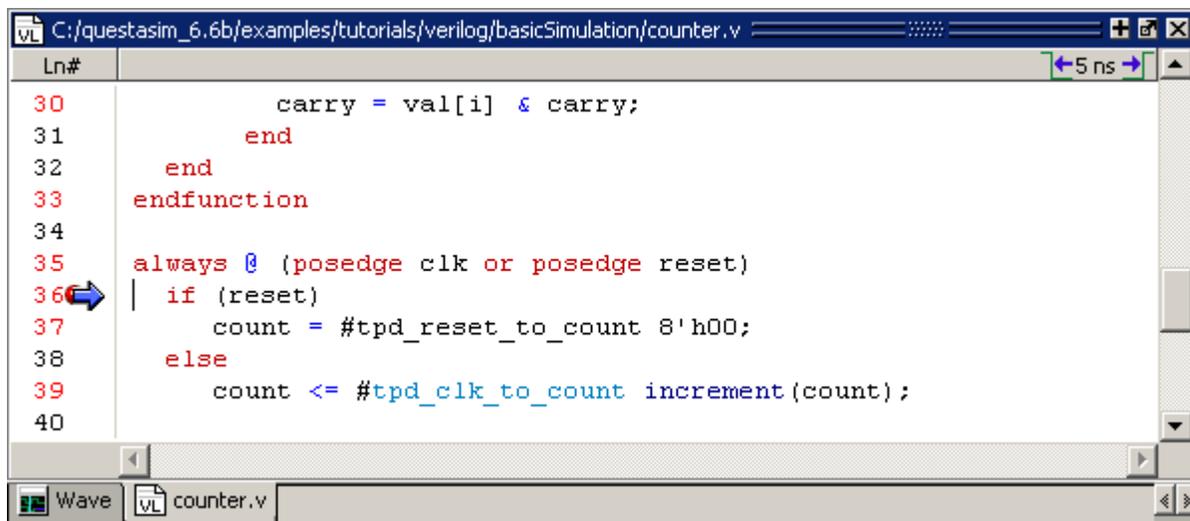
You can set individual file-line breakpoints in the Line number column of the Source Window.

Procedure

Click in the line number column of the Source window next to a red line number and a red ball denoting a breakpoint will appear (Figure 18-14).

The breakpoint markers (red ball) are toggles. Click once to create the breakpoint; click again to disable or enable the breakpoint.

Figure 18-14. Breakpoint in the Source Window



Setting Breakpoints with the bp Command

You can set a file-line breakpoints with the `bp` command to add a file-line breakpoint from the `VSIM>` prompt.

For example:

```
bp top.vhd 147
```

sets a breakpoint in the source file `top.vhd` at line 147.

Setting SystemC Breakpoints

Prerequisites

Your C Debug settings must be in place prior to setting a breakpoint since C Debug is invoked when you set a breakpoint within a SystemC module. Refer to “[Setting Up C Debug](#)” for more information. Once invoked, C Debug can be exited using the C Debug menu.

Editing Breakpoints

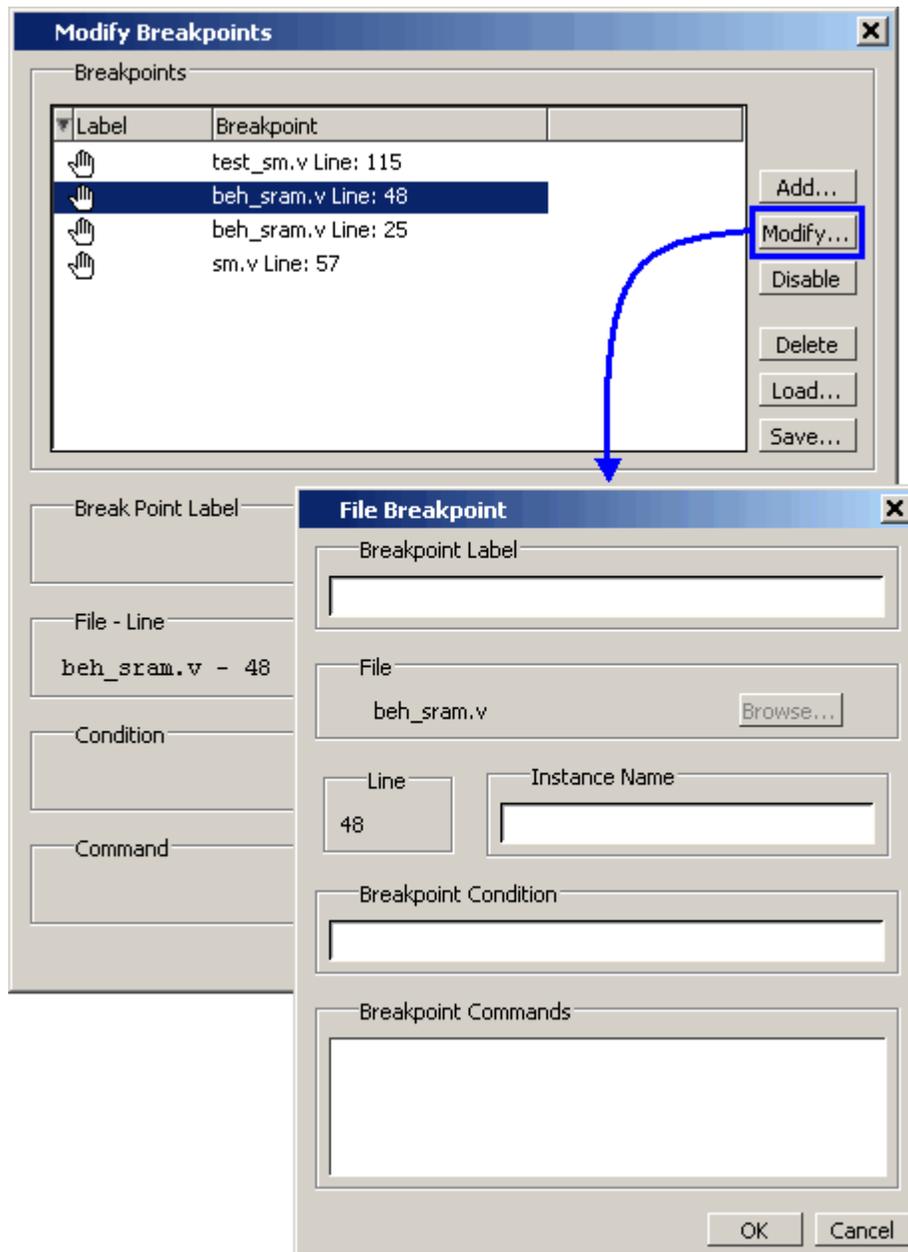
To edit a breakpoint in a source file, do any one of the following:

- Select **Tools > Breakpoints** from the Main menu.
- Right-click a breakpoint in your source file and select **Edit All Breakpoints** from the popup menu.
- Click the **Edit Breakpoints** toolbar button from the [Simulate Toolbar](#).

This opens the Modify Breakpoints dialog shown in [Figure 18-15](#). The Modify Breakpoints dialog provides a list of all breakpoints in the design organized by ID number.

1. Select a file-line breakpoint from the list in the Breakpoints field.
2. Click **Modify**, which opens the **File Breakpoint** dialog box, [Figure 18-15](#).

Figure 18-15. Editing Existing Breakpoints



3. Fill out any of the following fields to edit the selected breakpoint:

- Breakpoint Label — Designates a label for the breakpoint.
- Instance Name — The full pathname to an instance that sets a SystemC breakpoint so it applies only to that specified instance.
- Breakpoint Condition — One or more conditions that determine whether the breakpoint is observed. If the condition is true, the simulation stops at the breakpoint. If false, the simulation bypasses the breakpoint. A condition cannot refer

to a VHDL variable (only a signal). Refer to [Setting Conditional Breakpoints](#) for more information.

- **Breakpoint Command** — A string, enclosed in braces ({}), that specifies one or more commands to be executed at the breakpoint. Use a semicolon (;) to separate multiple commands.

i **Tip:** These fields in the File Breakpoint dialog box use the same syntax and format as the `-inst` switch, the `-cond` switch, and the command string of the `bp` command. For more information on these command options, refer to the `bp` command in the Reference Manual.

4. Click OK to close the File Breakpoints dialog box.
5. Click OK to close the Modify Breakpoints dialog box.

Deleting Individual Breakpoints

You can permanently delete individual file-line breakpoints using the breakpoint context menu.

Procedure

1. Right-click the red breakpoint marker in the file line column.
2. Select Remove Breakpoint from the context menu.

Deleting Groups of Breakpoints

You can delete groups of breakpoints with the Modify Breakpoints Dialog.

Procedure

1. Open the Modify Breakpoints dialog.
2. Select and highlight the breakpoints you want to delete.
3. Click the **Delete** button
4. **OK**.

Saving and Restoring Breakpoints

You can save your breakpoints in a separate *breakpoints.do* file or save the breakpoint settings as part of a larger *.do* file that recreates all debug windows and includes breakpoints.

1. To save your breakpoints in a *.do* file, select **Tools > Breakpoints** to open the Modify Breakpoints dialog. Click **Save**. You will be prompted to save the file under the name: *breakpoints.do*.

To restore the breakpoints, start the simulation then enter:

do breakpoints.do

2. To save your breakpoints together with debug window settings, enter

write format restart <filename>

The **write format** restart command creates a single *.do* file that saves all debug windows, file/line breakpoints, and signal breakpoints created using the **when** command. The file created is primarily a list of **add list**, **add wave**, and **configure** commands, though a few other commands are included. If the **ShutdownFile** *modelsim.ini* variable is set to this *.do* filename, it will call the **write format** restart command upon exit.

To restore debugging windows and breakpoints enter:

do <filename>.do

Note



Editing your source file can cause changes in the numbering of the lines of code. Breakpoints saved prior to editing your source file may need to be edited once they are restored in order to place them on the appropriate code line.

Setting Conditional Breakpoints

In dynamic class-based code, an expression can be executed by more than one object or class instance during the simulation of a design. You set a conditional breakpoint on the line in the source file that defines the expression and specifies a condition of the expression or instance you want to examine. You can write conditional breakpoints to evaluate an absolute expression or a relative expression.

You can use the SystemVerilog keyword **this** when writing conditional breakpoints to refer to properties, parameters or methods of an instance. The value of **this** changes every time the expression is evaluated based on the properties of the current instance. Your context must be within a local method of the same class when specifying the keyword **this** in the condition for a breakpoint. Strings are not allowed.

The conditional breakpoint examples below refer to the following SystemVerilog source code file *source.sv*:

Figure 18-16. Source Code for *source.sv*

```
1 class Simple;
2   integer cnt;
3   integer id;
4   Simple next;
5
6   function new(int x);
7       id=x;
8       cnt=0
```

```
9     next=null
10    endfunction
11
12    task up;
13        cnt=cnt+1;
14        if (next) begin
15            next.up;
16        end
17    endtask
18 endclass
19
20 module test;
21     reg clk;
22     Simple a;
23     Simple b;
24
25     initial
26     begin
27         a = new(7);
28         b = new(5);
29     end
30
31     always @(posedge clk)
32     begin
33         a.up;
34         b.up;
35         a.up
36     end;
37 endmodule
```

Prerequisites

Compile and load your simulation.

Note



You must use the +acc switch when optimizing with vopt to preserve visibility of SystemVerilog class objects.

Setting a Breakpoint For a Specific Instance

Enter the following on the command line:

```
bp simple.sv 13 -cond {this.id==7}
```

Results

The simulation breaks at line 13 of the *simple.sv* source file (Figure 18-16) the first time module a hits the expression because the breakpoint is evaluating for an id of 7 (refer to line 27).

Setting a Breakpoint For a Specified Value of Any Instance.

Enter the following on the command line:

```
bp simple.sv 13 -cond {this.cnt==8}
```

Results

The simulation evaluates the expression at line 13 in the *simple.sv* source file (Figure 18-16), continuing the simulation run if the breakpoint evaluates to false. When an instance evaluates to true the simulation stops, the source is opened and highlights line 13 with a blue arrow. The first time `cnt=8` evaluates to true, the simulation breaks for an instance of module Simple b. When you resume the simulation, the expression evaluates to `cnt=8` again, but this time for an instance of module Simple a.

You can also set this breakpoint with the GUI:

1. Right-click on line 13 of the *simple.sv* source file.
2. Select Edit Breakpoint 13 from the drop menu.
3. Enter

```
this.cnt==8
```

in the **Breakpoint Condition** field of the **Modify Breakpoint** dialog box. (Refer to Figure 18-15) Note that the file name and line number are automatically entered.

Run Until Here

The Source window allows you to run the simulation to a specified line of code with the “**Run Until Here**” feature. When you invoke **Run Until Here**, the simulation will run from the current simulation time and stop on the specified line unless:

- The simulator encounters a breakpoint.
- Optionally, the **Run Length** preference variable causes the simulation run to stop.
- The simulation encounters a bug.

Note



Run Until Here will not execute if you are running a fully optimized design. You must run the simulation in non-optimized mode or set `+acc` arguments to enable you to execute Run Until Here. Refer to [Preserving Object Visibility for Debugging Purposes](#) and [Design Object Visibility for Designs with PLI](#).

To specify **Run Until Here**, right-click on the line where you want the simulation to stop and select **Run Until Here** from the pop up context menu. The simulation starts running the moment the right mouse button releases.

The simulator run length is set in the Simulation Toolbar and specifies the amount of time the simulator will run before stopping. By default, **Run Until Here** will ignore the time interval entered in the **Run Length** field of the Simulation Toolbar unless the

PrefSource(RunUntilHereUseRL) preference variable is set to 1 (enabled). When **PrefSource(RunUntilHereUseRL)** is enabled, the simulator will invoke **Run Until Here** and stop when the amount of time entered in the **Run Time** field has been reached, a breakpoint is hit, or the specified line of code is reached, whichever happens first.

For more information about setting preference variables, refer to [Simulator GUI Preferences](#).

Source Window Bookmarks

Source window bookmarks are graphical icons that give you reference points within your code. The blue flags mark individual lines of code in a source file and can assist visual navigation through a large source file by marking certain lines. Bookmarks can be added to currently open source files only and are deleted once the file is closed.

Setting and Removing Bookmarks

You can set bookmarks in the following ways:

- Set an individual bookmark.
 - a. Right-click in the Line number column on the line you want to bookmark then select **Add/Remove Bookmark**.
- Set multiple bookmarks based on a search term refer to [Searching for All Instances of a String](#).

To remove a bookmark:

- Right-click the line number with the bookmark you want to remove and select **Add/Remove Bookmark**.
- Select the **Clear Bookmarks** button in the **Source** toolbar.

Setting Source Window Preferences.

You can customize a variety of settings for Source windows. For example, you can change fonts, spacing, colors, syntax highlighting, underlining of hyperlinked code, and so on.

Prerequisite

Select **Tools > Edit Preferences**. This opens the **Preferences** dialog box.

Procedure

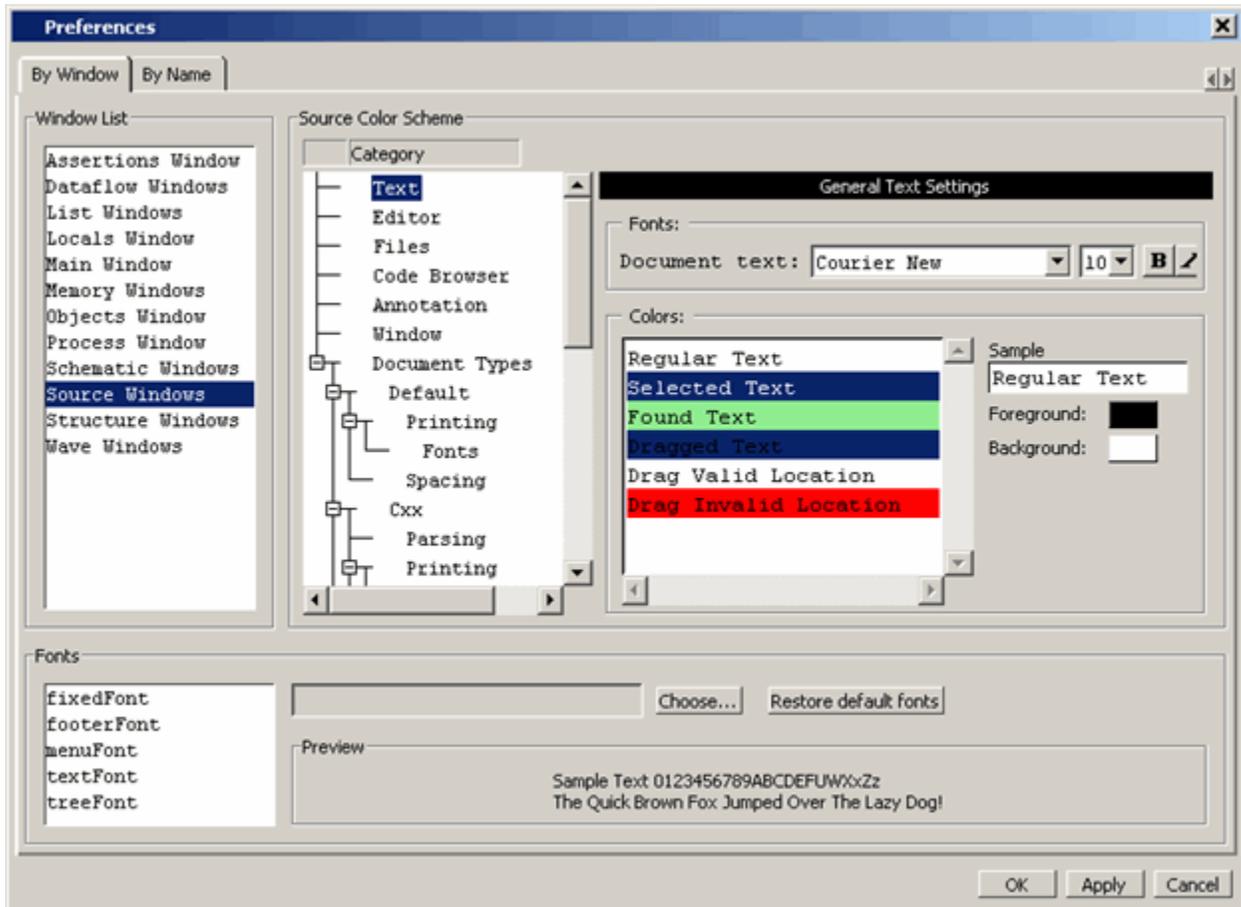
There are two tabs that change Source window settings:

1. By Window tab ([Figure 18-17](#)) — Sets the Color schemes and fonts for the Source window.

Source Window
Setting Source Window Preferences.

- a. Select **Source Windows** from the **Window List** pane.
- b. Select a **Category** in the **Source Color Scheme** pane or a font in the **Fonts** pane.
- c. Change the attributes.
- d. **OK**

Figure 18-17. Preferences By - Window Tab



Chapter 19

Using Causality Traceback

The Causality Traceback feature is designed to help you determine the cause of any signal event or all possible drivers of a signal. It allows you to trace backward through simulation time to find both event drivers and the logic behind the drivers. Causality Traceback uses the optimization utility (`vopt -debugdb` or `vdbg`) to detect combinatorial and sequential logic events, and saves data about those events to your working library. When you invoke the simulator with the `-debugdb` switch for the `vsim` command, event data is collected in a `.dbg` file. The `.dbg` file is a connectivity and structure database that can be used for current simulation and post simulation analysis.

You can initiate a causality trace from multiple debugging windows (Wave, Source, Objects, Schematic, Structure) or from the command line. In addition, you may designate any arbitrary time as the start time for the trace.

After a causality trace is complete, the design context is automatically changed and all signals found in the path are selected. You may view the details of the trace within the GUI in the Wave, Source, Objects, Schematic, Structure, and Active Driver Path Details windows, which are automatically updated with the latest trace results.

During a trace analysis it is possible that multiple input values are changing at the same time. When this occurs, a Multiple Drivers window opens, allowing you to choose how the trace should continue.

Usage Flow for Causality Traceback

The recommended usage flow for causality traceback is described by the following steps:

1. Create a library for your work.

`vlib <library_name>`

2. Compile your design into the library.

`vcom` and/or `vlog`

3. Optimize your design and collect combinatorial and sequential logic data.

`vopt +acc <filename> -o <optimized_filename> -debugdb`

The `+acc` switch maintains visibility into your design for debugging while the `-debugdb` switch saves combinatorial and sequential logic events to the working library.

4. Load your design (elaboration).

`vsim -debugdb <optimized_filename>`

The **`-debugdb`** switch instructs the simulator to look for combinatorial and sequential logic event data in the working library, then creates the debug database (*vsim.dbg*) from this information.

The default filename for the *.dbg* file is *vsim.dbg*. If you want to create a different name, use the following command syntax:

`vsim -debugdb=<custom_name>.dbg -wlf <custom_name>.wlf <optimized_filename>`

The *<custom_name>* must be the same for the *.dbg* file and the *.wlf* file.

5. Log simulation data.

`log -r /*` or **`add wave -r /*`**

It is advisable to log the entire design. This will provide the historic values of the events of interest plus its drivers. However, to reduce overhead, you may log only the regions of interest.

You may use the **`log`** command to simply save the simulation data to the *.wlf* file; or, use the **`add wave`** command to log the simulation data to the *.wlf* file *and* display simulation results as waveforms in the Wave window.

6. Run the simulation.
7. Initiate a causality trace from the GUI or from the command line.

Abbreviated Usage Flow

The above usage flow may be abbreviated as follows:

1. Create a library for your work.

`vlib <library_name>`

2. Compile your design.

`vcom` and/or **`vlog`**

3. Load your design

`vsim -voptargs="+acc" -debugdb <design_name>`

The **`-voptargs="+acc"`** switch for the **`vsim`** command maintains visibility into your design for debugging, and the **`-debugdb`** switch calls the **`vdbg`** command.

4. Log your design

`log -r /*` or **`add wave -r /*`**

5. Run the simulation
6. Initiate a causality trace from the GUI or from the command line.

This abbreviated flow does not give you the control over the optimization process provided by the recommended flow.

Post-sim Debug

If you have logged the design, you can perform post-simulation Causality Traceback. Simply open the library (directory) containing the saved *.wlf* file and enter the following command:

```
vsim -view vsim.wlf
```

The simulator will automatically look for the *.dbg* file.

If you have used a custom filename for the *.dbg* and *.wlf* files, use:

```
vsim -view <custom_name>.wlf
```

You may also use the [dataset open](#) command if you are entering the command from within ModelSim.

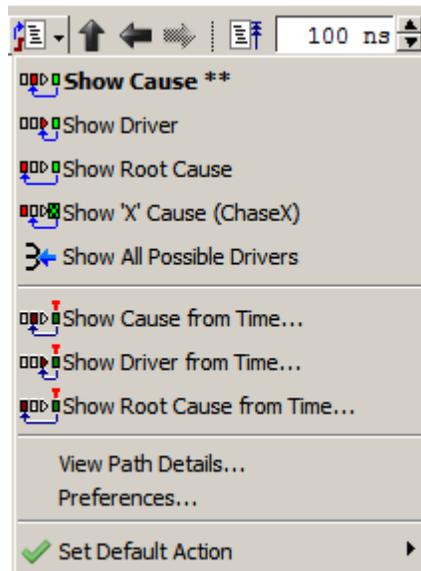
Initiating Causality Traceback from the GUI

Menu and toolbar button selections in the GUI allow you to initiate causality traceback for:

- [Tracing to the First Sequential Process](#) (Show Cause)
- [Tracing to the Immediate Driving Process](#) (Show Driver)
- [Tracing to the Root Cause](#) (Show Root Cause)
- [Tracing to the Root Cause of an 'X'](#) (Show 'X' Cause)
- [Finding All Possible Drivers](#) (Show All Possible Drivers)

The Event Traceback toolbar button provides access to these traces.  When you press-and-hold this button, a drop-down menu appears ([Figure 19-1](#)).

Figure 19-1. Event Traceback Toolbar Button Menu



You can initiate a causality trace from any arbitrary time by selecting **Show Cause from Time**, **Show Driver from Time**, or **Show Root Cause from Time** in the toolbar button menu above. You may also use the time indicator in the Source window to set a starting time for the causality trace.

Note



The Show Cause and Show Root Cause options will display a warning if you simulate with the -novopt switch. You will only have access to the Show Driver option in the Source window. Full causality traceback functionality requires optimization of your design with vopt or vsim -voptargs. Refer to [Usage Flow for Causality Traceback](#) for more information.

Tracing to the First Sequential Process

You can initiate a causality trace to the first sequential process from the Wave, Objects, Schematic, or Source windows.

Initiating the Trace from the Wave Window

Trace from a signal event to the sequential process(es) that caused the event by doing the following:

1. Select a signal of interest in the Wave window.
2. Perform one of the following actions:
 - Double-click (left mouse button) an event of interest in the waveform of the selected signal.

Or,

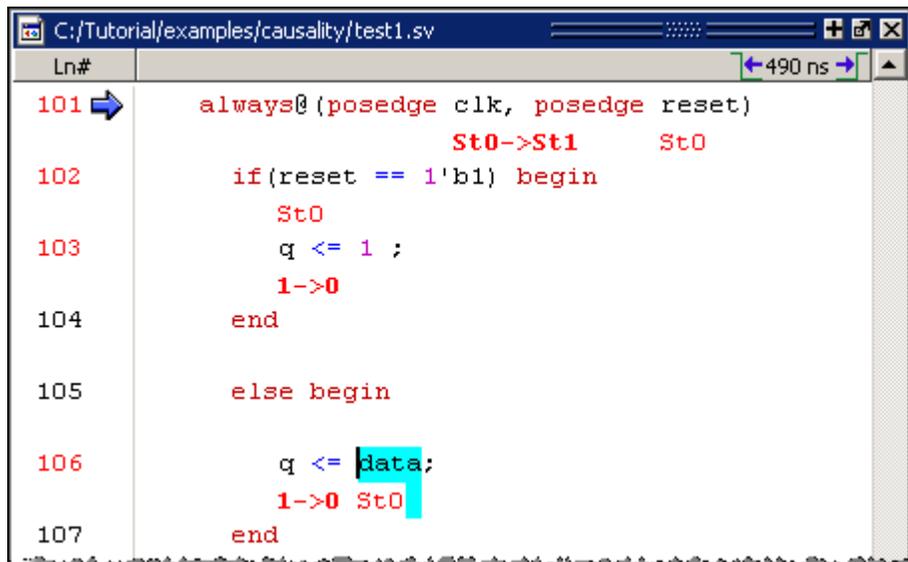
- Click an event of interest in the waveform of the selected signal, then click the **Event Traceback** toolbar button.



Either of these actions initiates a trace to find the sequential process(es) that caused the selected event.

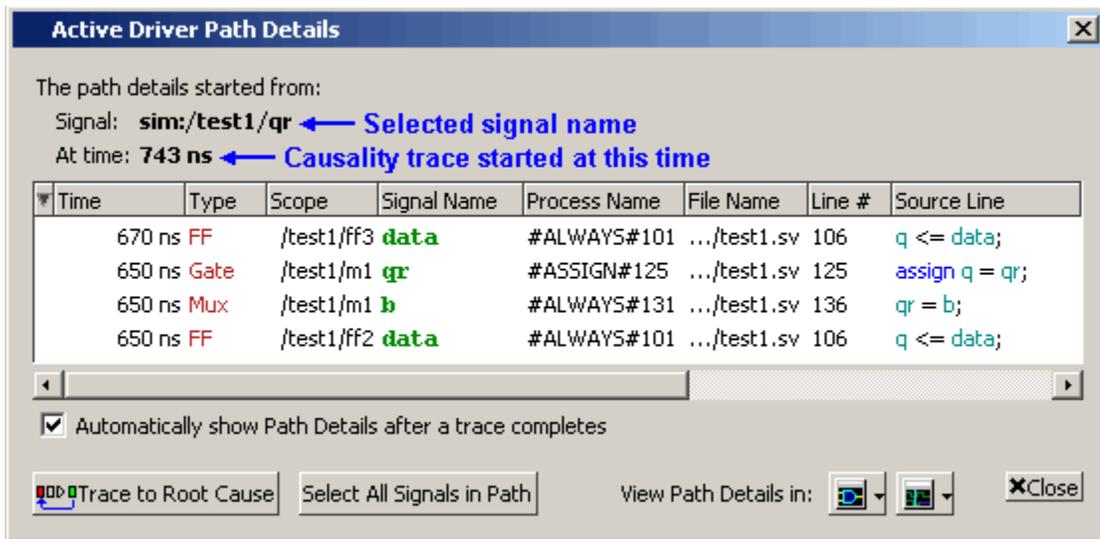
When the causality trace ends, an annotated Source window opens with the causal process highlighted (Figure 19-2).

Figure 19-2. Cause is Highlighted in Source Window



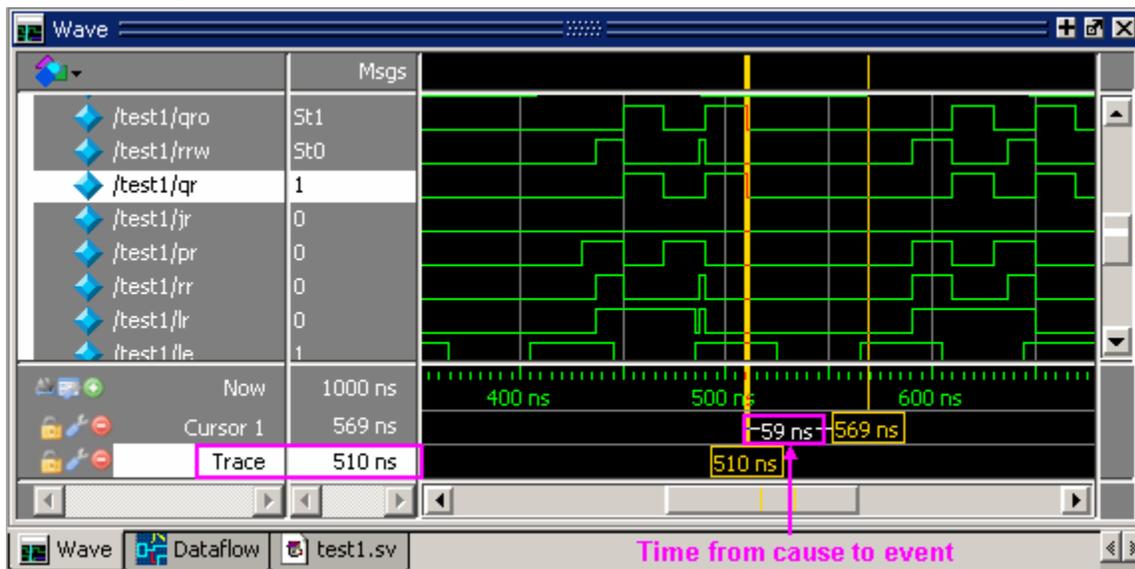
An Active Driver Path Details window also opens. This window displays information about the sequential process(es) that caused the selected event (Figure 19-3). It shows the selected signal name, the time of each process in the causality path to the first sequential process, and details about the location of the causal process in the code.

Figure 19-3. Active Driver Path Details Window



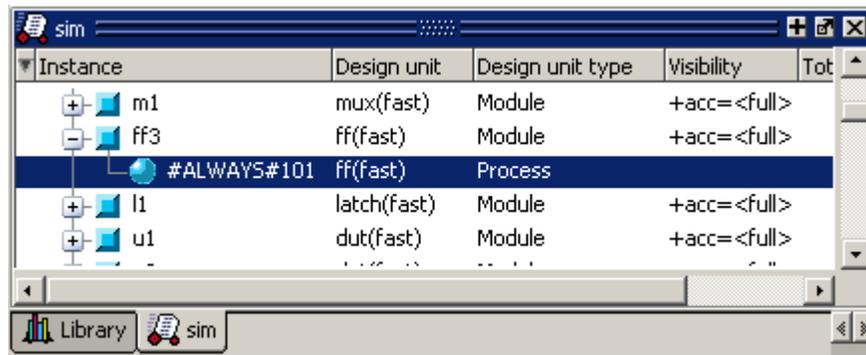
An active cursor named “Trace” is added to the Wave window at the time of the process that caused the selected event. The time from the causal process to the selected event is displayed as the relative time between the cursors (Figure 19-4).

Figure 19-4. Active Cursor Show Time of Causal Process



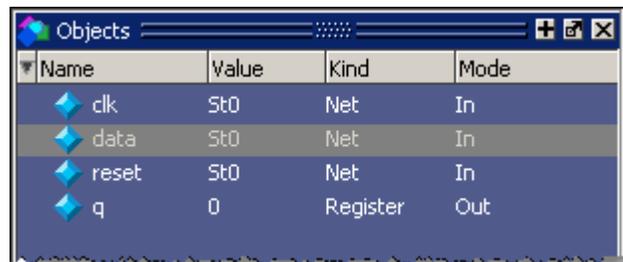
The causal process is highlighted in the Structure window (Figure 19-5).

Figure 19-5. Causal Process Highlighted in Structure Window



and the causal signal is highlighted in the Objects window (Figure 19-6).

Figure 19-6. Causal Signal Highlighted in the Objects Window



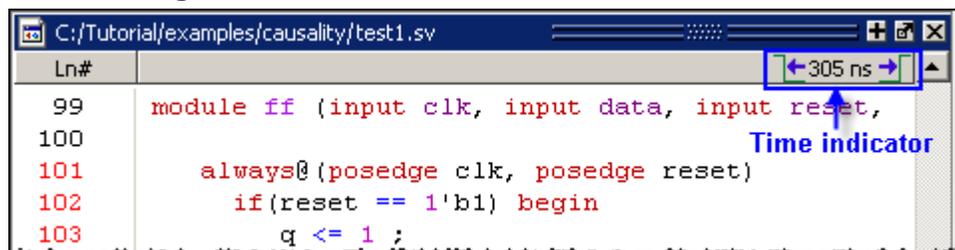
The Transcript window displays the command line equivalent of the GUI actions:

```
find drivers -source -time {<time>} -cause <signal>
```

Initiating the Trace from the Source Window

The Source window includes a time indicator in the top right corner (Figure 19-7) that displays the current simulation time, the time of the active cursor in the Wave window, or a user-designated time.

Figure 19-7. Time Indicator in Source Window

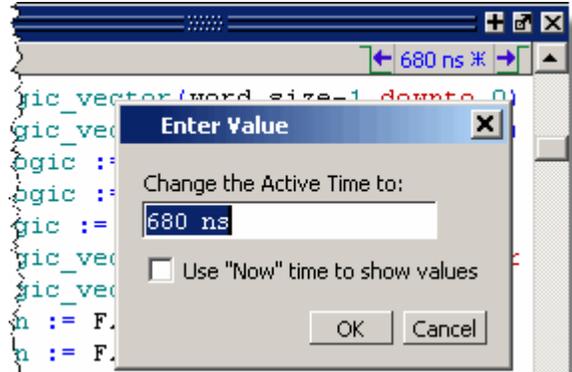


You can use this time indicator to designate a start time for a causality trace.

1. Click the time indicator to open the **Enter Value** dialog box (Figure 19-8).

2. Change the value to the starting time you want for the causality trace.
3. Click the OK button.

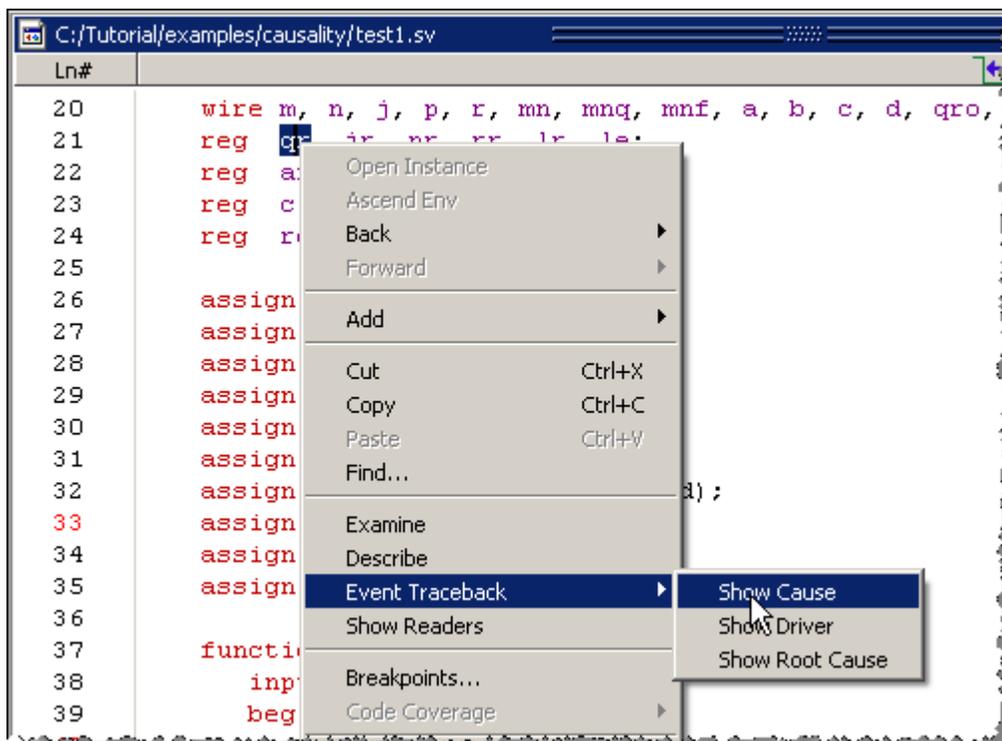
Figure 19-8. Enter an Event Time Value for Causality Tracing



To initiate a causality trace from the Source window, do the following:

1. Highlight a signal of interest in the Source window.
2. Right-click anywhere in the Source window to open a popup menu.
3. Select **Event Traceback > Show Cause** in the popup menu to initiate a causality trace. [Figure 19-9](#) shows the selection in the Source window's right-click popup menu.

Figure 19-9. Select Show Cause from Popup Menu



When the trace is complete, the Active Driver Path Details window displays all signals in the causality path, the Objects window highlights all signals in the path, and the Source window jumps to the assignment code that caused the event and highlights the code.

Initiating the Trace from the Objects or Schematics Windows

The Objects and Schematic windows use the time of the selected cursor in the Wave window, or the Time indicator in the Source window – whichever was set last – as the starting time for the event trace.

To initiate a causality trace from the Objects or Schematic windows:

1. Select a signal.
2. Perform one of the following actions:

- Click the **Event Traceback** button in the [Simulate Toolbar](#). 

Or,

- Right-click anywhere in either window and select **Event Traceback > Show Cause** from the popup menu.

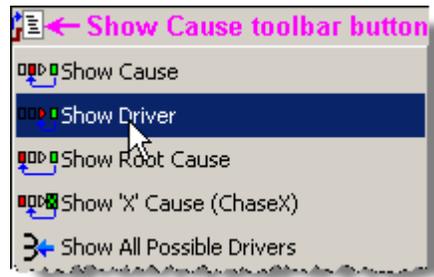
When the trace is complete, the Active Driver Path Details window displays all signals in the causality path, the Objects window highlights all signals in the path, and the Source window jumps to the assignment code that caused the event and highlights the code.

Tracing to the Immediate Driving Process

You can trace to the immediate driving process(es) using the Event Traceback toolbar button or by right-clicking the Wave window and using the popup menu. The immediate driving process may be a combinatorial or sequential assignment.

1. After the simulation, click (left mouse button) a signal of interest in the Wave window. Click on the selected signal's waveform to place an active cursor at an event of interest.
2. Perform one of the following actions:
 - Click and hold the **Event Traceback** toolbar button until a drop-down menu appears ([Figure 19-10](#)), then select **Show Driver** from the drop-down menu.

Figure 19-10. Selecting Show Driver from Show Cause Drop-Down Menu



- Right-click anywhere in the waveform pane and select **Event Traceback > Show Driver** from the popup menu (Figure 19-11). The time shown in parenthesis is the time at which the causality trace will start.

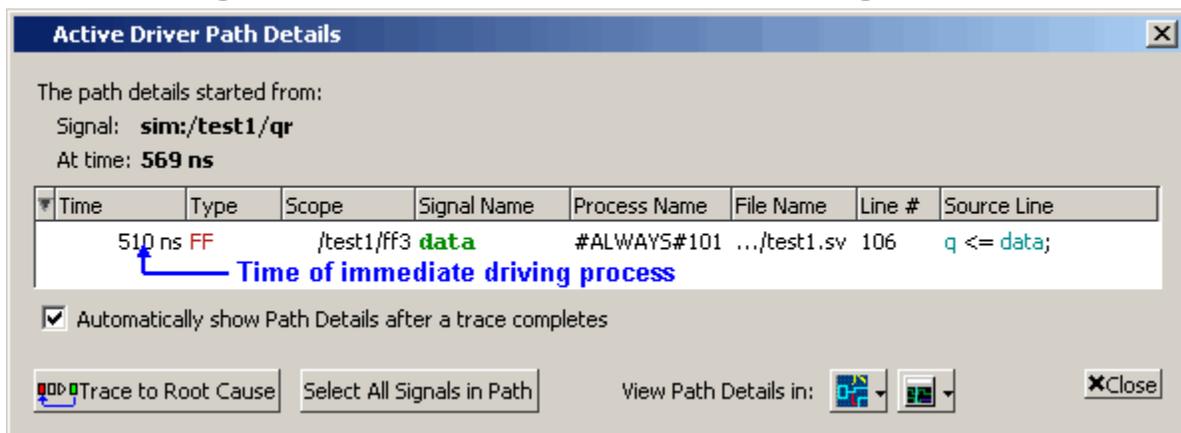
Figure 19-11. Right-click Menu – Show Driver



Causality Traceback examines the *.dbg* database for the immediate driving process(es) of the selected signal event, then opens a Source window with the code of the driving process(es) highlighted (Figure 19-2).

Selecting **Show Driver** also opens the Active Driver Path Details window (Figure 19-12). This dialog shows the selected signal name, the start time of the causality trace (At time), the time of the driving process, and details about the driving process.

Figure 19-12. Details of the Immediate Driving Process



The Transcript window displays the command line equivalent of the GUI actions.

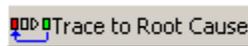
```
find drivers -source -time {<time>} <signal>
```

Tracing to the Root Cause

Causality Traceback allows you to trace an event back as far as possible - that is, to the root cause of that event. Tracing to the root cause may cross multiple clock cycles and even multiple clock domains.

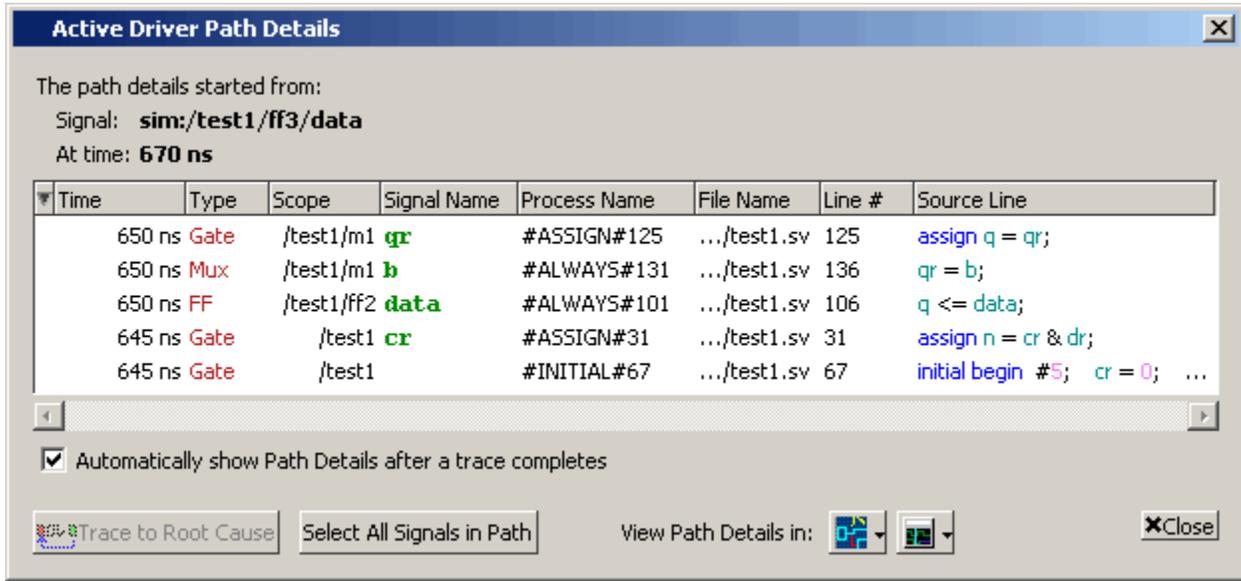
Initiate a trace to the root cause of a signal event as follows:

1. Select a signal of interest in the Wave window.
2. Click the selected signal's waveform at any point to place a cursor there. The time of this cursor is the start time of the causality trace.
3. Initiate a root cause trace using either of the following methods:
 - Right-click anywhere in the waveform pane and select **Event Traceback > Show Root Cause** from the popup menu.
 - Click and hold the Event Traceback button until the drop-down menu appears, then select **Show Root Cause** from the menu.
 - If you have performed a trace to the first sequential process (Show Cause), or a trace to the immediate process (Show Drivers), you can then initiate a trace to the root cause from the Active Driver Path Details window. Simply click the **Trace to Root Cause** button to find the root cause of the event of interest.



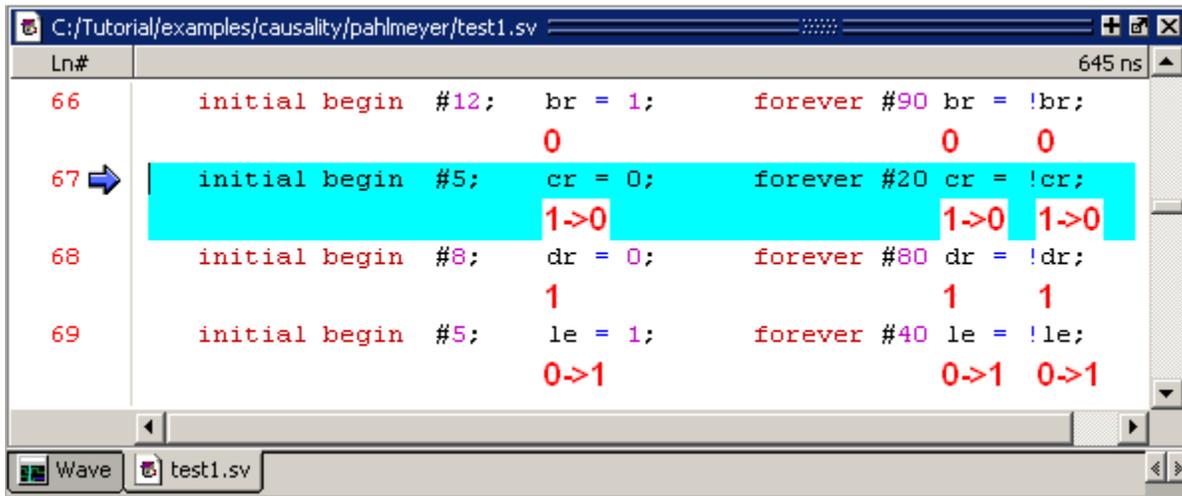
The Active Driver Path Details window displays a list of all the signals linked from the root cause to the event of interest, as shown in [Figure 19-13](#).

Figure 19-13. Trace Event to Root Cause



The Source window jumps to the root cause source code and highlights the relevant line (Figure 19-14).

Figure 19-14. Root Cause Highlighted in Source Window



The Transcript window displays the command line equivalent of the GUI actions:

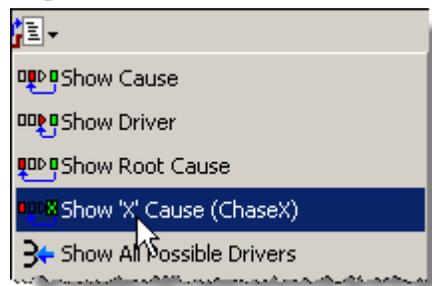
```
find drivers -source -time {<time>} -root <signal>
```

The -root switch for the `find drivers` command initiates the trace to the root cause of the selected event.

Tracing to the Root Cause of an 'X'

Causality Traceback can now be used to search for the source of an 'X' signal value. This analysis provides the automatic identification of the root cause of an 'X' over multiple cycles and clock domains. The analysis is accessed in the GUI via the "Show 'X' Cause (ChaseX)" menu pick (Figure 19-15). Or, you may use the **-chasex** switch for the **find drivers** command.

Figure 19-15. Show X Cause

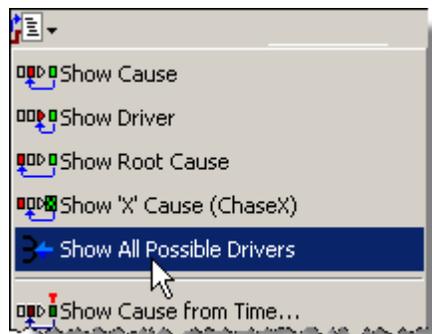


Finding All Possible Drivers

You can find and display all possible driving assignments of a selected signal by doing the following:

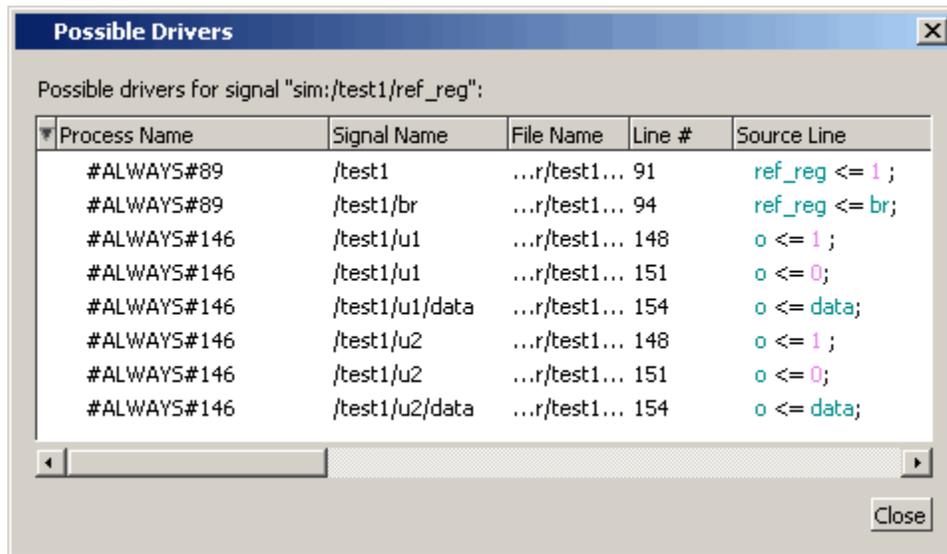
1. Select a signal of interest in the Wave window.
2. Click and hold the **Event Traceback** button until the drop-down menu appears.
3. Select **Show All Possible Drivers** from the drop-down menu (Figure 19-16).

Figure 19-16. Show All Possible Drivers Menu Selection



This action will open the Possible Drivers window and display all possible driving assignments of the selected signal (Figure 19-17) without regard to the time of any particular signal event.

Figure 19-17. Possible Drivers Window



The Transcript window displays the command line equivalent of the GUI actions:

```
find drivers -possible <signal>
```

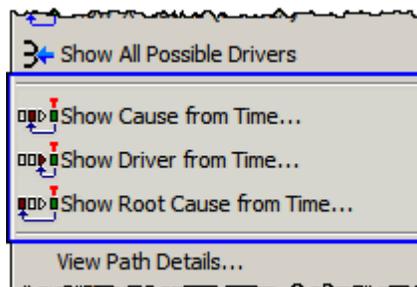
The `-possible` switch for the `find drivers` command initiates the search for all possible driving assignments of the selected signal.

Tracing from a Specific Time

The Causality Traceback feature allows you to initiate a trace to the cause of a signal event from any arbitrary time.

Click and hold the **Show Cause** button until the drop-down menu appears (Figure 19-18).

Figure 19-18. Selecting a Specific Time for a Trace



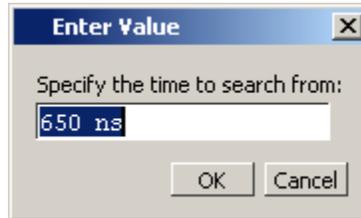
You may specify an event time and trace to:

- the first sequential process (**Show Cause from Time...**)
- the immediate driving process (**Show Driver from Time...**)

- the root cause (**Show Root Cause from Time...**)

When you make any one of these three selections, the Enter Value dialog box opens (Figure 19-19).

Figure 19-19. Enter Value Dialog Box



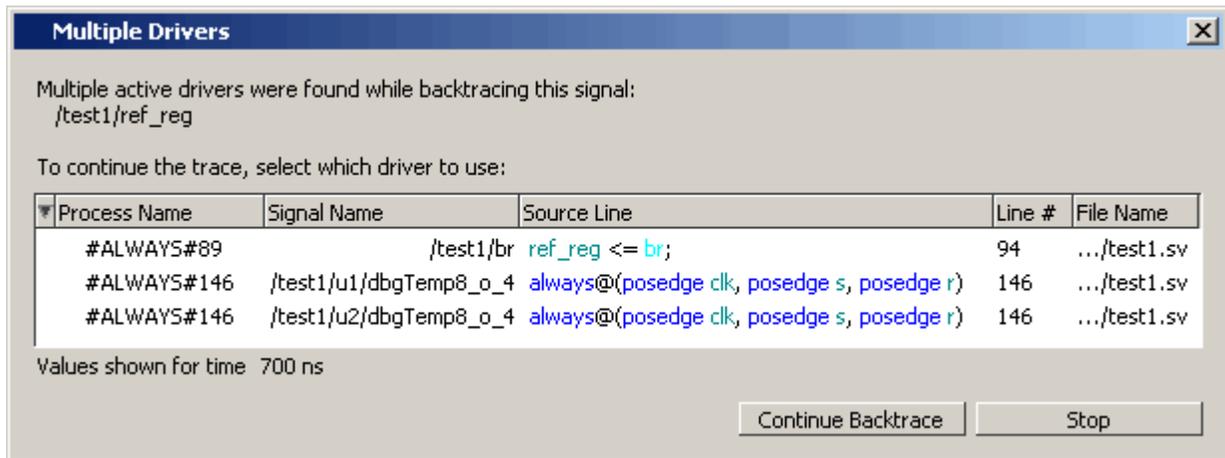
Enter a starting time for the causality trace and click the OK button.

Handling Multiple Drivers

Depending on the complexity of the design, some signal events may be driven by multiple processes. In such cases, you must make a decision about which path the causality trace will take.

If a signal event has multiple driving processes, the Multiple Drivers window opens and displays all drivers (Figure 19-20). To continue the causality trace you must select a driving process and click the **Continue Backtrace** button.

Figure 19-20. Multiple Drivers



Viewing Causality Path Details

The Active Driver Path Details window (Figure 19-3) contains two buttons for viewing causality path details – one for viewing path details in a dedicated Schematic window and one for viewing details in a dedicated Wave window (Figure 19-21).

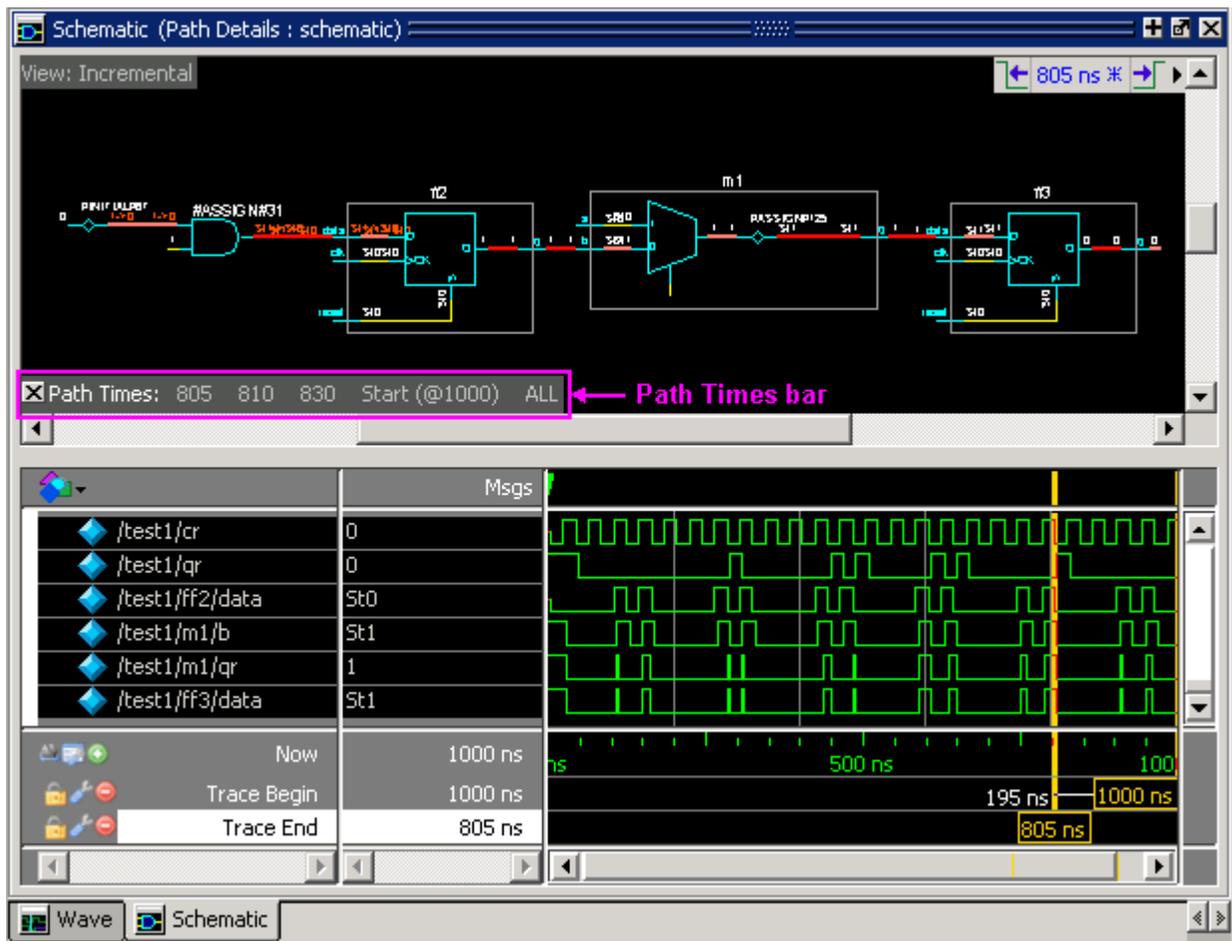
Figure 19-21. View Path Details Buttons



When you click the Schematic Window button, a dedicated **Schematic (Path Details)** window opens (Figure 19-22). It displays the causality path in the top half of the window (the schematic in the Incremental view) and lists all causality path signals in the bottom half (the Wave viewer) of the window. The causality path, from the beginning of the trace to the end, is highlighted in red in the schematic.

When you perform another causality trace, all signals contained in the path found by the new trace are added to the previous trace in the schematic.

Figure 19-22. Causality Path Details in the Schematic Window



In the Wave viewer (Figure 19-22), a cursor named “Trace Begin” marks the beginning of the causality trace; a “Trace End” cursor marks the end of the trace. In the Schematic view, the path of the trace is highlighted in red.

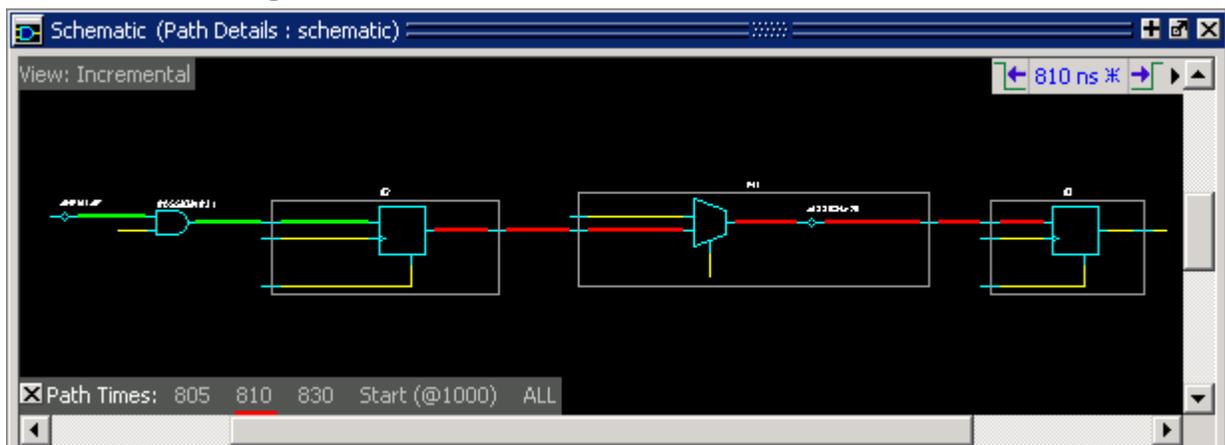
When you select a signal in the Wave viewer portion of the Schematic window, the signal is highlighted in the schematic. You can click through the signals in the Wave view to explore the connectivity of the causality path in the schematic view.

The times displayed in the Path Times bar, at the bottom left of the Schematic view, correspond to the times found during the trace. The Path Times bar also includes a “Start” and an “ALL” label. The times, Start, and ALL labels are clickable and will perform the following actions:

- Click a time to see the signals that were changing at the selected time highlighted in red. Signals that changed at a prior traced time (if one exists) are highlighted in green.
- Click **Start** to see the signal used to run the trace highlighted in red.
- Click **ALL** to see all signals identified during the trace highlighted in red.

For example, Figure 19-23 shows what happens when time “810” is selected. Signals that were changing at the selected time are red and those that changed a prior traced time are green.

Figure 19-23. Time 810 Selected in Path Times Bar



Notice that the selected time in the Path Times bar is underlined in red to emphasize that it is the selected time. In addition, the Active Time label in the upper right hand corner displays the selected time.

You can close the Path Times bar by clicking the X button in the bar. To reopen the bar, click the right mouse button to open a popup menu and select **Event Traceback > View Path Times**.

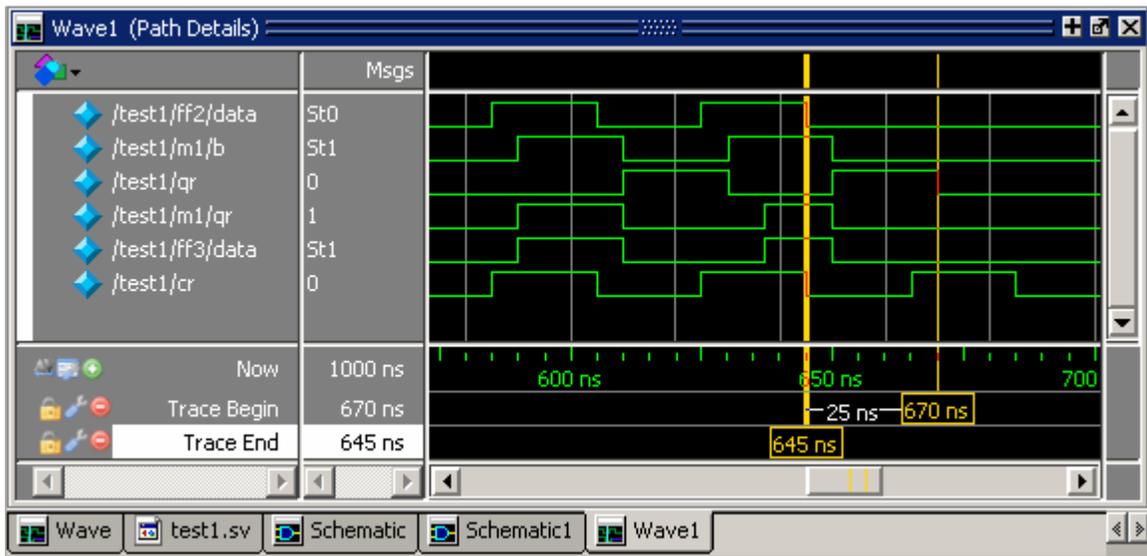
Note



The Path Times bar is only displayed in the dedicated Schematic (Path Details) window that results from a causality trace.

Returning to the Active Driver Path Details window (Figure 19-3), when you click the Wave Window button a dedicated Wave window opens that will include “(Path Details)” as part of its title (Figure 19-24). This Wave window lists each signal in the path of the causality trace and includes “Trace Begin” and “Trace End” cursors. When you perform another trace, this dedicated window is updated with signals in the path of the new trace, with updated “Trace Begin” and “Trace End” cursors.

Figure 19-24. Causality Path Details in the Wave Window



Initiating Causality Traceback from the Command Line

Causality traceback can be initiated from the command line with the [find drivers](#) command. This command can be used for current or post-simulation debugging. For the command to work properly you must:

- Use the **-debugdb** switch with [vopt](#), then the **-debugdb** switch with [vsim](#).

Or,

- If [vopt](#) is not used, simulate the design with the **-debugdb** and **-voptargs="+acc"** arguments with the [vsim](#) command.

See [Usage Flow for Causality Traceback](#).

All arguments for the [find drivers](#) command must precede the signal name. Please read the argument descriptions for more information, and refer to the [find drivers](#) command in the Command Reference for proper command syntax and complete details on how this command is used.

Setting the Report Destination with Command Line Options

The following command line options can be used with the [find drivers](#) command for setting the destination for reporting causality trace results.

Table 19-1. Setting Causality Traceback Report Destination

Command line option	Description
-transcript	Specifies that trace results are reported to the Transcript window in tabular format (unless the -compact argument is used). It is the default behavior if the -schematic, -source, or -wave options are not used.
-source	Opens the Source window with the source file that contains the line of code found by the trace. Scrolls to show that line and highlights the driving signal. If the -possible switch is used then this option is not allowed.
-wave	Specifies that all signals in the path found by the trace are added to a dedicated Wave window, and cursors are added that show the beginning and ending times of the trace. The dedicated Wave window is cleared of signals before displaying the results of a new trace.
-schematic	All signals contained in the causality path found by the trace are added into a dedicated Schematic window. The net segments connecting the beginning and ending signals are automatically selected.

Text Report Formatting in the Transcript Window

The following command line options are available for reporting causality trace results into the Transcript window. The default reporting format will display the path information, one signal per line, with each “field” contained in a separate column (aligned vertically). Certain fields within the data (i.e. parent scope, signal name) are automatically truncated to a specific number of characters controlled by the -width option. The -noclip option can be used to disable the character length check.

Table 19-2. Text Report Formatting

Command line option	Description
-compact <string>	Displays causality trace results in compact format, using a specified text string to separate the fields.
-tcl	Displays trace results in a TCL list.
-width	Specifies the maximum size of each column when data is returned to the transcript in tabular form.

Table 19-2. Text Report Formatting

Command line option	Description
-noclip	Allows columns to be arbitrarily long when returned to the transcript.
-last	Returns the results from the last completed trace to the transcript. ¹

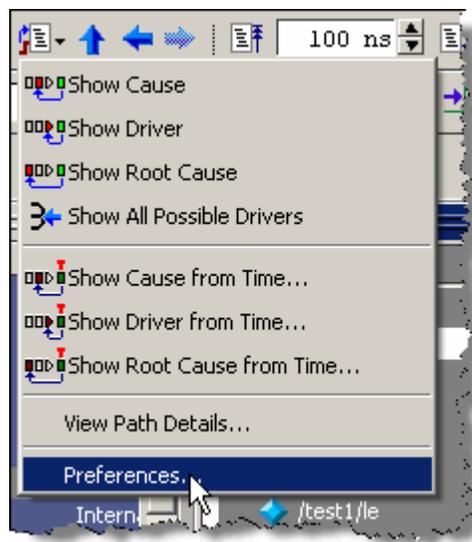
1. The -last option is useful for trying the various format options. Allows you to quickly see how each format option (-compact, -tcl, -width, and -noclip) affects the output.

Setting Causality Traceback Preferences

You can set Causality Traceback preferences as follows

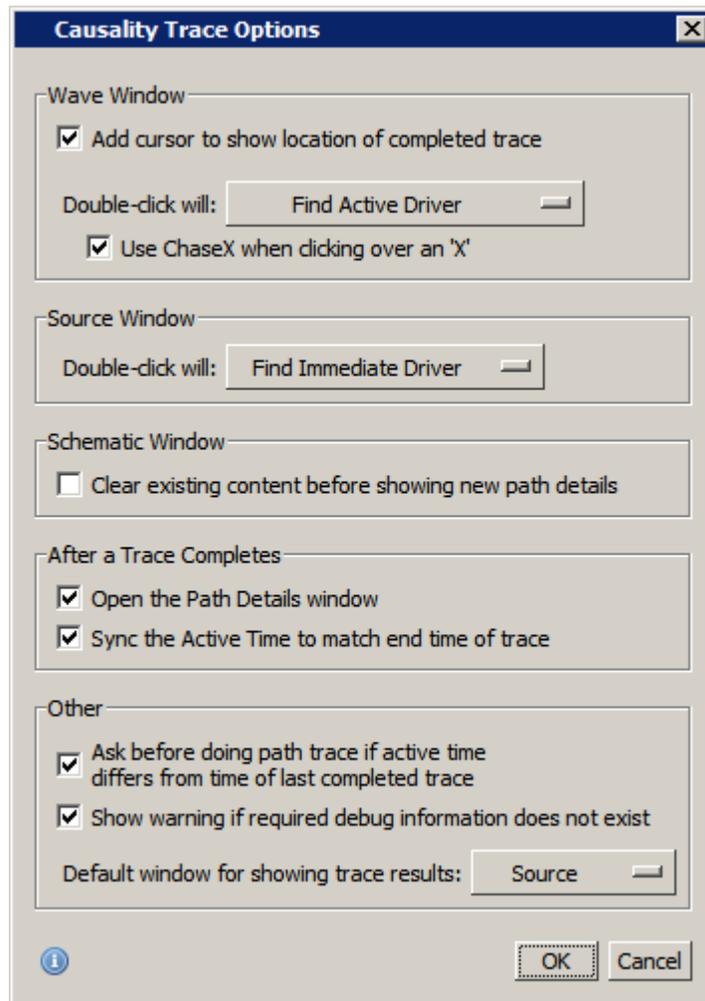
1. Click and hold the Event Traceback button until the drop-down menu opens
2. select Preferences (Figure 19-25).

Figure 19-25. Select Preferences From Event Traceback Menu



This will open the Causality Trace Options dialog box (Figure 19-26).

Figure 19-26. Causality Trace Options



Wave Window Options

- As you can see, all but one of the options are on by default. A cursor, named “Trace,” is added to the Wave window to show the location of the completed trace.
- Double-clicking a signal in the Wave Window can be set to one of the following choices:
 - Do Nothing
 - Show Drivers in Schematic
 - Show Drivers in Dataflow
 - Find Active Driver
 - Find Immediate Driver
 - Find Root Cause
 - Find All Drivers

Source Window Options

- Double-clicking a signal in the Source Window can be set to one of the following actions:
 - Find Active Driver
 - Find Immediate Driver
 - Find Root Cause
 - Find All Drivers

Schematic Window Options

- By default, new causality traces are added to existing traces in the Schematic window. Check this selection to remove existing traces before showing new causality path details.

After a Trace Completes Options

- You can choose whether or not the **Active Path Driver Details** window will automatically open after a trace is completed. To open the **Active Path Driver Details** window separately, click and hold on the **Event Traceback** button and select **View Path Details**.
- You can choose whether or not the Active Time display in the Schematic, Source, and Dataflow windows updates the Wave window cursor to move to the time at the end of a trace. You must select **Wave Window > Add cursor** in the **Causality Trace Options** dialog box to show location of completed trace selected for this to work.

Other Options

- You may elect to have Causality Traceback ask before doing a causality path trace if the active time differs from the time of the last completed trace.
- You can choose to have a warning issued if the design was simulated without the **-debugdb** option and a debugging database is not available. Refer to [Usage Flow for Causality Traceback](#) for more information about the vsim **-debugdb** option.
- You can choose the default window to show the results of a trace:
 - a. Source Window — (default) Opens with the causal process highlighted.
 - b. Schematic Window — Opens with the causal path displayed and a Wave pane showing the driving signal waveforms. Refer to [Viewing Causality Path Details](#) for more information.
 - c. Wave Window — opens a new Wave window populated with the driving signal(s) and waveforms.

Chapter 20

Code Coverage

Code coverage is the only verification metric generated automatically from design source in RTL or gates. While a high level of code coverage is required by most verification plans, it does not necessarily indicate correctness of your design. It only measures how often certain aspects of the source are exercised while running a suite of tests.

Missing code coverage is usually an indication of one of two things: either unused code, or holes in the tests. Because it is automatically generated, code coverage is a metric achieved with relative ease, obtained early in the verification cycle. 100% code coverage can be achieved even for designs containing impossible to achieve coverage (because of sections containing unused code) by using a sophisticated exclusions mechanism (see “[Coverage Exclusions](#)”). Code coverage statistics are collected and can be saved into the Unified Coverage DataBase for later analysis.

This chapter includes the following topics related to code coverage.

Overview of Code Coverage Types	900
Language and Datatype Support	900
Usage Flow for Code Coverage Collection	901
Specifying Coverage Types for Collection	902
Enabling Simulation for Code Coverage Collection	903
Saving Code Coverage in the UCDB	904
Code Coverage in the UCDB	905
Code Coverage in the Graphic Interface	906
Code Coverage Types	908
Statement Coverage	909
Branch Coverage	909
Condition and Expression Coverage	911
Toggle Coverage	924
Finite State Machine Coverage	933
Coverage Exclusions	933
What Objects can be Excluded?	933
Auto Exclusions	934
Methods for Excluding Objects	934
Toggle Exclusion Management	945
Exclude Nodes from Toggle Coverage	946
Saving and Recalling Exclusions	952
Coverage Reports	954

Notes on Coverage and Optimization 959

Note



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

Overview of Code Coverage Types

ModelSim code coverage provides graphical and report file feedback on the following:

- **Statement coverage** — counts the execution of each statement on a line individually, even if there are multiple statements in a line.
- **Branch coverage** — counts the execution of each conditional “if/then/else” and “case” statement and indicates when a true or false condition has not executed.
- **Condition coverage** — analyzes the decision made in “if” and ternary statements and can be considered as an extension to branch coverage.
- **Expression coverage** — analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage.
- **Toggle coverage** — counts each time a logic node transitions from one state to another.
- **FSM coverage** — counts the states, transitions, and paths within a finite state machine.

For details related to each of these types of coverage, see “[Code Coverage Types](#)”.

Language and Datatype Support

ModelSim code coverage supports VHDL and Verilog/SystemVerilog language constructs. Code coverage collects data separately for Verilog tasks and functions and VHDL subprograms, collectively referred to as “subprograms.” Data for subprograms is stored in the UCDB as separate regions, and is reported separately when either the [coverage report](#) or [vcover report](#) commands are used. All subprograms - Verilog tasks and functions as well as VHDL subprograms - are displayed in the Structure window of the GUI. (See [Code Coverage in the Graphic Interface](#).)

Code coverage does not work on SystemC design units.

Statement and Branch coverage have no limitations on support, however, sometimes optimizations can make it appear that statements or branches are uncovered. See “[Notes on Coverage and Optimization](#)” for more details.

For condition and expression coverage datatype support, see “[Condition and Expression Coverage](#)”.

For FSM coverage datatype support, see “[Finite State Machine Coverage](#)”.

For toggle coverage datatype support, see “[Toggle Coverage](#)”.

Usage Flow for Code Coverage Collection

To collect coverage data for a design, you must actively select the type of code coverage you want to collect, and then enable the coverage collection mechanism for the simulation run. You can view coverage results during the current simulation run or save the coverage data to a UCDB for post-process viewing and analysis. The data can be saved either on demand, or at the end of simulation (see “[Saving Code Coverage Data On Demand](#)” and “[Saving Code Coverage at End of Simulation](#)”).

Code coverage is not collected on any code that is run at elaboration time (loading the design). An example of such code might be a constant function that calculates the array range of a vector signal.

i **Tip:** Design units compiled with **-nodebug** are ignored by coverage: they are treated as if they are excluded. However, toggle coverage of ports compiled with **-nodebug** is supported **if and only if** **-nodebug** is used without any options. For example, options like “**-nodebug=ports**” will disable toggle coverage.

The basic flow for collecting code coverage in a ModelSim simulation is as follows:

1. Compile the design and specify the types of coverage to collect:

```
vlog top.v proc.v
```

```
vopt top -o opttop +cover
```

The **vlog** (or **vcom**, if design is VHDL) command compiles the specified files. The **vopt** command performs global analysis and optimizations on the design. The **-o** specifies the output name for the optimized version of the design. The **+cover** argument to the **vopt** command designates all coverage types for collection.

You may wish to apply coverage arguments differently, depending on whether you want to collect coverage for a specific source file, or just a module/sub-module, or the entire design. See “[Specifying Coverage Types for Collection](#)” for coverage application options.

2. Enable coverage collection during simulation:

```
vsim -coverage opttop
```

Coverage is enabled for the entire design using the optimized design *opttop*. See “[Enabling Simulation for Code Coverage Collection](#)” for further details.

3. Optionally, you can save the collected information for post-process viewing and analysis:

```
coverage save -onexit top.ucdb
```

This command saves the coverage data at the end of simulation, in the current directory in *top.ucdb*. See “[Saving Code Coverage in the UCDB](#)” for a list of all methods for saving data to a UCDB.

4. Run simulation with coverage enabled:

```
run -all
```

Specifying Coverage Types for Collection

When specifying the coverage types for collection, you are essentially instructing the code to collect coverage statistics when coverage collection is enabled at run time. Since extra instructions reduce simulation performance, you should only enable code for which you intend to collect coverage statistics.

You can apply coverage to:

- specific **source files** in the design — by supplying the +cover arguments to [vcom](#) or [vlog](#) during compile:

```
vlog top.v proc.v cache.v +cover=bcesfx -coveropt 1
```

- specific **modules or instance** of design — by supplying the +cover arguments to [vopt](#), using the +<selection> modifier to designate the desired design units or instances:

```
vlog top.v proc.v cache.v
```

```
vopt -o top_opt +cover=bcesxf+moduleA +cover=st+/top/proc/cache
```

```
vsim -coverage top_opt
```

- **entire design**, globally — by supplying the +cover arguments to [vopt](#):

```
vlog top.v proc.v cache.v
```

```
vopt -o top_opt +cover=bcesxf
```

```
vsim -coverage top_opt
```

or by supplying the +cover arguments using `vsim -voptargs` (when not specifically using `vopt`):

```
vlog top.v proc.v cache.v
```

```
vsim -coverage top -voptargs="+cover=bcesfx"
```

For information on the use of +cover= arguments and how the union of coverage arguments apply, see “[Union of Coverage Types](#)”.

Union of Coverage Types

For all coverage type arguments specified with “+cover=”, the arguments applied are a union of all arguments for a given module or design unit. For example, if module A was compiled with the argument +cover=xf (extended toggle and FSM) and the entire design (containing modules A, B and C) was optimized with +cover=bce, the coverage results would be:

```
Module A - bcefx
Module B - bce
Module C - bce
```

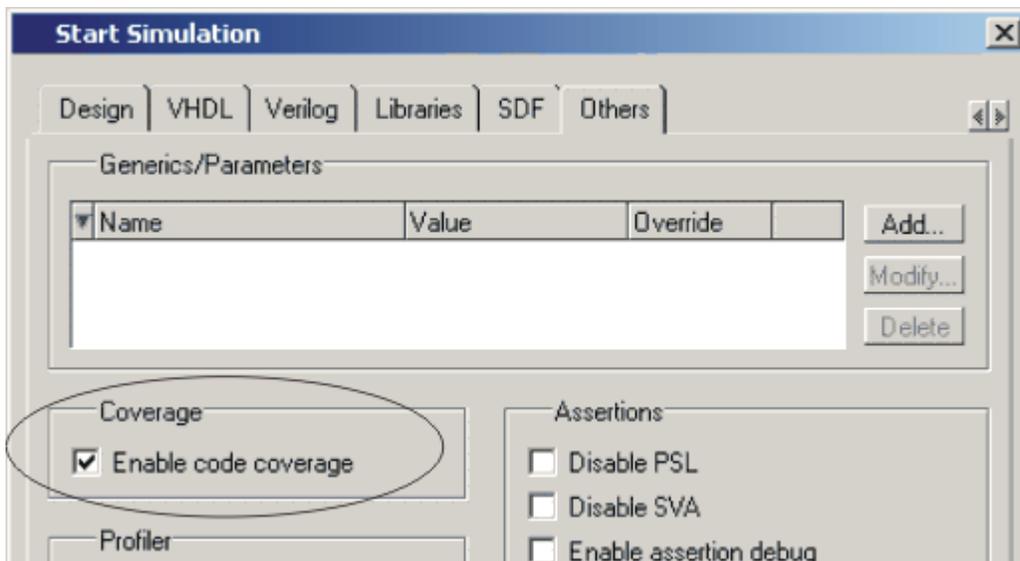
This union of arguments does not apply to the optimization number specified with -coveropt <1-5> to vcom, vlog, or vopt (see “[CoverOpt](#)” for details on -coveropt levels). In this case, any optimization level applied to a design unit or module takes precedence over the globally specified -coveropt level.

Enabling Simulation for Code Coverage Collection

Once the coverage types have been specified for coverage (“[Specifying Coverage Types for Collection](#)”), enable the simulation to collect the code coverage statistics using one of the following methods:

- CLI command: Use the **-coverage** argument to **vsim**. For example,
vsim -coverage work.top
- GUI: **Simulate > Start Simulation > Others > Enable Code Coverage** checkbox, as shown in [Figure 20-1](#).

Figure 20-1. Enabling Code Coverage in the Start Simulation Dialog



Saving Code Coverage in the UCDB

When you run a design with coverage enabled you can save the code coverage that was collected for later use, either on demand or at the end of simulation. By default, even if coverage is enabled, the tool will not save the data unless you explicitly specify that the data should be saved.

Often, users simulate designs multiple times, with the intention of capturing different coverage data from each test for post-process viewing and analysis. When this is the case, the naming of the tests becomes important. By default, the name ModelSim assigns to a test is the same as the UCDB file base name. If you fail to name the test you run explicitly, you can unintentionally overwrite your data.

To explicitly name a test **before** saving the UCDB, use a command such as:
coverage attribute -test mytestname

Saving Code Coverage Data On Demand

Options for saving coverage data dynamically (during simulation) or in coverage view mode are:

- GUI: **Tools > Coverage Save**

This brings up the Coverage Save dialog box, where you can specify coverage types to save, select the hierarchy, and output UCDB filename.

- CLI command: `coverage save`

During simulation, the `coverage save` command saves data from the current simulation into a UCDB file called *myfile1.ucdb*:

```
coverage save myfile1.ucdb
```

While viewing results in Coverage View mode, you can make changes to the data (using the `coverage attribute` command, for example). You can then save the changed data to a new file using the following command:

```
coverage save myfile2.ucdb
```

To save coverage results only for a specific design unit or instance in the design, use a command such as:

```
coverage save -instance <path> ... <dbname>
```

The resulting UCDB, *<dbname>.ucdb*, contains only coverage results for that instance, and by default, all of its children. For full command syntax, see [coverage save](#).

- SystemVerilog System Tasks and Functions (captures code coverage only):

```
$coverage_save (not recommended)  
$coverage_save_mti (not recommended)
```

The non-standard SystemVerilog `$coverage_save_mti` system function saves code coverage data only. It is not recommended for that reason. The `$coverage_save` system function is defined in the IEEE Std 1800; current non-compliant behavior is deprecated and therefore also not recommended. For more information, see “[Simulator-Specific System Tasks and Functions](#).”

Saving Code Coverage at End of Simulation

By default, coverage data is not automatically saved at the end of simulation. To enable the auto-save of coverage data, set a legal filename for the data using any of the following methods:

- Set the *modelsim.ini* file variable: `UCDBFilename=<filename>`

By default, `<filename>` is an empty string (`""`).

- Specify at the `Vsim>` prompt: **coverage save -onexit** command

The **coverage save** command preserves instance-specific information. For example:

```
coverage save -onexit myoutput.ucdb
```

- Execute the SystemVerilog command:

```
$set_coverage_db_name(<filename>)
```

If more than one method is used for a given simulation, the last command encountered takes precedence. For example, if you issue the command **coverage save -onexit vsim.ucdb**, but your SystemVerilog code also contains a `$set_coverage_db_name()` task, with no name specified, coverage data is not saved for the simulation.

Code Coverage in the UCDB

ModelSim stores saved coverage statistics in a Unified Coverage DataBase (UCDB) file, a single persistent database that is the repository for all coverage data — both code coverage and functional coverage.

Once the UCDB coverage data is saved, you can:

- Analyze coverage statistics in the GUI, either interactively with an active simulator, or in a post-processing mode with `vsim -viewcov` (see “[Usage Flow for Code Coverage Collection](#)”)
- Run and view reports on the collected code coverage data (see “[Coverage Reports](#)”)
- Exclude certain data from the coverage statistics (see “[Methods for Excluding Objects](#)”)
- View, merge, and rank sets of code coverage data without elaboration of the design or a simulation license. You can also merge test data with a verification plan.

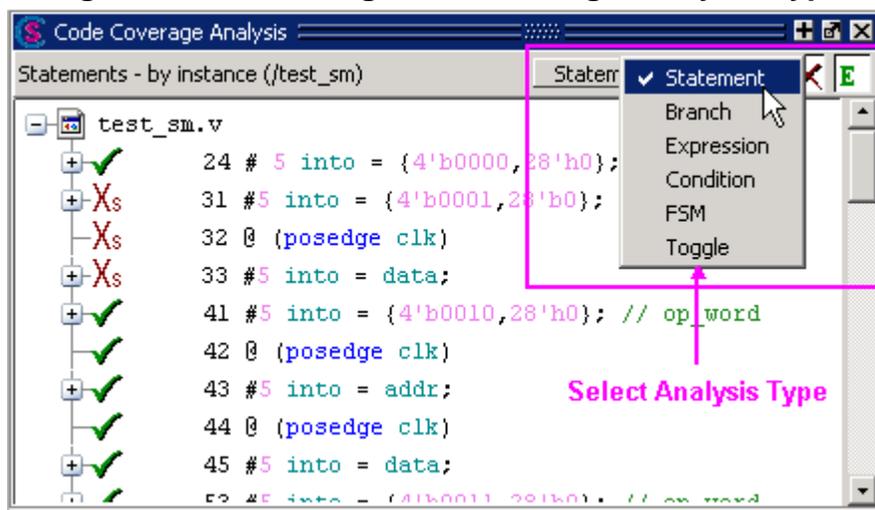
For information on working with both functional coverage and code coverage in the verification of your design, see the “[Coverage and Verification Management in the UCDB](#)” chapter.

For coverage aggregation details, see “[Calculation of Total Coverage](#)”.

Code Coverage in the Graphic Interface

When you simulate a design with code coverage enabled, coverage data is primarily displayed in the Code Coverage Analysis, Instance Coverage, and Coverage Details windows. In the Coverage Analysis window you can elect to display Statement, Branch, Expression, Condition, FSM, or Toggle coverage by clicking the Analysis Type selector ([Figure 20-2](#)).

Figure 20-2. Selecting Code Coverage Analysis Type



The coverage data in the Code Coverage Analysis window is displayed “by instance” or “by file” depending on whether the “sim” tab (Structure window) or “Files” tab is active.

Additional Code Coverage Data is displayed in the Object, Source, and Structure windows. To view coverage data in the Objects window, right click anywhere in the column title bar and select **Show All Columns** from the popup menu. When you double-click an item in the Code Coverage Analysis window or the Objects window, it will open a Source window with the selected item highlighted. For details, see [Table 20-1](#).

All subprograms - Verilog tasks and functions as well as VHDL subprograms - are displayed in the Structure window. Since VHDL allows multiple subprograms of the same name but different arguments, in the same hierarchical scope, the argument signature (list of arguments and their types) is displayed along with the subprogram name in order to differentiate overloaded subprogram names. The signature appears in the “design unit” column instead of the design unit name.

The Instance Coverage window also displays subprograms, as well as instances, and their respective code coverage data.

You can also write coverage statistics in different text and HTML reports (see “[Coverage Reports](#)”). You can save raw coverage data to a UCDB (see “[Code Coverage in the UCDB](#)”) and recall, or merge it with coverage data from previous simulations.

The table below summarizes the coverage windows.

Table 20-1. Code Coverage in Windows

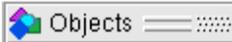
Coverage window	Description
Code Coverage Analysis	<p>Use this window to perform in-depth analysis of incomplete coverage numbers.</p> <p>Pulldown menu options for viewing missed coverage and details: Statement Analysis, Branch Analysis, Condition Analysis, Expression Analysis, Toggle Analysis, FSM Analysis.</p> <p>Displays exclusions with or without comments, missed coverage (anything with less than 100% coverage) for the selected design object or file, as well as details for each object. When the Details window is open, you can click on each line to display details of object.</p> <p>See “Code Coverage Analysis Window”.</p>
Details 	<p>Displays details of missed statement, branch, condition, expression, toggle, and FSM coverage, as well as exclusions and comments. When you select items in Code Coverage Analysis windows, the details populate in this window. Used to perform in-depth analysis of incomplete coverage numbers. See “Coverage Details Window”.</p>
Instance Coverage 	<p>Use this window as the primary navigation tool when exploring code coverage numbers.</p> <p>Displays coverage statistics for each instance. It recursively shows all child instances under the currently selected region in the Structure window. Use this window for analysis based on sorting by coverage numbers. See “Instance Coverage Window”.</p>
Objects 	<p>Can be used to view and analyze Toggle Coverage.</p> <p>Displays toggle coverage statistics when you right-click any column heading and select Show All Columns. Various columns show the toggle numbers collected for each variable and signal shown in the window. See “Viewing Toggle Coverage Data in the Objects Window”.</p>
Source 	<p>Most useful for statement and branch coverage analysis.</p> <p>Displays source code for covered items. See “Coverage Data in the Source Window”.</p>

Table 20-1. Code Coverage in Windows

Coverage window	Description
Structure (sim) 	Use this window mainly as a design navigation aid. Displays coverage data and graphs for each design object or file, including coverage from child instances compiled with coverage arguments. By default, the information is displayed recursively. You can select to view coverage by local scopes only by deselecting Code Coverage > Enable Recursive Coverage Sums . Columns are available for all types of code coverage. See “ Code Coverage in the Structure Window ” and “ Coverage Aggregation in the Structure Window ”.

Understanding Unexpected Coverage Results

When you encounter unexpected coverage results, it may be helpful to keep in mind the following special circumstances related to collecting coverage statistics:

- Optimizations affect coverage results. See “[Notes on Coverage and Optimization](#)”.
- Poorly planned or executed merges can produce unexpected results. One example: if you improperly apply the `-strip` and `-install` options of coverage edit).
- Package bodies, whether VHDL or SystemVerilog, are not instance-specific: ModelSim sums the counts for all invocations no matter who the caller is.
- All standard and accelerated VHDL packages are ignored for coverage statistics calculation.
- You may find that design units or instances excluded from code coverage will appear in toggle coverage statistics reports. This happens when ports of the design unit or instance are connected to nets that have toggle coverage turned on elsewhere in the design.
- Verilog cells (modules surrounded by ``celldefine / `endcelldefine`, and modules found using `vlog -y` and `-v search`) do NOT have code coverage enabled by default. In addition, coverage for cells that have been optimized will not appear in reports. For more information, refer to the `-covercells` arguments of the `vlog` command.

Code Coverage Types

Code coverage types are:

- [Statement Coverage](#)
- [Branch Coverage](#)
- [Condition and Expression Coverage](#)
- [Toggle Coverage](#)

- [Finite State Machine Coverage](#)

Statement Coverage

Statement coverage is the most basic form of coverage supported by ModelSim. The metric for statement coverage is the count of how many times a given statement is executed during simulation. Multiple statements may be present on a single line of HDL source code. Each such statement is processed independently of other statements on the same line. Statement coverage counts are prominently displayed in the Source window. They are present in most of the other coverage windows as well.

Statement coverage statistics for “for” loops are presented using two separate entries relating to one line of code: the first entry is the number of times the “for” statement was entered, while the second is the number of times the loop was repeated. Consider the following statement displayed in a coverage report:

```
31          1    ***0***          for i in SETS-1 downto 0 loop
31          2    ***0***
```

The statement on line 31 displays counts in two entries: the count “1” refers to how many times the loop was entered, the count “2” refers to how many times the loop was repeated.

Branch Coverage

Branch coverage is related to branching constructs such as “if” and “case” statements. True branch and “AllFalse” branch execution are measured (see “[AllFalse Branches](#)”). In order to achieve 100% branch coverage, each branching statement in the source code must have taken its true path, and every AllFalse branch must have been taken.

Verilog and SystemVerilog ‘if ... else if ... [else]’ chains are analyzed with a coverage model as shown in the example below.

Example 20-1. Branch Coverage

```
module top;
integer i=10;
initial begin
    #3 i = 18;
    #3 i = 2;
    #1 $finish();
end
always @ (i) begin
    if (i == 16)
        $display("sweet");
    else if (i == 2)
        $display("terrible");
    else if (i == 10)
        $display("double digits at last");
    else if (i == 18)
```

```

        $display("can vote");
    else
        $display("just another birthday"); end endmodule

```

When this example is run to completion and the branch coverage is collected and saved to *top.ucdb*, the “vcover report top.ucdb -details” command produces the following report:

Example 20-2. Coverage Report for Branch

```

Coverage Report by file with details

File: top.v
Branch Coverage:
  Enabled Coverage      Active      Hits      Misses % Covered
  -----
  Branches              5          3          2       60.0

=====Branch Details=====

Branch Coverage for file top.v --

-----IF Branch-----
   12                      3      Count coming in to IF
   12          1          ***0***      if (i == 16)
   14          1          1          else if (i == 2)
   16          1          1          else if (i == 10)
   18          1          1          else if (i == 18)
   20          1          ***0***      else
Branch totals: 3 hits of 5 branches = 60.0%

```

The 60% coverage number is derived, in that five bins under the initial 'if' have been inferred — 1 'if' branch, 3 'else if' branches, and 1 'else' branch. Three of these branches were executed.

If the final 'else' had not been present in the example, the coverage score would remain the same, but instead of listing an 'else' count, the report would list an 'All False Count' value of 0.

Case and Branches

For “case” statements, the case expression itself is not considered a branch. Rather, each case item is considered a separate and independent branch. To achieve 100% branch coverage in a “case” statement, each case item must have been executed during simulation.

In order to gain more coverage detail on the HDL expressions used in branching statements, the Condition and Expression Coverage features may be used (see “[Condition and Expression Coverage](#)”).

AllFalse Branches

For “if” statements without a corresponding “else”, an implicit branch known as the “AllFalse” branch is measured. If the “if” condition is executed and found to be false, the all false branch is considered to be hit. In the following VHDL example,

```
if (fsel = "10") then
    z <= a;
elsif (fsel = "11") then
    z <= b;
end if;
```

the AllFalse branch is hit when “fsel” is not equal to “10” or “11”. In the following Verilog example:

```
if (fsel == INDF_ADDRESS) begin
    fileaddr <= fsr[6:0];
end
```

the AllFalse branch is hit when “fsel” is not equal to INDF_ADDRESS.

You can exclude an AllFalse branch from participation in branch coverage using the **-allfalse** argument to a pragma exclusion or the [coverage exclude](#) command. See “[Exclude Implicit \(AllFalse\) Branches](#)” and “[coverage on and coverage off Pragma Syntax](#)” for further details.

Code coverage includes and all false bin for Verilog *case* statements that do not contain a “default” clause. Coverage reports and the GUI will show the *case* all false data similar to the way *if* statement all false data is shown. Case statement all false branches can be excluded also in a similar manner. (See [Exclude AllFalse Branches in Case Statements](#).) Also, if the [CoverExcludeDefault](#) variable in the *modelsim.ini* file is used to exclude case statement default clauses, it will also exclude case statement all false branches.

Missing Branches in VHDL and Clock Optimizations

In cases where you have a VHDL process serving as an edge-triggered flip-flop, the default ModelSim optimizations convert this process to an optimized process that is only activated on the rising edge of the clock. Because this process is NEVER activated on the falling edge of the clock, that branch is not executed and, thus, not counted for code coverage. Because of this, the code coverage algorithm excludes the branch for code coverage. The exclusion is reported in the Source window, with a special indicator showing that the branch is excluded for clock optimization.

You can turn off clock optimization in VHDL code by compiling the design with the [CoverOpt](#) *modelsim.ini* variable or the `vcom/vlog -coveropt` argument set to 2.

Condition and Expression Coverage

Condition coverage analyzes the decision made in "if" and ternary statements and can be considered an extension to branch coverage.

Expression coverage is similar to condition coverage: it analyzes the activity of expressions on the right-hand side of assignment statements, and counts when these expressions are executed.

For expressions involving logical operators, a truth table is constructed and counts are tabulated for conditions matching rows in the truth table.

i **Tip: Not Covered** — Expressions whose result is greater than one bit wide are not counted for coverage; they are silently ignored.

By default, only the collection of FEC style metrics is enabled (see “[Reporting Condition and Expression Coverage](#)”), though several condition and expression coverage metrics can be calculated and presented using ModelSim. Use them accordingly, to suit your purposes:

- **Focused Expression Coverage (FEC)** — A row based coverage metric which emphasizes the contribution of each expression input to the expression’s output value.

FEC measures coverage for each input of an expression. If all inputs are fully covered, the expression has reached 100% FEC coverage. In FEC, an input is considered covered only when other inputs are in a state that allow it to control the output of the expression. Further, the output must be seen in both 0 and 1 states while the target input is controlling it. If these conditions occur, the input is said to be fully covered. The final FEC coverage number is the number of fully covered inputs divided by the total number of inputs. As of ModelSim 10.0, FEC is fully compliant with the more widely known MC/DC coverage metric (Multiple Condition/Multiple Decision). See “[FEC Coverage Detailed Examples](#)” for further details on report output and analysis.

- **User Defined Primitive (UDP)** — The term is borrowed from the Verilog language, which uses the same basic table format to model user-defined primitives. Coverage for UDP is enabled through the use of `vcom/vlog/vopt -coverudp`.

A UDP table describes the full range of behavior for a given expression. Each row corresponds to a coverage bin. If the conditions described by a row are observed during simulation, that row is said to be hit. All rows in the UDP table must be hit for UDP coverage to reach 100%. Row minimization is attempted by use of wildcard matches. See “[UDP Coverage Details and Examples](#)” for further details on UDP analysis.

- **Sum-of-Products** — based on UDP data

Sum-of-Products checks that each set of inputs that satisfies the expression (results in a “1”) must be exercised at least once, but not necessarily independently.

- **Basic Sub-Condition** — based on UDP data

Basic sub-condition checks that each subexpression has been both true and false.

Effect of Short-circuiting on Expression and Condition Coverage

By default, the simulator follows LRM rules for short-circuit evaluation for Verilog and VHDL expressions. In brief, for Verilog, the `&&`, `||`, and ternary operators short-circuit. And for VHDL,

expressions are short-circuited when their operands are of boolean or bit types, and the expression is purely composed of logical operators.

For example, in the following expression, if A has a value of '0', the term B || C will never be evaluated:

```
Z <= A && (B || C);
```

Short-circuit evaluation remains in effect per LRM rules when coverage is enabled.

You may want to analyze the coverage results when short-circuit evaluation is turned off, and all terms in an expression are considered in an evaluation. To achieve this effect, use the **-nocovershort** argument to vlog/vcom/vopt. Generally, expression and condition coverage percentages are lower when short-circuit evaluation is active, since, on average, fewer inputs are considered when evaluating expressions and conditions.

A brief short-circuit status is given in the Details window and the text coverage report for each condition and expression.

Reporting Condition and Expression Coverage

FEC / UDP:

- By default, when coverage is enabled with the “+cover=ec” argument (where “=ec” enables expression and condition coverage) to vcom/vlog or vopt, only FEC coverage statistics are collected and reported.
 - You can turn on/off FEC collection using the **-coverfec** and **-nocoverfec** argument to vcom/vlog, or vopt.
 - You can turn on/off UDP collection (off by default) using the **-coverudp** and **-nocoverudp** argument to vcom/vlog, or vopt.
- You can print condition and expression coverage truth tables in coverage reports by:
 - GUI: Select **Coverage Reports > Condition Coverage** or **Expression Coverage** (see “[Coverage Reports](#)”)
 - Command Line: with [coverage report -details](#). Works when one or more of the rows has a zero hit count. To force the table to be printed even when an expression is 100% covered, apply the **-all** switch to the coverage report command.
- Detailed analysis metrics are reported using a command such as:

[coverage report -details](#)

or

[vcover report -details](#)

- The default maximum limit for the number of rows allowed in a table is 192. You can customize the limit for FEC / UDP using the `vlog` or `vcom -maxfecrows / -maxudprows` arguments, or the `FecEffort` `modelsim.ini` file variable.

Sum-of-Products / Basic Sub-Condition:

- The Sum-of-Products and Basic Sub-Condition calculations are based on the UDP data and can be reported in detail using a command such as:

```
vcom/vlog -coverudp
```

```
then
```

```
coverage report -details -metricanalysis
```

```
or
```

```
vcover report -details -metricanalysis
```

FEC Coverage Detailed Examples

Following are examples detailing the default coverage (FEC) report tables.

Example 20-3. FEC Coverage - Simple Expression

Let's examine the following FEC report table for the expression (a & b & c) when it receives input vectors {101, 011, 111}:

```
# -----Focused Expression View-----
# Line      31 Item    1    #1 tempreg1 <= (a & b & c);
# Expression totals: 2 of 3 input terms covered = 66.6%
#
#   Input Term   Covered   Reason for no coverage   Hint
#   -----
#           a           Y
#           b           Y
#           c           N   '_0' not hit                Hit '_0'
#
#   Rows:       Hits   FEC Target           Matching input patterns
#   -----
#   Row  1:           1   a_0                   { 011 }
#   Row  2:           1   a_1                   { 111 }
#   Row  3:           1   b_0                   { 101 }
#   Row  4:           1   b_1                   { 111 }
#   Row  5:   ***0***   c_0                   { 110 }
#   Row  6:           1   c_1                   { 111 }
#
# NOTE:
# * Order of matching input pattern values: {a,b,c}
```

Each FEC report consists of two tables;

- The first table reports coverage on a per-input basis. For inputs that are not covered, the report gives a brief reason for the lack of coverage. The “Hint” column provides

information on how to get the input covered. In the FEC report above, input 'c' was not covered because the coverage bin '_0' associated with this input (i.e. c_0) did not receive any hits. The hint says that to get 'c' FEC covered, an input pattern matching c_0 (i.e. {110}) must be applied to this expression during simulation. Matching input patterns are always strings of 1's and 0's separated by whitespace.

- The second table goes a step deeper and expands each input into its coverage bins. The table lists the Rows, Hits, FEC Target and Matching input patterns. The matching input patterns are always strings of 1's and 0's separated by whitespace.

In the FEC report above, consider the first row containing the FEC Target (or bin) of a_0: where a is the input and _0 is the value of that input. The full tag of a_0 indicates that this row delivers FEC testing when a's value is 0. This bin was incremented 1 time, since the input vector {011} was seen. By definition a is 0 for every input vector on the a_0 list. Similarly, the input vector for the a_1 list - row 2 in the table - was observed once. Again, by definition, the a_1 list vectors are identical to the a_0 list except with the 'a' bit equal to 1. This is always the case for each pair of FEC rows (non-short circuit logic only).

Walking through the truth table in this way, one can see how FEC ensures that each input a, b, and c has been shown to independently affect the expression output. For example, for the conditions of FEC to be satisfied, when an a_0 input vector flips to the corresponding a_1 vector - i.e., only bit 'a' changes to 1, with the other bits unchanged - the output value of the expression MUST also change.

In effect, this type of coverage metric can help determine if there is a functional bug in the logic that is feeding the targeted input (FEC Target). It is a powerful tool in that it helps minimize the risk that an expression is masking potential bugs in the logic feeding each of its inputs.

If FEC coverage indicates any bins are missed (such as c_0 in Row 5 of [Example 20-3](#)) you know that none of your tests ever produced a value of '1' when other inputs are in a state that allow it to control the output. You should then work on the design/stimulus to improve FEC coverage. One method of raising FEC coverage numbers is to modify test stimulus such that appropriate patterns appear at the expression's inputs. The matching input vectors in the report can help in this process.

Figure 20-3. Focused Expression Report Sample

```
# -----Focused Expression View-----
# Line      28 Item   1   #1 tempreg2 <= (~a & c) | (a & b & ~c & d) | (a & ~b & c & d);
# Expression totals: 3 of 4 input terms covered = 75.0%
#
#   Input Term   Covered   Reason for no coverage   Hint
# -----
#           a           Y
#           c           Y
#           b           N   '_0' hit but '_1' not hit   Hit '_1' for output ->0 or ->1
#           d           Y
#
# Rows:   Hits(->0)   Hits(->1)   FEC Target   Matching input patterns(->0)   Matching input patterns(->1)
# -----
# Row 1:           0           1   a_0           { 0011 }           { 011- 01-0 }
# Row 2:           E           E   a_1           { 111- 11-0 }           { 1011 }
# Row 3:           1           0   c_0           { 00-- 1001 }           { 1011 }
# Row 4:           0           2   c_1           { 1111 }           { 01-- 1101 }
# Row 5:           1           1   b_0           { 1001 }           { 1101 }
# Row 6:           0           0   b_1           { 1111 }           { 1011 }
# Row 7:           E           -   d_0           { 1010 1100 }           -
# Row 8:           -           1   d_1           -           { 1011 1101 }
#
# NOTE:
# * Order of matching input pattern values: {a,c,b,d}
```

A Deeper Look at the Theory of FEC

A given expression input can operate in an inverting or non-inverting mode. When the value of an expression input is '0', with all other terminals in their quiescent states and the output at '1', the input is said to be operating in an inverting mode. Similarly, when the value of an input is '1', with all other inputs in their quiescent states and the output at '1', the input is said to be operating in a non-inverting mode.

An expression can be categorized as unimodal or bimodal. If each expression input only ever operates in one mode, that expression is said to be a unimodal expression. If at least one input can operate in both inverting and non-inverting modes, that expression is said to be a bimodal expression.

A classic example of a unimodal expression is an 'AND' gate:

Table 20-2. AND Gate

A	B	A&B	FEC Row
0	0	0	--
0	1	0	a_0
1	0	0	b_0
1	1	1	a_1, b_1

Consider input 'a'. FEC rows with a_0 only ever result in an expression output of '0'. FEC rows with a_1 only ever result in an output of '1'. Similar holds for b_0 and b_1. Therefore this is a unimodal expression with all inputs operating permanently in non-inverting mode. It isn't hard

to extrapolate that NAND's are unimodal expressions with all inputs operating permanently in inverting mode.

Now let's look at an 'XOR' gate:

Table 20-3. XOR Gate

A	B	A&B	FEC Row
0	0	0	a_0, b_0
0	1	1	a_0, b_1
1	0	1	a_1, b_0
1	1	0	a_1, b_1

Consider input 'a'. There is an FEC row with a_0 which evaluates the expression to '0', and a different FEC row with a_0 that evaluates the expression to '1'. The same holds for a_1, b_0, and b_1. Therefore, both inputs of this expression operate in inverting and non-inverting modes. This is a classic case of a bimodal expression.

To determine if a bimodal input is FEC covered, it must have been shown to independently control the output from within one mode. This implies the output must change if the bimodal input changes while all other inputs are seen in a quiescent state. This algorithm avoids a false coverage hit on the XOR when 'a' and 'b' transition simultaneously, e.g. {11} -> {00}. Only transitions {11} -> {01} and {10} -> {00} will result in a_0 being FEC covered. For input 'a' to be fully FEC covered, both a_0 and a_1 must be FEC covered. Note that during transition {11} -> {00}, input 'a' switches from inverting mode to non-inverting mode. Hence the transition does not count towards FEC coverage for a_0.

Example 20-4. FEC Coverage - Bimodal Expression

Let's examine the following FEC report table for the expression ((~a & c) | (a & b & ~c & d) | (a & ~b & c & d)) when it receives input vectors {0100, 1001, 1111}.

```
# -----Focused Expression View-----
#
# Line      28 Item   1   #1 tempreg2 <= (~a & c) | (a & b & ~c & d) | (a & ~b &
# c & d);
# Expression totals: 3 of 4 input terms covered = 75.0%
#
#
# Input Term   Covered   Reason for no coverage                               Hint
# -----
#             a           Y
#             c           Y
#             b           N   '_0' hit but '_1' not hit           Hit '_1' for output
->0 or ->1
#             d           Y
#
#
#Rows: Hits(-->0) Hits(-->1) FEC Target Matching input patterns(-->0) Matching
input patterns(-->1)
#-----
```

```

# Row 1:      0      1  a_0      { 0011 }      { 011- 01-0 }
# Row 2:      E      E  a_1      { 111- 11-0 }      { 1011 }
# Row 3:      1      0  c_0      { 00-- 1001 }      { 1011 }
# Row 4:      0      2  c_1      { 1111 }      { 01-- 1101 }
# Row 5:      1      1  b_0      { 1001 }      { 1101 }
# Row 6:      0      0  b_1      { 1111 }      { 1011 }
# Row 7:      E      -  d_0      { 1010 1100 }      -
# Row 8:      -      1  d_1      -      { 1011 1101 }
#
# NOTE:
# * Order of matching input pattern values: {a,c,b,d}
#

```

As in the simple [Example 20-3](#), the first table reports coverage on a per-input basis. In the FEC report above, input 'b' was not covered because both rows corresponding to this input were hit for the same output value (i.e. 'b' changed but the output didn't change).

The second table expands each input into its coverage bins. In the FEC report above, consider the first row containing the FEC Target (or bin) of a_0, where a is the input and _0. The hits and matching input patterns have been divided based on the output of the expression when applying the input pattern. This is done to ensure that while qualifying an input terminal as FEC covered, it has been shown to independently control the output while operating in one mode, i.e. making sure that it receives '_0' and '_1' hits for different output values.

The bin corresponding to 'a_0' was incremented 1 time for output value 1, as one input vector was seen from the list of matching patterns for output value 1 { 011- 01-0 }. Similarly, the input vector for the a_1 list - row 2 in the table - was observed once for output 0. Again, by definition, the a_1 list vectors are identical to the a_0 list except with the 'a' bit equal to 1. This is always the case for each pair of FEC rows for non-short circuit logic. Since input 'a' receives hits in both the '_0' and '_1' rows for different output values, 'a' is considered 100% FEC covered.

Even though input 'b' receives hits in both '_0' and '_1' rows, it is not considered FEC covered. This is because both the rows are hit for the same output value (i.e. 0). In cases where an input is not FEC covered, use the reason and hint to improve the test stimulus, or potentially modify the design if a design issue is found. For example, in this particular case, one method of raising FEC coverage numbers is to modify the test stimulus such that pattern {1101} (matching input pattern for b_0 for output 1) or {1011} (matching input pattern for b_1 for output 1) appear at the expression's inputs during simulation.

Note that input 'd' in this expression is a unimodal input. The '-' characters in the output value columns represent values that are impossible to hit. Similarly holds for the '-' characters in the Matching Input Pattern values column. (Consider the 'AND' gate: it is impossible to hit output value ->1 for a_0).

Observe the NOTE: at the bottom of the report. Matching input patterns are binary strings of 1's and 0's, separated by whitespace. It can be difficult to tell which binary value corresponds to which input. The NOTE: provides the positional order of inputs in the binary strings used in the report. In this case the order is "acbd", rather than "abcd". This is because symbol c occurred earlier in the expression than symbol b.

Usage Tip

After spending some time with FEC tables for bimodal expressions, one can observe that inputs which are FEC covered have at least one non-zero value in both the "->0" and "->1" columns. Any input with two '0's in a given column will be uncovered. It can be efficient to scan down the ->0 and ->1 columns looking for strings of '0's, then concentrate on those inputs and their matching input patterns.

FEC and Short-circuiting

For some expressions, it is not required to evaluate all the inputs within the expression once the output has been determined. For more detail on short-circuit expression evaluation, refer to [“Effect of Short-circuiting on Expression and Condition Coverage”](#).

FEC is supported in the presence of short-circuiting. Short circuit expressions can be treated in the same manner as the conventional expressions described above, except for the fact that quiescent states can now include don't cares as well. This may lead to asymmetry of input patterns for '_0' and '_1' rows. The following example shows a FEC report table for the expression (a && b && c) when it receives input vectors {001, 100, 111}

```
# -----Focused Expression View-----
# Line      38 Item    1   #1 tempreg1 <= (a && b && c);
# Expression totals: 2 of 3 input terms covered = 66.6%
#
#   Input Term   Covered Reason for no coverage   Hint
# -----
#           a           Y
#           b           Y
#           c           N   '_0' not hit           Hit '_0'
#
#   Rows:      Hits   FEC Target           Matching input patterns
# -----
#   Row  1:           1   a_0           { 0-- }
#   Row  2:           1   a_1           { 111 }
#   Row  3:           1   b_0           { 10- }
#   Row  4:           1   b_1           { 111 }
#   Row  5:      ***0***   c_0           { 110 }
#   Row  6:           1   c_1           { 111 }
#
# NOTE:
#   * Order of matching input pattern values: {a,b,c}
```

Note the matching input pattern for row 1 in the above example. Once input 'a' has been evaluated to '0', the evaluation of the other inputs is not required. Note the asymmetry in input patterns for rows 'a_0' and 'a_1'. They no longer differ only in the 'a' bit.

There is a further difference from non-short circuit coverage: Covering each expression input may require a different level of effort. To be FEC covered, input 'c' requires considerably more precise stimulus vectors than input 'a'.

Exclusions and FEC

Exclusions are row based for both unimodal and bimodal expressions. The second (more detailed) table of the FEC report contains the rows that are excluded. The first table's rows cannot be excluded. Since rows '_0' and '_1' are not linked to each other for unimodal expressions, excluding one simply implies that only the other should be hit for that input term to be considered fully covered. For bimodal expressions, excluding one row out of the '_0' '_1' pair breaks the link between their outputs. If one row is excluded, the non-excluded row becomes independent. This means that the corresponding input terminal will be considered fully covered when the non-excluded row is hit for any output value.

Consider the sample report shown in [Figure 20-3](#). For this example, the input vectors applied to the expression were 0100, 1101 and 1000. Rows 2 and 7 of the FEC table were excluded by using pragma exclusion, as follows:

```
//coverage off -item e 1 -fecexprrow 2 7
#1 tempreg2 <= ((~a & c) | (a & b & ~c & d) | (a & ~b & c & d));
```

Note that instead of excluding these rows using a pragma, row 2 and 7 of this expression could have been excluded using [coverage exclude](#). Assuming that the expression is defined on line 28 in *adder64.v*, the coverage exclude command would be:

```
coverage exclude -srcfile adder64.v -fecexprrow 28 2 7
```

In this example, excluding the row corresponding to 'a_1' (row 2) breaks its pair with 'a_0'. Input terminal 'a' is now considered covered irrespective of whether 'a_0' is hit for output '0' or '1'. Similarly, input terminal 'd' is considered fully covered when 'd_1' is hit.

UDP Coverage Details and Examples

By default, UDP coverage is not enabled for collection. You can enable the collection of UDP statistics using [vcom/vlog/vopt -coverudp](#).

During evaluation of a condition or expression, a truth table is constructed and counts are kept for each row of the truth table that occurs. UDP truth tables are composed of columns that correspond to each input of the targeted condition or expression. The right-most column corresponds to the expression's output value. The table rows correspond to combinations of input and output values.

Values can be '0', '1', or '-' (don't-care). 'Z' values are automatically excluded. Also automatically excluded are rows corresponding to ternary expressions where the two data inputs are the same and the select input is "don't care". The [vlog, vopt](#) or [vsim -noexcludeternary](#) command argument can be used to override this automatic exclusion from coverage.

When the simulator evaluates an expression, each UDP row is examined. If the current values match the given row, the row is said to be hit, and its hit count increments. If all rows have non-0 hit counts, the expression or condition has reached 100% coverage.

Examples

An example of UDP coverage for conditions is shown in [Example 20-5](#); with a condition with vectors in [Example 20-6](#); and [Example 20-7](#) shows an example of UDP expression coverage.

Example 20-5. UDP Condition Truth Table

For example, consider the following IF statement:

```
Line 180: IF (a or b) THEN x := 0; else x := 1; endif;
```

It reflects a truth table as shown in [Table 20-4](#):

Table 20-4. Condition UDP Truth Table for Line 180

Truth table for line 180				
	counts	a	b	(a or b)
Row 1	5	1	-	1
Row 2	0	-	1	1
Row 3	8	0	0	0
unknown	0			

Row 1 indicates that $(a \text{ or } b)$ is true if a is true, no matter what b is. The "counts" column indicates that this combination has executed 5 times. The '-' character means "don't care." Likewise, row 2 indicates that the result is true if b is true no matter what a is, and this combination has executed zero times. Finally, row 3 indicates that the result is always zero when a is zero and b is zero, and that this combination has executed 8 times. The unknown row indicates how many times the line was executed when one of the variables had an unknown state.

If more than one row matches the input — as is the case in the example with an input vector $\{1,1\}$ — each matching row (in this case, Row 1 and 2) is counted. If you would prefer no counts to be incremented on multiple matches, set “[CoverCountAll](#)” to 0 in your *modelsim.ini* file to reverse the default behavior. Alternatively, you can use the [-covercountnone](#) argument to [vsim](#) to disable the count for a specific invocation.

Example 20-6. Vectors in UDP Condition Truth Table

Values that are vectors are treated as subexpressions external to the table until they resolve to a boolean result. For example, take the IF statement:

```
Line 38:IF ((e = '1') AND (bus = "0111")) ...
```

A UDP truth table will be generated in which `bus = "0111"` is evaluated as a subexpression and the result, which is boolean, becomes an input to the truth table. The truth table looks as follows:

Table 20-5. Condition UDP Truth Table for Line 38

Truth table for line 38				
	counts	e	(bus="0111")	(e='1') AND (bus = "0111")
Row 1	0	0	-	0
Row 2	10	-	0	0
Row 3	1	1	1	1
unknown	0			

Index expressions also serve as inputs to the table. Conditions containing function calls cannot be handled and will be ignored for condition coverage.

If a line contains a condition that is uncovered — that is, some part of its truth table was not encountered — that line will appear in the **Coverage Analysis** window when you select **Condition Analysis** from the Analysis Type pulldown menu (see [Figure 20-2](#)). When that line is selected, the condition truth table will appear in the Details window. Double-click the line to highlight it in the Source window.

In general, if branch and condition coverage are turned on but NOT expression coverage, the ternary is treated as a (b && d) condition and a true and false branch. But if expression coverage is on, the entire RHS is analyzed as a single expression, and you will not see a condition or branch.

In regards to (b==c) condition, this is regarded as a single relational expression which is evaluated as a primary input, and thus the condition is a single term and is rejected for condition coverage. (It doesn't check whether b and c are scalars.) If however, b and c are scalars, and you were to rewrite it as (b ~^ c) or (b xnor c), then it would be recognized as a boolean operator on scalar operands and would be accepted for condition coverage.

In short, condition coverage is considered for all logical operators, but not for "relational" operators (==, <=, >=, !=). The entire relational subexpression is treated as a primary input.

Example 20-7. Expression UDP Truth Table

For the statement:

```
Line 236: x <= a xor (not b(0));
```

The following truth table results, with the associated counts.

Table 20-6. Expression UDP Truth Table for line 236

Truth table for line 236				
	counts	a	b(0)	(a xor (not b(0)))
Row 1	1	0	0	1
Row 2	0	0	1	0
Row 3	2	1	0	0
Row 4	0	1	1	1
unknown	0			

If a line contains an expression that is uncovered (some part of its truth table was not encountered) that line appears in the **Coverage Analysis** window when you select **Expression Analysis** from the Type pulldown menu. When that line is selected, the expression truth table appears in the Details window and the line will be highlighted in the Source window.

VHDL Condition and Expression Type Support

Condition and expression coverage supports bit, boolean and std_logic types. Arbitrary types are supported when they involve a relational operator with a boolean result. These types of subexpressions are treated as an external expression that is first evaluated and then used as a boolean input to the full condition. The subexpression can look like:

(var <relop> const)

or:

(var1 <relop> var2)

where var, var1 and var2 may be of any type; <relop> is a relational operator (e.g., ==, <, >, >=); and const is a constant of the appropriate type.

Expressions containing only one input variable are ignored, as are expressions containing vectors. Logical operators (e.g., and, or, xor) are supported for std_logic/std_ulogic, bit, and boolean variable types.

When condition or expression coverage is enabled, all VHDL expression inputs are converted to one of 4 states: 0, 1, X, or Z. In particular, a common scenario is for U to be converted to X, which can sometimes visibly affect simulation results.

Verilog/SV Condition and Expression Type Support

For Verilog/SV condition and expression coverage, as in VHDL, arbitrary types are supported when they involve a relational operator with a boolean result. Expressions containing only one

input variable are ignored, as are expressions resulting in vector values. Logical operators (&& || ^, for example) are supported for one-bit net, logic, and reg types.

Toggle Coverage

Toggle coverage is the ability to count and collect changes of state on specified nodes, including:

- Verilog and SystemVerilog signal types: wire, reg, bit, enum, real, shortreal, and integer atoms (which includes shortint, int, longint, byte, real, integer, and time). SystemVerilog integer atoms are treated as bit vectors of the appropriate number of bits, and counts are kept for each bit. Aggregate types (arrays, structs, packed unions) are handled by descending all the way to the leaf elements and collecting coverage on each leaf bit.
- VHDL signal types: boolean, bit, bit_vector, enum, integer, std_logic/std_ulogic, and std_logic_vector/std_ulogic_vector. Aggregate types (arrays, records) are handled by descending all the way to the leaf elements and collecting coverage on those bits.

There are two modes of toggle coverage operation - standard (or 2-state) and extended (or 3-state). Extended coverage allows a more detailed view of test bench effectiveness and is especially useful for examining the coverage of tri-state signals. It helps to ensure, for example, that a bus has toggled from high 'Z' to a '1' or '0', and a '1' or '0' back to a high 'Z'. (See [Standard and Extended Toggle Coverage](#).)

When compiling or simulating, specify standard (2-state) toggle using the “t” code, and extended (3-state) toggle using the “x” code. See “[Specifying Toggle Coverage Statistics Collection](#)” for more information on this topic.

Toggle coverage can be excluded from statistics collection, though proper management of exclusions is important. See “[Toggle Exclusion Management](#)” for information on this topic.

Toggle Coverage and Performance Considerations

Toggle coverage can be expensive in terms of performance since it applies to many objects in simulations. It also turns off several important optimizations used by Questa. If Toggle coverage is collected indiscriminately across an entire design, slowdowns of 10x or more may be observed. The "Toggle Ports Only" flow - viewing only the ports when collecting toggle coverage - helps reduce this impact (see "[Toggle Ports Only Flow](#)"). Other methodologies may help as well; for example, only collecting toggle coverage on a subset of simulations, or on a subset of regions within a simulation.

In addition, the following [vcom](#), [vlog](#), and [vsim](#) options also can be used to control performance and capacity when toggle coverage is in effect: -togglecountlimit, -togglewidthlimit, -togglevlogint, -togglemaxintvalues, -togglevlogreal, -togglemaxrealvalues, -togglefixedsizearray, and -togglemaxfixedsizearray.

VHDL Toggle Coverage Type Support

Supported types for toggle coverage are: boolean, bit, enum, integer, std_logic/std_ulogic, and arrays and records of these types, including multi-dimensional arrays and arrays-of-arrays.

VHDL multi-dimensional arrays and arrays-of-arrays are not treated as toggle nodes by default. To treat them as toggle nodes, use the `-togglefixedsizearray` argument to the `vsim` command, enable the `ToggleFixedSizeArray` variable in `modelsim.ini`. VHDL multi-dimensional arrays and arrays-of-arrays are supported, providing that the leaf level elements consist of supported data types. These arrays are broken into their leaf level elements, and toggle coverage for each element is calculated individually.

For VHDL enum's, counts are recorded for each enumeration value and a signal is considered "toggled" if all the enumerations have non-zero counts.

For VHDL integers, a record is kept of each value the integer assumes and an associated count. The maximum number of values recorded is determined by a limit variable that can be changed on a per-signal basis. The default is 100 values. The limit variable can be turned off completely with the `-notoggleints` option for the `vsim` command or setting `ToggleNoIntegers` in the `modelsim.ini` file. The limit variable can be increased by setting the `vsim` command line option `-togglemaxintvalues`, setting `ToggleFixedSizeArray` in the `modelsim.ini` file, or setting the Tcl variable `ToggleMaxIntValues`. A VHDL integer is considered 100% toggled if at least two different values were seen during simulation. If only one value was ever seen, it is considered 0% toggled.

For VHDL arrays, toggles are counted when the array has less than `ToggleWidthLimit` elements (see "[Limiting Toggle Coverage](#)"). Toggle coverage works for VHDL arrays by descending to the bit elements at the leaves of the array, and then collecting counts for each leaf bit.

Verilog/SV Toggle Coverage Type Support

Supported types for toggle coverage are wire, reg, bit, logic, fixed-size multi-dimensional array (both packed and unpacked), real, enum, integer atoms (i.e. integer, time, byte, shortint, int, and longint), packed unions, and structures (both packed and unpacked) with fields of types supported for toggle coverage. For objects of non-scalar type, toggle counts are kept for each bit of the object. Dynamic arrays, associative arrays, queues, unpacked unions, classes, class-like objects such as mailbox and semaphore, and events do not participate in toggle coverage collection.

SystemVerilog integer atom types are treated as toggle nodes unless the `-notogglevlogints` argument is applied to the `vsim` command line, or the `ToggleVlogIntegers` variable is turned off in `modelsim.ini`. The simulator breaks up integer atoms into individual bits and counts them as toggle nodes.

SystemVerilog real types (real, shortreal) are not treated as toggle nodes by default. To treat them as toggle nodes, use the `vsim` command's `-togglevlogreal` argument or turn on the

[ToggleVlogIntegers](#) variable in *modelsim.ini*. When toggle collection is in effect for SV real types, a record is kept of each value the real assumes and an associated count. The maximum number of values recorded is determined by a limit variable. The default is 100 values. The limit variable can be increased by setting the vsim command line option **-togglemaxrealvalues**, or setting [ToggleMaxRealValues](#) in the *modelsim.ini* file. A SystemVerilog real is considered 100% toggled if at least two different values were seen during simulation. If only one value was ever seen, it is considered 0% toggled.

SystemVerilog packed types include sophisticated data structures such as packed struct, packed union, tagged packed union, multi-dimensional packed arrays, enumerated types, and compositions of these types. By default, toggle coverage is reported for each dimension or member of such types. However, you can control this by making use of the **-togglepackedasvec** argument to the [vsim](#) command. This option causes coverage to be reported as if the object was an equivalent one-dimensional packed array with the same overall number of bits. The ["TogglePackedAsVec"](#) *modelsim.ini* variable provides a default value for **-togglepackedasvec**.

Objects of enumerated types are considered to be covered if all of the enumeration values occur during simulation. However, the **-togglevlogenumbits** argument to the [vsim](#) command can be used to cause the object to be treated as an equivalent packed array of bit or logic type. The ["ToggleVlogEnumBits"](#) *modelsim.ini* variable provides a default value for **-togglevlogenumbits**.

SystemVerilog unpacked array support is limited to fixed-size arrays. See the **-togglefixedsizearray** and **-togglemaxfixedsizearray** arguments to the [vsim](#) command. Other kinds of unpacked arrays (dynamic arrays, associative arrays, and queues) are not supported for toggle coverage. The **-togglepackedasvec** option only applies to the packed dimensions of multi-dimensioned SystemVerilog arrays.

SystemVerilog unpacked structs are supported as long as all struct elements consist of supported data types. Unpacked structs are broken into their fields and toggle coverage for each field is calculated individually.

Toggle Ports Only Flow

At times the amount of data collected during Toggle Coverage can be overwhelming. It is possible to limit the amount of data by only collecting coverage on ports. Internal signals can be left out of the UCDB, thus reducing both storage and processing requirements.

This approach is known as the "Toggle Ports Only" flow. It is enabled by using the [TogglePortsOnly](#) *modelsim.ini* variable or the **-toggleportsonly** switch to [vlog](#), [vcom](#), [vopt](#), or [vsim](#).

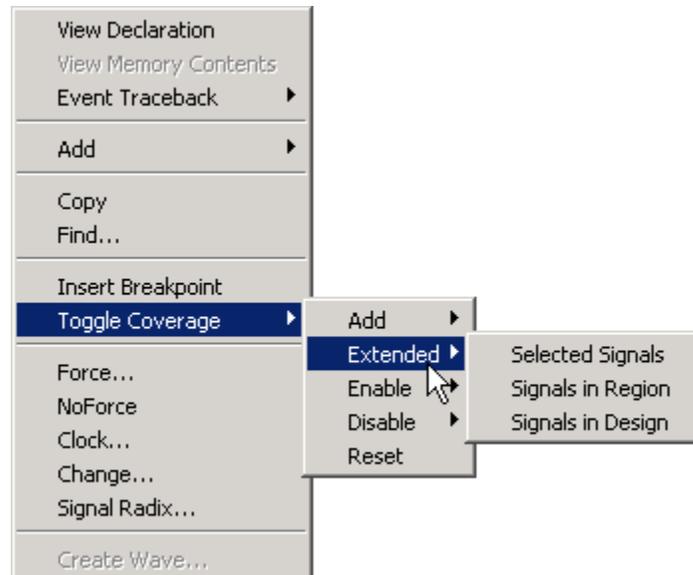
The coverage reports and GUI will only show numbers associated with togglenodes that are ports in the design. Similarly, coverage aggregation calculations only involve ports. The Toggle Ports Only flow correctly handles simulator port collapsing. Other approaches such as "toggle

add -ports ..." and "coverage exclude ..." don't work as smoothly or intuitively when port collapsing is present.

Viewing Toggle Coverage Data in the Objects Window

To view toggle coverage data in the Objects window right-click in the window to open a context popup menu mouse-over the **Toggle Coverage** selection to open the sub-menu (Figure 20-4).

Figure 20-4. Toggle Coverage Menu



The sub-menu allows you to **Add** toggle coverage for the selected item(s) in the Objects window; include **Extended** toggle coverage; **Enable** or **Disable** toggle coverage; or **Reset**. Another sub-menu allows you to choose **Selected Signals**, **Signals in Region**, or **Signals in Design**.

Toggle coverage data is displayed in the Objects window in multiple columns, as shown below. Right-click the column title bar and select **Show All Columns** from the popup menu to make sure all Toggle coverage columns are displayed. There is a column for each of the six transition types. Click (left mouse button) any column name to sort data for that column. See [GUI Elements of the Objects Window](#) for more details on each column.

Figure 20-5. Toggle Coverage Data in the Objects Window

Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% Z
into	0100000000...	Reg	Internal	119628	119629	0	0	0	0	32	11	34.38%	34.38%	11.46%	0%
outof	0000000000...	Reg	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
rst	0	Reg	Internal	2	1	0	0	0	0	1	1	100%	100%	33.33%	0%
clk	1	Reg	Internal	83222	83223	0	0	0	0	1	1	100%	100%	33.33%	0%
out_wire	0000000000...	Net	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
dat	0000000000...	Net	Internal	23401	28607629308634542	119620	114418			32	6	18.75%	21.88%	47.92%	60.94%
addr	0000100000	Net	Internal	26006	26007	0	0	0	0	10	4	40%	40%	13.33%	0%
loop	xxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	S10	Net	Internal	15602	15601	0	0	0	0	1	1	100%	100%	33.33%	0%
wr_	S11	Net	Internal	7803	7803	0	0	0	0	1	1	100%	100%	33.33%	0%

Understanding Toggle Counts

This section defines what is considered as a “toggle” during a simulation.

All toggle coverage ignores zero-delay glitches: they do not count. Also, initialization values are not counted.

Standard and Extended Toggle Coverage

Standard (2-state) toggle coverage only counts 0L->1H and 1H->0L transitions.

Extended (3-state) toggle coverage counts these transitions plus four possible Z transitions (0L->Z, 1H->Z, Z->0L, Z->1H)

There are three different modes for extended toggle coverage. The modes range from optimistic (mode 1) to pessimistic (mode 3). Select the mode that corresponds best to your coverage methodology and goals. Mode selection can be done on a per-design unit basis using vcom/vlog options, or on a more global basis using vopt or vsim options.

- Mode 1: 0L->1H & 1H->0L & any one Z transition (to/from Z)
- Mode 2: 0L->1H & 1H->0L & one transition to Z & one transition from Z
- Mode 3: 0L->1H & 1H->0L & all four Z transitions.

Conversion of Extended Toggles

It is common in extended toggle coverage that not all togglenodes in a given scope are capable of producing Z values. For this reason, all modes of extended toggle coverage calculation allow 100% coverage for lh and hl, as long as no Z edge occurs. An extended togglenode with no Z edges observed in simulation is implicitly converted to a 2-state togglenode. The togglenode therefore only has 0L->1H and 1H->0L counts. Such togglenodes will have their mode of extended toggle coverage reported as '2-STATE' in coverage reports. If such a togglenode is

eventually merged (via vcover merge or similar) with a simulation that contains Z edges, the merge is pessimistic: the merged result contains all the Z edges.

Coverage Computation for Toggles

Both regular and extended toggle coverage are calculated using the general coverage calculation algorithm: $\text{cover} = \{\# \text{ covered bins}\} / \{\# \text{ total bins}\}$. In regular mode, each togglenode has 2 bins. In extended mode, there are a different number of bins based on the exact extended toggle mode.

First, nothing changes with respect to 0L->1H and 1H->0L bins. These are 2 bins and are always counted that way. However, the various Z bins are counted differently:

```
if (extended toggle mode is 1) then
  # total bins = 3
  if (lz or zl or hz or zh) then
    # covered++
  endif
else if (extended toggle mode is 2) then
  # total bins = 4
  if (lz or hz) then
    # covered++
  endif
  if (zl or zh) then
    # covered++
  endif
else if (extended toggle mode is 3) then
  # total bins = 6
  each of zl, zh, lz, hz are counted individually for # covered
endif
```

The # total bins will affect the "Active" count in [vcover stats](#) and any reference to "toggle nodes" in reports

Toggle Nodes that Span Hierarchy

In hierarchical designs, many signals span multiple levels of hierarchy through port connections. At each level of the design, such signals have different hierarchical names. For example:

```
/top/clock
/top/dut/clock
/top/dut/deep/fsm/clock
```

may all be different names for the same signal. In ModelSim, we use the term "alias name" to refer to the set of duplicate names. We use the term "canonical name" to refer to a unique name that can be used to describe the overall signal. In almost all cases, the "canonical name" is the top-most name of the signal in the hierarchy. Note that a canonical name is just another alias in the overall set of aliases. In other words, the canonical name can be considered an alias name, too.

In the example above, */top/clk* is the canonical name, and the other names are alias names. When toggle coverage is enabled for such designs, one has to consider the issue of multiply counting and reporting statistics on the signal's various alias names.

ModelSim uses the following rules when processing hierarchical togglenodes:

1. When reporting toggle coverage in a single du or instance, no consideration is given to alias names vs. canonical names. All togglenodes declared in the du or instance are shown and their toggle counts and percentages are displayed. Examples of this occur in the Objects window, the Details window, the HTML report for a given du or instance, and the output of the "coverage report -code t -details" command.
2. When reporting aggregated toggle results in a hierarchy, a given signal is only considered once when determining toggle coverage numbers. If more than one name for a given togglenode is present in the hierarchy, only one of the names for the togglenode is used when recursively aggregating toggle coverage numbers (the canonical name is used if it is present). Recursive aggregation of toggle coverage numbers occurs in several places, including:
 - The Structure window's Toggle % column, when "Enable Recursive Coverage Sums" is enabled, as it is by default
 - The Structure window's "Total Coverage" column
 - Total Coverage in the UCDB Browser
 - The hierarchical numbers in HTML Report's
 - The output of the "toggle report" command

At times, the existence of alias names for high level toggle nodes can create confusion for users when viewing toggle coverage numbers. For example, if you enable toggle reporting on one specific instance using "toggle add -instance", you might find that toggle coverage numbers start appearing in other instances. Likewise, if you exclude a togglenode in one instance, you might find that togglenodes (ports and internal signals) disappear in other instances. This behavior occurs when nets span multiple instances and a set of alias names is present. The behavior is normal and occurs by default: it does not compromise coverage numbers or performance.

The best way to have full access to all alias names of hierarchical toggle nodes is to use the **vopt** +acc=p switch and **vsim** -nocollapse. Without the use of these, many ports on hierarchical signals are not visible to the coverage reporting system. The +acc=p switch can degrade simulator performance significantly, so only use it when necessary for analysis. Use the +acc=p+<selection> modifier whenever possible in order to prune down the affected area of the design. The +acc=p switch is not necessary if you are using the -novopt flow.

To see a complete list of all alias names on each hierarchical signal, you can use the -duplicates switch with the **toggle report** command.

If you see that some toggle nodes (typically aliased ports) are missing in a coverage report, run an exclusion report ([coverage report](#) with exclusions enabled) to see if those nodes might have been excluded on a different alias, using a command such as:

```
coverage report -excluded -code t
```

If you are only interested in monitoring the coverage numbers of ports on selected blocks in your design, you might also consider using the following command:

```
toggle report -select ports
```

Extended Toggle Mode Across Hierarchy

It is possible that a signal that spans multiple levels of hierarchy will have different toggle modes applied to it. Consider the following compile commands:

```
vlog +cover=x+/top
```

```
vlog +cover=t+/top/dut
```

In this scenario the compiler applies "extended toggle mode" to */top/clock*, and regular toggle mode to */top/dut/clock* and lower aliases. In such cases, the mode that is applied to the canonical name dominates the mode(s) applied to other aliases of the signal. So, */top/clock*, */top/dut/clock*, and */top/dut/deep/fsm/clock* would all be collected in extended toggle mode.

If there is more than one extended toggle mode applied to the same hierarchical signal at both compile time and simulation time, the option with the highest priority overrides the setting. Vsim time is lowest priority and vlog/vcom time is highest priority. For example, if a design is compiled with **-extendedtogglemode 1**, but simulated with **-extendedtogglemode 2**, the compiler option overrides the simulator option, so the **-extendedtogglemode** is applied to the design as 1.

Specifying Toggle Coverage Statistics Collection

You can specify that toggle coverage statistics be collected for a design, either standard toggle or extended, using any of the following methods:

- Compile (vcom/vlog) using the argument `+cover=` with either 't' or 'x'. See "[Specifying Coverage Types for Collection](#)" for more information.
- Entering the `toggle add` command at the command line.
- Select **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** in the Main window menu.

Using the Toggle Add Command

The `toggle add` command allows you to initiate toggle coverage at any time from the command line. Upon the next running of the simulation, toggle coverage data will be collected according

to the arguments employed (i.e., the `-full` argument enables collection of extended toggle coverage statistics).

Arguments specified with `toggle add` command override those set by the `+cover=t(x)` arguments set at compile time (`vlog/vcom`). The `toggle add` command also allows you to change toggle modes (from standard to extended or vice versa) during simulation. 0L->1H and 1H-> 0L transition counts are maintained no matter how many times you switch modes. Z transitions, however, are reset.

Note



If you do a `toggle add` command on a group of signals, then '`toggle add -full`' on the same signals will convert them to extended toggle coverage mode (all six transitions). (See [Standard and Extended Toggle Coverage](#).) Similarly, if you do a '`toggle add`' command on extended toggle coverage mode toggles (six transitions), then it will convert them into standard coverage toggles (two transitions).

- When the `toggle add` command is given, the result '0' (number of toggles added) means that any(all) existing extended toggle nodes were converted to standard toggles.
- When the '`toggle add -full`' command is given, then result '0' (number of toggles added) means that any(all) existing toggle nodes were converted to extended toggles.
- For SystemVerilog struct, conversion will apply to all fields of the structure and not to any particular type of field.

Using the Main Window Menu Selections

You can enable toggle coverage by selecting **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** from the Main window menu. These selections allow you to enable toggle coverage for Selected Signals, Signals in Region, or Signals in Design.

After making a selection, toggle coverage statistics will be captured the next time you run the simulation.

Limiting Toggle Coverage

The `ToggleCountLimit` *modelsim.ini* variable limits the toggle coverage count for a toggle node. After the limit is reached, further activity on the node will be ignored for toggle coverage. All possible transition edges must reach this count for the limit to take effect. For example, if you are collecting toggle data on 0->1 and 1->0 transitions, both transition counts must reach the limit. If you are collecting "full" data on 6 edge transitions, all 6 must reach the limit. The default setting for this variable is 1. If the limit is set to zero, then it is treated as unlimited.

If you want different toggle count limits on different design units, use the `-togglecountlimit` argument for `vcom` or `vlog`. The `-countlimit` argument for the `toggle add` command will set a count limit on a specific node.

If you want to override the `ToggleCountLimit` variable everywhere, like for a batch run, use the `-togglecountlimit` argument for `vsim`.

The `ToggleWidthLimit` `modelsim.ini` variable limits the maximum width of signals that are automatically added to toggle coverage with the `+cover=t` argument to `vcom` or `vlog`. The default limit is 128. A value of 0 is taken as "unlimited." This limit is designed to filter out memories from toggle coverage. The limit applies to Verilog registers and VHDL arrays. If the register or array is larger than the limit, it is not added to toggle coverage.

You can change the default toggle width limit on a design unit basis with the `-togglewidthlimit` argument for `vcom`, `vlog`, or `vsim`.

The `-widthlimit` argument for the `toggle add` command will set the width limit for signals on a specific node.

Finite State Machine Coverage

Because of the complexity of state machines, FSM designs can contain a higher than average level of defects. It is important, therefore, to analyze the coverage of FSMs in RTL before going to the next stages of synthesis in the design cycle.

Refer to the chapter “[Finite State Machines](#)” for additional information.

Coverage Exclusions

When code coverage is enabled for an entire design, or for the purposes of debugging a particular segment of the design, you may want to exclude coverage for individual design units, files, lines of code, objects, etc. Coverage exclusions are used for this purpose.

Coverage objects can be excluded using the GUI, with the `coverage exclude` CLI command, or with source code pragmas. When exclusions are applied, they are saved into the UCDB along with all coverage count data. This allows you to generate reports later in Coverage View mode which match the most recent simulation state.

When exclusions have been applied, they are saved into the UCDB, allowing you to generate reports based on the most recent snapshot of coverage state.

What Objects can be Excluded?

You can exclude the following from coverage statistics collection:

- Any number of lines or files containing various constructs such as states, branches, expressions and conditions.
- Condition and expression truth table rows (see “[Exclude Rows from UDP and FEC Truth Tables](#)”).

- Toggles (see “[Toggle Exclusion Management](#)”).
- FSM transitions and states (see “[FSM Coverage Exclusions](#)”).
- Functional coverage (see “[Excluding Functional Coverage](#)”).
- Assertions (see “[Excluding Assertions and Cover Directives](#)”).

Exclusions can be instance or file specific. You can also exclude nodes from toggle statistics collection using “[coverage exclude -code t](#)”, “[coverage exclude -togglenode](#)”, or “[toggle disable](#)”.

Auto Exclusions

ModelSim automatically applies exclusions to certain code constructs. One good example is assertion code, which is normally not considered part of the “design”. It is not normally desired to have assertions participate in code coverage.

Another case is FSM state exclusions. By default, when a state is excluded, all transitions to and from that state are excluded. To explicitly control FSM auto exclusions, set the [vsim](#) argument `-autoexclusionsdisable`. For example, you can use the following command to turn off the auto exclusion feature for FSMs:

```
vsim <your vsim args> -autoexclusionsdisable=fsm
```

To change the default behavior of the tool, set the variable [AutoExclusionsDisable](#) in the `modelsim.ini` file.

The Source window and coverage reports display special indications on lines that contain auto exclusions (E_A icon in GUI, “Ea” text in a text report). See “[Coverage Data in the Source Window](#)”.

Methods for Excluding Objects

ModelSim includes the following mechanisms for excluding coverage from the UCDB.

- From within the GUI:
 - File window: Right-click on a file and select **Code Coverage > Exclude Selected File** from the popup menu.
 - Code Coverage Analysis window: Right-click on an object and select **Exclude Selection** or **Exclude Selection For Instance <inst_name>** from the popup menu.
 - Source window: Right-click a line in the Hits or BC column to:
 - Select any of the exclusion options listed in the menu to exclude (or unexclude) by line, by line for an instance, by file, or exclude with a comment. In the case of a multi-line statement, branch, condition or expression, be sure to right-click on

the last line of the item to correctly apply the exclusion. If an “if” tree has an AllFalse branch, the exclusion must be applied to the first “if” statement. See “[AllFalse Branches](#)” for further details.

- Exclude objects with a comment, or edit existing comments on exclusions — in Coverage View mode only — using the comment related menu items. These menu items are unavailable during live simulation.
- Using the CLI:
 - The [coverage exclude](#) command excludes code coverage items. Examples:

```
coverage exclude -du <du_name>//excludes particular design unit
coverage exclude -du * //excludes entire design
```

See “[Exclude Individual Metrics with CLI Commands](#)”.

- Using source code pragmas to exclude individual code coverage (bces) metrics. See “[Exclude Individual Metrics with Pragmas](#)”.
- Using an exclusion filter file, used with a .do file when running simulation with -coverage. See “[Default Exclusion Filter File](#)”.

Exclude Individual Metrics with CLI Commands

Specific CLI commands are available to:

- [Exclude Rows from UDP and FEC Truth Tables](#)
- [Exclude True or False Branch of if Statements](#)
- [Exclude Implicit \(AllFalse\) Branches](#)
- [Exclude AllFalse Branches in Case Statements](#)
- [Exclude Any/All Coverage Data in a Single File](#)
- Manage toggle coverage and exclusions (see “[Toggle Exclusion Management](#)”)

Exclude Rows from UDP and FEC Truth Tables

You can exclude lines and rows from condition and expression FEC truth tables (and UDP, if enabled for coverage) using the **coverage exclude** command or the “coverage off” pragma. For more details, see the [coverage exclude](#) command and “[coverage on and coverage off Pragma Syntax](#)”.

Exclude True or False Branch of if Statements

You can exclude either the true or false branch of an ‘if’ statement by excluding the lines where the branch occurs. For example, consider the following:

```
if (a = '1') then -- line 30
```

```
    Z <= Y;
else          -- line 32
    Z <= X;
end if;
```

To exclude the true branch in this example, you enter the command:

```
coverage exclude -code b -srcfile a.vhd -linerange 30
```

To exclude the false branch:

```
coverage exclude -code b -srcfile a.vhd -linerange 32
```

Excluding line 30 excludes the true branch, while excluding line 32 excludes the false branch. A special case applies when there is no “else”, or an “elsif” is used instead. In that case, an “AllFalse” branch is created, whose exclusion must be set explicitly. See “[Exclude Implicit \(AllFalse\) Branches](#)” for details.

Exclude Implicit (AllFalse) Branches

You can also explicitly exclude branches which are implicit at the end of an if/ elsif/elsif tree. Note that this only applies if there is no trailing "else" at the end of the tree. For example:

```
if (a = '1') then -- line 30
    Z <= Y;
elsif (b = '1') then -- line 32
    Z <= X;
end if;
```

In the event that either *a* or *b* remains at '1' throughout the simulation, you will end up with less than 100% coverage, since the "AllFalse" branch of this construct was never exercised. (i.e. the case of *a* = '0' and *b* = '0').

To explicitly exclude that implicit "AllFalse" branch, you must apply the `-allfalse` option to a coverage exclude command on line 30:

```
coverage exclude -code b -srcfile a.vhd -linerange 30 -allfalse
```

If `-code b` is not used, all code coverage types on that line would be excluded, too.

Exclude AllFalse Branches in Case Statements

The `coverage exclude` command supports exclusion of the all false branch in "case" statements. The all false branch is bound to the item number of the case head itself exactly as the all false branch for "if" statements. You can exclude such a branch using the `-allfalse` switch as in the following examples:

```
coverage exclude -scope <inst_name> -line <ln> -code b -allfalse  
coverage exclude -src <file_name> -line <ln> -item b <in> -allfalse
```

Exclude Any/All Coverage Data in a Single File

In cases where you would like to exclude all types of coverage data in a given source file, use the [coverage exclude](#) command. It can be used to exclude any / all of the following types of coverage:

- Lines within a source file
- Rows within a condition or expression truth table
- Instances or design units
- Transitions or states within a Finite State Machine

Example 20-8. Creating Coverage Exclusions with a .do File

Suppose you are doing a simulation of a design and you want to exclude selected lines from each file in the design and all mode INOUT toggle nodes. You can put all exclusions in a .do file and name it, say, *exclusions.do*. The contents of the *exclusions.do* file might look like this:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77
coverage exclude -srcfile pqr.vhd
coverage exclude -du * -togglenode * -inout
```

This excludes lines 12, 55, and 67 to 90 (inclusive) of file xyz.vhd; lines 3 to 6, 9 to 14, and 77 of abc.vhd; all lines from pqr.vhd, and all INOUT toggle nodes in the design.

After compiling using **+cover** switch, you can load and run the simulation with the following commands:

```
vsim -coverage <design_name> -do exclusions.do
run -all
```

In order to view details about the exclusions applied, such as which exclusion commands failed, enable the transcript mechanism prior to running vsim by entering the following line at the top of the your .do file (*exclusions.do*).

Default Exclusion Filter File

The Tcl preference variable PrefCoverage(pref_InitFilterFrom) specifies a default exclusion filter file to read when a design is loaded with the **-coverage** switch. By default this variable is not set. See “[Simulator GUI Preferences](#)” for details on setting and changing this variable. You can use this setting to automatically load an *exclusions.do* file at startup, thus avoiding the command “-do exclusions.do” in the example above.

Exclude Individual Metrics with Pragmas

ModelSim also supports the use of source code pragmas to selectively turn coverage off and on for individual code coverage metrics. Pragmas allow you to turn statement, branch, condition, expression, toggle and FSM coverage on and off independently.

The “coverage on”, “coverage off”, and “coverage never” pragmas are currently supported for use with branch, condition, expression, statement, toggle, and FSM coverage exclusively. They have no effect on Functional coverage.

- For toggle coverage, signal is excluded from coverage if its declaration appears within the confines of the “coverage off” section of code (see [“Exclude Nodes from Toggle Coverage”](#)).
- For FSM coverage, FSM is excluded from coverage when the declaration of a ‘current state’ variable appears within the confines of the “coverage off” region of code. The individual transitions of the FSM are not affected by the exclusion (see [“FSM Coverage Exclusions”](#) for FSM coverage pragma).

Note



Multiple coverage items on a single line of code are numbered, from left to right in ascending order. If a branch statement occurred on the same line as another type of coverage object (such as an assignment) in the source code, the item number displayed for the additional coverage object may change from one report to the next, depending on whether branch coverage was enabled.

The pragmas supported are as follows.

```
coverage on           See "coverage on and coverage off Pragma Syntax"
coverage off
coverage never
coverage fixed_value
coverage fsm_off
coverage toggle_ignore
pragma synthesis_off
pragma synthesis_on
pragma translate_off
pragma translate_on
vcs coverage on
vcs coverage off
vnavigatoroff
vnavigatoron
```

Verilog vs. VHDL

- **Verilog:** Pragmas are enforced at the level of the file. If a pragma is placed in the middle of a design unit, its influence extends beyond the end of the design unit to the end of the file.

- **VHDL:** Pragmas are enforced at the level of the design unit. For example, if you place "-- coverage off" inside an architecture body, all the following statements in that architecture are excluded from coverage; however, statements in all subsequent design units are included in statement coverage (until the next "-- coverage off"). If you place a pragma outside a design unit scope, it is active until the end of the next design unit.

Usage

- Each pragma is preceded in the code line by either a "//" (Verilog) or "--" (VHDL). For example:

```
// coverage never      (Verilog)
-- coverage never      (VHDL)
```

- Bracket the line(s) you want to exclude with these pragmas. For example:

```
-- coverage off b
...
...
-- coverage on b
```

- The "pragma" keyword can also be replaced with either "synopsys", "mentor", or "synthesis".
- Pragmas can often nest in source code, often without the developer's awareness. This can result in coverage being turned on or off at unexpected intervals. For more information, see "[Pragma Usage and Nesting](#)".

i **Important:** The 'coverage on' pragma, as well as other synonymous pragmas (such as synthesis_on, etc.), do not support nested behavior. They turn on coverage, irrespective of any number of 'coverage off' pragmas encountered earlier in the file.

- The "coverage off" and "coverage on" pragmas turn on and off coverage for specified items. If no coverage items are specified, all items are affected.
- The "coverage on|off" pragma applies to branches, conditions, expressions, statements, toggles and FSMs.

For information on toggle exclusions, see "[Toggle Exclusion Management](#)".

The "fsm_off" pragma selectively turns coverage off and on for FSM state variables and their associated transitions:

```
// coverage fsm_off <fsm_name> (Verilog)
-- coverage fsm_off <fsm_name> (VHDL)
```

For more detailed information, see "[FSM Pragma Exclusions](#)".

- The "coverage never" pragma turns off coverage completely. It takes effect at compile time, and thus can never provide coverage statistics for specified areas of the design unless the "coverage never" pragma is removed and the design recompiled. In contrast,

“coverage off” and “coverage on” pragmas are simply report-time options, and thus the coverage statistics for the specified items can be toggled on and off via the GUI or the [coverage exclude](#) command.

- The “coverage fixed_value” pragma is better described as an “inclusion” pragma, which allows you to fix the values of the input terminal of an expression or condition that has been added. The resulting UDP and FEC tables only contain those rows whose input terminals have hit the specified input values.

The pragma must be placed within the declarative region of the module or architecture, after the declaration of the signal which the input_terminal constitutes, within the same scope.

Syntax is (// for Verilog, -- for VHDL):

```
//coverage fixed_value "<input_terminal>" (<fixed_value>)  
--coverage fixed_value "<input_terminal>" (<fixed_value>)
```

where:

<input_terminal> is any condition or expression in the scope in which the pragma is added, and in the same form as it appears in the coverage report.

<fixed_value> is '0', '1', or 'Z', which can be expressed as a boolean, integer, bit, or std_logic literal expression, or as any constant signal or an expression, whose evaluated value comes out as constant (0 or 1). The value must be enclosed in parentheses '()'.

Examples :

```
//coverage fixed_value "a" (3'h0)  
// coverage fixed_value "3'{a,b,c} > 1" (0)  
-- coverage fixed_value "a" (const1 && const2)
```

where, *const1* and *const2* are two constants.

coverage on and coverage off Pragma Syntax

Two forms of the coverage on|off pragma are available:

- One form is used for general exclusions of specific coverage types, or for all types if none are specified. Syntax for general exclusions:

```
coverage {on | off} [bcesft]
```

- Another form, called fine-grained exclusion, is used to exercise very precise control over exclusions in complex code constructs. Syntax for fine-grained exclusions:

```
coverage {on | off}  
  [-item {b c e s} {[<int> | <int>-<int>]+}]  
  [-allfalse]  
  [-udpcondrow [<int>|<int>-<int>]+]
```

```
[-udpexprrow [<int> | <int>-<int>]+]
[-feccondrow [<int> | <int>-<int>]+]
[-fecexprrow [<int> | <int>-<int>]+]
```

where:

[bcesft]

selectively turns on/off branch (“b”), condition (“c”), expression (“e”), statement (“s”), FSM state variables and associated transitions (“f”), and/or toggle (“t”) coverage. If no coverage type is specified, all are affected.

-item

selectively turns on/off branch (“b”), condition (“c”), expression (“e”), and/or statement (“s”) coverage(s) for only the line of code immediately following the pragma. Requires both a specification for type of coverage and an integer specifying the item numbers (<int>) for the line immediately following pragma. Coverage items are numbered in ascending order, from left to right, beginning with 1. Item numbers can be specified as an integer or a series of integers (item 1 or items 2-4). Multiple items may be specified separated by whitespace.

-allfalse

affects only the “all false” branch from the branch coverage of a specified “if” statement. This argument is only valid for use with “coverage {on|off} -item b <item#>”.

The term “AllFalse” is being used as a name for an implicit branch at the end of an “if” or “if-else” decision tree. The AllFalse branch is considered to be hit when none of the conditions in the decision tree are true. An AllFalse branch doesn’t exist for any decision tree which ends with a bare “else”. For further details on “all false” and how it applies to the code, see [“Branch Coverage”](#).

-udpcondrow

affects only the specified rows from the UDP table of the specified condition. Valid for use with coverage on | off “-item c <item#>” when UDP coverage is enabled with vcom/vlog/vopt -coverudp.

-feccondrow

affects only the specified rows from the FEC table of the specified condition. Valid for use with coverage on | off “-item c <item#>”.

-udpexprrow

affects only the specified rows from the UDP table of the specified expression. Valid for use with coverage on | off “-item e <item#>” when UDP coverage is enabled with vcom/vlog/vopt -coverudp.

-fecexprrow

affects only the specified rows from the FEC table of the specified expression. Valid for use with coverage on | off “-item e <item#>”.

Usage

- The effect of any “coverage on|off” fine-grained pragma exclusion is limited to the next valid line of source code after excluding all blank lines and comments.

- The `-item` argument to “coverage on/off” selectively turns on/off coverage for statements, branches, conditions and/or expressions for the next line of code.
- You do not need to add a matching “coverage off” directive when using fine-grained exclusions (i.e. any pragma specified with a `-item` option).
- You can combine two or more coverage items in one coverage pragma if the item numbers are the same. For example,

```
(// coverage off -item bs 2)
```

turns off coverage for statement #2 and branch #2 of the next line.

- To exclude/include different item numbers for different coverage items, use two different pragmas.

```
// coverage on -item b 2  
// coverage on -item c 3-5
```

turns on coverage for branch #2 and condition #3,4,5 of the next line.

- You can use multiple item numbers and ranges in one pragma.

```
// coverage on -item s 1 3-5 7-9 11
```

- For Branches, enabling/disabling coverage for the (case) branch automatically enables/disables coverage for all its branches even if they are not on the same line of the (case) branch. You can override the coverage of a certain branch by an additional pragma that changes coverage of this branch.
- For Conditions, you can selectively turn coverage on/off for certain FEC condition rows using the `-feccondrow` (or `-udpcondrow`, if UDP collection is enabled). For example,

```
// coverage off -item c 1 -feccondrow 1 3-5
```

excludes UDP condition rows (1, 3, 4, 5) of the first condition item from coverage.

- For Expressions, the same functionality can be achieved using `-fecexprrow` options, or `-udpexprrow`, if UDP collection is enabled).

```
// coverage off -item e 1 -fecexprrow 1-4 6
```

excludes FEC expression rows (1, 2, 3, 4, 6) of the first expression item from coverage.

Examples

- Exclude all the following conditions, and expressions, statement, branches, until a “// coverage on” pragma is reached, or the end of the design unit is reached.

```
// coverage off
```

- Exclude 1st row from the FEC table of the first condition on the next line

```
-- coverage off -item c 1 -feccondrow 1
```

- Exclude the 2nd branch and 2nd statement from the next line

```
-- coverage off -item bs 2
```

- Exclude 3rd, 5th and 6th statements from statement coverage

```
// coverage off -item s 3 5-6
```

- Include coverage for 2nd and 3rd branches, and condition #4 of the next line

```
// coverage on -item b 2-3
// coverage on -item c 4
```

- Exclude only the AllFalse branch from the 4th branch on the next line

```
-- coverage off -item b 4 -allfalse
```

- Exclude both of these rows from the following line of code:

- 1st row in UDP table and 2nd row in FEC table of 2nd expression,
- and the 3rd row in UDP table and 4th row in FEC table of 2nd condition:

```
-- coverage off -item ce 2 -udpcondrow 3 -feccondrow 4 -udpexprrow 1 -fecexprrow 2
```

Pragma Usage and Nesting

During the course of development, the use of pragmas within long source files or included files can lead to situations in which pragmas are nested. Synthesis pragmas have an effect on coverage, as well, which further complicates the issue. This can lead to unintended or confusing coverage behavior. The coverage behavior that ModelSim exhibits as it encounters nested pragmas is described in the following scenarios. In all cases, notes are generated by ModelSim to inform you of when coverage has been enabled in the sourced code. These notes can be disabled using the `vsim` argument “-suppress 2071”.

Consider how code coverage is enabled in the following examples of nested pragmas.

Example 20-9. When Code Coverage Is Turned On

```
1 //coverage off --> LINE 1
2 /* Coverage is turned OFF in this region */
3 //coverage off
4 /* Coverage is still turned OFF in this region */
5 //coverage on
6 /* Third pragma, Coverage turned ON after this region */
7 //coverage on --> LINE 7
8 /* Fourth pragma, Coverage is already turned ON in this region */
```

One might logically assume that coverage collection would be enabled by the fourth pragma. However, it is the third pragma (on line 5) which enables the code coverage collection for the remainder of the code. This is due to the fact that ModelSim doesn't support nested pragmas as one might expect.

A note appears in the Transcript window stating the line responsible for enabling the coverage:

```
# ** Note: (vopt-2071) src/test.v(5) : enabling code coverage.
```

No warning or note appears for the fourth pragma, since the coverage has already been enabled.

Example 20-10. Nesting and Code Coverage Types

```
1 //coverage off --> LINE 1
2 /* Coverage is turned OFF in this region */
3     //coverage off
4     /* Coverage is still turned OFF in this region */
5     //coverage on bce
6     /* Third pragma, Coverage turned ON after this region */
7 //coverage on --> LINE 7
8 /* Fourth pragma, Coverage is already turned ON in this region */
```

Case 2a — Code compiled with ‘+cover’:

If the code is compiled with all coverage enabled, the pragma on line 5 enables only branch, condition and expression; coverage for all types is enabled at line 7. The following notes are issued:

```
** Note: (vopt-2071) src/test.v(5) : enabling branch , condition and
expression coverage.
** Note: (vopt-2071) src/test.v(7) : enabling code coverage.
```

Case 2b — Code compiled with ‘+cover=bcs’:

If, however, the code compiled enables only branch, condition and statement coverage, then line 5 (//coverage on bce) enables only branch and condition coverage until line 7, where all coverage is enabled. The following notes are issued:

```
** Note: (vopt-2071) src/test.v(5) : enabling branch and condition
coverage.
** Note: (vopt-2071) src/test.v(7) : enabling code coverage.
```

The rules governing how coverage is enabled/disabled with pragmas apply to any pragmas which are synonymous with 'coverage on' and 'coverage off', including:

```
// [pragma | synopsis | mentor | synthesis | vcs] translate_on
// [pragma | synopsis | mentor | synthesis | vcs] translate_off
// [pragma | synopsis | mentor | synthesis | vcs] synthesis_on
// [pragma | synopsis | mentor | synthesis | vcs] synthesis_off
// [pragma | synopsis | mentor | synthesis | vcs] override_off
// [pragma | synopsis | mentor | synthesis | vcs] override_on
```

These pragmas, however, do not operate on individual coverage types (b,c,e,s,t, or f). They enable/disable all types of code coverage as a whole.

Toggle Exclusion Management

Toggle coverage as it relates to exclusions are more complex to configure than other coverage types because of the historical fact that the “toggle” command existed before code coverage was introduced in ModelSim. Thus, there are multiple ways of achieving the same effects.

Two Methods for Excluding Toggles

ModelSim offers two independent flows for excluding toggle coverage:

- Manual flow —
Using this flow, you manually add/enable/disable toggles with the following commands:
 - **toggle add** — tells the simulator to cover a set of toggles. This automatically enables the added toggles. It requires that nets and/or variables be visible to the simulator (in other words, it will not work in a completely optimized simulation.)
 - **toggle disable** — disables previously added or enabled toggles.
 - **toggle enable** — enables previously disabled toggles.
- Compiler/Simulator flow —
Using this flow, you set up ModelSim to recognize toggles in the design during compilation, and enable the collection/exclusion of toggle data during simulation using the following commands:
 - **vcom/vlog +cover=t** — prepares to add all toggles found in the compiled design units, except pragma-excluded toggles. See “[Exclude Nodes from Toggle Coverage](#)”.

i **Tip: Important:** Since pragma-excluded toggles are parsed in the source compilation, they are only relevant to the compiler/simulator flow with `+cover=t` and `-coverage`. The pragma exclusion has no effect on toggle add, disable, or enable.

- **vsim -coverage** — required in order to add all the toggles previously found by the compiler.
- **vcover report -excluded** — prints an exclusions report detailing all toggles specific toggles that were recognized in the design during compilation. The generated report is actually an executable `.do` file that you can run at a later time.
- **coverage exclude [disable|enable] -pragma** — dynamically enables or disables reporting on toggle nodes that were previously excluded by the compiler.

In this method, both “`vcom/vlog +cover=t`” and “`vsim -coverage`” are required in order to add toggles for coverage.

These two flows of managing toggle coverage and exclusions are quite distinct. You can apply toggle exclusions by executing an exclusions report as a `.do` file (TCL format), as mentioned

above. However, this *.do* file only consistently reproduces the same set of enabled toggles in the compiler/simulator flow. This is because the exclude commands in the exclusions report depend on having a given set of toggles currently enabled. In other words, if you introduce any toggle add/enable/disable commands before applying a set of toggle exclusions from a saved *.do* file, the resulting set of toggle exclusions will not be identical to your original set. The toggle exclusions can only be applied with respect to currently enabled toggles, i.e., not to a pristine, “toggle-free” environment.

This is important, because nothing in ModelSim prevents you from mixing commands from the manual and compiler/simulator flows, however you should only do so with a solid understanding of how they interact.

Exclude Nodes from Toggle Coverage

The toggle disable command is intended to be used as follows:

1. Enable toggle statistics collection for all signals using the **+cover=t** or **+cover=x** argument to **vcom** or **vlog**, or use the **toggle add** command at run time.
2. Exclude certain signals by disabling them with the “toggle disable” command.
3. Exclude individual transitions of a toggle node with the **coverage exclude -togglenode** command. The command syntax is:

```
coverage exclude -togglenode <node_path_list> [-du <path_list> |  
-scope <path_list>] -trans <transition_list>
```

For example:

```
coverage exclude -togglenode mybit myreg -trans 01 0z
```

excludes transitions 0->1 and 0->Z from toggle nodes *mybit* and *myreg*.

Transition names are not case sensitive and can be any of the following six transitions:
01 10 0Z 1Z Z0 Z1

You can re-enable toggle statistics collection on nodes whose toggle coverage has previously been disabled via the toggle disable command using the **toggle enable** command.

Toggle Pragma Exclusion

You can also exclude toggle nodes using any of the following three pragmas:

1. “coverage off [t]” — excludes any signal placed after “coverage off” and before “coverage on” in the code. For example, in the following Verilog code:

```
//coverage off t  
reg mysignal;  
//coverage on
```

the signal *mysignal* appears as “pragma excluded” in toggle coverage reports. See “[coverage on and coverage off Pragma Syntax](#)” for further details.

2. “coverage toggle_ignore” — excludes toggles, including specific bus bits.

Verilog command syntax:

```
//coverage toggle_ignore <simple_signal_name> [<list>]
```

VHDL command syntax:

```
-- coverage toggle_ignore <simple_signal_name> [<list>]
```

where <list> is a space-separated list of bit indices or ranges. A range is two integers separated by ':' or '-'. If <list> is not specified, the entire signal is excluded.

The following additional rules apply to the use of these pragmas:

- o The pragma must be placed within the declarative region of the module or architecture in which <simple_signal_name> is declared.
- o Glob-style wildcards are supported. For example, to exclude reg_123, reg_234, reg_345 from toggle coverage, you can simply enter:

```
//coverage toggle_ignore "reg*"
```

Or, to exclude all toggles from coverage for a specific module, you can enter the following within that module:

```
//coverage toggle_ignore "*"
```

To exclude a specific index of a register, such as reg_123[2], reg_234[2], reg_345[2]:

```
//coverage toggle_ignore "reg*" "2"
```

- o If using a range, the range must be in the same ascending or descending order as the signal declaration.
3. "coverage never"

The behavior of the "`// coverage toggle_ignore`" matches that of the "`// coverage on/off`" pragma — that is, toggle nodes will be pragma excluded and can only be included in coverage by clearing the pragma exclusion at the vsim command line.

However, with the "`// coverage never`" pragma, toggle node data structures will not be created and the nodes cannot be included later.

The precedence order of these three pragma exclusions will be :

```
"coverage never" > "coverage toggle_ignore" > "coverage on/off"
```

Simultaneous Behavior of Pragmas

Both "`// coverage on/off`" and "`// coverage toggle_ignore`" will work for toggle coverage. If pragmas "`// coverage on/off`" and "`// coverage toggle_ignore`" are used simultaneously then "`//`

coverage toggle_ignore” pragma will override “// coverage on/off” pragma. For example, in the following code:

```
module top;
  reg temp;
  // coverage off
  reg temp1;
  //coverage on
  // coverage toggle_ignore temp
endmodule
```

both *temp* and *temp1* will be ignored.

In the following code:

```
module top;
  // coverage toggle_ignore temp1
  reg temp;
  // coverage on
  reg temp1;
endmodule
```

temp is ignored. This is consistent with the behavior of “// coverage toggle_ignore” because default behavior is coverage ON.

Exclude Bus Bits from Toggle Coverage

You can use the “<list>” specifier in the “coverage toggle_ignore” pragma to exclude bus bits. You can also use the “[toggle add -exclude <list>](#)” command.

Re-enable Toggles Excluded with Pragmas

Toggles that have been previously excluded using pragmas can be re-enabled (in the compiler/simulator flow) using either the “coverage exclude -code t -pragma -clear” or “coverage exclude -pragma -clear -togglenode” commands. If the compiled database includes a set of pragma-excluded toggle nodes, these CLI commands override any pragma exclusions and include the specified toggles in coverage statistics.

Exclude enum Signals from Toggle Coverage

You can exclude individual VHDL and SystemVerilog enums or ranges of enums from toggle coverage and reporting by specifying enum exclusions in source code pragmas or by using the **-exclude** argument to the [toggle add](#) command. See “[Toggle Exclusion Management](#)” for important information specific to toggle exclusions.

FSM Coverage Exclusions

In ModelSim, any transition or state of an FSM can be excluded from the coverage reports using the [coverage exclude](#) command at the command line or by pragma. See the [coverage exclude](#) command for syntax details.

Exclude FSM with coverage exclude Command

The [coverage exclude](#) command is used to exclude the specified transitions from coverage reports. Consider the following coverage report:

```
# FSM Coverage for du fsm_top--
#
# FSM_ID: state
#   Current State Object : state
#   -----
#   State Value MapInfo :
#   -----
#           State Name           Value
#   -----
#           idle                  0
#           rd_wd_1               8
#           wt_blk_1              3
#           wt_wd_1                1
#           ctrl                  10
#           wt_wd_2                2
#           wt_blk_2              4
#           wt_blk_3              5
#   Covered Transitions :
#   -----
#           Trans_ID             Transition             Hit_count
#   -----
#           0                    idle -> idle             9
#           2                    idle -> wt_wd_1             4
#           3                    idle -> wt_blk_1             2
#           4                    idle -> rd_wd_1             6
#           5                    rd_wd_1 -> rd_wd_2             6
#           7                    wt_blk_1 -> wt_blk_2             2
#           9                    wt_wd_1 -> wt_wd_2             4
```

The following are some examples of commands used to exclude data from the coverage report:

coverage exclude -du fsm_top -fstate <state> S1

excludes FSM state S1 from coverage in the design unit *fsm_top*. <state> is the current state variable. By default, when a state is excluded, all transitions to and from that state are excluded (see “[Auto Exclusions](#)”).

coverage exclude -du fsm_top -ftrans state

excludes all transitions from the FSM whose FSM_ID is *state* in the design unit *fsm_top*.

coverage exclude -du fsm_top -ftrans state {idle -> wt_wd_1} {idle -> rd_wd_1}

excludes specified transitions (2 and 3) from the FSM whose FSM_ID is *state*, in the design unit *fsm_top*. If whitespace is present in the transition, it should be surrounded by curly braces.

```
coverage exclude -scope /fsm_top/a1 -ftrans state
```

excludes all the transitions from the */fsm_top/a1* instance, in the FSM whose FSM_ID is *state*, in instance */fsm_top/a1*.

```
coverage exclude -scope /fsm_top/a1 -ftrans state {idle -> rd_wd_1} {idle ->  
rd_wd_1}
```

excludes specified transitions (numbers 3 and 4) from the FSM whose FSM_ID is *state*, in instance */fsm_top/a1*.

Using -clear with coverage exclude

When the **-clear** option is used with [coverage exclude](#), it re-enables the reporting of those transitions which have been excluded. The transitions are specified in the same manner as that for the coverage exclude command. For example:

```
coverage exclude -clear -du fsm_test -ftrans <state_var> {idle -> rd_wd_1} {idle ->  
rd_wd_1}
```

re-enables the reporting of the specified transitions.

FSM Pragma Exclusions

You can use coverage pragmas to selectively turn coverage off and on for FSM state variables and their associated transitions. Three pragmas are available:

1. “coverage off [f]” — excludes any FSM states and associated transitions that appear after “coverage off” and before “coverage on” in the code. Current state variables declared between “// coverage off” and “// coverage on” pragmas are excluded from the FSM coverage; however, the FSM is recognized. This is consistent with the current behavior of the “// coverage fsm_off” pragma. For example:

```
module mid(clock);  
input clock;  
// coverage off  
reg [1:0] cst;  
localparam s0 = 2'b00, s1 = 2'b01;  
// coverage on  
always @(posedge clock)  
begin  
    case (cst)  
        s0:    cst = s1;  
        s1:    cst = s0;  
    endcase  
end  
endmodule
```

The FSM will be recognized but will be excluded from coverage. See “[coverage on and coverage off Pragma Syntax](#)” for further details.

2. “coverage fsm_off” —

In Verilog, the pragma is:

```
// coverage fsm_off {<state_var_name> | -ftrans <state_var_name>
[<transition_list>] | -fstate <state_var_name> [<state_list>]}
```

In VHDL, the pragma is:

```
-- coverage fsm_off {<state_var_name> | -ftrans <state_var_name>
[<transition_list>] | -fstate <state_var_name> [<state_list>]}
```

where, a transition in <transition_list> is of the form: state1->state2.

Entire FSM(s) with current state variable <state_var_name> are excluded from coverage if '-fstate' and '-ftrans' options in the pragma are not used. The pragma should come after the object declaration; otherwise, it will have no effect.

The following examples demonstrate the use of pragmas for excluding individual states, individual transitions, or(and) the entire FSM:

a. The following code:

```
//coverage fsm_off cst
```

excludes the entire FSM that has state_variable *cst*.

b. The following code:

```
// coverage fsm_off cst -fstate cst2 s1 s2
```

excludes the entire FSM that has state_variable *cst* and excludes those states and transitions which constitute *s1* and *s2* of another FSM (*cst2*).

c. The following code:

```
//coverage fsm_off -ftrans cst s1->s0
```

excludes only the *s1->s0* transition of the FSM *cst*.

d. The following code:

```
//coverage fsm_off -fstate cst1 s1 -ftrans cst2 s2->s0 s3->s1
```

excludes state *s1* and all related transitions of *s1*, and excludes *s2->s0* and *s3->s1* transitions of the other FSM (*cst2*).

The state and transition name (strings) will be same as those recognized by the FSM and as reported in the vsim FSM coverage report.

Warnings are printed only if code coverage is turned on with the `+cover=f` argument during compile (with `vcom` or `vlog`). If an FSM coverage pragma is specified and coverage is turned on, the Warning may look like the following:

```
** Warning: [13] fsm_safe1.vhd(18): Turning off FSM coverage for
"state".
```

If an FSM coverage pragma is specified before the object declaration, the Warning may appear as follows:

```
** Warning: [13] fsm_safe1.vhd(17): Can't find decl "state" for turning
off FSM coverage.
```

3. “coverage never” —

The behavior of `/// coverage fsm_off` matches that of `/// coverage on/off` pragma — the FSM will be pragma excluded, and can only be included in coverage by clearing the pragma exclusion at the vsim command line.

The precedence order of these three pragma exclusions is :

```
// coverage never > // coverage fsm_off > // coverage on/off"
```

Simultaneous Behavior of Pragmas

If `/// coverage on/off` and `/// coverage fsm_off` are used simultaneously then `/// coverage fsm_off` pragma will override the `/// coverage on/off` pragma. For example:

```
module top(clock);
input clock;
// coverage on
reg [1:0] cst;
// coverage fsm_off cst
localparam s0 = 2'b00, s1 = 2'b01;
always @(posedge clock)
begin
    case (cst)
        s0:  cst = s1;
        s1:  cst = s0;
    endcase
end
endmodule
```

Even though coverage was ON while 'cst' was encountered, the 'fsm_off' pragma will override it to turn off coverage for FSM (cst).

Saving and Recalling Exclusions

You may specify files and line numbers or condition and expression FEC truth table rows (see below for details) that you wish to exclude from coverage statistics. You can then create a `.do` file that contains these exclusions with the coverage report command as follows:

coverage report -excluded -file <filename>.do

You can load this *.do* file during a future analysis with the `vsim` command as follows:

```
vsim -coverage <design_name> -do exclusions.do
```

For example, the contents of the *exclusions.do* file might look like the following:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77
coverage exclude -srcfile pqr.vhd
```

This excludes lines 12, 55, and 67 to 90 (inclusive) of file *xyz.vhd*; lines 3 to 6, 9 to 14, and 77 of *abc.vhd*; and all lines from *pqr.vhd*.

This *exclude.do* file can then be used as follows:

1. Compile your design with the `+cover=<argument>` to:
 - **vopt**, if using 3-step vopt flow
 - **vcom** or **vlog**, if not explicitly running vopt (2-step vopt flow)
2. Load and run your design with:

```
vsim -coverage <design_name> -do "do exclude.do; run -all"
```

To avoid running the `"-do exclude.do"` explicitly, you can set the default exclusion filter to run the *exclusion.do* file automatically upon invocation.

i **Tip:** To view exclusion failures, edit the *.do* file and add `"transcript on"` at the beginning of the file. You can then check the generated transcript after executing the *.do* file to see which exclusions have failed.

See the ["Exclude Rows from UDP and FEC Truth Tables"](#) for details.

Example 20-11. Excluding, Merging and Reporting on Several Runs

Suppose you are doing a number of simulations, *i*, numbered from 1 to *n*.

1. Use `vlib` to create a working library.
2. Use `vcom` and/or `vlog` to compile.
3. Use `vsim` to load and run the design:

```
vsim -c <design_i> -do "log -r /*;
coverage save -onexit <results_i>;
run -all; do <exclude_file_i.do>; quit -f"
```

Note, you can have different exclude files `<exclude_file_i>` for each run *i*, numbered from 1 to *n*.

4. Use `vcover merge` to merge the coverage data:

```
vcover merge <merged_results_file> <results_1> <results_2> ...  
<results_n>
```

5. Use `vcover report` to generate your report:

```
vcover report [switches_you_want] -output <report_file>  
<merged_results_file>
```

Exclusions are invoked during vsim, in step 3.

All the various results files `<results_i>` contain the exclusion information inserted at step 3.

The exclusion information for the merged results file is derived by ORing the exclusion flags from each vsim run. So, for example, if runs 1 and 2 exclude `xyz.vhd` line 12, but the other runs don't exclude that line, the exclusion flag for `xyz.vhd` line 12 is set in the merged results since at least one of the runs excluded that line. Then the final **vcover report** will not show coverage results for file `xyz.vhd` line 12.

Let's suppose your `<exclude_file_i>` are all the same, and called `exclude.do`.

The contents of `exclude.do` file could be:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90  
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77  
coverage exclude -srcfile pqr.vhd -linerange all
```

This will exclude lines 12, 55, and 67 to 90 (inclusive) of file `xyz.vhd`; lines 3 to 6, 9 to 14, and 77 of `abc.vhd`; and all lines from `pqr.vhd`.

Coverage Reports

Create a coverage report from the command line or with the GUI, using:

- the **Coverage Report** dialog
- the `coverage report` command — use when a simulation is loaded; creates an organized list of report data, including toggle data
- the `toggle report` command — produces an unordered list of unique toggles
- the `vcover report` command — produces textual output of coverage data from UCDB generated by a previous code or functional coverage run

HTML reports are also available through the **Coverage Report** dialog, or by applying the `-html` argument to the `coverage report` and `vcover report` commands. See “[Generating HTML Coverage Reports](#)” for more information.

Report Contents

By default, the [coverage report](#) contains a summary of coverage information for the indicated design unit, file, or instance. A summary of code coverage numbers is given for each coverage type. When you specify **-details** with the coverage report command, the summary information is followed by coverage details which correspond to each active coverage type:

- For statements — a code listing is given along with counts and exclusion details. This is similar to the Source window when in coverage mode.
- For branches — a code listing is given and it appears, similar to that shown in the Source window.
- For conditions and expressions — detailed row-by-row FEC and/or UDP tables are printed, along with hit counts for each row. See “[Reporting Condition and Expression Coverage](#)” for more information.
- For toggles — a listing of missed togglenodes is printed.
- For FSM's— a listing of missed states and transitions is printed.

Code Coverage Profiles

A code coverage “profile” is a sequence of coverage items (statements, branches, conditions and expressions) associated with their respective file/line/item numbers. This sequence can change if a condition becomes constant and branches and statements are removed, or if a particular statement right-hand-side becomes a constant and the statement is optimized away. When the sequence is changed, a new profile is said to exist.

When an instance of a module is “inlined”, the code for that module is copied into the “parent” module for that instance. Then, further optimizations might be performed on that code, depending on any parameters or constant inputs to the module. This frequently results in statements being optimized away, conditions becoming constants, branches being optimized away, and so forth. So, the code coverage “profile”, or sequence of code coverage items changes. When there are several instances of a module that are likewise inlined, some with different parameters and constant inputs, the result is that the family of instances may result in many different code coverage “profiles”.

The solution for resolving different profiles of coverage lies in reporting coverage by-instance. When you report the data by-instance, you can see exactly which statements are there (no longer optimized away) and what their coverage counts are. Whereas, if you report the data by-du or by-file, ModelSim attempts to merge these different profiles, which may result in apparently contradictory counts. (Branch counts won't match the corresponding statement counts, and so forth.) That is why it is recommended to perform reporting on a by-instance basis.

Branch Coverage and Numbering of Items in Coverage Report

Multiple coverage items on a single line within a report are numbered, from left to right in ascending order. If a branch statement occurred on the same line as another type of coverage object (such as an assignment) in the source code, the item number for the other type of coverage object displayed in a coverage report may change from one report to the next, depending on whether branch coverage was enabled.

Using the coverage report Command

The `coverage report` command produces textual output of coverage statistics or exclusions of the current simulation.

Example 20-12. Reporting Coverage Data from the Command Line

Here is a sample command sequence that outputs a code coverage report and saves the coverage data:

```
vlog ../rtl/host/top.v
vlog ../rtl/host/a.v
vlog ../rtl/host/b.v
vlog ../rtl/host/c.v

vopt +cover=bcefsx top -o top_opt
vsim -c -coverage top_opt
run 1 ms
coverage report -file d:\\sample\\coverage_rep.txt
coverage save d:\\sample\\coverage.ucdb
```

The `vlog` command compiles Verilog and SystemVerilog design units. The `+cover=bcefs[t|x]` argument applied to either `vopt` (for [Three-Step Flow](#)) or `vlog` (for [Two-Step Flow](#)) prepares the design and specifies the types of coverage statistics to collect:

- b = branch coverage
- c = condition coverage
- e = expression coverage
- f = finite state machine coverage
- t = toggle coverage (two-state)
- s = statement coverage
- x = toggle coverage (four-state)

The `-coverage` option for the `vsim` command enables code coverage statistics collection during simulation.

The **-file** option for the `coverage report` command specifies a filename for the coverage report: `coverage_rep.txt`. And the `coverage save` command saves the coverage data to `d:\sample\coverage.ucdb`.

Using the toggle report Command

The `toggle report` command displays a list of all nodes that have not transitioned to both 0 and 1 at least once, and the counts for how many times each node toggled for each state transition type. Also displayed is a summary of the number of nodes checked, the number that toggled, the number that didn't toggle, and a percentage that toggled.

The **toggle report** command is intended to be used as follows:

1. Enable statistics collection with the `+cover=t` argument to either `vlog/vcom` (if not explicitly running `vopt`), or to `vopt` for the three-step `vopt` flow.
2. Run the simulation with the `run` command.
3. Produce the report with the `toggle report` command.

By default, the report only shows togglenodes that did not toggle. In order to show all toggle nodes, including those that have already toggled, use “`toggle report -all`”.

Figure 20-6. Sample Toggle Report

```

Toggle Report
Node      1H->0L  0L->1H  0L->Z   Z->0L   1H->Z   Z->1H
-----
/test_sm/dat(7)      0        1        0        1   23404   23404
/test_sm/dat(6)   2600     2600   10403   10404   13001   13001
/test_sm/dat(5)   2600     2601   13001   13002   10403   10403
/test_sm/dat(4)   2600     2600   18203   18204    5201    5201
/test_sm/dat(3)      0        1        0        1   23404   23404
/test_sm/dat(2)   2600     2600   10403   10404   13001   13001
/test_sm/dat(1)   2600     5202    5200    7802   18204   15603
/test_sm/dat(0)  10401    13002   10402   13004   13002   10401
-----
Total Node Count    =    587
Toggled Node Count  =    131
Untoggled Node Count =    456

Toggle Coverage     =   22.32 %
  
```

You can produce this same information using the `coverage report` command.

Port Collapsing and Toggle Coverage

The simulator collapses certain ports that are connected to the same signal in order to improve performance. Collapsed signals will not appear in the toggle coverage report. If you want to ensure that all signals in the design appear in your toggle report, use the `-duplicates` switch with the [toggle report](#) command. Also, you should selectively apply `vopt +acc=p` to avoid optimizing signals away from the design and the toggle report. Also, make sure to selectively apply the `+acc=p` switch to `vopt` and `vsim -nocollapse`. See “[Toggle Nodes that Span Hierarchy](#)” for further information.

Ordering of Toggle Nodes

The ordering of nodes in the report may vary depending on how you specify the signal list. If you use a wildcard in the signal argument (e.g., `toggle report -all -r /*`), the nodes are listed in the order signals are found when searching down the context tree using the wildcard. Multiple elements of the same net will be listed multiple times. If you do not use a wildcard (e.g., `toggle report -all -r /*`), the nodes are listed in the order in which they were originally added to toggle coverage, and elements are not duplicated.

Using the Coverage Report Dialog

To create a coverage report using the ModelSim GUI, access the Coverage Report dialogs by right-clicking any object in the Files or Structure (sim) windows and selecting **Code Coverage > Code Coverage Reports**; or, select **Tools > Coverage Report > Text** or **HTML** or **Exclusions**.

See “[Generating Coverage Reports](#)” for details on the Coverage Report dialogs.

Setting a Default Coverage Reporting Mode

You can specify a default coverage mode that persists from one ModelSim session to the next through the preference variable `PrefCoverage(DefaultCoverageMode)`. The modes available allow you to specify that lists and data given in each report are listed by: file, design unit, or instance. By default, the report is listed by file. See [Simulator GUI Preferences](#) for details on changing this variable.

You may also specify a default coverage mode for the current invocation of ModelSim by using the `-setdefault [byfile | byinstance | bydu]` argument for either the [coverage report](#) or the [vcover report](#) command.

XML Output

You can output coverage reports in XML format by checking **Write XML Format** in the Coverage Report dialog or by using the `-xml` argument to the [coverage report](#) command.

The following example is an abbreviated “By Instance” report that includes line details:

```
<?xml version="1.0" ?>
- <coverage_report>
- <code_coverage_report lines="1" byInstance="1">
- <instanceData path="/concat_tester/CHIPBOND/control_inst" du="micro"
  sec="rtl">
- <sourceTable files="1">
  <fileMap fn="0" path="src/Micro.vhd" />
</sourceTable>
  <statements active="65" hits="64" percent="98.5" />
  <stmt fn="0" ln="83" st="1" hits="2430" />
  <stmt fn="0" ln="84" st="1" hits="30" />
  <stmt fn="0" ln="85" st="1" hits="15" />
  <stmt fn="0" ln="86" st="1" hits="14" />
  <stmt fn="0" ln="87" st="1" hits="15" />
  ...
  ...
```

“fn” stands for filename, “ln” stands for line number, and “st” stands for statement.

There is also an XSL stylesheet named *covreport.xsl* located in
<install_dir>/examples/tutorials/vhdl/coverage, or
<install_dir>/examples/tutorials/verilog/coverage.

Use it as a foundation for building your own customized report translators.

HTML Output

You can output coverage reports in HTML format by checking **Write HTML Format** in the Coverage Report dialog or by using the **-html** argument to the [coverage report](#) command.

For more information on HTML output reports, see “[Generating HTML Coverage Reports](#)”.

Coverage Reporting on a Specific Test

You can output coverage reports for specific tests in all formats, except XML, by using the **-testextract** argument to either [vcover](#) report or the [coverage report](#) command, or to the **-html** argument for either of those commands.

For more information on HTML output reports, see “[Generating HTML Coverage Reports](#)”.

Notes on Coverage and Optimization

The optimization process removes constructs in your design that are not functionally essential, such as code in a procedure that is never called. These constructs can include statements, expressions, conditions, branches, and toggles. This results in a trade-off between aggressive optimization levels and the ease with which the coverage results can be understood. While aggressive levels of optimization make the simulation run fast, your results may at times give you the mistaken impression that your design is not fully covered. This is due to the fact that

native code is not generated for all HDL source code in your design. Those fragments of HDL code that *do not* result in native code generation are never instrumented for coverage, either. And thus, those fragments of code do not participate in coverage gathering, measurement, or reporting activities.

When observing the Source window, you can tell which statements do not participate in coverage activities by looking at the Statement and Branch Count columns on the left of the window. If those columns are completely blank (no numbers or 'X' symbols at all), then the associated statements have been optimized out of the simulation database, and they will not participate in coverage activities.

Customizing Optimization Level for Coverage Runs

It is conceivable that you will achieve 100% coverage in an optimized design, even if certain statements or constructs have been optimized away. This is due to the fact that at the lowest level, all coverage calculations are of the form “Total Hits / Total Possible Hits = % Coverage”. Constructs that have been optimized out of the design do not count as Possible Hits. Also, because the statements never execute, they never contribute to Total Hits. Thus, statements that are optimized out of the design do not participate in coverage results in any way. (This is similar to how statements that you explicitly exclude from coverage don't contribute to coverage results.)

By default, ModelSim enables a reasonable level of optimizations while still maintaining the logic necessary for the collection of coverage statistics (for details, see [CoverOpt](#) *modelsim.ini* file variable). If you achieve 100% coverage with the default optimization level, the results are as viable as achieving 100% coverage with no optimizations enabled at all.

You can customize the default optimization levels used when coverage is enabled for the simulation as follows:

- To change optimizations applied to all runs —
Change the value (1 - 5) of CoverOpt *modelsim.ini* variable from the default level. Refer to [CoverOpt](#) for information on the available optimization levels.
- To change optimizations applied to a specific run —
Set the -coveropt argument to [vlog](#), [vcom](#), or [vopt](#). For example:

```
vcom +cover=cbesxf -coveropt 2
```

Refer to [CoverOpt](#) for information on the available optimization levels.

For more information about the tradeoffs of optimization, refer to “[Optimizing Designs with vopt](#)”.

Interaction of Optimization and Coverage Arguments

In ModelSim, the default level of optimization in vopt is -O5. When -novopt is used with vsim at the command line, the default is -O4.

The **CoverWeight** *modelsim.ini* variable corresponds to the number options of the vlog/vcom -coveropt command line option. The valid settings are 1 through 5. The lower the number, the fewer optimizations are enabled. Fewer optimizations translate into greater design visibility, yet slower performance.

CoverOpt works as follows: After all other optimization-control options have been processed, the specified level of CoverOpt optimizations is applied. All CoverOpt can do is turn OFF certain optimizations known to be harmful or confusing to coverage. CoverOpt never turns on an optimization that was not enabled already. Some optimizations are always turned off when code coverage is in effect, and some +acc flags are always turned on when code coverage is in effect (such as when line numbering is correctly preserved).

The CoverOpt setting gives you a level of control over how much optimization is applied to your design when specifying coverage types for collection.

Chapter 21

Finite State Machines

A Finite State Machine (FSM) reflects the changes a state-based design has gone through from the start of simulation to the present. Transitions indicate state changes and are described by the conditions required to enable them. Because of the complexity of FSMs, designs containing them can contain a high number of defects. It is important, therefore, to analyze the FSMs in RTL before going to the next stages of synthesis in the design cycle.

FSM Recognition	963
FSM Coverage	967
FSM Multi-State Transitions	968
Collecting FSM Coverage Metrics	968
Reporting Coverage Metrics for FSMs	970
Viewing FSM Information in the GUI	971
FSM Coverage Metrics Available in the GUI	973
Advanced Command Arguments for FSMs	975
Recognized FSM Note	976
FSM Recognition Info Note	976
FSM Coverage Text Report	978

FSM Recognition

ModelSim recognizes VHDL and Verilog FSMs during compilation or optimization when you are [Collecting FSM Coverage Metrics](#) or [Viewing FSM Information in the GUI](#) and when they fit the following criteria:

- There should be a finite number of states which the state variable can hold.
- The next state assignments to the state variable must be made under a clock.
- The next state value must depend on the current state value of the state variable.

State assignments that do not depend on the current state value are considered reset assignments.

ModelSim recognizes the following VHDL and Verilog FSM design styles:

- FSMs using a current state variable.
- FSMs using current state and next state variables.
- FSMs using multiple next-state variables, where all are used as a buffer assignment.
- FSMs using fixed, non-floating parameters/generics (only supported when using vopt).
- FSMs using a single or multiple Case statements.
- FSMs using a single or multiple If-Else statements.
- FSMs using mixed If-Else and Case statements.
- FSMs using a VHDL wait/select statement.
- FSMs using a VHDL when/else statement.
- FSMs using complex “if” conditions with AND or OR operators.
- FSMs defined using a one-hot or one-cold style (supported for Verilog only).
- FSMs using non-static next state assignments. The non-static next state variable cannot be a port, it should be an object or variable expression.
- FSMs using a current- or next-state variable as a VHDL record or SystemVerilog struct field. Nested structures are not supported.
- FSMs using a current- or next-state variable as a VHDL or Verilog index expression.
- FSMs using any integral SystemVerilog types like logic, int, bit_vector, enum, packed struct. Typedefs of these types are also supported.
- Verilog FSMs having state variable assignment to a 'x (unknown) value. X-assignment is enabled by default.

Unsupported FSM Design Styles

ModelSim does not recognize the following design styles:

- FSMs using complex “if” conditions where a current state variable is ANDed with another current state variable.
- Using Verilog part-select expressions as a current-state variable.
- Using VHDL slice expressions as a current-state variable.
- Defining a single FSM in multiple modules.

FSM Design Style Examples

The following examples illustrate several supported FSM design styles.

Example 21-1. Verilog Single-State Variable FSM

```

module fsm_1proc (output reg out, input [7:0] inp, input clk, en, rst);

    typedef enum {s0, s1, s2, s3, s4, s5, s6, s7} state_t;
    state_t state;

    always_ff @(posedge clk, posedge rst)
        if (rst)
            state <= s0;
        else
            casex (state)
                s0: begin out <= inp[0]; if (en) state <= s1; end
                s1: begin out <= inp[1]; if (en) state <= s2; end
                s2: begin out <= inp[2]; if (en) state <= s3; end
                s3: begin out <= inp[3]; if (en) state <= s4; end
                s4: begin out <= inp[4]; if (en) state <= s5; end
                s5: begin out <= inp[4]; if (en) state <= s6; end
                s6: begin out <= inp[6]; if (en) state <= s7; end
                s7: begin out <= inp[7]; if (en) state <= s0; end
                default: begin out <= inp[5]; state <= s1; end
            endcase
    endmodule

```

Example 21-2. VHDL Single-State Variable FSM

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity fsm is port( in1 : in signed(1 downto 0);
                   in2 : in signed(1 downto 0);
                   en  : in std_logic_vector(1 downto 0);
                   clk: in std_logic;
                   reset : in std_logic_vector( 3 downto 0);
                   out1 : out signed(1 downto 0));
end fsm;

architecture arch of fsm is
    type my_enum is (s0 , s1, s2, s3,s4,s5);
    type mytype is array (3 downto 0) of std_logic;
    signal test : mytype;
    signal cst : my_enum;

    begin
        process(clk,reset)
        begin
            if(reset(1 downto 0) = "11") then
                cst <= s0;
            elsif(clk'event and clk = '1') then
                case cst is
                    when s0 =>  cst <= s1;

```

```
        when s1 =>    cst <= s2;
        when s2 =>    cst <= s3;
        when others =>  cst <= s0;
    end case;
end if;
end process;
end arch;
```

Example 21-3. Verilog Current-State Variable with a Single Next-State Variable FSM

```
module fsm_2proc (output reg out, input [7:0] inp, input clk, en, rst);

    typedef enum {s0, s1, s2, s3, s4, s5, s6, s7} state_t;
    state_t c_state, n_state;

    always_ff @(posedge clk, posedge rst)
        if (rst)
            c_state <= s0;
        else
            if (en) c_state <= n_state;

    always_comb
        casex (c_state)
            s0: begin out = inp[0]; n_state = s1; end
            s1: begin out = inp[1]; n_state = s2; end
            s2: begin out = inp[2]; n_state = s3; end
            s3: begin out = inp[3]; n_state = s4; end
            s4: begin out = inp[4]; n_state = s5; end
            s5: begin out = inp[4]; n_state = s6; end
            s6: begin out = inp[6]; n_state = s7; end
            s7: begin out = inp[7]; n_state = s0; end
            default: begin out = inp[5]; n_state = s1; end
        endcase

endmodule
```

Example 21-4. VHDL Current-State Variable and Single Next-State Variable FSM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.pack.all;
entity fsm is port( in1 : in signed(1 downto 0);
                   in2 : in signed(1 downto 0);
                   en  : in std_logic_vector(1 downto 0);
                   clk: in std_logic;
                   reset : in std_logic_vector( 3 downto 0);
                   out1 : out signed(1 downto 0));
end fsm;

architecture arch of fsm is
    type my_enum is (s0 , s1, s2, s3,s4,s5);
    type mytype is array (3 downto 0) of std_logic;
    signal test : mytype;
    signal cst, nst : my_enum;
```

```
begin
  process (clk, reset)
  begin
    if (reset(1 downto 0) = "11") then
      cst <= s0;
    elsif (clk'event and clk = '1') then
      cst <= nst;
    end if;
  end process;

  process (cst)
  begin
    case cst is
      when s0 => nst <= s1;
      when s1 => nst <= s2;
      when s2 => nst <= s3;
      when others => nst <= s0;
    end case;
  end process;
end arch;
```

FSM Coverage

ModelSim recognizes FSMs in your design during the compilation stages prior to simulation. The simulation stage collects coverage metrics about which states and transitions were used while simulating the test bench with the DUT.

The following metrics are collected for FSMs:

- **State Coverage Metric** — determines how many FSM states have been reached during simulation.
- **Transition Coverage Metric** — determines how many transitions have been exercised in the simulation of the state machine.
- **Multi-state transition coverage** — tracks the various possible sequences of transitions that have been exercised in the simulation of the state machine. This is also referred to as Sequence coverage.

Related Topics

- [Collecting FSM Coverage Metrics](#)
- [FSM Coverage Metrics Available in the GUI](#)
- [Code Coverage](#)
- [Coverage Aggregation in the Structure Window](#)

FSM Multi-State Transitions

A multi-state transition is also known as a state sequence. In the coverage domain, using the switch `-fsmmultitrans` with `vcom` or `vlog` or `vopt` yields a metric sometimes known as sequence coverage, since it measures the progress of an FSM through a sequence of states. The FSM recognition messages shows multi-state transitions as:

```
#      S0 => S1 => S0
```

When you specify `-fsmmultitrans` you will be able to view this information in the:

- [FSM Recognition Info Note](#) in the transcript, specifically the multi-state transition table.
- [FSM Coverage Text Report](#), specifically the covered transition and uncovered transition tables.
- Details window
- Missed FSMs window

Collecting FSM Coverage Metrics

You can enable the recognition and collection of coverage metrics for FSMs in your design through the use of command arguments in your simulation flow.

Prerequisites

- Evaluate the commands and switches in [Table 21-1](#) to determine which are required for your flow.

Table 21-1. Commands Used for FSM Coverage Collection

Task	Command/Switch	Required
Enable the collection of FSM coverage metrics for individual design units.	<code>vcom</code> or <code>vlog</code> with <code>+cover=f¹</code>	Yes ²
Enable the collection of FSM coverage metrics for the complete design.	<code>vopt</code> with <code>+cover=f¹</code>	Yes ²
Produce a detailed report in the transcript for each recognized FSM.	<code>vcom</code> or <code>vlog</code> or <code>vopt</code> with <code>-fsmverbose</code>	No

Table 21-1. Commands Used for FSM Coverage Collection

Task	Command/Switch	Required
Enable the reporting and collection of coverage metrics for FSM Multi-State Transitions	vcom or vlog or vopt with -fsmmultitrans	No
Collect coverage metrics for the design under simulation.	vsim with -coverage	Yes

1. You can also use +cover with no arguments, which enables collection of all coverage metrics.
2. You must specify +cover=f at some point in the procedure with vcom or vlog or vopt.

Procedure

1. Compile your design units with vcom or vlog.
Include any switches from [Table 21-1](#) in addition to any other command arguments required for your design and environment.
2. Optimize your design with vopt.
Include any switches from [Table 21-1](#) in addition to any other command arguments required for your design and environment.
3. Load your simulation using the -coverage switch with the vsim command. The -coverage switch is required.
4. Run your simulation with the [run](#) command.

Examples

- Enable FSM coverage on the complete design.

```
vcom a.vhdl b.vhdl
vcom top.vhdl
vopt +cover=f top -o opt_top
vsim -coverage opt_top
run -all
```
- Enable coverage for all types on the complete design.

```
vlog *.v
vopt +cover top -o opt_top
vsim -coverage opt_top
run -all
```
- Enable FSM coverage, including multi-state transitions, on the complete design with verbose reporting.

```
vcom top.vhdl  
vcom a.vhdl b.vhdl  
vopt +cover=f -fsmverbose -fsmmultitrans top -o opt_top  
vsim -coverage opt_top  
run -all
```

- Enable FSM coverage only on selected design units.

```
vlog +cover=f a.v b.v  
vlog top.v  
vopt top -o opt_top  
vsim -coverage opt_top  
run -all
```

Results

- The vcom and vlog and vopt commands produce messages related to any recognized FSMs. Refer to the sections [“Recognized FSM Note”](#) and [“FSM Recognition Info Note”](#) for more information.
- The vsim command loads the simulation and changes the GUI layout to Coverage mode.
- Several of the GUI windows will contain coverage metrics, refer to the section [“FSM Coverage Metrics Available in the GUI”](#).

Related Topics

[FSM Recognition](#)
[FSM Coverage](#)
[Code Coverage](#)
[Code Coverage in the Graphic Interface](#)
[FSM Coverage Exclusions](#)

Reporting Coverage Metrics for FSMs

You can use the GUI or [coverage report](#) command to create a text report of coverage metrics for FSMs in your design.

Prerequisites

- Run a simulation to collect coverage metrics for FSMs.
- (Optional) Exclude transitions or states from coverage collection. This will allow you to reach 100% FSM coverage. Refer to the section [“FSM Coverage Exclusions”](#) for more information.

Procedure

1. Select **Tools > Coverage Report > Text**.

Displays the Coverage Text Report dialog box.

2. From the Report on dropdown, select one of the following:
 - All files — reports data for FSMs for all design units defined in each file. (-byfile with coverage report)
 - All instances — reports data for all FSMs in each instance, merged together. (-byinst with coverage report)
 - All design unit — reports data for all FSMs in all instances of each design unit, merged together. (-bydu with coverage report)
3. In the Coverage Type pane, ensure that **Fsms** is selected. (-code f with coverage report)
4. Alter any of the other options as needed.
5. Click **OK**

Results

- Writes the report (*report.txt*) to the current working directory.
- Opens a notepad window containing the *report.txt* file.

Related Topics

[FSM Coverage](#)
[FSM Coverage Metrics Available in the GUI](#)
[Code Coverage](#)
[Generating Coverage Reports](#)
[Coverage Reports](#)

Viewing FSM Information in the GUI

You can enable the creation of debug information for FSMs in your design through the use of command arguments in your simulation flow.

Prerequisites

- Invoke ModelSim in GUI mode.

- Evaluate the commands and switches in [Table 21-2](#) to determine which are required for your flow.

Table 21-2. Commands Used to Capture FSM Debug Information

Task	Command/Switch	Required
Retain visibility of and capture debugging information about FSMs for individual design units.	<code>vcom</code> or <code>vlog</code> with <code>+acc=f¹</code>	Yes ²
Retain visibility of and capture debugging information about FSMs for the complete design.	<code>vopt</code> with <code>+acc=f¹</code>	Yes ²

1. You can also specify `+acc` with no arguments, which retains visibility for most design elements.
2. You must specify `+acc=f` at some point in the procedure with `vcom` or `vlog` or `vopt`.

Procedure

1. Compile your design units with `vcom` or `vlog`.
Include any switches from [Table 21-2](#) in addition to other switches required for your design and environment.
2. Optimize your design with `vopt`.
Include any switches from [Table 21-2](#) in addition to other switches required for your design and environment.
3. Load your simulation with `vsim`.
ModelSim does not generate any FSM debug information if you specify `-novopt`.
4. Select **View > FSM List**
Displays the [FSM List Window](#), which lists all recognized finite state machines.
5. Double-click on an FSM in the FSM List window.
Displays the FSM in an [FSM Viewer Window](#).
6. Add an FSM to the Wave window:
 - a. Right-click on an FSM from the FSM List window
 - b. Select **Add to Wave > Selected FSM**

The FSM is automatically displayed in the Wave window as a group of signals with the label: **FSM (<current_state>)**. You can change the name of the group through the FSM Display Options dialog (**FSM List > Options**)

The FSM Viewer and the Wave window are dynamically linked, allowing you to analyze states and their transitions, based on your cursor position.

7. Run your simulation with the **run** command.
8. Rearrange the GUI so can see both the FSM Viewer and Wave windows simultaneously.
9. Link the FSM Viewer window to the cursor of the Wave window:
 - a. Select **FSM View > Track Wave Cursor**
 - b. View how states change by moving the cursor in the wave window, either by dragging the cursor left and right or by using the Find Previous/Next Transition buttons in the Wave Cursor toolbar.

Green indicates current state and yellow the previous state for the cursor location.

Related Topics

[FSM Recognition](#)

FSM Coverage Metrics Available in the GUI

The GUI presents coverage metrics in several windows within the GUI. This section provides an overview of where you can find this information.

- Missed FSMs window — lists states and transitions that have not been fully covered during simulation.

Transitions are listed under states, and source line numbers for each transition are listed under their respective transitions.

You must select a design unit from the Structure window that contains a state machine before anything will appear in this window

The icon that appears next to the *state* variable name is also a button , which opens the FSM in the [FSM Viewer Window](#).

The Missed FSMs window is linked to several windows:

- When you double-click on a state or transition, the FSM will open in an FSM Viewer window.
 - When you select a state or transition, it will be highlighted in the Coverage Details window.
 - When you select the source line number for the transition, a Source window will open and display the line number you have selected.
- Coverage Details window — provides detailed coverage information about any FSM item selected in the Missed FSMs window.

- Columned windows — provide FSM coverage metrics in the Structure, Files, Objects and Instance Coverage windows, as indicated in [Table 21-3](#).

Table 21-3. FSM Coverage Columns

Column	Structure window	Files window	Objects window	Instance Coverage window	Information
State %	X	X	X		state hits divided by states
State Graph	X	X		X	displays a green bar when 90% or greater, otherwise the bar is red
States Hit	X	X	X	X	
States Missed	X	X		X	
States	X		X	X	
Transition %	X	X		X	transition hits divided by transitions
Transition Graph	X	X		X	displays a green bar when 90% or greater, otherwise the bar is red
Transitions Hit	X	X		X	
Transitions Missed	X	X		X	
Transitions	X			X	

Related Topics

[FSM Coverage](#)
[Code Coverage](#)
[Code Coverage in the Graphic Interface](#)

Advanced Command Arguments for FSMs

Table 21-4 lists ModelSim command arguments related to FSM recognition and their consolidated syntax literals.

Table 21-4. Additional FSM-Related Arguments

Command	Argument	Description	Literal
vcom vlog vopt	-fsmimplicittrans	Controls recognition of implied same-state transitions.	i
	-fsmresettrans -nofsmresettrans	Controls recognition of implicit asynchronous reset transitions.	r
	-fsmsingle -nofsmsingle	Controls recognition of FSMs having a single-bit current-state variable.	s
	-fsmxassign -nofsmxassign	Controls recognition of FSMs containing an X assignment.	x
	-fsmmultitrans	Controls detection and reporting of multi-state transitions.	m

Consolidated FSM Recognition Arguments

You can use any combination of the FSM arguments (Table 21-4) in a consolidated form:

-fsm=[imrsx]

Any of the FSM arguments can be negated by prefixing its literal with “-”, for example:

-fsm=-r-xs

This example would disable recognition of implicit asynchronous reset transitions and FSMs containing an X assignment, and enable recognition of FSMs having single-bit current-state variable.

Related Topics

[Collecting FSM Coverage Metrics](#)
[Viewing FSM Information in the GUI](#)

Recognized FSM Note

Output from: [vlog](#), [vcom](#), [vopt](#)

Note ID: [vlog-143](#), [vcom-143](#), [vopt-143](#)

The [vcom](#) or [vlog](#) or [vopt](#) commands write a note to the transcript whenever they recognize an FSM. The note is of the format:

```
** Note: (<command>-143) Recognized <n> FSM in module "<module>".
```

Parameters

[Table 21-5](#) defines the replaceable values from the Recognized FSM note.

Table 21-5. Recognized FSM Note Parameters

Keyword	Description
<command>	Specifies the command (vlog , vcom , vopt) that issued the Note.
<n>	Specifies the number of FSMs recognized in the module.
<module>	Specifies the name of the module.

Example

```
# ** Note: (vlog-143) Recognized 1 FSM in module "decode_top".  
# ** Note: (vlog-143) Recognized 1 FSM in module "psi_coder".
```

Related Topics

[FSM Recognition](#)

FSM Recognition Info Note

Output from: [vlog](#), [vcom](#), [vopt](#) with the `-fsmverbose` switch

Note ID: [vlog-1947](#), [vcom-1947](#), [vopt-1947](#)

The [vcom](#) or [vlog](#) or [vopt](#) commands write a note to the transcript for every recognized FSM when you use the `-fsmverbose` switch.

Parameters

Table 21-6 defines the information in the FSM Recognition Info note.

Table 21-6. FSM Recognition Info Note Parameters

Keyword	Description
FSM recognized in	Specifies the module containing the FSM.
Current State Variable	Specifies the name, file, and line number of the current state variable.
Next State Variable	Specifies the name, file, and line number of the next state variable.
Clock	Specifies the name of the clock controlling the FSM
Reset States	Specifies the reset states of the FSM
State Set	Specifies the complete list of state names in the FSM.
Transition table	Lists all possible state transitions and any related line numbers.
Multi-state Transition table ¹	Lists all possible multi-state transitions.
INFO	Provides additional information about the FSM, such as: <ul style="list-style-type: none"> • identifying unreachable states. • identifying which states have no transitions, other than to a reset state. • identifying RTL code that will never be executed.

1. This section appears only when you specify -fsmmultitrans

Example

```
# ** Note: (vlog-1947)   FSM RECOGNITION INFO
# Fsm recognized in : decode_top
# Current State Variable : present_state : ./rice_src/sysv/decode_top.sv(19)
# Next State Variable : next_state : ./rice_src/sysv/decode_top.sv(19)
# Clock : pins.clk
# Reset States are: { S0 , XXX }
# State Set is : { S0 , S1 , XXX }
# Transition table is
# -----
# S0 => S1      Line : (32 => 34)
# S0 => S0      Line : (24 => 24)
# S0 => XXX     Line : (30 => 30)
# S1 => S0      Line : (36 => 38) (24 => 24)
# S1 => XXX     Line : (30 => 30)
# XXX => S0     Line : (24 => 24) (42 => 42)
# XXX => XXX    Line : (30 => 30)
# -----
# Multi-state transition table is
# -----
# S0 => S1 => S0                               (Loop)
# S0 => S1 => XXX
# S0 => S1 => XXX => S0                         (Loop)
```

```
#      S0 => XXX => S0                      (Loop)
#      S1 => S0 => S1                      (Loop)
#      S1 => S0 => XXX
#      S1 => XXX => S0
#      S1 => XXX => S0 => S1                (Loop)
#      XXX => S0 => S1
#      XXX => S0 => S1 => XXX              (Loop)
#      XXX => S0 => XXX                    (Loop)
#      -----
```

Related Topics

[FSM Recognition](#)
[FSM Multi-State Transitions](#)

FSM Coverage Text Report

Generated by:

- [coverage report -code f -details](#)
- **Tools > Coverage Report > Text**

This report contains available coverage metrics for the FSMs in your design.

Parameters

The FSM Coverage Report contains the following sections:

- Header — specifies whether the report was generated by file (-byfile), instance (-byinstance), or design unit (-bydu).
- FSM Coverage — coverage metrics for States and Transitions
- FSM_ID — the name of the current state variable.
- State Value MapInfo — a mapping of the state names to internal values.
- Covered States — coverage metrics for each state
- Covered Transitions — coverage metrics for each transition, including multi-state transitions if you use the -fsmmultitrans switch.
- Uncovered Transitions — a list of all transitions that have no coverage metrics.
- Summary — the same information as the FSM Coverage table at the top of the report.

Example

This examples shows an FSM coverage report, where the metrics are reported by file.

```
# Coverage Report by file with details
#
# File: test.vhdl
```

```

# FSM Coverage:
#   Enabled Coverage      Active      Hits % Covered
#   -----
#   States                3          3    100.0
#   Transitions          13         10    76.9
#
# =====FSM Details=====
#
# FSM Coverage for file test.vhdl --
#
# FSM_ID: cst
#   Current State Object : cst
#   -----
#   State Value MapInfo :
#   -----
#           State Name      Value
#           -----
#           s0                0
#           s1                1
#           s2                2
#   Covered States :
#   -----
#           State           Hit_count
#           -----
#           s0              20
#           s1              16
#           s2              16
#   Covered Transitions :
#   -----
#           Trans_ID      Transition      Hit_count
#           -----
#           0             s0 -> s1      16
#           1             s0 -> s0       2
#           2             s1 -> s2      16
#           4             s2 -> s0      16
#           5             s0->s1->s2    16
#           7             s1->s2->s0    16
#           9             s2->s0->s1    14
#           10            s0->s1->s2->s0  16
#           11            s1->s2->s0->s1  14
#           12            s2->s0->s1->s2  14
#   Uncovered Transitions :
#   -----
#           Trans_ID      Transition
#           -----
#           3             s1 -> s0
#           6             s0->s1->s0
#           8             s1->s0->s1
#
#   Summary      Active      Hits % Covered
#   -----
#   States      3          3    100.0
#   Transitions 13         10    76.9

```

Related Topics

- [FSM Coverage](#)
- [FSM Coverage Metrics Available in the GUI](#)
- [Code Coverage](#)
- [Code Coverage in the Graphic Interface](#)
- [Coverage Reports](#)

Chapter 22

Verification with Assertions and Cover Directives

Note



The functionality described in this chapter requires an additional license feature for ModelSim SE. Refer to the section "[License Feature Names](#)" in the Installation and Licensing Guide for more information or contact your Mentor Graphics sales representative.

This chapter discusses methods for using VHDL, PSL, and SystemVerilog assertions and cover directives for design verification with ModelSim. The chapter is organized into four sections:

- [Overview of Assertions and Cover Directives](#)
- [PSL Assertions and Cover Directives](#)
- [Using SVA Assertions and Cover Directives](#)
- [Using -assertdebug to Debug with Assertions and Cover Directives](#)

ModelSim implements assertion verification capabilities via `assert`, `cover`, and `assume` directives. In the discussions that follow, the term "assertion" is used to indicate both assertion properties and verification directives unless otherwise noted.

ModelSim supports the simple subset of PSL constructs and semantics as described in the IEEE Std 1850-2005, *IEEE Standard for Property Specific Language (PSL)*. Also, the following formal types are supported: `bit`, `bitvector`, `boolean`, `numeric`, `string`, and `hdltype`.

Refer to `<install_dir>/docs/technotes/sysvlog.note` for a list of supported SystemVerilog features.

We strongly encourage you to obtain and refer to a copy of the IEEE Std 1850-2005 for PSL as well as the IEEE Std 1800-2009 for SystemVerilog.

Overview of Assertions and Cover Directives

The use of assertions and cover directives falls into the category of "white box" testing – they specify and validate the expected behavior of a design. Assertions and cover directives are written directly into the design in order to observe the values of signals and states, possibly over a span of time. This allows the verification tool to observe (and log) when particular events occur or assumptions are violated.

In SystemVerilog, two types of assertions are defined – immediate and concurrent. ModelSim supports both assertion types.

- **Immediate assertions** evaluate immediately and may be inserted in any procedural code. They can be specified anywhere a procedural statement can be specified and are executed like a statement in a procedural block.
- **Concurrent assertions** describe behavior that spans time and, therefore, can be used for checking temporal properties. Unlike immediate assertions, the evaluation model is based on a clock — that is, a concurrent assertion is evaluated only at the occurrence of a clock tick.

Cover directives, like concurrent assertions, are temporal - they are evaluated on specified clock edges over a span of time.

The crucial difference between an assertion and a cover directive is that the assertion declares that something must always hold. What is of interest is the assertion failure, which is a design bug (or perhaps an assertion bug.) A cover directive declares that something should occur sometimes. What is of interest is the cover success, which is a measure of coverage. Furthermore, it is interesting to count how many times the cover success occurred. A cover directive in PSL or a cover statement in SystemVerilog is a form of functional coverage: user-defined coverage. For more on functional coverage in general, see “[Verification with Functional Coverage](#)”.

For information on how assertion and cover directives and participate in the total coverage aggregation, see “[Coverage Aggregation in the Structure Window](#)”.

Assertion Coding Guidelines

Writing assertions, like writing RTL code, requires knowledge of which constructs are more efficient than others in terms of how they affect simulation performance. This section provides a few key guidelines that will help you improve simulation throughput - as well as a few key things you should avoid. We assume a basic understanding of assertion terminology, sequence operators, and syntax. The coding examples below are written in SystemVerilog but the concepts apply equally to PSL.

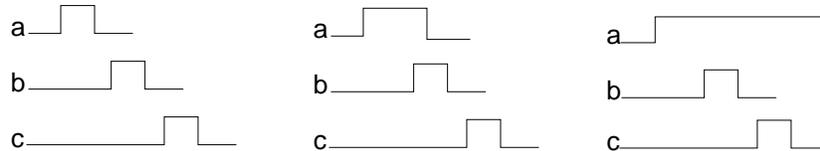
1. Keep directives simple. Create named assertions that you then reference from the assert directive (e.g., `assert check1`).
2. Keep properties and sequences simple too. Build complex assertions out of simple, short assertions/sequences.
3. Whenever possible, reference a single clock. Properties referencing multiple clocks require far more simulation time than properties referencing a single clock.
4. Do not use implication with *never* directives. You will rarely get what you want if you use implication with a *never*.

5. Create named sequences so you can reuse them in multiple assertions.
6. Be aware of "unexpected matches." For example, the following PSL assertion:

```
assert always a->next(b->next(c));
```

will match all of the following conditions (as well as others):

Figure 22-1. Assertion Matches



7. Avoid long or infinite time ranges. For example, if there is an effective timeout in a sequence, make the time range for the timeout as short as possible given the overall functional/timing requirements of the design. Using an infinite time range in an assertion means that it is never able to fail.

Take the example of a simple handshake protocol which requires an acknowledgment for every ready indication:

```
property handshake_check;
  always @(posedge clk) rdy |-> ##[1:$] acpt ##1 !acpt;
endproperty
assert property(handshake_check);
```

A much more simulation-efficient way of expressing the concept of *eventually* in an assertion is to use the *goto* operator as follows:

```
property handshake_check;
  always @(posedge clk) rdy |-> acpt[->1] ##1 !acpt;
endproperty
assert property(handshake_check);
```

Within a sequence, the use of large or unbounded time range can severely impact simulation performance. The reason for this is that a separate thread is spawned for each possibility in the legal range. For example, the sequence,

```
(a ##1 b[*1 to 8000] ##5 c ##1 d)
```

can result in 8000 separate threads of the form:

```
(a ##1 b[*1] ##5 c ##1 d);
(a ##1 b[*2] ##5 c ##1 d);
...
(a ##1 b[*8000] ##5 c ##1 d);
```

8. Use system functions like *\$rose* and *\$fell* to avoid inadvertently spawning a new thread or several new threads each cycle. In the line below,

```
(!a[*0:$] ##1 a) |-> b;
```

a thread will be started at every clock edge to check if *a* is not true. A better way to write this is:

```
$rose(a) |-> b;
```

9. Use a qualifying condition when repetitively checking ($[\rightarrow n]$) for multiple occurrences of a condition in the antecedent expression of an assertion. Often, writing assertions involves the need to check for multiple occurrences of an expression to trigger when additional expressions are evaluated. In the examples below, the intent is to check for 48 occurrences (non-consecutive) of signal *a*; and on the 48th time signal *a* is true, signal *b* is also required to be true.

```
a[->48] |-> b;
```

When re-written in its equivalent form below, the above property is extremely expensive in terms of spawning new threads. Since a brand new thread is started each and every cycle signal *a* is not true, threads grow at an nearly an exponential rate. And previously started threads, in turn, spawn new threads each subsequent cycle due the unbounded time range when signal *a* is false.

```
(!a[*0:$] ##1 a)[*48] |-> b;
```

10. In general, be very careful when using the non-consecutive ($[= n]$) operator, but especially on the left-hand side of an implication. Consider the property:

```
property p3;  
  @(posedge clk) a ##1 d[=2] ##1 c |-> ##1 e;  
endproperty  
assert property (p3);
```

It is easy to incorrectly interpret this property as *a* followed by 2 non-consecutive occurrences of *d* followed at least 1 cycle later by *c*; which is then followed one cycle by *e*, at which time the property should pass. However, as written, the property allows for both *c* and *e* to assert after the second occurrence of *d* but not pass until the third occurrence of *d* which could be sometime after *e*. This is completely unexpected behavior, causing many to assume an assertion bug has been found. However the behavior is correct because $d[=2]$ is equivalent to $(d[*1:$] ##1 d)[*2] ##1 d[*1:$]$. It is the last $d[*1:$]$ which keeps a thread from the left-hand side (LHS) of the implication alive until the third occurrence of *d*. In order for a property with implication to pass, all threads started from both the LHS and right-hand side (RHS) of the implication must complete. In this example the threads from the LHS do not complete until signal *d* occurs a third time, even if all threads from the RHS have already completed. To avoid this behavior the $d[=2]$ can be replaced by $d[->2]$ to get the intended behavior.

11. Specify behaviors accurately. Take the SVA sequence below:

```
sequence easy;  
  always @(posedge clk) a ##1 b ##1 c;  
endsequence
```

This sequence named *easy* appears to be straight-forward, and it is. It states that *a* is followed by *b* which is followed by *c* (all with delay of a single cycle/clock). However, if the correct behavior requires *a* and *b* signals to either remain asserted or to de-assert in the next cycle, then this simple sequence will not check for the expected behavior. The following modifications will:

```
a ##1 a & b ##1 a & b & c;
```

Or, depending on expected behavior:

```
a ##1 !a & b ##1 !a & ! b & c;
```

Another example: If a sequence is needed that says *a* happens at a clock edge followed by *b* in 4 to 8 clock cycles followed by *c*, it can be written as:

```
sequence s1;
  always @(posedge clk) a ##[4:8]b ##1 c;
endsequence
```

This accurately represents the above requirement. In most cases, however, the requirement is: when *a* asserts, it is to be followed by *b* asserting in 4 to 8 clock cycles; and the first time *b* asserts within the [4:8] cycle range it should be followed by *c*. This is represented by:

```
sequence s2;
  always @(posedge clk) a ##1 !b[*3:7] ##1 b ##1 c;
endsequence
```

The difference between sequences *s1* and *s2* is that in *s2*, *c* has to follow the first occurrence of *b* in [*4:8] range whereas in *seq1*, *c* can follow any occurrence of *b* in [*4:8]. In most cases the requirement is that of *s2*.

12. This is an example of a badly written cover sequence:

```
cover sequence (@(posedge clk)
  dll_state == DL_INACTIVE [*1:$] ##1 dll_state == DL_INIT [*1:$]
  ##1 dll_state == DL_ACTIVE);
```

A thread will be started at every clock edge as long as the *dll_state* is *DL_INACTIVE* which really makes no sense. A better way to write this is to use the *cover property* statement:

```
cover property (@(posedge clk)
  $changed(dll_state) |->
  $past(dll_state == DL_INACTIVE) ##0 dll_state == DL_INIT
  [*1:$] ##1 dll_state == DL_ACTIVE;
```

13. Be careful what you do in an *assertion pass* statement. SV assertions have an action block which contains an *assertion pass* statement as well as an *assertion failure* statement. If an assertion has a pass statement, then the pass statement gets executed on both real and vacuous passes. Unless you care about vacuous passes you should use the

assert control task *\$assertvacuousoff* to turn off executing of pass action blocks for vacuous passes.

14. Take into account reset conditions. You don't want to see false failures due to an assertion failing because either the design is not yet initialized or that a reset occurs during operation.
15. For local var usage please refer to:
www.mentor.com/resources/techpubs/upload/mentorpaper_35466.pdf

Using Assert Directive Names

PSL 1.1 provides for named directives via the use of a label. You are not allowed to have a label that duplicates another symbol in the same scope. In other words, you cannot explicitly label a PSL directive with a name that already exists (as, say, a signal).

In the absence of a label, ModelSim generates assert directive names for reporting information about assertions. For example:

```
property p0 is always a -> b;  
assert p0;
```

The name generated for this assert directive will be *assert__p0*. Generically, the syntax of the generated name is:

```
assert__<property name>.
```

However, if you write the same directive in this manner:

```
assert always a -> b;
```

there is no property name, so ModelSim will generate a name like *assert__0* (i.e., a number appended to "assert__").

Using the SystemVerilog Bind Construct

The SystemVerilog **bind** construct allows you to bind a Verilog design unit to another Verilog design unit, to a VHDL design unit, or to a SystemC module. This is especially useful for binding SystemVerilog assertions to your VHDL, Verilog, SystemC and mixed designs during verification. For details, see [Using SystemVerilog bind Construct in Mixed-Language Designs](#).

Processing Assume Directives

Designers use assume directives to constrain static verification. Because they are intended for formal tools, assume directives have no meaning in simulation. However, by default, ModelSim simulates assume directives as if they are assert directives and displays them in the Assertions window.

You can configure how ModelSim processes assume directives using the `-assume` and `-noassume` switches for the `vsim` command or the `SimulateAssumeDirectives` variable in the `modelsim.ini` file.

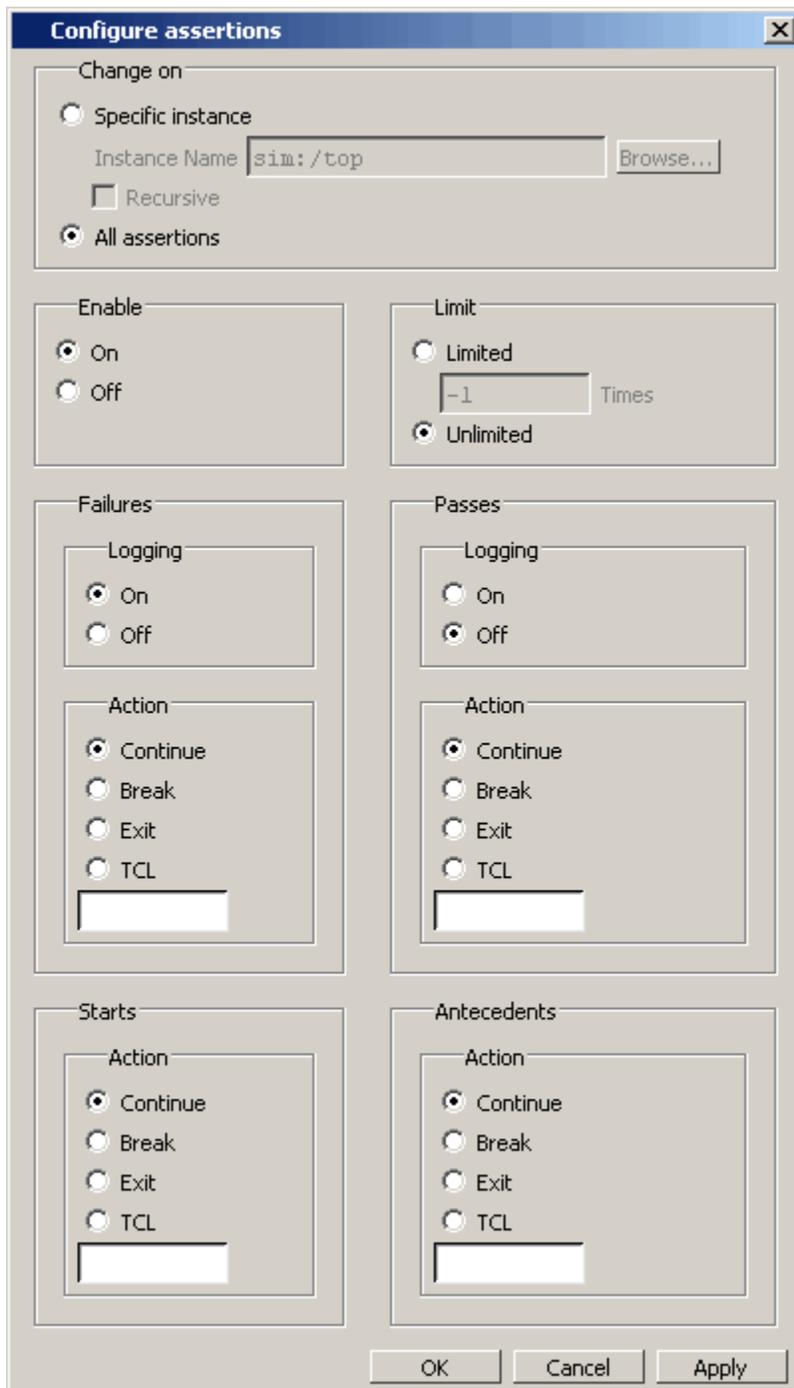
Configuring Assertions

The various tasks required to configure assertions via the command line or the GUI include:

- [Enabling Assertions](#)
- [Enabling Memory and Performance Profiling Data](#)
- [Enabling/Disabling Assertion Failure and Pass Logging](#)
- [Configuring Message Logging](#)
- [Setting Break Severity for Assertions](#)
- [Enabling/Disabling Assertion Failure and Pass Logging](#)
- [Setting Assertion Failure Limits](#)
- [Setting Assertion Actions](#)
- [Changing the Default Configuration of Cover Directives](#)

The GUI interface for configuring assertions is the Configure Assertions dialog, accessed via the **Assertions > Configure** menu selection when the Assertions window is active (Figure 22-2).

Figure 22-2. Configure Assertions Dialog



Enabling Assertions

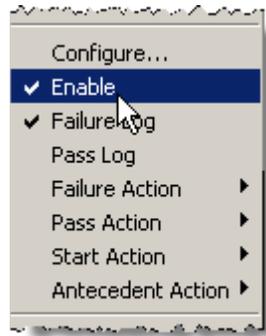
You can enable assertions by selecting “On” in the Enable section of the Configure Assertions dialog (Figure 22-2) or by using the [assertion enable](#) command. The assertion enable command allows you to turn on or off assertions of a specific language (SystemVerilog, PSL, VHDL) or type (concurrent or immediate).

The default value of the `AssertionEnable` variable in the `modelsim.ini` file is on ('1'), enabling all VHDL, PSL, and SystemVerilog assertions. You can override this variable by specifying:

assertion enable -off

You may also enable assertions by right-clicking an assertion in the Assertions window and selecting **Enable** from the popup menu (Figure 22-3). The selection acts as a toggle.

Figure 22-3. Assertion Enable Menu Selection

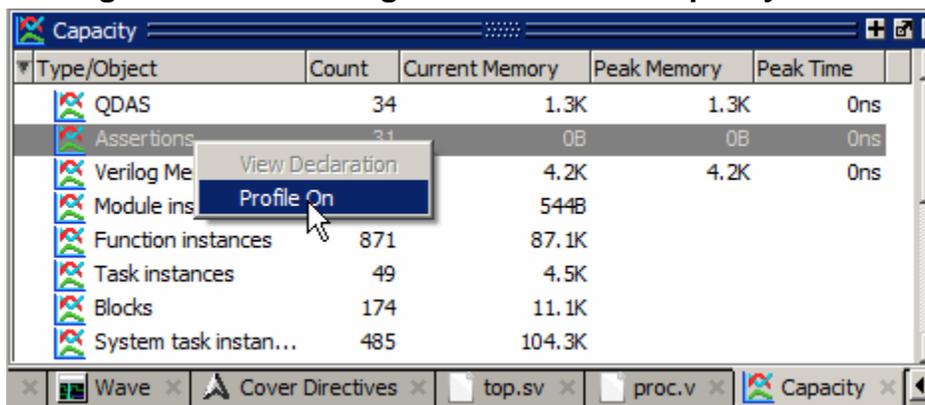


This is the equivalent of selecting **Assertions > Enable** from the menus when the Assertions window is active.

Enabling Memory and Performance Profiling Data

You can enable the collection of fine grained memory usage data for assertions and cover directives with the `assertion profile on` command. This command may be given at any time during simulation. You can also enable profiling in the Capacity window by right-clicking Assertions, Cover Directives, or Covergroups and selecting Profile On from the popup menu (Figure 22-4).

Figure 22-4. Selecting Profile On from Capacity Pane



The memory profile data is displayed in the Assertions and Cover Directives windows (select **View > Coverage > Assertions** and **View > Coverage > Cover Directives** to open these windows). Three columns in the Assertions and Cover Directives windows display this fine

grained profile information: Current Memory, Peak Memory, and Cumulative (number of threads).

Assertions that create the most threads take the most time to simulate. Therefore, this information is not only useful for memory profiling but also helps in performance profiling as well. The cumulative number of threads can help in scenarios where an assertion creates many short lived threads. For example, in the following assertion,

```
assert property (@(posedge clk) a | => b);
```

if 'a' is true throughout the simulation, the assertion will create a thread at every clock edge and the thread will remain alive for exactly one clock cycle. This assertion may not be one of the primary consumers of memory space but it will produce a high cumulative thread count.

The **-threadthreshold** switch for the [assertion profile](#) command can be used to identify assertion/cover directive threads that are exploding in memory. The correct syntax for the **assertion profile** command is:

```
assertion profile [-threadthreshold <number_of_threads>] on|off
```

The simulator will generate a message at every clock edge if the number of thread created by an assertion or cover directive is more than the threshold. For example,

```
assert profile -threadthreshold 100
```

will write the following message to the Transcript window as the simulation runs when the threshold of 100 threads is crossed:

```
# ** Note: Assertion thread threshold reached. Thread count = 110, Memory  
= 4.8KB  
# Time: 215 ns Scope: test.assert01 File: ./src/profile01.sv Line: 9
```

Note that this message is printed at every clock edge when the thread threshold is crossed. So if the threshold is too low, it will cause deluge of messages in the Transcript.

The assertion profile command is independent of **-assertdebug** switch for the [vsim](#) command (which stores assertion pass/fail data in the *.wlf* file) so can be used even if **vsim** was not invoked with **-assertdebug**.

This feature can be used to quickly identify assertions and cover directives that consume most memory and also are performance intensive. It helps identify assertions and cover directives that should be reviewed and rewritten in order to reduce the number of threads for better performance.

Configuring Message Logging

You control message logging for SystemVerilog assertions via severity tasks (\$fatal, \$error, \$info, \$warning) in an action block. You can determine which messages actually print in ModelSim by changing variables in the *modelsim.ini* file or via the Runtime Options dialog.

To set permanent defaults for message logging, edit the [IgnoreSVAError](#), [IgnoreSVAFatal](#), [IgnoreSVAInfo](#), and [IgnoreSVAWarning](#) variables in the *modelsim.ini* file.

To edit the message logging for the current simulation run only, select **Simulate > Runtime Options** and click the Message Severity tab in the Runtime Options dialog. Check the appropriate box(es) under Verilog ([Figure 22-5](#)) and click OK.

Figure 22-5. Selecting Message Logging

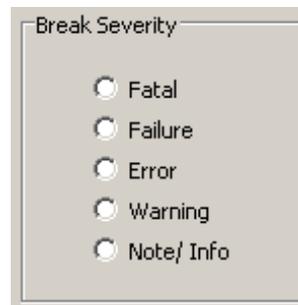


Setting Break Severity for Assertions

By default, a severity level of **Failure** causes a simulation break. To change this default permanently, edit the [BreakOnAssertion](#) variable in the *modelsim.ini* file.

To edit the severity for the current simulation run only, select **Simulate > Runtime Options** and click the Message Severity tab. Check the appropriate severity level ([Figure 22-6](#)) and click OK.

Figure 22-6. Setting Immediate Assertion Break Severity



Enabling/Disabling Assertion Failure and Pass Logging

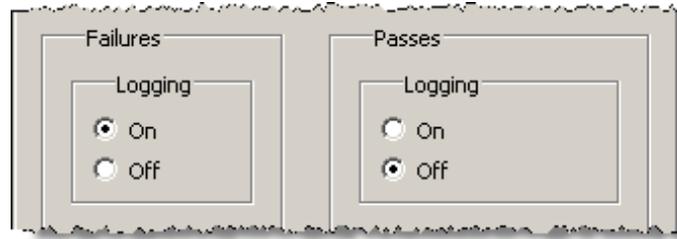
You can enable or disable failure and pass logging using the [assertion fail](#) or the [assertion pass](#) commands, respectively.

You can change the permanent defaults by setting the [AssertionFailLog](#) and [AssertionPassLog](#) variables in the *modelsim.ini* file.

To enable or disable an assertion's failure or pass logging from the GUI, right-click an assertion in the Assertions window and select **Failure Log** or **Pass Log** from the popup menu (Figure 22-3). The selection acts as a toggle.

You can also select **Assertions > Configure** from the menu bar (or, right-click an assertion and select **Configure**). This opens the **Configure assertions** dialog, where you can enable/disable failure or pass logging.

Figure 22-7. Enabling/Disabling Failure or Pass Logging



Note



Assertion pass logging can be enabled only if the [AssertionDebug](#) variable in the `modelsim.ini` file is on (set to 1), or if the `-assertdebug` option is used with the `vsim` command.

Setting Assertion Failure Limits

The assertion failure limit determines how many times ModelSim processes an assertion before disabling it for the duration of the simulation. By default, the failure limit is set to "unlimited." In other words, assertions will not be disabled during the simulation. You can change the permanent default by setting the [AssertionLimit](#) variable in the `modelsim.ini` file. Or, you can set failure limit using the `assertion fail` command. For example,

```
assertion fail -r / -limit 4 mydesign
```

sets the failure response limit to 4 for all assertions in `mydesign`. Each assertion failure will be responded to a maximum of 4 times during the current simulation. The `-r /` argument indicates that the assertion command should start at the root of `mydesign` and find all assertions.

To set the assertion failure limit with the GUI, make the Assertions window active and select **Assertions > Configure** from the menu bar; or, right-click an assertion in the Assertions window and selecting **Configure**. This opens the **Configure assertions** dialog where you can change the limit for all assertions or for the selected assertion in the Limit section of the dialog (Figure 22-8).

Figure 22-8. Setting Assertion Failure Limits



The assertion count is not verified until the end of the current time stamp. If multiple threads are active for a given property and if all of them fail at the same time, then all fail messages are reported. You may see more fail messages than the limit you set.

ModelSim continues to respond to assertions if their limit has not been reached. The limit applies to the entire simulation session and not to any single simulation run command.

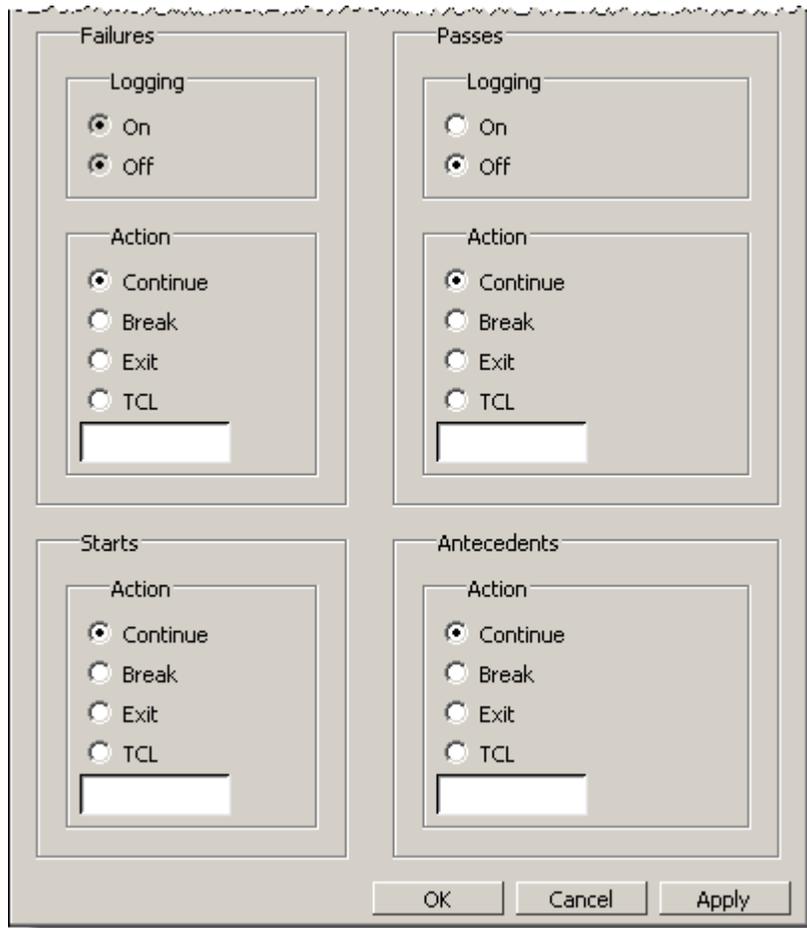
Setting Assertion Actions

You can set the action that will take place during simulation when an assertion passes or fails, when an assertion evaluation starts, and when an assertion antecedent is matched. ModelSim can take any one of four actions:

- Continue — (default) No action taken. This is the default value if you do not specify this switch.
- Break — Halt simulation and return to the ModelSim prompt.
- Exit — Halt simulation and exit ModelSim.
- TCL — Execute a tcl subroutine call.

You can set the assertion action with the [assertion action](#) command or in the GUI. To set the action in the GUI, select **Assertions > Configure** from the menus when the Assertions window is active, or right-click an assertion in the Assertions window and select **Configure**. This will open the Configure Assertions dialog, where you can set the assertion actions.

Figure 22-9. Set Assertion Actions

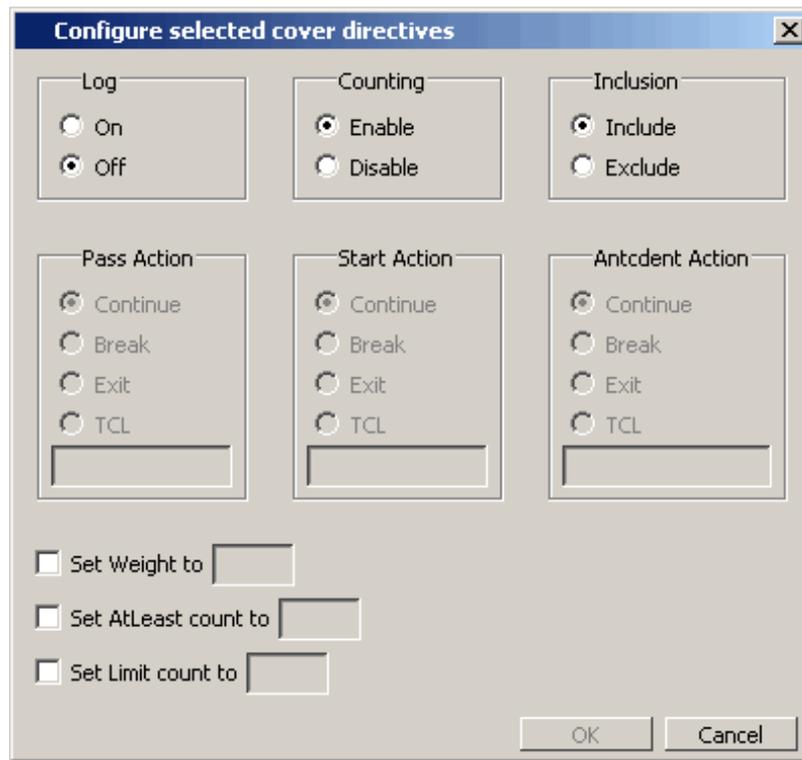


Changing the Default Configuration of Cover Directives

After compiling the design, you may want to edit the default configuration for individual cover directives. Follow these steps to edit the default values:

1. Select **View > Coverage > Cover Directives** to see your directives in the Cover Directives window.
2. Make the Coverage Directives window active and select one or more cover directives. Then, select **Cover Directives > Configure** in the menu bar, or right-click the selected cover directive and select **Configure Directive** from the popup menu. This opens the "Configure selected cover directives" dialog ([Figure 22-10](#)).

Figure 22-10. Configure Selected Cover Directives Dialog



This dialog allows you to enable/disable directive counting and logging, include/exclude cover directives from coverage statistics calculations, set a weight for directives, and specify a minimum number of times a directive should fire. You can also set the action for coverage directive passes, starts, and antecedent matches.

Setting Cover Directive Actions

In the “Configure selected cover directives” dialog you can choose from four different actions for cover directive passes, starts, and antecedent matches:

- Continue — No action is taken.
- Break — Halt simulation and return to the Questa prompt.
- Exit — Halt simulation and exit Questa.
- TCL — Execute designated tcl subroutine.

Weighting Cover Directives

As shown in [Figure 22-10](#), you can assign weights to cover directives. Weighting affects the aggregated coverage statistics in the currently selected design region. A directive with a weight of 2 has twice the effect of a directive with a weight of 1. Conversely, assigning a directive a

weight of 0 would omit the directive from the statistics calculation. See “[Coverage Aggregation in the Structure Window](#)” for more details.

Weighting is a decision you make as to which cover points are more important than others within the context of the design and the objectives of the test bench. You can change weighting based on the simulation run so specific runs could be setup with different test bench objectives. In this way, weighting is a good way of filtering how close the test bench is to achieving its objectives.

For example, the likelihood that each type of bus transaction could be interrupted in a general test is very low as interrupted transactions are normally rare. But you might want to ensure the design handles the interrupt of all types of transactions and recovers properly from them. To accomplish this, you can construct a test bench so the stimulus is constrained to ensure that all types of transactions are generated and that the probability of transactions being interrupted is relatively high. For that test bench, the weighting of the interrupted transaction cover points would probably be higher than the weightings of uninterrupted transactions (or other coverage criteria).

Choosing AtLeast Counts for Cover Directives

The AtLeast count is a minimum threshold of coverage that gives you some confidence that the run was meaningful. You do not need to set this threshold on every directive, but you should understand which minimal thresholds make for a useful simulation run based on your design and the objectives of the verification session.

For example, say your test bench requires a certain level of PCI traffic during the simulation. 30 PCI STOP transactions might be a proxy measure of sufficient PCI traffic, so you would set an AtLeast count of 30 on the "PCI STOP" cover directive. Another example might be that a FIFO full should have been achieved at least once as that would indicate that enough activity occurred during the simulation to reach a key threshold. So, your "FIFO full" directive would get an AtLeast count of 1.

Limiting Cover Directives

Cover directives that are evaluated frequently during simulation can adversely affect runtime efficiency. To improve runtime efficiency you can disable cover directives after a number of counts with the **Set Limit count to** option in the Configure selected cover directives dialog ([Figure 22-10](#)), or with the **-limit <count>** argument for the [fcover configure](#) command. For example,

```
fcover configure -limit 5 /top/refresh_during_rw
```

limits the evaluation of the *refresh_during_rw* cover directive to a count of 5 then disables it. Once a cover directive is disabled, the assertion engine (which implements cover directives) no longer makes a kernel call associated with that directive, thus improving simulation runtime efficiency.

Simulating Assertions

If any assertions were compiled, the `vsim` command automatically invokes the assertion engine. If you do not want to simulate compiled assertions, use the `-nopsl` switch to ignore PSL assertions or the `-nosva` to ignore SystemVerilog assertions. You can perform the same action in the GUI by selecting **Disable PSL** and **Disable SVA** in the Others tab of the Start Simulation dialog when you load the design with the **Simulate > Start Simulation** menu selections.

If you want to have access to the details of assertion failures in the Assertion Debug pane of the Wave window, use the `-assertdebug` switch with `vsim`; or, select **Enable assertion debug** in the Others tab of the Start Simulation dialog when you load the design with the **Simulate > Start Simulation** menu selections.

To invoke assertion thread viewing of specific assertions, use the `-enable` switch with the `atv log` command after loading the design with `vsim`, and before the simulation is run.

Maintaining Assertion Counts

In previous versions of ModelSim (prior to version 10.0a), a detailed count of assertion actions – such as passes, failures, vacuous passes, disabled, active, etc. – were maintained only if the design was compiled with `+acc=a` and the simulation run with the `-assertdebug` option for the `vsim` command. Now the `-assertcover` option for `vsim` allows you to generate assertion counts even on fully optimized designs (those compiled without `+acc=a`) without the performance impact of using `-assertdebug`.

The specific actions counted depends on whether the simulation is run with the `-assertcover` or the `-assertdebug` option for `vsim` [or without either option](#).

For example, consider the following assertion:

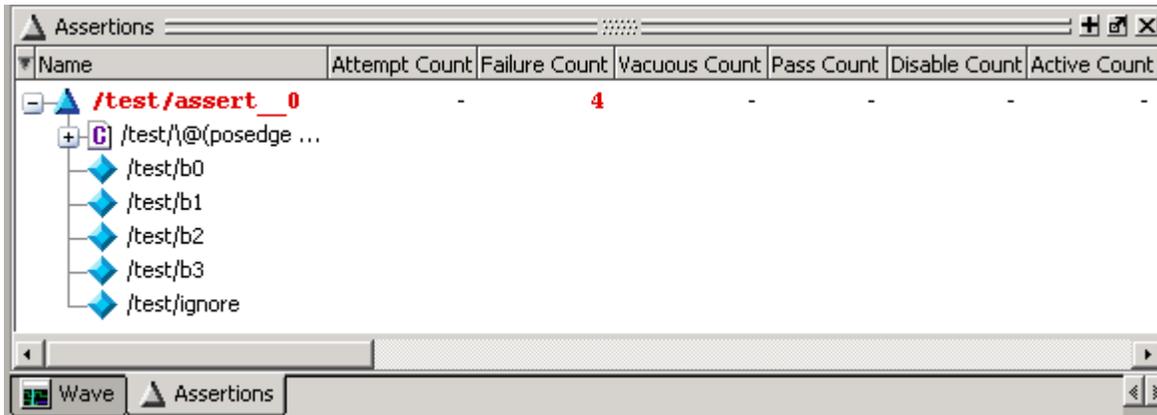
```
assert property (@(posedge clk) disable iff (ignore) (b0 | => b1 ##1 b2 ##1 b3));
```

In the following discussion we show what happens to the assertion counts for this assertion when the simulation is run first without `-assertcover` or `-assertdebug`; then with `-assertcover`; and then with `-assertdebug`.

Assertion Counts without `-assertcover` or `-assertdebug`

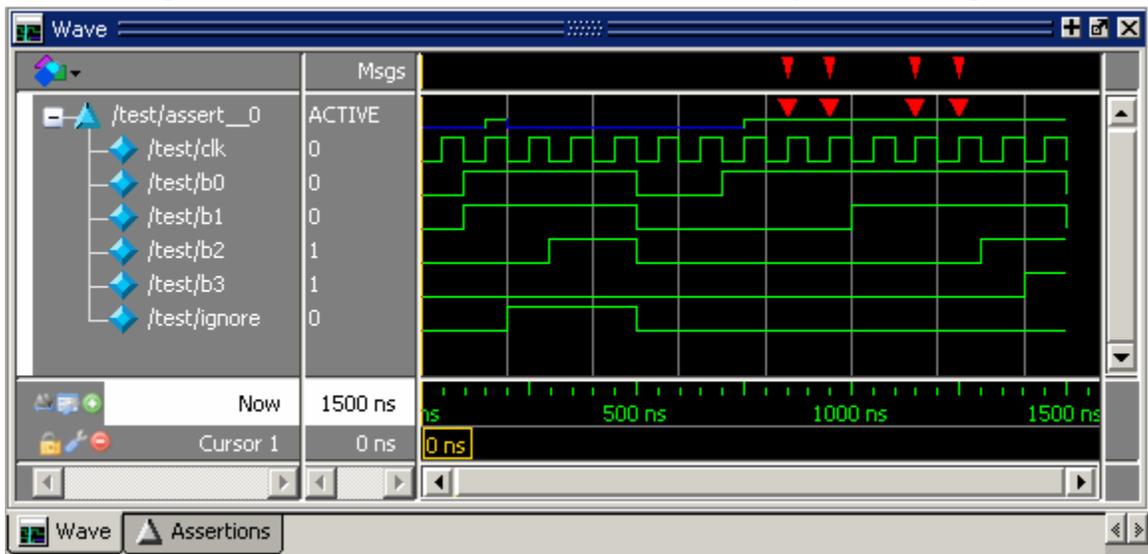
When the simulation is run without `-assertcover` or `-assertdebug` the Assertions window only shows the Failure Count, which in this case is 4 ([Figure 22-11](#)).

Figure 22-11. Failure Counts in the Assertions Window



If the design has been compiled with the `+acc=a` option, you can view the assertion waveform in the Wave window, as shown in Figure 22-12. The assertion failures are indicated in the Wave window by red triangles.

Figure 22-12. Assertion Failures Indicated by Red Triangles



Assertion Counts with `-assertcover`

The `-assertcover` option for `vsim` allows you to generate assertion counts on a fully optimized design without using the `+acc=a` option during compile, and without incurring the impact on performance that occurs when `-assertdebug` is used.

When the simulation is run with the `-assertcover` option on the assertion property above, the assertion in the Assertions Window contains a different column for each assertion count, as shown in Figure 22-13.

Figure 22-13. Assertion Counts in the Assertions Window

Name	Attempt Count	Failure Count	Vacuous Count	Pass Count	Disable Count	Active Count
/test/assert_0	15	4	3	1	4	3

- Attempt Count - 15 — The number of times the assertion was attempted, which basically corresponds to the number of clock edges for the assertion.
- Failure Count - 4— The number of start attempts that resulted in failure.
- Vacuous Count - 3— The number of start attempts when the assertion passed vacuously.
- Pass Count - 1— The number of start attempts when the assertion passed.
- Disable Count - 4 — The number of start attempts that were disabled due to the disable condition being TRUE.
- Active Count - 3— The number of start attempts that are currently active.

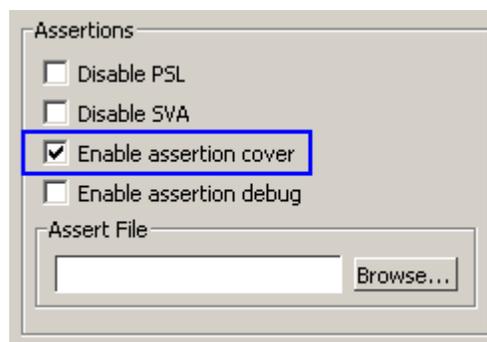
For these counts, note that:

$$\text{Attempt} = \text{Failure} + \text{Vacuous} + \text{Pass} + \text{Disable} + \text{Active}$$

To enable assertion coverage with the GUI:

1. Select **Simulate > Start Simulation** to open the Start Simulation dialog.
2. Open the **Others** tab.
3. In the Assertions section, select **Enable assertion cover** (Figure 22-14).

Figure 22-14. Enable Assertion Coverage

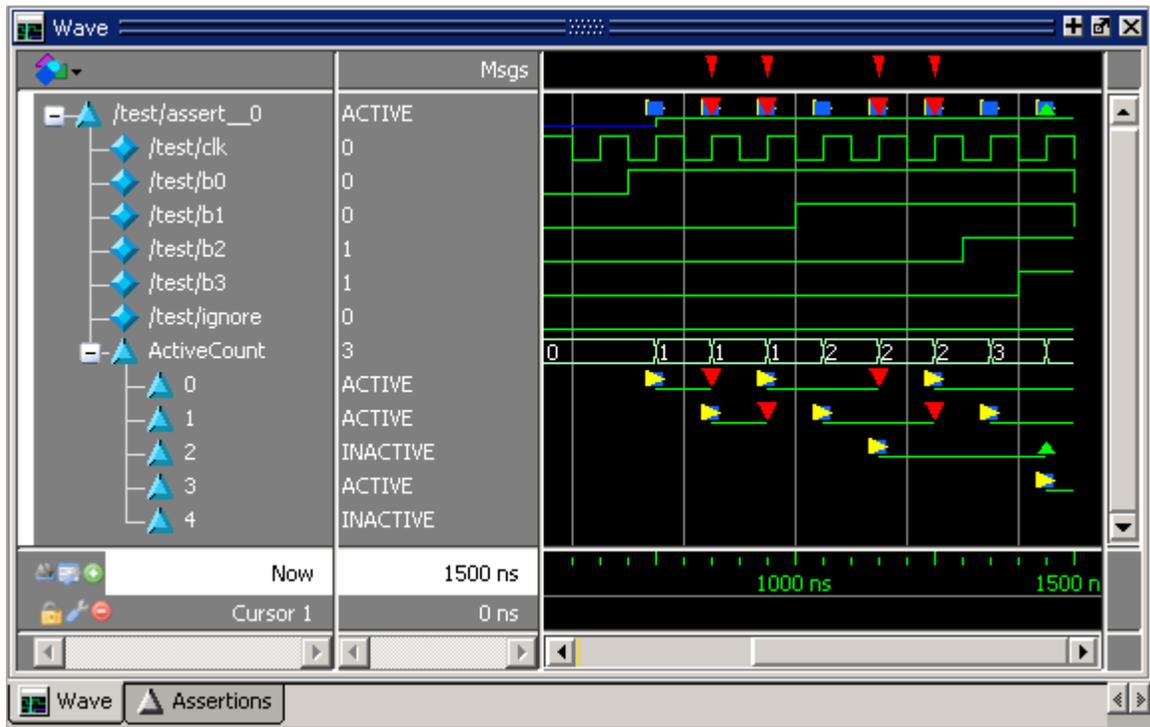


These assertion counts can also be enabled with the [AssertionCover](#) modelsim.ini variable.

Assertion Counts with -assertdebug

When simulation is run with -assertdebug, the assertion property is displayed in the Wave window as shown in Figure 22-15.

Figure 22-15. Assertion Indicators When -assertdebug is Used



In addition to the red triangles used to denote assertion failures, the Wave window includes the following assertion indicators when -assertdebug is used:

- blue square = start of assertion thread
- green triangle = assertion pass
- yellow triangle = antecedent match

In addition, the Wave window includes the assertion's ActiveCount, which is the number of active assertion threads at the current time.

The Assertions Window contains a different column for each assertion count, as shown in (Figure 22-13).

Figure 22-16. Counts Columns in Assertions Window

Name	Attempt Count	Failure Count	Vacuous Count	Pass Count	Disable Count	Active Count	Peak Active Count
/test/assert_0	15	4	3	1	4	3	3

- **Attempt Count - 15** — The number of times the assertion was attempted, which basically corresponds to the number of clock edges for the assertion.
- **Failure Count - 4** — The number of start attempts that resulted in failure. In this case, the attempts that started at 750, 850, 950 and 1050 ns resulted in failures
- **Vacuous Count - 3** — The number of start attempts when the assertion passed vacuously. In this case, the start attempts whenever 'b0' was FALSE - at 50, 550, 650 ns.
- **Pass Count - 1** — The number of start attempts when the assertion passed. In this case, for the attempt that started at 1150 ns.
- **Disable Count - 4** — The number of start attempts that were disabled due to the disable condition being TRUE. In this case the attempts that started at 150, 250, 350 and 450 ns.
- **Active Count - 3** — The number of start attempts that are currently active. In this case, the attempts that started at 1250, 1350 and 1450 ns have not yet completed.
- **Peak Active Count - 4** — This represent the maximum number of start attempts active at any time.

For these counts, note that:

$$\text{Attempt} = \text{Failure} + \text{Vacuous} + \text{Pass} + \text{Disable} + \text{Active}$$

Note *Assertion Success* is a term used to describe coverage statistics for assertions. Assertion Successes are those assertions that never failed and passed at least once. In the absence of "-assertdebug," Assertion Passes are not counted, and Assertion Successes are those assertions that never failed.

Analyzing Assertions and Cover Directives

The following tasks can be used for analyzing assertions and cover directives:

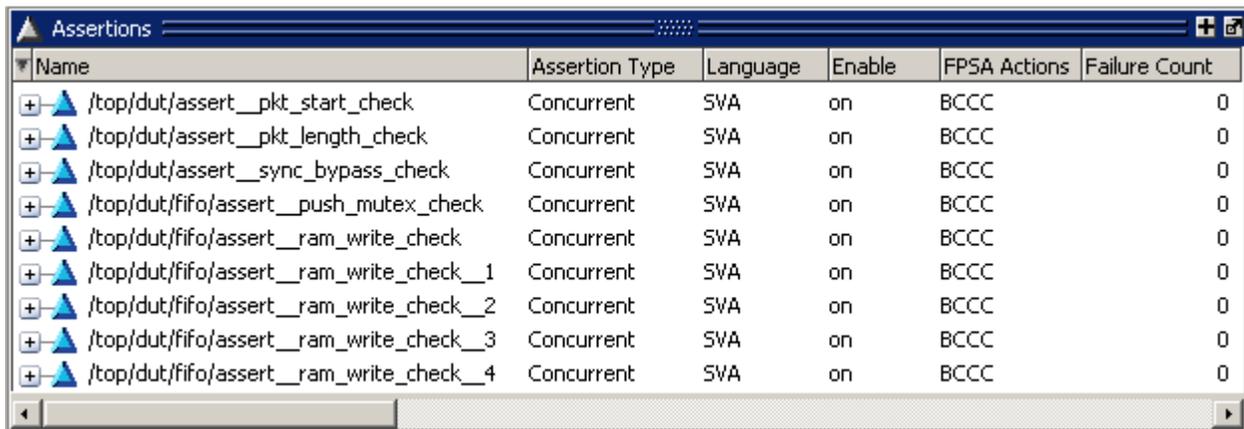
- [Viewing Assertions in the Assertions Window](#)
- [Viewing Cover Directives in the Cover Directives Window](#)
- [Viewing Memory Profile Data](#)
- [Viewing Assertions and Cover Directives in the Wave Window](#)
- [Comparing Assertions](#)

Viewing Assertions in the Assertions Window

The Assertions windows displays simulation data about assertions. To open the Assertions window, select **View > Coverage > Assertions** from the menus.

[Figure 22-17](#) shows SystemVerilog assertions in the Assertions window. SV assertions are indicated by a light blue triangle. PSL assertions (not shown) are indicated by a purple triangle.

Figure 22-17. SystemVerilog Assertions in the Assertions Window



Name	Assertion Type	Language	Enable	FPSA Actions	Failure Count
+ ▲ /top/dut/assert_pkt_start_check	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/assert_pkt_length_check	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/assert_sync_bypass_check	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/fifo/assert_push_mutex_check	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/fifo/assert_ram_write_check	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/fifo/assert_ram_write_check_1	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/fifo/assert_ram_write_check_2	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/fifo/assert_ram_write_check_3	Concurrent	SVA	on	BCCC	0
+ ▲ /top/dut/fifo/assert_ram_write_check_4	Concurrent	SVA	on	BCCC	0

The Assertions window lists all embedded and external assert directives that were successfully compiled and simulated during the current session. The plus sign ('+') to the left of the Name field lets you expand the assertion hierarchy to show its elements (properties, sequences, clocks, and HDL signals).

The Assertions window includes several columns for displaying information about assertions. See [GUI Elements of the Assertions Window](#) for a description of each field.

When assertions fire with failure messages, the Assertions window displays the name and failure count in red, both during simulation and in post-simulation mode ([Figure 22-18](#)).

Figure 22-18. Assertion Failures Appear in Red

Name	Assertion Type	Language	Enable	FPSA Actions	Failure Count
assert_ram_write_check_6	Concurrent	SVA	on	BCCC	0
assert_ram_write_check_7	Concurrent	SVA	on	BCCC	0
assert_ram_write_check_8	Concurrent	SVA	on	BCCC	0
assert_ram_write_check_9	Concurrent	SVA	on	BCCC	0
assert_ram_write_check_10	Concurrent	SVA	on	BCCC	1
assert_ram_read_check	Concurrent	SVA	on	BCCC	0
assert_ram_read_check_1	Concurrent	SVA	on	BCCC	0
assert_ram_read_check_2	Concurrent	SVA	on	BCCC	0

You can use the `assertion count` command to return the sum of the assertion failure counts for a specified set of assertion directive instances. This command returns a "No matches" warning if the given path does not contain any assertions.

Assertions Window Display Options

The Assertions window can display assertion directives in hierarchical (tree) mode (Figure 22-19) or in the flattened form (Figure 22-17).

Figure 22-19. Hierarchy Display Mode

Name	Assertion Type	Design Unit	Design Unit Type	Language	Enable
avm_pkg					
top					
pins_if					
di_handshake	Concurrent	rdyacpt_pins_if	VHDL	SVA	on
do_handshake	Concurrent	rdyacpt_pins_if	VHDL	SVA	on
hs_chk					
di_data_hold	Concurrent	rdyacpt_pins_if	VHDL	SVA	on
do_data_hold	Concurrent	rdyacpt_pins_if	VHDL	SVA	on
dut					
interleaver_svc_pkg					

The hierarchical display mode can be enabled or disabled by doing any one of the following:

- When the Assertions window is docked, select **Assertions > Display Options > Hierarchy Mode** from the Main menus.
- Right-click in the Assertions window and select **Display Options > Hierarchy Mode** from the popup menu.

The Display Options menu also includes the following options:

- The **Recursive Mode** option displays all assertions at and below the selected hierarchy instance, the selection being taken from a Structure window. (i.e., the **sim** tab). Otherwise only items actually in that particular scope are shown.
- The **Show All Contexts** option displays all instances in the design. It does not following the current context selection in a structure pane. The Show All Context display mode implies the recursive display mode as well, so the **Recursive Mode** selection is automatically grayed out.
- The **Show Concurrent Asserts** option displays only concurrent assertions.
- The **Show Immediate Asserts** option displays only immediate assertions.

Viewing Cover Directives in the Cover Directives Window

The Cover Directives window displays information about cover directives. To open the Cover Directives window, select **View > Coverage > Cover Directives**.

Figure 22-20 shows PSL cover directives in the Cover Directives window. PSL cover directives are indicated by a purple chevron. SystemVerilog cover directives (not shown) are indicated by a light blue chevron.

Figure 22-20. PSL Cover Directives in the Cover Directives Window

Name	Language	Enabled	Log	Count	AtLeast	Limit	Weight	Cmplt %	Cmpltd
+ /tb/c_reset	PSL	✓	Off	1	1	nlim...	1	100%	█
- /tb/c_refresh	PSL	✓	Off	1	1	nlim...	1	100%	█
- /tb/!(posedge clk)									
- /tb/clk									
+ /tb/cover_refresh									
+ /tb/cntrl/c_refresh_during_rw	PSL	✓	Off	0	1	nlim...	1	0%	█

The Cover Directives window displays accumulated cover directive statistics at the current simulation time, including percentages and a graph for each directive and instance. The plus sign ('+') to the left of the Name field lets you expand the directive hierarchy to show its elements (properties, sequences, clocks, and HDL signals). Refer to [GUI Elements of the Cover Directives Window](#) for a description of each column.

Display Options for Cover Directives

Display options allow you to display cover directives in a **Recursive Mode** or in a **Show All Contexts** mode. For details, see [Changing the Cover Directives Window Display Options](#).

Filtering Data in the Assertions and Cover Directives Window

You can filter the Assertions and Cover Directives data displayed by selecting **Assertions > Filter > Setup** or **Cover Directives > Filter > Setup**, depending on which window is active. For details, see “[Filtering Functional Coverage Data](#)”.

Viewing Memory Profile Data

The `assertion profile` command generates a fine grained profile of memory usage for assertions and cover directives. The results are displayed in the Memory, Peak Memory, Peak Memory Time, and Cumulative Threads columns of the Assertions and Cover Directives windows.

- The **Memory** column tracks the current memory used by the assertion or cover directive.
- The **Peak Memory** column tracks the peak memory used by the assertion or cover directive.
- The **Peak Memory Time** column indicates the simulation run time at which the peak memory usage occurred.
- The **Cumulative Threads** column counts the cumulative thread count for the assertion.

While the Cumulative Threads count is not specifically about memory, it is designed to highlight those assertions and cover directives that are starting too many attempts, such as the following assertion:

```
assert property ((@posedge clk) a | => b);
```

If ‘a’ is true throughout the simulation, then the above assertion will start a brand new attempt at every clock. An attempt, once started, will only be alive until the next clock. So this assertion will not appear abnormally high in the **Memory** and **Peak Memory** columns, but it will have a high count in the **Cumulative Threads** column.

Viewing Assertions and Cover Directives in the Wave Window

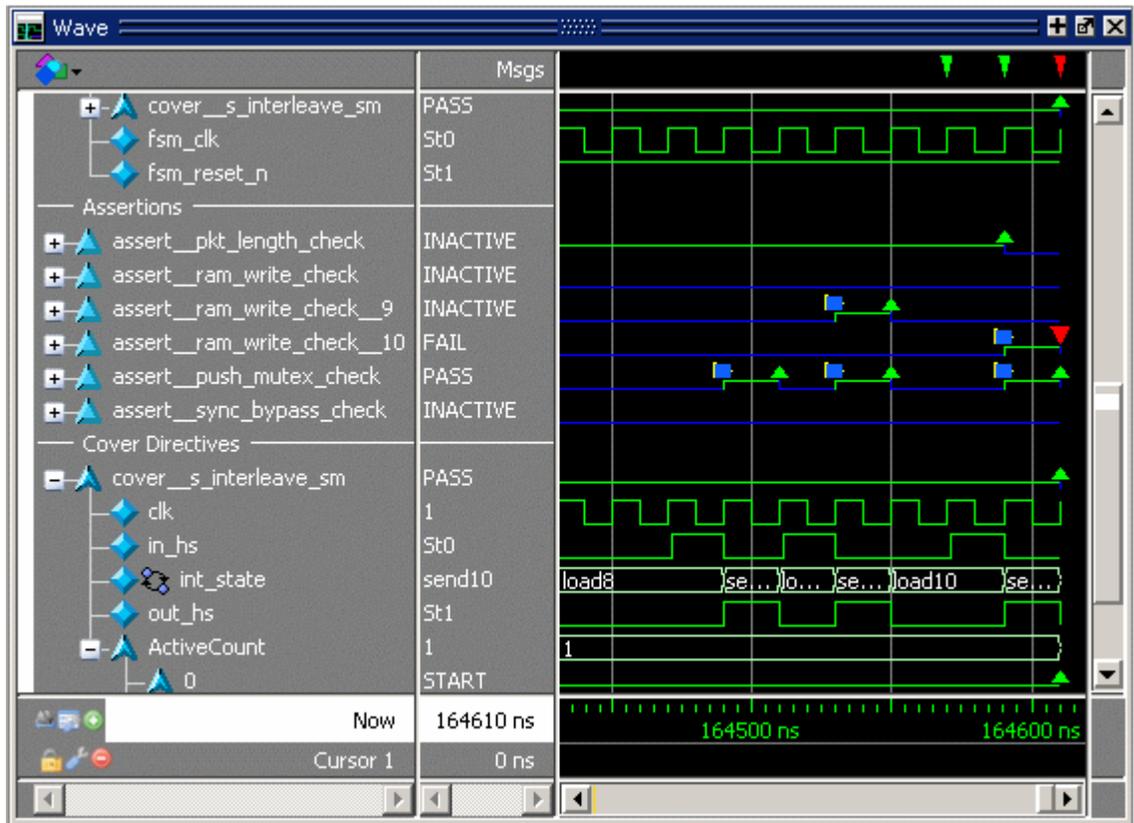
You can view assertions and cover directives in the Wave window just like any other signal in your design. Use one of the following methods to add directives to the Wave window:

- To add all assertions in your design to the Wave window,
 - Select a single object in the Assertions window, then select **Add > To Wave > Objects in Design** from the main menus.
 - Select all objects in the Assertions window, then select **Add > To Wave > Selected Objects** from the main menus
 - Right-click the selected assertions, then select **Add Wave > Objects in Design** from the popup menu.

- Select all objects in the window, then click the **Add Selected To Window** button in the [Standard Toolbar](#). 
- To add all cover directives in your design to the Wave window:
 - Select a single directive in the Cover Directives window, then select **Add > To Wave > Functional Coverage in Design** from the main menus.
 - Select all directives in the Cover Directives window, then select **Add > To Wave > Selected Functional Coverage** from the main menus.
 - Right-click the selected cover directives, then select **Add Wave > Functional Coverage in Design** in Design from the popup menu.
 - Select all directives in the window, then click the **Add Selected To Window** button in the [Standard Toolbar](#). 
- To place a single assertion or cover directive in the Wave window:
 - Drag the object from the its window and drop it into the Wave window, or simply drop it onto the Wave tab if it is showing.
 - Select the object, then click the **Add Selected To Window** button in the [Standard Toolbar](#).
 - Select the object then select **Add > To Wave > Selected Objects** from the menu bar.
- Right-click any selected assertion and select **Add Wave > Selected Objects** from the popup menu; or right-click any selected cover directive and select **Add Wave > Selected Functional Coverage** from the popup menu.

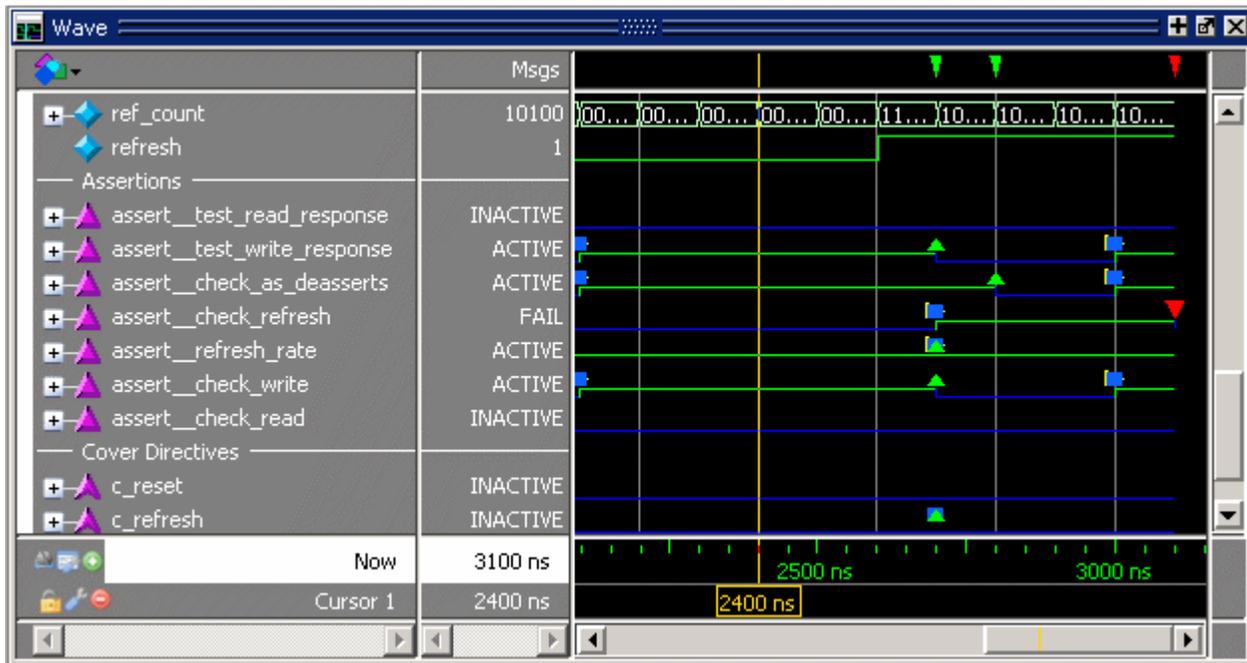
ModelSim represents assertions and cover directives as signals or waveforms in the Wave window. The Wave window in [Figure 22-21](#) shows several SystemVerilog assertions and a single cover directive. SystemVerilog assertions are represented by light blue triangles in the pathnames column. SystemVerilog cover directives are represented by light blue chevrons.

Figure 22-21. SystemVerilog Assert and Cover Directives in the Wave Window



The Wave window in [Figure 22-22](#) shows several PSL assertions and cover directives. PSL assertions are represented by magenta triangles. PSL cover directives are represented by magenta chevrons.

Figure 22-22. PSL Assert and Cover Directives in the Wave Window



The name of each assertion and cover directive comes from the assertion code. The plus sign ('+') to the left of the name indicates that an assertion or cover directive is a composite trace and can be expanded to show its elements (properties, sequences, clocks, and HDL signals). Note that signals are flattened out; hierarchy is not preserved.

The value in the value pane is determined by the active cursor in the waveform pane. The value will be one of ACTIVE, INACTIVE, PASS, FAIL, or ANTCDENT.

The waveform for an assertion or cover directive represents both continuous and instantaneous information.

- Continuous information is either active or inactive. The directive is active anytime it matches the first element in the directive. When active, the trace is green; when inactive it is blue.
- Instantaneous information is represented as a start, pass, or fail event. A start event is shown as a blue square. A green triangle represents a pass. And a red triangle indicates a fail.

A yellow triangle represents an antecedent match (Figure 22-23). The yellow triangle is displayed only if the directive is browseable and assertion debug is on (vsim -assertdebug). The yellow triangle is shown for each thread of the assertion under ActiveCount in the assertion (see Using the Assertion Active Thread Monitor). The signal values of the assertion also reflect the antecedent match (ANTCDENT).

Figure 22-23. Antecedent Matches Indicated by Yellow Triangle

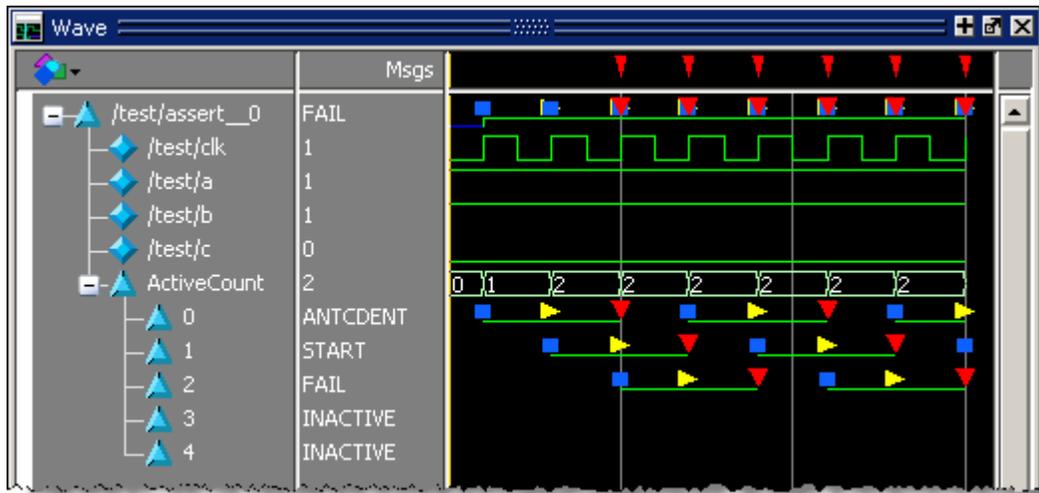


Table 22-1 summarizes the graphic elements for assertions and cover directives used in the Wave and ATV windows (see [Viewing Assertion Threads in the ATV Window](#)):

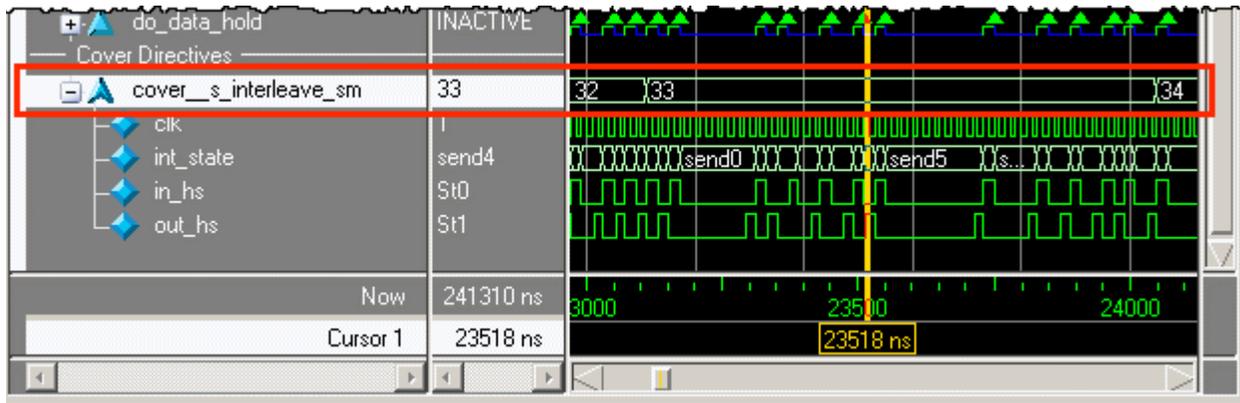
Table 22-1. Graphic Elements for Assertions and Cover Directives

Graphic element	Meaning
blue line	assertion or cover directive is inactive
green line	assertion or cover directive is active
blue square	assertion or cover directive starts
green triangle	assertion or cover directive passed
red triangle	assertion or cover directive failed
yellow triangle	antecedent match occurred in assertion

Displaying Cover Directives in Count Mode

You can change the coverage directive waveform in the Wave window so it displays in count mode format, which shows the instantaneous waveform value as a decimal integer. To change to count-mode format, right-click a coverage waveform name and select **Cover Directive View > Count Mode**.

Figure 22-24. Viewing Cover Directive Waveforms in Count Mode



Count mode can be useful for gauging the effectiveness of stimulus over time. If all cover directive counts are static for a long period of time, it may be that the stimulus is acting in a wasteful manner and can be improved.

Comparing Assertions

ModelSim's compare feature allows you to compare assertions (which includes any assertion-like object such as `accAssertion`, `accCover`, `accEndpoint`, or `accImmediateAssert`.) There is no cross-compare with assertion types outside the set listed, and assertion compare is further limited to like types only. That is, both the reference and test items must be of the same type.

Comparing assertion signals differs from comparing normal HDL signals/ports because assertion signals have two attributes:

- The current assertion state (ACTIVE | INACTIVE)
- The current assertion event (START | PASS | FAIL | EVAL)

Assertions expand to show child signals but these child signals don't participate in the compare evaluation. Child signals are, however, visible in the compare waveforms when you expand compare assertions.

Setting Up the Assertions Compare

You can set up and run an assertion compare using the **compare** commands or the menu-based Waveform Comparison Wizard. For example, a comparison using compare commands may look like the following:

```
add wave /top/assert_sig
run 1000 ns
dataset open vwim_test.wlf
compare start sim vsim_test
compare add sim:/top/assert_sig vsim_test:/top/assert_sig
compare run
```

All existing compare commands are supported for comparing assertion signals. Refer to the Command Reference for syntax and command descriptions.

The Waveform Comparison Wizard will guide you through the selection of a reference dataset and a test dataset. Assertions within those datasets are compared along with other signals. You can start the Wizard is by selecting **Tools > Waveform Compare > Comparison Wizard**.

The Compare Signal

When two assertion signals are compared — for example, *vsim_pass:/top/my_assertion_sig* and *vsim_fail:/top/my_assertion_sig* — a third virtual signal is created:

```
compare:/top/my_assertion_sig<>my_assertion_sig\
```

The *compare* signal created is composed of the reference signal and the test signal. Differences between the reference and test assertion signals are highlighted in red in the *compare* signal when it is displayed in the Wave Window. Assertion differences cannot be viewed in the ATV window.

Two Types of Assertion Differences

There are two types of assertion differences:

- **Instantaneous difference** — When the assertion event (START | PASS | FAIL | EVAL) is different but the state of the assertion (ACTIVE | INACTIVE) is the same.

For example, considering two datasets *vsim_top* and *vsim_ntop* with an assertion signal *my_assertion_sig*.

```
vsim_top:/top/my_assertion_sig is PASS_INACTIVE at 20 ns
```

```
vsim_ntop:/top/my_assertion_sig is FAIL_INACTIVE at 20ns
```

This is an instantaneous difference the difference will marked at time 20 ns and the width of the difference marker will be equal to the width of the PASS/FAIL symbol.

- **Range difference** — When there is a state change (ACTIVE->INACTIVE) or vice-versa, between the reference and test assertion, irrespective of the event on the assertions.

Child Signals

An assertion object is composed of child signals. It is the evaluation of these child signals that determine the assertion event (START/PASS/FAIL). If you choose to expand the assertion, the difference marker is propagated to the child signals as well, but this may not necessarily mean a change in value on the child signal at that specific time — the difference could have occurred earlier.

If the reference signal has child signals but the test signal does not, or vice-versa, waveform compare will still work because compare cares only about the absolute event on the assertion. If there is a difference, it will be marked.

Saving Metrics to the UCDB

You can save assertion and cover directive metrics to the Unified Coverage Database (UCDB) with the [coverage save](#) command.

Note



For easiest viewing and tracking of assertions and cover directives in the GUI and coverage reports, it is recommended that you name all assertions and directives in the source files. See [Assertion/Cover Directive Naming Conventions](#) for more information.

When coverage save is used without switches and arguments, all assertion and cover directive metrics are saved to the UCDB. For details, see [Saving Assertion and Cover Directive Metrics](#).

Excluding Assertions and Cover Directives

If the -code option is not specified with the [coverage exclude](#) command, all assertions and cover directives, as well as all code coverage types, will be excluded from the coverage database.

To exclude assertions and cover directives from a loaded coverage database (.ucdb), use the coverage exclude command options, as follows:

```
coverage exclude -assertpath <path_to_assert>
```

```
coverage exclude -dirpath <path_to_directive>
```

The coverage exclude -assertpath and -dirpath options are only operational in the Coverage View mode. During active simulation, these command options have no effect. See [coverage exclude](#) for full syntax details.

Creating Assertion Reports

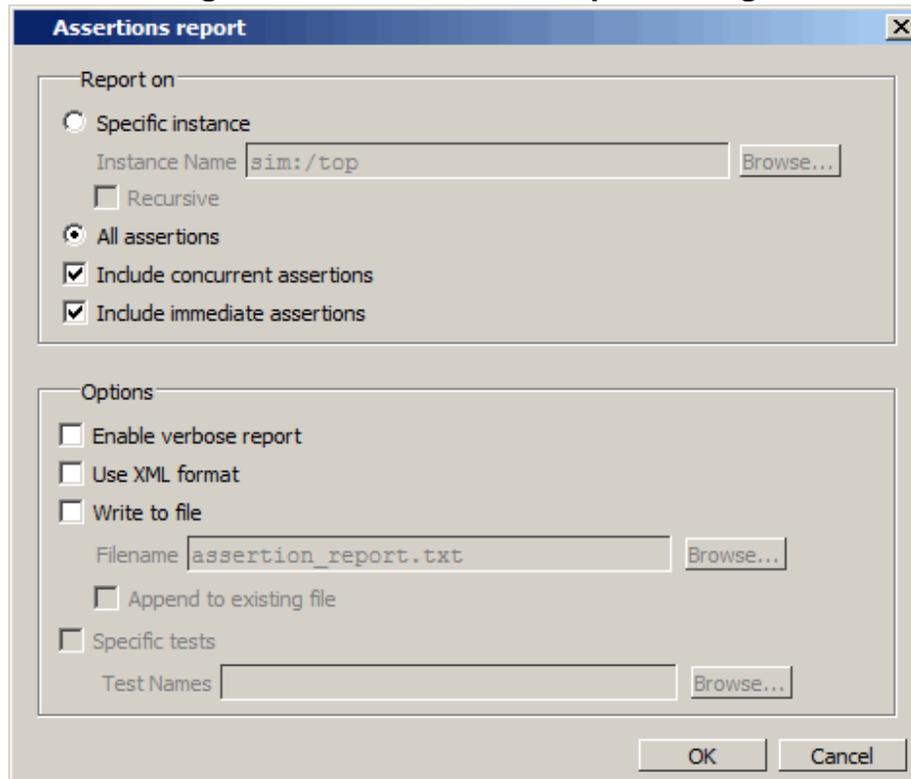
You can create reports on assertions and cover directives using dialogs accessible through the GUI or via commands entered at the command line prompt. SVA immediate and concurrent assertions, VHDL immediate assertions, and PSL assertions are all included in the assertion report. The Assertion Type column in Assertions window distinguishes the immediate and concurrent assertions. You can create assertion reports by:

- [Using the Assertions Report Dialog](#)
- [Specifying an Alternative Output File for Assertion Messages](#)

Using the Assertions Report Dialog

To save an ASCII file of assertion coverage results, right-click anywhere in the Assertions window and select **Report** from the popup context menu. This opens the Assertions report dialog (Figure 22-25).

Figure 22-25. Assertions Report Dialog



Specifying an Alternative Output File for Assertion Messages

You can specify an alternative output file for recording assertion messages. To do this, invoke **vsim** with the **-assertfile <filename>** switch. By default assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file. You can set a permanent default for the alternative output file using the [AssertFile](#) variable in the *modelsim.ini* file.

Note



The output file defined by the **-assertfile <filename>** switch will also contain VHDL assert messages.

Limitations

In some circumstances, processing cover directive will produce too many matches, causing the cover count to be too high. The problem occurs with coverage of sequences like $\{\{a;b\} | \{c;d\}\}$ or $\{a[*1 to 2]; b[*1 to 2]\}$. In this instance, the same sequence for the same input at the same start time may succeed simultaneously in multiple ways. The first sequence may succeed with a

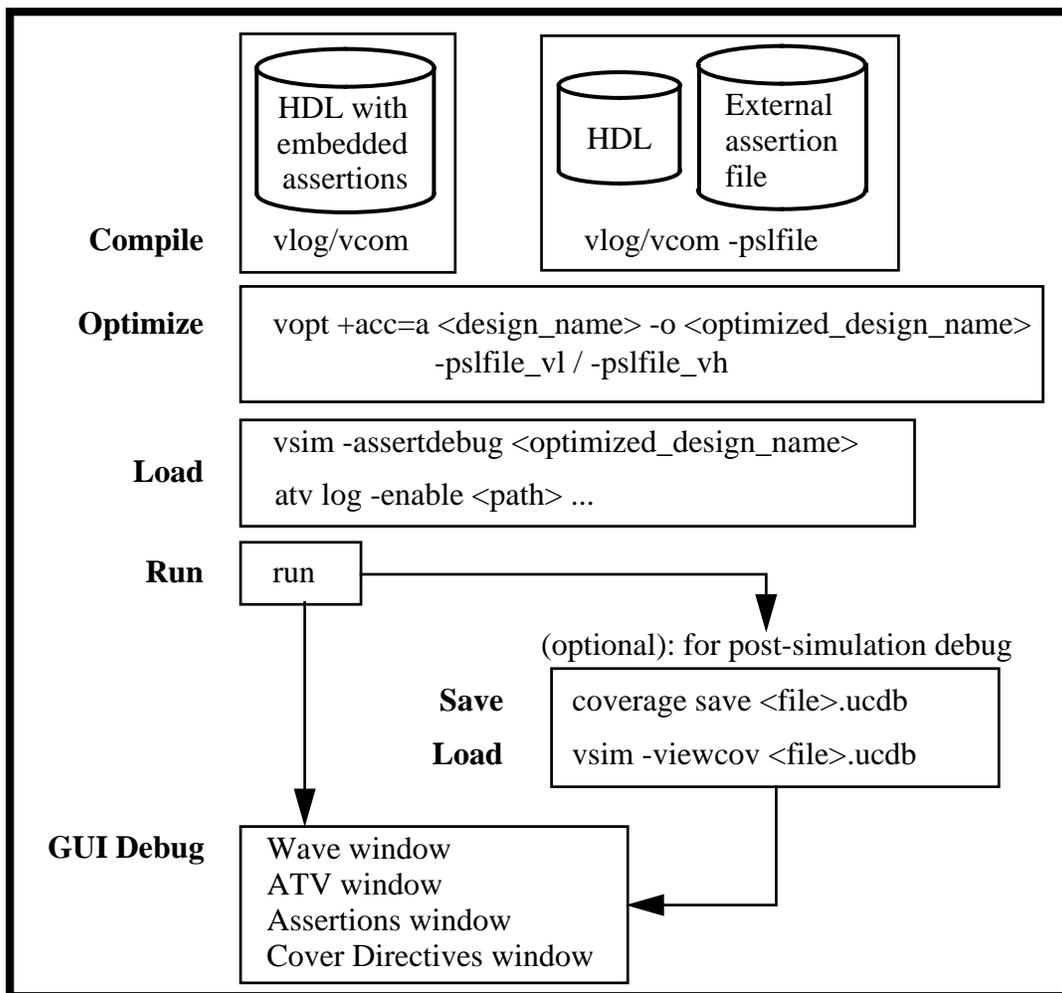
and c followed on the next cycle by b and d; this satisfies both the simultaneous {a;b} and {c;d} sequences. Logically, the evaluation should increment the count once and only once for a single directive with a given set of inputs from a given start time, but the ModelSim implementation will increment the count twice.

PSL Assertions and Cover Directives

PSL assertions can be embedded in your code or supplied in an external file, as shown in [Figure 22-26](#), while SystemVerilog assertions are always embedded in your code.

If PSL assertions are embedded, `vlog/vcom` will compile them automatically. If the assertions are in a separate file, use the `-pslfile` switch for either the `vlog` or `vcom` command.

Figure 22-26. Usage Flow for PSL Assertions



When using optimization (`vopt +acc`), you may still specify assertions at compile time, but you may also specify an external PSL file when you optimize your design with the `vopt` command.

Use the **-pslfile_vl** switch for PSL files that apply to Verilog modules and the **-pslfile_vh** switch for PSL files that apply to VHDL modules.

There are several advantages to loading PSL files along with **vopt**:

- Rather than specifying a PSL file for every invocation of the compilers, you can put all assertions in one file and specify that to **vopt**.
- Vunits can be bound to design unit instances only when provided to **vopt** using **-pslfile_vl/-pslfile_vh**.

The **vopt** command maintains assertions that were compiled with **vcom** or **vlog** whether they are embedded or external vunits.

Using PSL Directives in Procedural Blocks

PSL directives (**assert**, **cover** etc.) are supported in procedural blocks (always and initial). These directives are treated as if they were written outside and no inferences are made from the procedural blocks.

PSL Limitations in 0-In

ModelSim support for PSL is a superset of the 0-In support for PSL. Any language feature you can use in 0-In is supported in ModelSim. Consult the 0-In documentation for further details.

Common PSL Assertions Coding Tasks

Common tasks associated with coding assertions include:

- [Embedding Assertions in Your Code](#)
- [Writing PSL Assertions in an External File](#)
- [PSL Clocked and Unclocked Properties](#)
- [Using PSL ended\(\) in HDL Code](#)

Embedding Assertions in Your Code

One way of looking at assertions is as design documentation. In other words, anywhere you would normally write a comment to capture pre-conditions, constraints or other assumptions as well as to document the proper functionality of a module, process, or subprogram, use assertions to capture the information instead.

PSL Syntax

PSL assertions are embedded using metacomments prefixed with 'psl'. For example:

```
-- psl sequence s0 is {b0; b1; b2};
```

The PSL statement can be multi-line. For example:

```
-- psl sequence s0 is  
-- {b0; b1; b2};
```

Note that the second line did not require a 'psl' prefix. Once in PSL context, the parser will remain there until a PSL statement is terminated with a semicolon (';').

Restrictions for PSL Embedded Assertions

Embedded assertions have the following restriction as to where they can be embedded:

- Assertions can be embedded anywhere inside a Verilog module, interface, program, package, or compilation unit. They cannot be inside initial blocks, always blocks, tasks, functions, or clocking blocks. They also cannot be embedded in UDPs.
- Assertions can be embedded only in declarative and statement regions of a VHDL entity or architecture body and in VHDL packages. They cannot be embedded in VHDL procedures and functions.
- In a VHDL statement region, assertions can appear at places where concurrent statements may appear. If they appear in a sequential statement, ModelSim will generate an error.
- There is no support for a default clock in VHDL or Verilog packages.
- There is no support for directives in VHDL or Verilog packages.
- Endpoints must be parameterized and must have @clock specifications in VHDL or Verilog packages.

Note



The endpoint construct is not part of the IEEE Std 1850-2005. However, it was present in the original Accellera PSL standard upon which ModelSim's PSL support was based. Support of the endpoint construct will be maintained in ModelSim.

Examples

Example 11-1 shows how embedded assertions should appear in your code.

Example 22-1. Embedding Assertions in Your Code

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use WORK.constants.all;
entity dram_control is
generic ( BUG : Boolean := TRUE );
port ( clk      : IN      std_logic;
      reset_n  : IN      std_logic;
      as_n     : IN      std_logic;
      addr_in  : IN      std_logic_vector(AIN-1 downto 0);
      addr_out : OUT     std_logic_vector(AOUT-1 downto 0);
      rw       : IN      std_logic; -- 1 to read; 0 to write
      we_n     : OUT     std_logic;
      ras_n    : OUT     std_logic;
      cas_n    : OUT     std_logic;
      ack      : OUT     std_logic );
end entity dram_control;

architecture RTL of dram_control is

    type memory_state is (IDLE, MEM_ACCESS, SWITCH, RAS_CAS, OP_ACK, REF1,
REF2);
    signal mem_state : memory_state := IDLE;

    signal col_out   : std_logic; -- Output column address
                                -- = 1 for column address
                                -- = 0 for row address

    signal count     : natural range 0 to 2;          -- Cycle counter
    signal ref_count : natural range 0 to REF_CNT;    -- Refresh counter
    signal refresh   : std_logic;                    -- Refresh request

--psl default clock is rising_edge(clk);
-- Check the write cycle
-- psl property check_write is always {fell(as_n) and not rw} |=> {
--   [*0 to 5];
--   (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(7 downto 4)));
--   (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(3 downto
0)))[*2];
--   (ras_n = '0' and cas_n = '0')[*2];
--   ack};

--psl assert check_write;

begin
.

```

HDL Code Inside PSL Statements

Verilog and VHDL statements may be placed in either embedded PSL meta-comments or in external vunits. When they are embedded in your code, you must use PSL block meta-comment tags. For example, suppose you have a module called *test* with the following embedded PSL:

```
/* psl begin
```

```
    default clock = (posedge clk);
    A_E:assert always {b0} | => b1;
    always @(posedge clk)
        $display($time, " TEST : posedge clk.");
end
*/
```

In addition, you have a vunit binding to *test* as follows:

```
vunit v2(test)
{
    default clock = (posedge clk);
    A_V:assert always {b0} | => b1;
    always @(posedge clk)
        $display($time, " VUNIT : posedge clk.");
}
```

The compiler(**vlog/vcom**) -nopsl switch disables any embedded PSL parsing. This will prevent parsing of any code within the PSL metacomment including any HDL code in the metacomment. It has no effect on the vunit parsing in any way. If you provide a vunit file using the -pslfile switch then the entire vunit will be parsed and code will be generated for it.

If you simulate with the **vsim** -nopsl switch, evaluation of all PSL assume/assert/cover directives and endpoints will be disabled. It will not, however, affect any HDL code which was present in a PSL metacomment or in a vunit.

Four possible simulation scenarios (using the Verilog example files located at `<install_dir>/examples/psl/verilog/nopsl_switch`) are as follows:

1. The -nopsl is *not* used during compile or simulation.

```
vlog doctest.v -pslfile doctest.psl
vsim -c test -do "run -all"
```

This will display the TEST and VUNIT messages and evaluate assertions A_V and A_E.

2. The -nopsl switch is only used during simulation.

```
vlog doctest.v -pslfile doctest.psl
vsim -c test -do "run -all" -nopsl
```

This will display both the TEST and VUNIT messages.

3. The -nopsl switch is used only during compile.

```
vlog doctest.v -pslfile doctest.psl -nopsl
vsim -c test -do "run -all"
```

This will display only the VUNIT messages and evaluate assertion A_V.

4. The -nopsl switch is used during compile and simulation.

```
vlog doctest.v -pslfile doctest.psl -nopsl
```

```
vsim -c test -do "run -all" -nops1
```

This will display only the VUNIT messages.

Writing PSL Assertions in an External File

PSL assertions in an external file are grouped into vunits.

Syntax

```
vunit name [(<HDL\_design\_unit>)]  
{  
  default clock = <clock\_decl>;  
  <assertions>;  
  ...  
}
```

name

The name of the vunit.

[<HDL_design_unit>](#)

The module or entity/architecture to which the vunit is bound. Can also be a design unit instance if you are running the simulation without optimization (see [PSL Assertions and Cover Directives](#)). Optional.

If the design unit is unspecified the vunit does not bind to anything. Unbound vunits may be "inherited" from other vunits using the PSL keyword **inherit**. This option is available only if you are running the simulation with optimization. (See [PSL Assertions and Cover Directives](#)).

[<clock_decl>](#)

The default clock declaration for the vunit.

[<assertions>](#)

Any number of verification directives or PSL statements.

Restrictions

The following restrictions exist when providing PSL assertions in a separate file.

- Vunits can be bound only to a module, entity, or architecture unless you are running the simulation without optimization (see [PSL Assertions and Cover Directives](#)).
- The PSL file and its corresponding HDL file must be compiled together if you are running the debug flow.

Examples

Example 11-2 shows how three assertions are written in one vunit using Verilog syntax.

Example 22-2. Writing Assertions in an External File

```
vunit check_dram_controller(dram_control)  
{
```

```
default clock = rose(clk);

// declare refresh sequence
sequence refresh_sequence = {
  !cas_n && ras_n && we_n; [*1];
  (!cas_n && !ras_n && we_n)[*2];
  cas_n && ras_n};

sequence signal_refresh = {[*24]; rose(refresh)};

property refresh_rate = always {rose(reset_n) || rose(refresh)} | =>
{signal_refresh};

assert refresh_rate;

property check_refresh = always ({rose(refresh)} | ->
  {(mem_state != IDLE)[*0:14]; (mem_state == IDLE); refresh_sequence}
  abort fell(reset_n));

assert check_refresh;

// Check the write cycle
property check_write = always {fell(as_n) && !rw} | => {
  [*0:5];
  (!ras_n && cas_n && (addr_out == addr_in[7:4]));
  (!ras_n && cas_n && (addr_out == addr_in[3:0]))[*2];
  (!ras_n && !cas_n)[*2];
  ack};

assert check_write;

// check the read cycle
property check_read = always {fell(as_n) && rw} | => {
  [*0:5];
  (!ras_n && cas_n && (addr_out == addr_in[7:4]));
  (!ras_n && cas_n && (addr_out == addr_in[3:0]))[*2];
  (!ras_n && !cas_n)[*3];
  ack};

assert check_read;
}
```

Sharing a Single .psl File Between VHDL and Verilog Models

The `'ifdef VHDL_DEF` and `'ifdef VLOG_DEF` macros allow you to share a single `.psl` file between VHDL and Verilog versions of a model. Use these macros inside the `.psl` file to bracket vunits written in both VHDL and Verilog.

Internally ModelSim adds a ``define VHDL_DEF` or `VLOG_DEF` depending on how you read in the `.psl` file (with `vcom`, `vlog`, or the `-pslfile_vh/-pslfile_vl` switches for `vopt`). For example, if you read in the following file using `vlog -pslfile` or `vopt -pslfile_vl`, ModelSim will ignore the first vunit and parse the second:

```

`ifdef VHDL_DEF
  vunit v1 ( top(a) )
  {
    default clock is rose(clk);
    property vh_clk is always (a and b);
    assert vh_clk;
  }
`endif
`ifdef VLOG_DEF
  vunit v1 ( top )
  {
    default clock = rose(clk);
    property vl_clk = always (a && b);
    assert vl_clk;
  }
`endif

```

Inserting VHDL library and use Clauses in External PSL Assertion Files

You can insert VHDL library and use clauses directly in external PSL assertion files. This lets you access packages such as Signal Spy even if the design unit (to which the vunit is attached) doesn't reference the package.

Here is an example that shows the use of Signal Spy:

```

library modelsim_lib;
use modelsim_lib.util.all;

vunit top_vunit(test) {
  signal vunit_local_sigA : bit := '0';
  signal vunit_loc_sigB : bit := '0';

  initial_proc: process
  begin
    --spy on a signal in a package
    init_signal_driver("/pack/global_signal", "vunit_local_sigA");
    --spy on a internal signal
    init_signal_driver("/test/aa/internal_signal_AA",
"vunit_loc_sigB");
    wait;
  end process initial_proc;

  assert (vunit_local_sigA -> vunit_loc_sigB);
}

```

Here are two points to keep in mind about library and use clauses in PSL files:

- If you already have the use clause applied to an entity, then you don't need to specify it for the vunit. The vunit gets the entity's complete visibility.
- If you have two vunits in a file and the use clause at the top, the use clause will apply only to the top vunit. If you want the use clause to apply to both vunits, you have to specify it twice. This follows the rules for use clauses as they apply to VHDL entities.

PSL Clocked and Unclocked Properties

PSL assertions in ModelSim may be clocked or unclocked.

PSL Unclocked Properties

ModelSim supports unclocked properties of the following form:

- `assert always <expr>`
- `assert never <expr>`
- `cover <expr>`

If these directives appear before any default clocking statements, and they are not individually clocked, then they are treated as unclocked. The `<expr>` is a simple boolean expression.

Default Clock

Any PSL assertion that is not individually clocked and appears below a default clock statement will be clocked by the default clock. For example:

```
default clock is rose(clk);
assert always sigb@rose(clk1)
assert always siga;
```

The first assertion is sensitive to `clk1`. The second assertion is sensitive to `clk` (the default clock).

PSL Partially Clocked Properties

The default clock also applies to partially clocked properties. For example:

```
default clock is rose(clk);
assert always (b0 |-> (b1@rose(clk1)))
```

In this case, only the RHS of the implication(|->) expression is clocked. The outermost property is unclocked, so default clock applies to this assertion.

Also, the complete assertion property, because it is not a simple expression, must be clocked. For example, if you have the following assertion:

```
assert always (b0 |-> (b1@rose(clk1)))
```

and no default clock preceding it, then since part of the property is unclocked, ModelSim will produce an error.

PSL Multi-Clocked Properties and the Default Clock

You need to be very careful when writing multi-clocked properties that also have a default clock, or you may produce unexpected results. For example, say you want to write a property

that means the following: if signal *a* is true at *rose(clk1)*, then at the next rising edge of *clk2*, signal *b* should be true. You would write the property like this:

```
assert always (a -> b@rose(clk2)) @rose(clk1);
```

In this property, the @ operator has more precedence than the always operator, so the property is interpreted like this:

```
assert always ((a -> b@rose(clk2)) @rose(clk1));
```

Note that the *always* operator is unlocked but the property under *always* is clocked. This is acceptable because ModelSim detects that the property is to be checked at every *rose(clk1)*.

Other Restrictions

The following are additional restrictions on clock declarations:

- There is no support for a default clock in VHDL packages.

Using PSL ended() in HDL Code

The PSL ended() construct is a built-in function whose value is set to TRUE for the simulation time unit whenever its sequence is matched. HDL code can read the value of this builtin.

Note



The endpoint construct is not part of the IEEE Std1850-2005. However, it was present in the original Accellera PSL standard upon which ModelSim's PSL support was based. Support of the endpoint construct will be maintained in ModelSim.

[Example 22-3](#) and [Example 22-4](#) are two complete examples that demonstrate the use of ended() in Verilog and VHDL code, respectively.

Example 22-3. Using PSL ended() in Verilog

```
module test;

reg clk;
initial clk = 0;
always #50 clk <= ~clk;

reg b1, b2;

// psl sequence s0(boolean b_f) = {b1[*2]; [*0:2]; b_f};

initial
begin
    b1 <= 0;    b2 <= 0;
    #100;      b1 <= 0;    b2 <= 0; //100
    #100;      b1 <= 0;    b2 <= 0; //200
    #100;      b1 <= 0;    b2 <= 0; //300
    #100;      b1 <= 1;    b2 <= 0; //400
end
```

```
#100;      b1 <= 1;   b2 <= 1; //500
#100;      b1 <= 1;   b2 <= 1; //600
#100;      b1 <= 0;   b2 <= 0; //700
#100;      b1 <= 0;   b2 <= 1; //800
#100;      b1 <= 0;   b2 <= 0; //900
#100;      b1 <= 0;   b2 <= 0; //1000
#100;
$finish;
end

always @(posedge clk)
begin
    if (ended(s0(b2)) == 1)
        $display($time, " Ended is true.");
end

always @(ended(s0(b2)), posedge clk)
    $display($time, " Ended triggered.");

endmodule
```

Example 22-4. Using ended() in VHDL

```
use STD.textio.all;

entity test is
end test;

architecture a of test is
    signal clk_0 : bit := '0';
    signal clk_1 : bit := '0';
    signal b1    : bit := '0';
    signal b2    : bit := '0';
begin

    clk_0 <= not clk_0 after 50 ns;
    clk_1 <= not clk_1 after 75 ns;

    -- psl begin
    --     sequence s0(Boolean b_f) is {b1[*2]; [*0 to 2]; b_f};
    --     sequence se0(Boolean clk_f) is {s0(b2)}@rose(clk_f);
    -- end

    endp_0 : process(clk_0)
        variable test_val_0 : BOOLEAN;
        variable vline : line;
    begin
        test_val_0 := ended(se0(clk_0));
        write(vline, now);
        write(vline, string'(": test_val_0 = "));
        write(vline, test_val_0);
        writeline(OUTPUT, vline);
    end process;

    endp_1 : process(clk_1)
        variable test_val_1 : bit;
        variable vline : line;
```

```

begin
  if (ended(se0(clk_1)) = true) then
    test_val_1 := '1';
  else
    test_val_1 := '0';
  end if;
  write(vline, now);
  write(vline, string(": test_val_1 = "));
  write(vline, test_val_1);
  writeline(OUTPUT, vline);
end process;

process
begin
  wait for 400 ns;  b1 <= '1';    b2 <= '0';  --400
  wait for 100 ns; b1 <= '1';    b2 <= '1';  --500
  wait for 200 ns; b1 <= '0';    b2 <= '0';  --700
  wait for 100 ns; b1 <= '0';    b2 <= '1';  --800
  wait for 100 ns; b1 <= '0';    b2 <= '0';  --900
  wait for 300 ns; b1 <= '1';    b2 <= '1';  --1210
  wait for 100 ns; b1 <= '1';    b2 <= '0';  --1300
  wait for 300 ns; b1 <= '0';    b2 <= '1';  --1600
  wait for 300 ns; b1 <= '1';    b2 <= '0';  --1900
  wait for 200 ns; b1 <= '1';    b2 <= '1';  --2100
  wait for 100 ns; b1 <= '0';    b2 <= '1';  --2200
  wait for 100 ns; b1 <= '0';    b2 <= '0';  --2300
  wait;
  -- infinite wait
end process;

end a;

```

Note



In VHDL, unused endpoints (defined but not used in the design) are optimized away during compilation.

Compiling and Simulating PSL Assertions

Common tasks for compiling and simulating assertions in your design include:

- [Compiling Embedded PSL Assertions](#)
- [Compiling an External PSL Assertions File](#)
- [Applying PSL Assertions During Elaboration/Optimization](#)
- [Making Changes to PSL Assertions](#)

Compiling Embedded PSL Assertions

Embedded PSL assertions are compiled automatically by default. If you have embedded PSL assertions that you don't want to compile, use the **-nopsl** switch with the **vlog** or **vcom** commands.

Compiling an External PSL Assertions File

To compile assertions in an external file, invoke the compiler with the **-pslfile** switch and specify the assertions file name. For example, in the following command:

```
vlog tadder.v adder.v -pslfile adder.psl
```

the *adder.psl* file is compiled by means of the **-pslfile** switch.

The design and its associated assertions file must be compiled in the same invocation.

Applying PSL Assertions During Elaboration/Optimization

You can also specify an external PSL vunit file with the **-pslfile_vh** or **-pslfile_vl** switches for the **vopt** command. For example:

```
vopt testbench -o mydesign -pslfile_vl tb.psl
```

See [PSL Assertions and Cover Directives](#) for more information.

Making Changes to PSL Assertions

If you make any changes to embedded assertions, you need to re-compile the design unit. If you make changes to an external PSL assertions file, you need to compile both the external PSL file and the design unit file to which the vunit binds in the same **vlog/vcom** invocation.

PSL Limitations

ModelSim supports the simple subset of PSL constructs and semantics as described in the IEEE Std 1850-2005, except the following:

- `next()`, `nondet`, and `nondet_vector()` built-in functions
- Union expressions
- OBE properties
- `assume_guarantee`, `restrict`, `restrict_guarantee`, `fairness` and `strong_fairness` directives
- Integer Range and Structure

The current release also has the following limitations.

- Vunits can be bound to design unit instances only when provided to **vopt** using `-pslfile_vl/-pslfile_vh`.
- Level-sensitive clock expressions are not allowed.
- There is no support for integer, structures, and union in the modeling layer.

- There is no support for post-simulation run of assertions (i.e., users cannot run the assertion engine in post-simulation mode).

Using SVA Assertions and Cover Directives

Assertions and Action Blocks in SVA

Action block can contain immediate assertion statements.

Immediate assertions are used to enforce a property as a checker. When the property for the assert directive is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the action block are executed. For example,

```
property abc(a,b,c);
  disable iff (a==2) @ (posedge clk) not (b ##1 c);
endproperty
env_prop: assert property (abc(rst,in1,in2)) pass_statement;
else fail_statement;
```

When no action is needed, a null statement is specified. If no statement is specified for else, then \$error is used as the statement when the assertion fails.

For SystemVerilog and VHDL immediate assertions, passes and failures cannot be enabled or disabled independently. So if [AssertionEnable](#) or [assertion enable](#) are used, both passes and failures are enabled for immediate assertions.

Deferred Assertions and Cover Directives

ModelSim supports deferred assertions, a special kind of immediate assertions, as well as deferred immediate assume and cover directives. Deferred assume directives are simulated like assert directives unless the -noassume switch is specified with the [vsim](#) command. (See [Processing Assume Directives](#).) Deferred assertions and cover directives are displayed in the Assertions window and can be added to the Wave window just like simple immediate assertions. They are stored in the UCDB with other assertions and cover directives and can be reported with the [coverage report](#) command.

SystemVerilog Cover Directives

SystemVerilog assertion coverage is performed with either or both of the following two **cover** directives.

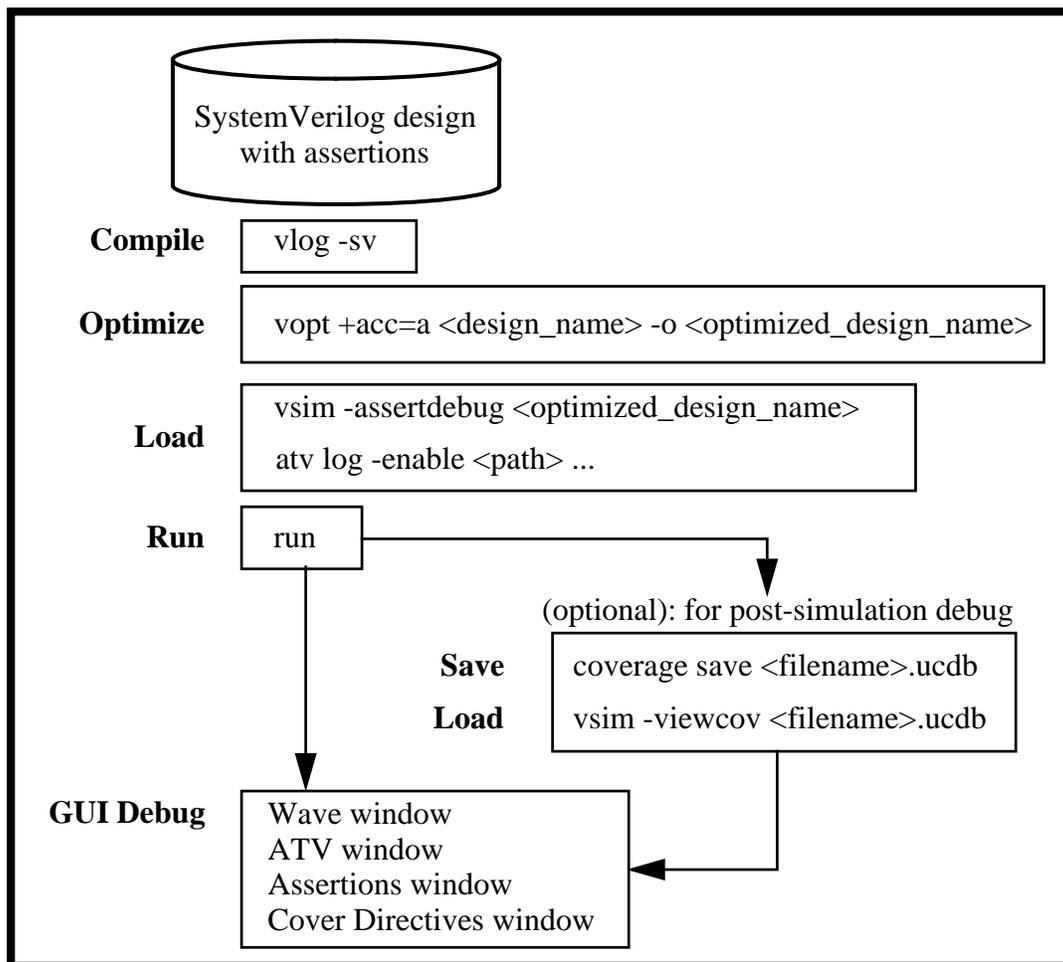
```
cover property ( property_spec ) statement_or_null
cover sequence (
  [clocking_event]
  [disable iff (expression_or_dist)]
  sequence expr) statement_or_null
```

The "statement_or_null" syntax indicates an optional statement to be executed every time the property succeeds or the sequence is matched.

SVA Usage Flow for Assertions and Cover Directives

ModelSim compiles SystemVerilog assertions along with other SystemVerilog code when either the source file ends in `.sv` or when you specify the `-sv` switch with the `vlog` command. Figure 22-27 shows the use of SystemVerilog assertions in the debug flow.

Figure 22-27. Usage Flow for SystemVerilog Assertions



The `vopt` command performs global optimizations on designs after they have been compiled with `vcom` or `vlog` and produces an optimized version of your design in the working directory. You must provide a name for this optimized version using the `-o` switch. You can then invoke `vsim` directly on that new design unit name.

In the course of optimizing a design, the **vopt** utility will remove objects deemed unnecessary for simulation — line numbers are removed, processes are merged, nets and registers may be removed, etc. For debugging, you preserve object visibility into your assertions by using the **+acc=a** argument with the **vopt** command. The **+acc=a** argument specifies which objects are to remain accessible for the simulation. In this case the “a” stands for assertions. (See [Preserving Object Visibility for Debugging Purposes](#).)

When you invoke **vsim** on the optimized version of the design, the simulator automatically loads any assertions that are present. The **-assertdebug** switch makes detailed assertion and cover directive information available for viewing in the debugging windows of the GUI (see the next section, [Viewing Debugging Information](#)).

If the **+acc=a** argument is used with the **vopt** command and the **-assertdebug** switch is set with the **vsim** command, the **coverage save** command will save the detailed assertion and cover directives information in the **.ucdb** file (see [Saving Assertion and Cover Directive Metrics](#)). This information can be called up and viewed in the debugging windows with the **vsim -viewcov <filename>.ucdb** command.

Using -assertdebug to Debug with Assertions and Cover Directives

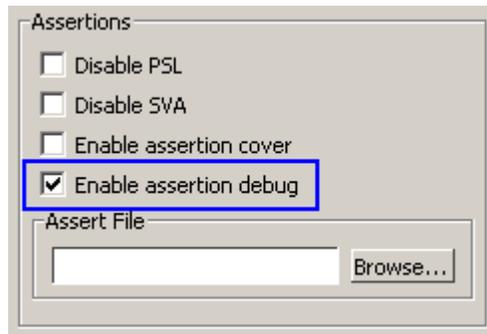
Viewing Debugging Information

Using the optional **-assertdebug** switch with the **vsim** command instructs ModelSim to examine all assertions and display assertion details in the Assertions window. The Assertions window displays information about assertion failures, passes, starts, and antecedents. See [Analyzing Assertions and Cover Directives](#) for additional details.

To enable assertion debugging with the GUI:

1. Select **Simulate > Start Simulation** to open the Start Simulation dialog.
2. Open the **Others** tab.
3. In the Assertions section, select **Enable assertion debug** ([Figure 22-28](#)).

Figure 22-28. Enable Assertion Debug



With assertion debugging enabled, assertion fail messages will be displayed in the transcript and will include the expression that caused the assertion failure. Assertion failures are also listed in the Message Viewer window (**View > Message Viewer**) under “Error.”

Local variable values corresponding to failed assertion threads are printed to the Transcript along with assertion error messages when **vsim** is run with **-assertdebug**. This can be turned on/off (default on) with `AssertionFailLocalVarLog modelsim.ini` variable or by using the CLI `assertion fail -lvlog` command. For example:

```
# ** Error: Assertion error.  
#   Time: 450 ns Started: 250 ns Scope: test File: src/lvlog1.sv Line: 19  
Expr: x==1  
#   Local vars : x = 0, s11 = '{x:{10, 0, 0, 0, 0, 0, 0, 0, 0, 0}, y:1}
```

The `atv log` command enables Assertion Thread Viewing (ATV) in ATV windows. To turn on ATV recording, an assertion path (or assertion paths) must be specified with `atv log -enable` prior to running the simulation. This allows thread data to be collected over the course of the entire simulation. Optionally, ATV recording can be enabled during a simulation when the simulator has stopped, but only threads that start after the current simulation time will be visible.

Note



The **-assertdebug** switch must be used with **vsim** prior to the `atv log` command.

There is a performance penalty when assertion debugging and assertion thread viewing are enabled. You should use these features only when you need to debug failures.

Enabling ATV Recording

For analyzing assertion problems, the Assertion Thread Viewer can be configured to record detailed data of assertion states during simulation. This allows you to analyze temporal expressions to determine when and where they are not holding.

Note



This is a memory and cpu-intensive process, so it is best to enable it selectively on an instance by instance basis.

If you want access to the Assertion Thread Viewer (ATV), you must activate ATV recording before the simulation is run with these two steps:

1. The `-assertdebug` switch must be enabled with the `vsim` command, or “Enable assertion debug” must be selected in the Others tab of the Start Simulation dialog.
2. ATV recording must be enabled with the `atv log` command.

An assertion path is specified with `atv log -enable <path>...` prior to running the simulation so thread data can be collected over the course of the entire simulation. More than one assertion path may be included in each `atv log` command.

For example, a correct procedure would be the following:

```
vopt +acc=a top -o dbgver
vsim -assertdebug dbgver
atv log -enable /top/assert_0 /top/assert_1 /top/assert_2
```

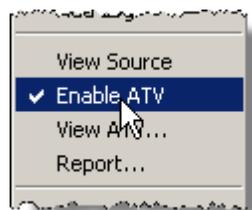
In this example, `top` is the top-level module in the design.

- The `+acc=a` argument is used with the `vopt` command to preserve assertion visibility for debugging; and the `-o` switch is used to name the optimized version of the design (`dbgver`).
- The `-assertdebug` switch is used with `vsim` to enable assertion debugging on the optimized version of the design.
- The `atv log` command line enables ATV recording for three assertions - `assert_0`, `assert_1`, and `assert_2` - in the `top` module.

You can also enable ATV recording with the GUI by doing one of the following when the Assertions window is active and one or more assertions is highlighted:

- Select **Assertions > Enable ATV** from the menus.
- Right-click (RMB) any highlighted assertion or assertions and select **Enable ATV** from the popup menu (Figure 22-29).

Figure 22-29. Enable ATV Menu Selection



When the GUI is used to enable ATV recording, the appropriate command will be echoed to the transcript (for enabling ATV in *.do* files).

Enabling and Disabling ATV Recording During Simulation

Optionally, ATV recording may be enabled or disabled during a simulation when the simulator has stopped. To enable, use the **atv log -enable** command as follows:

```
atv log -enable <path>...\
```

To disable, use **atv log -disable**:

```
atv log -disable <path>...
```

When you enable ATV recording during simulation, only threads that start after the current time will be visible.

You can disable ATV Recording with the GUI by unchecking the Enable ATV selection with **Assertions > Enable ATV**, or right-clicking an assertion and unchecking Enable ATV (as in [Figure 22-29](#)).

Example 22-5. Enable and Disable Assertion During Simulation

For example, you may want to only monitor a certain assertion thread, let's say */top/assert_1* between time T and time T+N. The code would be something like this:

```
run Tns
atv log -enable /top/assert_1
run Nns
atv log -disable /top/assert_1
run 1000ns
```

Saving Assertion and Cover Directive Metrics

You can save assertion and cover directive metrics to the Unified Coverage Database (UCDB).

Note



For easiest viewing and tracking of assertions and cover directives in the GUI and coverage reports, it is recommended that you name all assertions and directives in the source files. See [Assertion/Cover Directive Naming Conventions](#) for more information.

To make all assertion and cover directive metrics available for saving into a *.ucdb* file you must do all of the following steps:

- compile your design,
- use **+acc=a** with the **vopt** command,
- use **-assertdebug** with the **vsim** command

- use the `coverage save` command

If the **coverage save** command is used without switches and arguments, all assertion and cover directive metrics are saved to the UCDB. If other coverage types (branch, statement, condition, etc.) are specified with the **coverage save** command, then the **-assert** switch must be used to save assertions, like this:

```
coverage save -code bcest -assert
```

or the **-directive** switch must be used to save cover directives. like this:

```
coverage save -code bcest -directive
```

You can specify that only assertions and cover directives be saved with:

```
coverage save -assert -directive
```

If the **-assertdebug** switch is not used with **vsim**, the **coverage save** command will only save the fail metrics for assertions and cover directives. When **-assertdebug** is used, additional metrics saved for assertion directives include:

```
PassEnable  
PassLog  
PassCount  
VacuousPassCount  
DisabledCount  
AttemptedCount  
ActiveThreadCount  
PeakActiveThreadCount
```

The **-assertdebug** switch determines how assertion coverage numbers are calculated in the UCDB. When the **-assertdebug** switch is used, an assertion is considered covered if the `PassCount` is > 0 . If the **-assertdebug** switch is not used, an assertion is considered covered if the `FailCount` = 0.

You can also use the GUI to save assertion and cover directive metrics.

- Select **Tools > Coverage Save** from the Main window

This opens the Coverage Save dialog, where you can select the type of coverage you want to save, level of hierarchy to save, and output UCDB file name.

- Select **Simulate > Start Simulation** from the Main window

This opens the Start Simulation dialog. In the Others tab, check the “Enable code coverage” box and the “Enable assertion debug” box.

Once the data is saved, assertion and cover directive coverage can be analyzed using:

- the Assertions window (see [Viewing Assertions in the Assertions Window](#)),
- the Cover Directives window (see [Viewing Cover Directives in the Cover Directives Window](#)),
- the Wave window (see [Viewing Assertions and Cover Directives in the Wave Window](#)),
- columns in the Structure Window (Cover and Assertion hits and misses) (see [GUI Elements of the Structure Window](#)),
- the Instance Coverage window (see [GUI Elements of the Instance Coverage Window](#)),
- columns in the Verification Management Browser window (see [Coverage and Verification Management in the UCDB](#)),
- and the [coverage report](#) and [vcover report](#) commands.

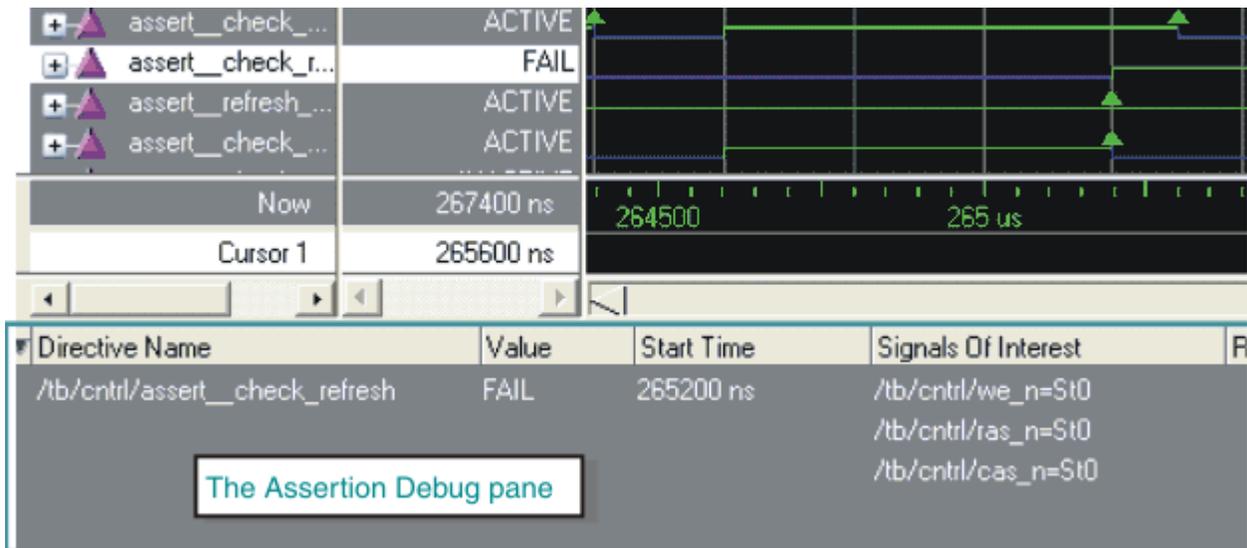
These methods are all available in live simulation mode as well as the Coverage View (post-processing) mode.

Analyzing Assertion Failures in the Assertion Debug Pane of the Wave Window

If you used the **-assertdebug** switch with the [vsim](#) command when you invoked the simulator, you can view the details of assertion failures in the Assertion Debug pane of the Wave window. The steps to view assertion failures are as follows:

1. To open the Assertion Debug pane in an undocked Wave window, select **View > Assertion Debug**. To view the debug pane when the Wave window is docked in the Main window, make the Wave window active then select **Wave > Assertion Debug**.
2. Click a red triangle on an assert directive waveform (the red triangle indicates a failed assert directive) to display debug information about the failed assertion.

Figure 22-30. Assertion Debug Pane in Wave Window



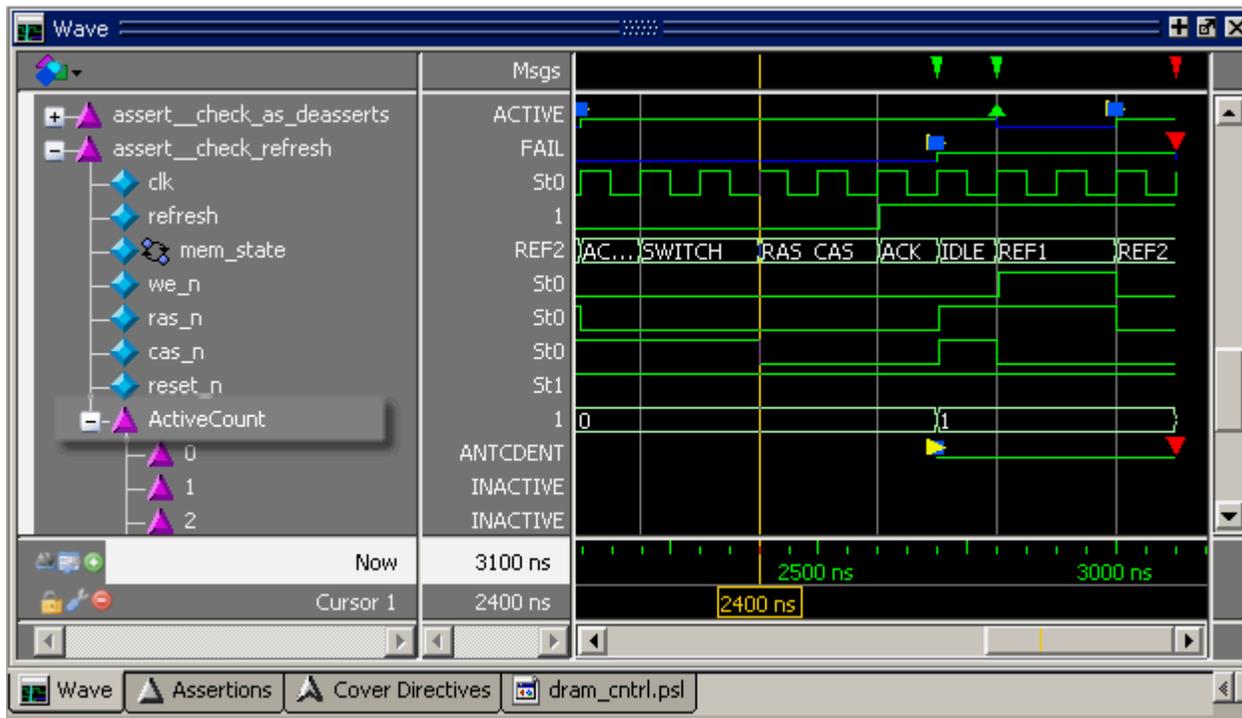
The Signals of Interest column displays the signals responsible for the assertion failure. You can analyze these signals further in the Dataflow window by right-clicking a signal and selecting Show Signal Drivers.

ModelSim supports the PSL **forall** keyword, which replicates designated assertions multiple times and reports PASS or FAIL on assert directives that contain replicators. The Replicator Parameters column displays the value of the replicator parameter for which the assertion failed.

Using the Assertion Active Thread Monitor

If you used the **-assertdebug** switch with the **vsim** command when you invoked the simulator, the Assertion Active Thread Monitor will display an ActiveCount object in the Wave window for each assertion and cover directive. This object tracks the number of currently active assertion or cover directive threads for the given instance. Expanding it will show individual threads (based on start time) starting and stopping (Figure 22-31).

Figure 22-31. The ActiveCount Object in the Wave Window - SystemVerilog



The Assertion Active Thread Monitor is controlled by the [BreakOnAssertion](#) .ini variable, whose default value is 1 (enabled). The number of rows the Assertion Active Thread Monitor displays is limited, and is controlled by the [AssertionActiveThreadMonitorLimit](#) .ini variable.

Note

A large number of active thread rows will result in large .wlf files and increased memory usage. The default for [AssertionActiveThreadMonitorLimit](#) is 5.

As with the main assertion Wave display, assertion start (blue square), pass (green triangle), fail (red triangle), and antecedent match (yellow triangle) symbols appear in the active thread monitor display. Right-clicking on one of these symbols reveals a **View ATV** menu selection which will show assertion evaluation attempt start times. Selecting a start time will open an assertion thread view. See [Viewing Assertion Threads in the ATV Window](#).

Viewing Assertion Threads in the ATV Window

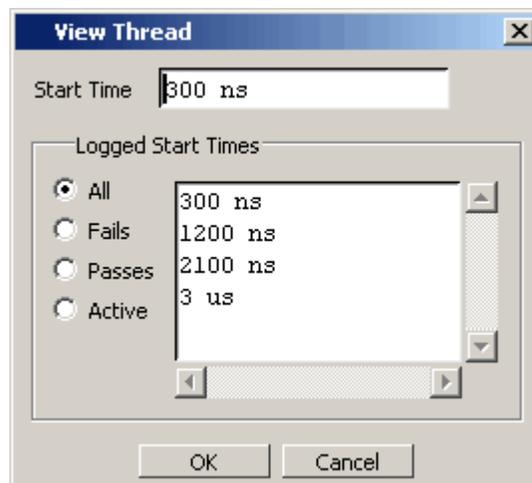
Assertion thread viewing is enabled for all assertions specified with the `atv log` command, before the simulation is run. After running a simulation, an ATV window can be opened a number of different ways.

- From the command line — The `add atv` command opens an ATV window for the specified assert or cover directive (designated by its pathname), at the specified evaluation attempt start time. For example:

add atv /top/assert_1 450 ns

- From the Assertion Active Thread Monitor in the Wave window — Right-click any assertion start (blue square), pass (green triangle), and fail (red triangle) symbol to open a popup menu. Select **View ATV**, then an assertion evaluation attempt start time. See [Using the Assertion Active Thread Monitor](#).
- From the menu bar — Select (highlight) an assertion in the Assertions window then select **Assertions > Add ATV** from the menus. This will bring up the View Thread dialog, which allows you to select an assertion evaluation attempt start time. Note that a given assertion instance typically has many evaluation attempt start times. Any one of these times may be selected from the dialog ([Figure 22-32](#)). Selecting (or typing) an entry will bring up the ATV view for that particular instance and starting time.

Figure 22-32. Selecting Assertion Thread Start Time



Note



If the Start Time and Logged Start Times fields are empty, either ATV recording has not been enabled or no evaluation attempts have occurred.

- From the Assertions window — Right-click (RMB) any assertion and select **View ATV** from the popup menu. Making this selection brings up the View Thread dialog ([Figure 22-32](#)), where you can select an evaluation attempt start time.

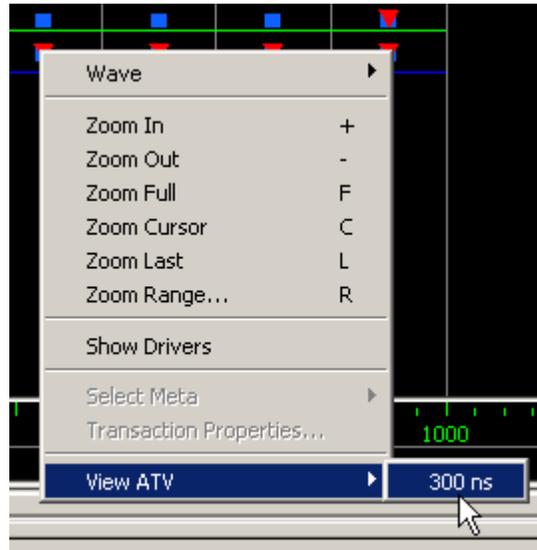
The View Thread dialogue allows you to filter the displayed list of logged thread starting times by failed, passed, and still-active attempts.

- From the Wave window — If the assertion is logged, there will be symbols on the assertion signal consisting of start objects (blue squares), pass objects (green triangles) and fail objects (red triangles). Right-clicking near one of these objects will open a pop-up menu with a **View ATV** selection and a sub-menu of evaluation attempt start times ([Figure 22-33](#)). Select a start time to bring up the ATV view for the selected assertion or cover directive.

Note

 If the sub-menu of evaluation attempt start times is empty, ATV recording has not been enabled.

Figure 22-33. Opening ATV from the Wave Window

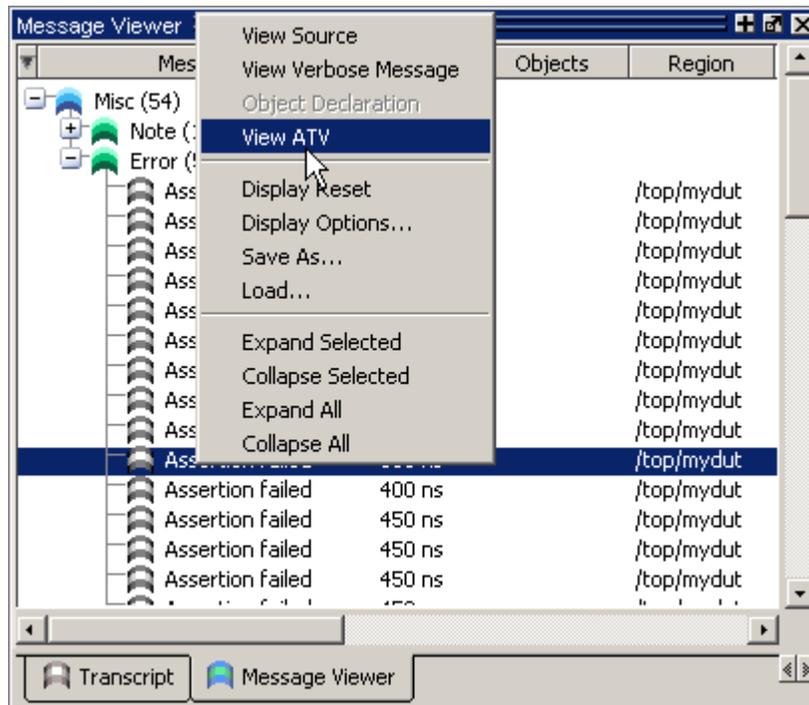


- From the Message Viewer window — Right-click an assertion error message to open a pop-up menu with a **View ATV** selection. Selecting this will bring up an ATV window for the evaluation attempt start time associated with that particular assertion error ([Figure 22-34](#)).

Note

 If the sub-menu of evaluation attempt start times is empty, ATV recording has not been enabled.

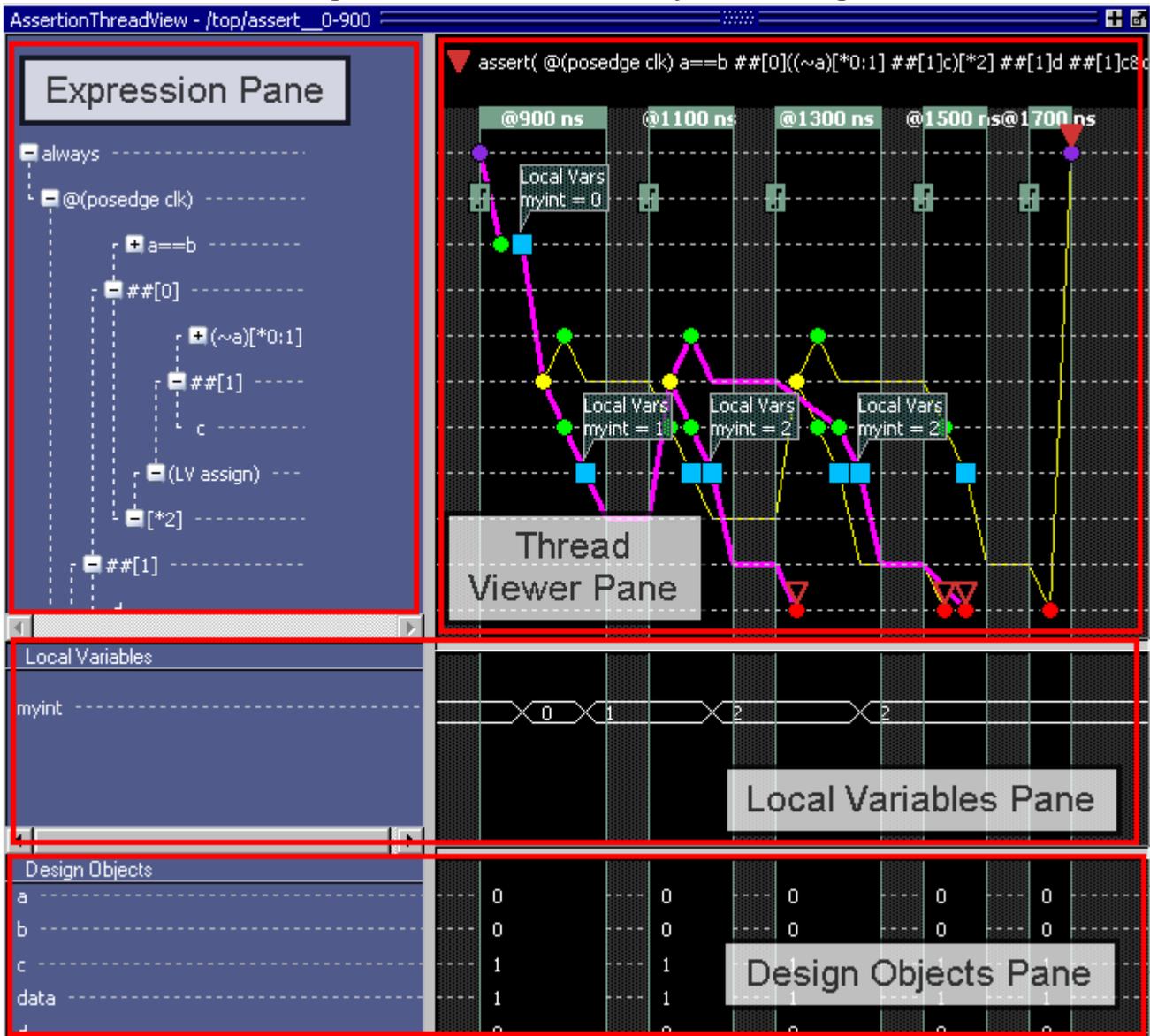
Figure 22-34. Opening ATV from the Message Viewer Window



Navigating Inside an ATV Window

The ATV window normally consists of four panes: the [Expression Pane](#), the [Thread Viewer Pane](#), the [Local Variables Pane](#), and the [Design Objects Pane](#) (Figure 22-35).

Figure 22-35. ATV Panes - SystemVerilog

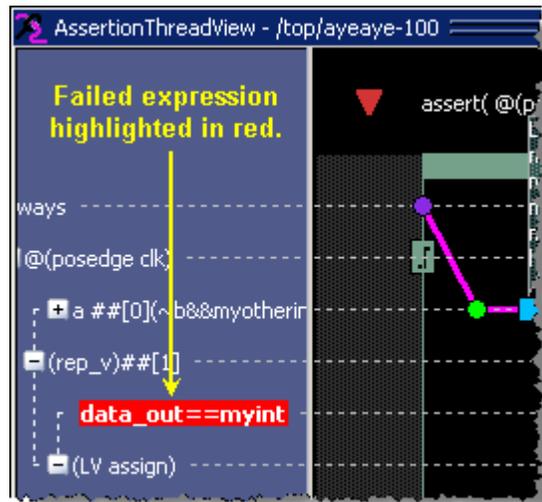


Expression Pane

The Expression pane is a hierarchical representation of the assertion. From bottom to top, each term in the assertion statement is represented following the left to right order of the original expression. From left to right, the hierarchy of expression statements is represented with the highest order terms on the left and the child terms on the right. Each term that has children can be expanded or collapsed in the viewer by clicking on the '+' and '-' symbols. To expand or collapse all terms, right-click anywhere in the Expression Pane and select **Expand All Terms** or **Collapse All Terms**.

Failed boolean expressions are highlighted in red (Figure 22-36).

Figure 22-36. Failed Expressions Highlighted Red



By default, assertion expressions are displayed in the Expressions Pane in descending order. You can change to ascending order by right-clicking in the Expressions Pane and selecting Ascending Order from the popup menu.

Thread Viewer Pane

The thread viewer pane is on the right, and shows the progress of the assertion threads over time for a given thread instance and starting time. Sub-threads that fork off of the main thread are shown, as well as boolean checks, waits, and thread terminations.

Time is displayed along the X axis. The black columns show an instant in time — i.e., @100ns, @200ns, @300ns, etc. — during which some part of the assertion expression is being evaluated. Every clock initiates an attempt to evaluate an assertion. From the simulation's point of view, simulation time is not advancing in the black areas during the evaluation; simulation time only advances in the dark grey areas.

Assertion statement progress is shown on the Y axis. The Y axis coordinates map directly to the expression terms shown in the thread expression pane. So, as a thread is seen jumping to a particular Y level, this shows that a particular term is being evaluated.

Understanding Time in the Thread Viewer Pane

It is vitally important to understand that time in the ATV window is not represented in a linear fashion like time in a Wave window. In the ATV window within a given time column (black), all the assertion activity seen has happened at exactly the same instant in the simulator. However, it is displayed in a manner that spaces out events so that individual thread progress can be seen and understood. Consequently, no correlation between events on differing threads can be implied unless two threads fork or join.

Local Variables Pane

To display the Local Variables Pane in the ATV window do one of the following:

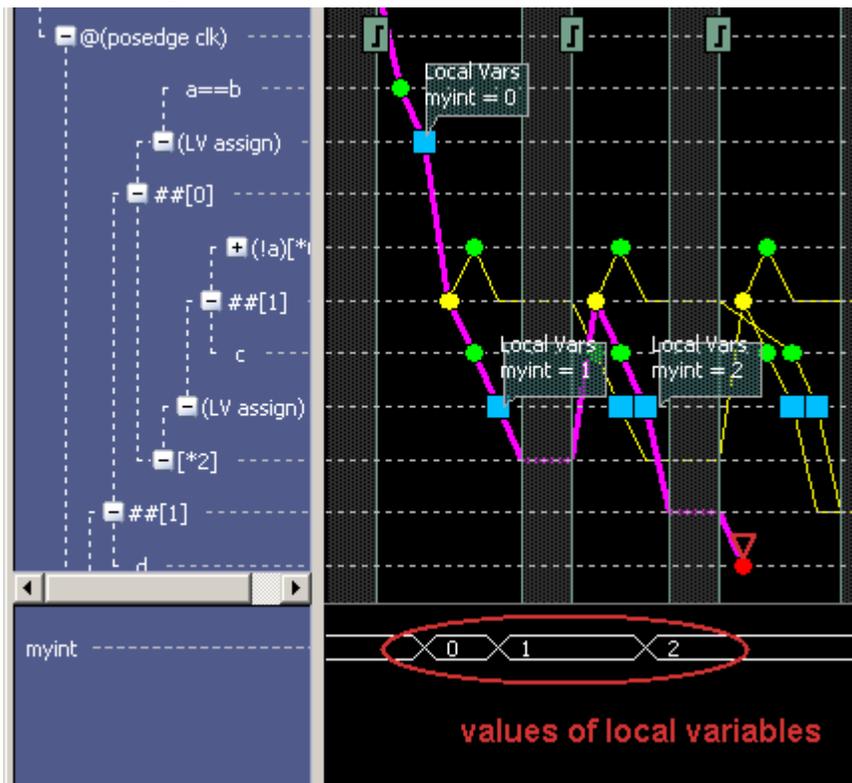
- Click the Show Local Var Pane icon in [ATV Toolbar](#). 
- When the ATV window is docked in the Main window, select **ATV > Show Local Vars**.
- When the ATV window is undocked, select **View > Show Local Vars**.

A solid blue square  indicates a local variable.

Local variable assignments are stripped out of the assertion statement at the top of the Thread Viewer Pane for readability, but they do appear in the Expression Pane as “(LV assign).” You can hover the cursor over any local variable icon (blue square) in the Thread Viewer pane to see the actual assignment; or turn on local variables annotation (see [Annotating Local Variables](#)).

When a thread is highlighted (see [Highlighting a Thread](#)) and it contains local variables, the values for the local variables on that highlighted thread will appear in the Local Variables Pane ([Figure 22-37](#)).

Figure 22-37. Values of Local Variables



Like other windows in the GUI, the display radix for the values in the Local Variables Pane are controlled by the [radix](#) command.

Design Objects Pane

To display the Design Objects pane in the ATV window do one of the following:

- Click the Show Design Objects icon in [ATV Toolbar](#).
- When the ATV window is docked in the Main window, select **ATV > Show Design Objects**.
- When the ATV window is undocked, select **View > Show Design Objects**.

The Design Objects pane contains information similar to the Wave window when used to display transactions. The left half contains the names of the design objects from the expression. The right half contains areas for each column aligned to the corresponding column in the Thread Viewer pane. Each area contains a list of textual representations of the values of the design objects in the expression at that time.

The values shown in the Design Objects pane are the values used when the assertion expression is evaluated. For PSL assertions, the values are at the time that the clock transitions. For SystemVerilog assertions, the values at the beginning of the simulation time step before any signals have transitioned. In either case, these values may be different than the values for these objects at the end of that simulation time step.

Actions in the ATV Window

Actions you can do to change what is seen in the ATV window include:

- [Find Sub-Expression That Caused Failure](#)
- [Viewing Multiple Clock Expressions](#)
- [Expanding and Contracting Expression Hierarchy](#)
- [Highlighting a Thread](#)
- [Hovering the Mouse for Thread and Directive Status](#)

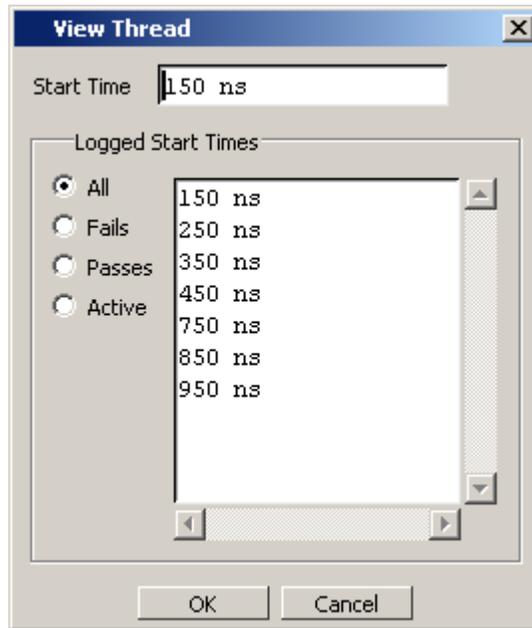
Find Sub-Expression That Caused Failure

The ATV window allows you to find which sub-expression caused an HDL expression to fail. For example, consider the assertion:

```
assert property (@(posedge clk) (b0 | => ((b1 && b2) || b3)));
```

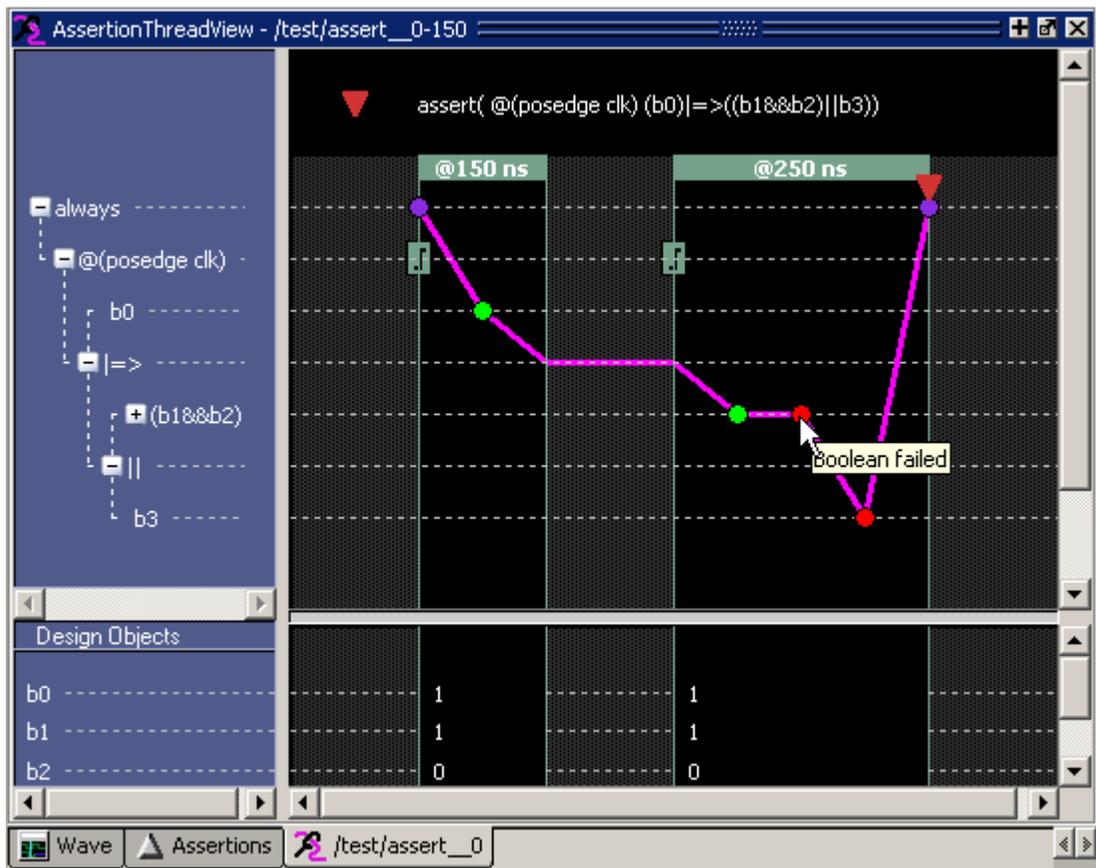
Even though `((b1 && b2) || b3)` is a boolean expression, the ATV window will show which sub-expression – `b1/b2/b3` – failed. In the Assertions window we right-click the assertion and select View ATV from the popup menu. This opens the View Thread dialog, where we'll elect to Start at 150 ns ([Figure 22-38](#)).

Figure 22-38. View Thread at 150 ns



This opens the ATV window for the assertion, as shown in Figure 22-39.

Figure 22-39. ATV Window Shows Where Boolean Sub-Expression Failed

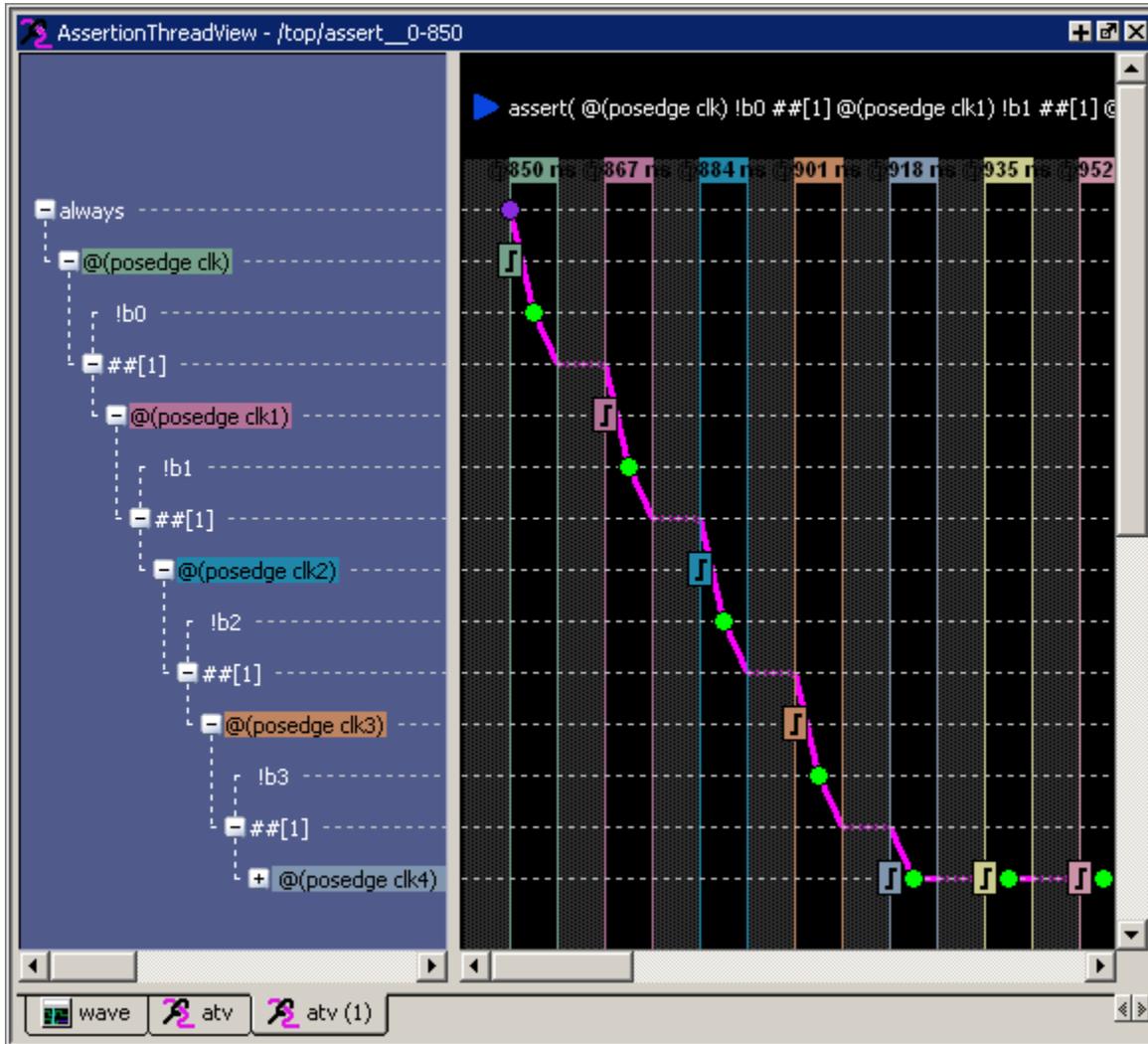


The red dots show where the boolean sub-expression failed.

Viewing Multiple Clock Expressions

The Expressions Pane displays all clock expressions in the assertion expression hierarchy. Each clock is given a different color in the Thread Viewer Pane (Figure 22-40). All clock symbols in the Thread Viewer Pane align with the appropriate clock expression in the Expression Pane.

Figure 22-40. ATV Window Displays All Clock Expressions



Expanding and Contracting Expression Hierarchy

You can expand and contract the expression hierarchy in the expression pane by clicking a plus sign, '+', to expand and a minus sign, '-', to contract. When doing this, the threads in the thread viewer pane will also expand and contract to exactly follow the expression pane.

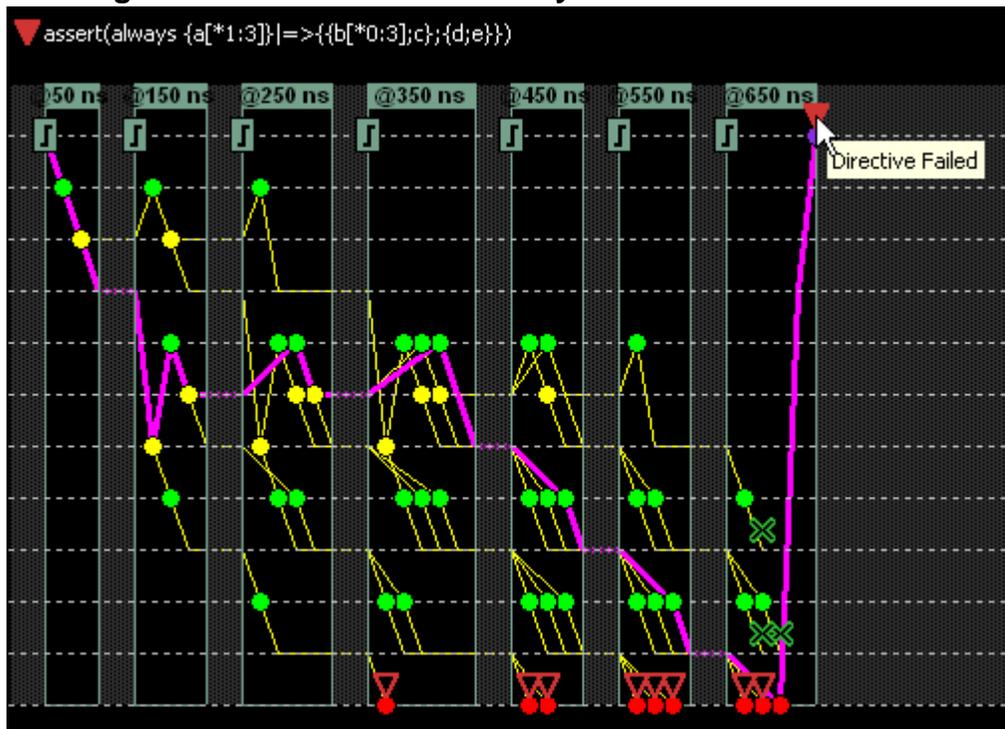
Highlighting a Thread

You can highlight any thread in the Thread Viewer Pane to differentiate it from the other threads by simply clicking on it with the left mouse button. Highlighting appears as a bold purple line. Depending on where the thread is clicked, any sub-threads that are forked and occur to the right of the click-point will also be highlighted. Going left, parent thread events which lead up to the current thread and selection point will also be highlighted. Any parent thread forks, other than the one which leads to the selected thread, will not be highlighted.

Highlighting for Root Thread Analysis

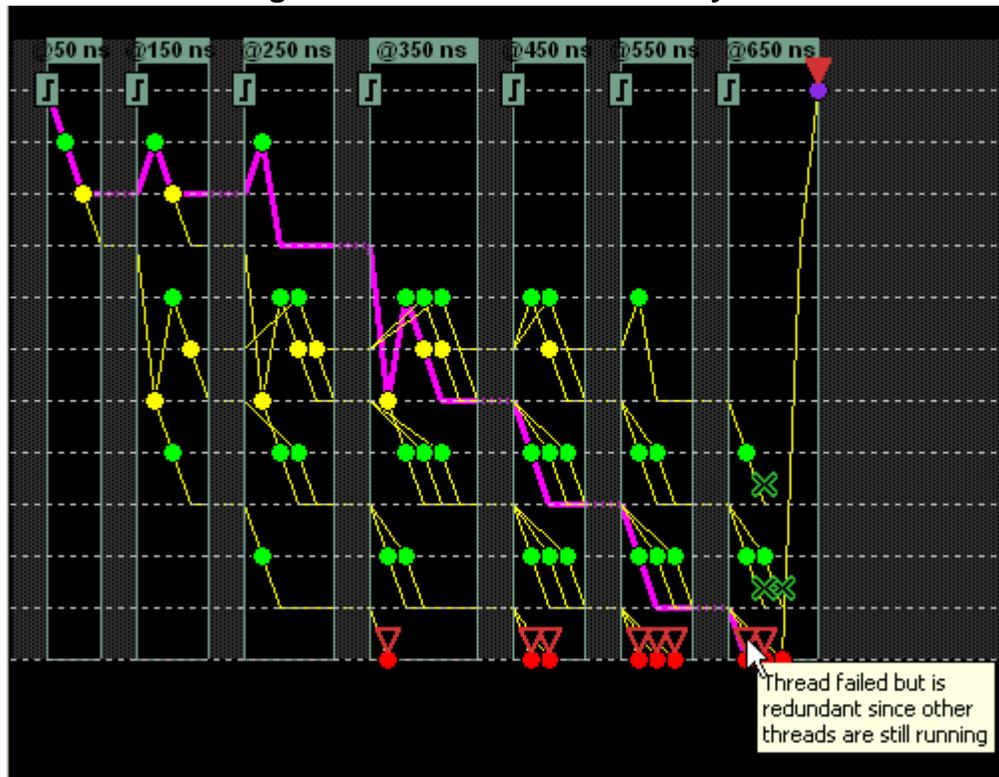
Highlighting provides a quick visual aid to root thread analysis of assertion failures. Red icons in the Thread Viewer Pane indicate failures. Clicking any red icon, like the Directive Failed icon (solid red triangle) in Figure 22-41, will highlight the path from the start thread to the failed thread.

Figure 22-41. Root Thread Analysis of a Directive Failure



In Figure 22-42, a Thread Failed icon (hollow red triangle) is clicked, showing the path from the start thread to the failure. In this case, the thread failure is redundant because other threads of the assert directive are still running.

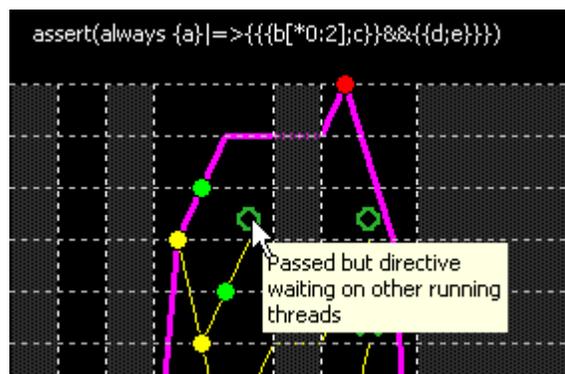
Figure 22-42. Root Thread Analysis



Hovering the Mouse for Thread and Directive Status

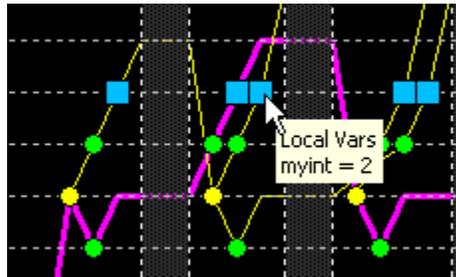
Hovering the mouse over any [ATV Window Graphic Symbols](#) reveals information about assertion thread or directive status. In [Figure 22-43](#), the cursor hovers over a hollow green circle and the status indicates “Passed but directive waiting on other running threads.”

Figure 22-43. ATV Mouse Hover Information



Hovering over a SystemVerilog local variable assignment symbol (a large blue square) reveals the values assigned to the particular local variables at that point ([Figure 22-43](#)).

Figure 22-44. ATV Mouse Hover Information - SystemVerilog



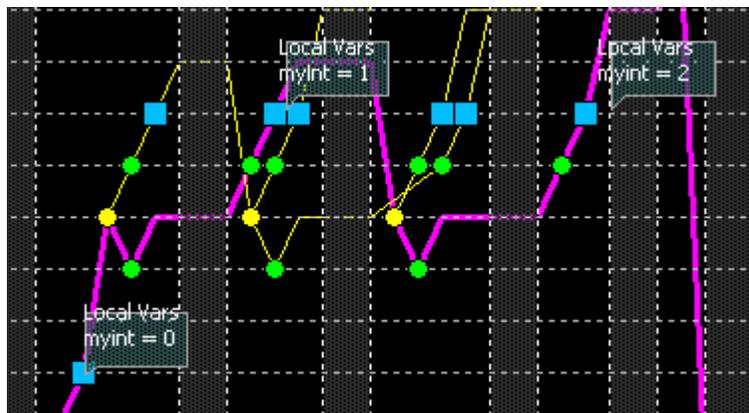
Annotating Local Variables

The ATV window allows you to annotate local variables in the highlighted thread of the Thread Viewer Pane using one of the following methods:

- Click the **Annotate Local Vars** icon in **ATV Toolbar**. 
- When the ATV window is docked in the Main window, select **ATV > Annotate Local Vars** from the Main menu bar.
- When the ATV window is undocked, select **View > Annotate Local Vars** from the menu bar.
- Right-click anywhere in the Thread Viewer Pane and select **Annotate Local Vars** from the popup menu.

When Annotate Local Vars is selected, annotation appears in the Thread Viewer Pane *only* on the highlighted thread, as shown in [Figure 22-45](#).

Figure 22-45. Local Variables Annotation in Thread Viewer Pane



Chapter 23

Verification with Functional Coverage

Note



The functionality described in this chapter requires an additional license feature for ModelSim SE. Refer to the section "[License Feature Names](#)" in the Installation and Licensing Guide for more information or contact your Mentor Graphics sales representative.

Functional coverage is user-defined coverage – in contrast with code coverage, which is automatically inferred from the source. At the most abstract level, functional coverage specifies some values to observe at certain times in a design or test bench, and counts how many times those values occur.

It should be noted, however, that complete functional coverage (100% coverage) does not necessarily indicate design correctness, or even that all bugs have been observed. Rather, functional coverage is a confidence metric. It is an implementation of a test bench – a way for the simulator to track that certain values and events occurred as expected while running the test bench. If the test bench is comprehensive and the coverage model (set of functional coverage metrics) correctly implements the test bench, and the design produces correct results with 100% functional coverage, then there is a high degree of confidence that all important bugs have been found.

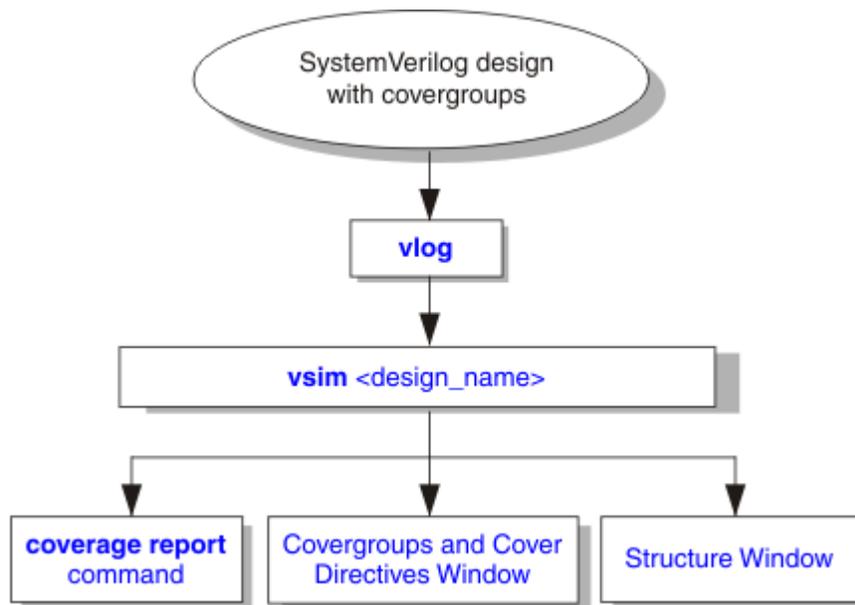
SystemVerilog implements functional coverage with covergroups and cover directives. Because cover directives are usually temporal and can inspect multiple signals and states in the same evaluation, they are usually inserted by designers in the design source as "white box" testing – i.e., they specify and validate the expected behavior of a design. This allows the verification tool to observe (and log) when particular events occur or assumptions are violated.

Because covergroups operate upon integral values and have limited temporal features, they are most often inserted in the test bench itself as "black box" testing. That is, the values monitored by covergroups are most often high-level test bench or design features, like transaction types, modes, addresses, opcodes, and so on.

Functional Coverage Flow

The following diagram shows the SystemVerilog functional coverage flow in ModelSim.

Figure 23-1. SystemVerilog Functional Coverage Flow



ModelSim compiles SystemVerilog functional coverage constructs along with other SystemVerilog code when either the source file ends in `.sv` or you specify the `-sv` argument to `vlog` command.

When you invoke `vsim` on the top-level of the design, the simulator automatically handles any functional coverage constructs that are present. Next, you run the simulation. You may optionally view coverage interactively in the GUI with commands such as ‘`coverage report`’, and/or save off coverage to the Unified Coverage DataBase (UCDB) with the ‘`coverage save`’ command. The UCDB can then be used for reporting, merging, ranking, or other types of verification analysis.

Configuring Covergroups, Coverpoints, and Crosses

SystemVerilog offers a rich set of facilities for configuring covergroups, coverpoints, and crosses through the `option` and `type_option` structures. To set values for these variables from the ModelSim command line interface or a `.do` file, use the `change` command. For example:

```
change covervar.type_option.weight 20
```

If set at the covergroup syntactic level, this command specifies the weight of this covergroup for computing the overall cumulative (or type) coverage of the saved database. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the cumulative (or type) coverage of the enclosing covergroup.

Note that a *type_option* is not accessible with the type name and "::" in the command line interface.

Functional Coverage Computation

SystemVerilog provides the following built-ins, available for use with ModelSim.

Predefined Coverage Methods

The SystemVerilog LRM defines two built-in methods (`get_coverage()` and `get_inst_coverage()`), applied at the the covergroup, coverpoint, and cross scope. All of these built-ins return a coverage metric as a floating point number that represents achieved coverage as a percentage (where 100.0 is the maximum possible coverage). Refer IEEE Std 1800-2009 LRM, Section 19.8 for details on these methods.

For a description of per-type coverage aggregation computation and information on how functional coverage participates in the total coverage aggregation, see [“Coverage Aggregation in the Structure Window”](#).

Predefined Coverage System Function

The SystemVerilog LRM defines the `$get_coverage()` built-in system function as "the overall coverage of all coverage group types." Additionally, *type_option.weight* at the covergroup scope is used to weight an individual covergroup type's contribution to the “overall cumulative (or type) coverage.” Keeping in mind that different module instances create different covergroup types, as described in the previous section, the `$get_coverage()` system task returns a weighted average as follows:

Numerator = sum over all covergroup types of (`type::get_coverage()` value * `type::type_option.weight`)

Denominator = sum over all covergroup types of (`type::type_option.weight`)

For details on this function, you can refer to IEEE Std 1800-2009 LRM, Section 19.9.

SystemVerilog Functional Coverage Terminology

The LRM refers to “cumulative” coverage and “type” coverage as the same thing; we use the term “type-based” coverage. The LRM sometimes uses “coverage” and “instance coverage” interchangeably; we use “instance-based coverage.”

IEEE Std 1800-2009 Option Behavior

The coverage system default behavior for SystemVerilog covergroups (ModelSim version 6.6 and higher) is compliant with the latest IEEE Std1800-2009 clarifications and changes.

Note



The command line options discussed in this section have no influence on coverpoint or cross options, and thus those options are not discussed here. See the LRM for complete details.

The changes as they relate to covergroup calculation and reporting — and instructions for reverting the settings — are summarized in [Table 23-1](#):

Table 23-1. Questa SIM and SystemVerilog IEEE 1800-2009 Options

1800-2009 Option	Default in Questa	To revert to pre-6.6 behavior, use:
option.per_instance	0 - no instance data is reported (it is saved, though)	set to 1, or use vsim -cvgperinstance
type_option.merge_instances	0 - algorithm used to calculate is average of instances	set to 1, or use vsim -cvgmergeinstances
option.get_inst_coverage	0 - only valid when merge_instances is set (1). Both get_inst_coverage and get_coverage return the same merged coverage result	set to 1

Example of Option Settings

Here's an example showing all three options being set:

```
covergroup cgtype (int lhs, rhs);
  type_option.merge_instances = 1;
  option.per_instance = 1;
  option.get_inst_coverage = 1;
  coverpoint vbl {
    bins b[] = { lhs, rhs };
  }
endgroup
```

The default in the 2009 standard, if you were to explicitly set it in the code, would be:

```
covergroup cgtype (int lhs, rhs);
  type_option.merge_instances = 0;
  option.per_instance = 0;
  option.get_inst_coverage = 0;
  coverpoint vbl {
    bins b[] = { lhs, rhs };
  }
endgroup
```

A report could be generated with the following report command:

coverage report -details

The report report generated would look like this:

```
# COVERGROUP COVERAGE:
# -----
# Covergroup          Metric      Goal/      Status
#                   Metric      At Least
# -----
# TYPE /top/cgtype  50.0%     100        Uncovered
#
# TOTAL COVERGROUP COVERAGE: 50.0%  COVERGROUP TYPES: 1
```

In this case, the coverage of the covergroup type is the average of the two instances. Since each of the two instances has 50% coverage, the type-based coverage is also 50%. The type-based coverage is computed as a weighted average based on the `option.weight` of each instance. The instances themselves do not appear for a reason explained in “[SystemVerilog 2009 option.per_instance](#)”.

Note



Since the type-based coverage is calculated from instances rather than child coverpoints and crosses, the `type_option.weight` specified in coverpoints and crosses has no effect on the type-based coverage of the covergroup.

SystemVerilog 2009 option.per_instance

The `option.per_instance` covergroup option has a clarified meaning in IEEE Std 1800-2009. Namely, you must set it in order to have instances appear in the report and the coverage database. Coverage database behavior is explicitly allowed to be vendor-specific, so the ModelSim simulator saves the instance anyway in the database when `option.per_instance` is set equal to 0. Without it, post-process merging would be crippled with a strict interpretation of the LRM.

Example 23-1. With `option.per_instance=1` or `vsim -cvgperinstance`

```
module rwb;

    typedef enum {red, white, blue} color;

    color c1;

    covergroup color_cg;
        coverpoint c1;
    endgroup : color_cg

    color_cg cg_inst = new();

    initial
    begin
        c1 = red;
        cg_inst.sample();
    end
```

```
endmodule : rwb
```

To run this code, execute the following commands:

```
vlib work  
vlog rwb.sv  
vsim -c rwb -do "run ; coverage report -details"
```

The resulting coverage report created is:

```
# COVERGROUP COVERAGE:  
# -----  
# Covergroup                Metric      Goal/  
#                           At Least      Status  
# -----  
# TYPE /rwb/color_cg        33.3%      100      Uncovered  
#   Coverpoint color_cg::c1 33.3%      100      Uncovered  
#  
# TOTAL COVERGROUP COVERAGE: 33.3%  COVERGROUP TYPES: 1  
#
```

In this report, no bins are reported for the type because the instances have not been merged into the type. The type coverage is computed only as the weighted average of the coverage of the instances -- so all that must be shown are the instances themselves.

Now, if you add the `-cvgperinstance` switch to the `vsim` command and generate another coverage report, the `per_instance` behavior can be thought of as a debug/analysis aid. Use the `-cvgperinstance` switch as follows:

```
vsim -cvgperinstance -c rwb -do "run -all ; coverage report -details"
```

This produces the following coverage report:

```
# coverage report -details  
# COVERGROUP COVERAGE:  
# -----  
# Covergroup                Metric      Goal/  
#                           At Least      Status  
# -----  
# TYPE /rwb/color_cg        33.3%      100      Uncovered  
#   Coverpoint color_cg::c1 33.3%      100      Uncovered  
#   Covergroup instance \rwb/cg_inst 33.3%      100      Uncovered  
#   Coverpoint c1           33.3%      100      Uncovered  
#     covered/total bins:    1          3  
#     missing/total bins:    2          3  
#     bin auto[red]          1          1      Covered  
#     bin auto[white]        0          1      ZERO  
#     bin auto[blue]         0          1      ZERO  
#  
# TOTAL COVERGROUP COVERAGE: 33.3%  COVERGROUP TYPES: 1  
#
```

SystemVerilog 2009 `type_option.merge_instances`

The `merge_instances` option did not exist in IEEE Std 1800-2005, therefore any SystemVerilog code which contains it may not compile or may trigger compile warnings when run in any version prior to 6.4. The option was introduced in IEEE Std 1800-2009, which also defines its default behavior. Questa’s default in effect is the same as the LRM default behavior.

The default algorithm used to calculate the coverage score for a covergroup type is an average-of-instances algorithm. See “[SystemVerilog 2009 option.per_instance](#)” and “[Legacy Behavior and Option Recommendations](#)” for related details.

SystemVerilog 2009 `option.get_inst_coverage`

The ModelSim simulator’s interpretation of `option.per_instance` was as an enabler for instance-based coverage as well as whether instances appeared in the report and database. This has been clarified in SystemVerilog 2009. By "enabler" we mean essentially whether the `get_inst_coverage()` method can provide instance based coverage, which can be different than the type (cumulative) coverage from the `get_coverage()` method. This enabling functionality is now set with `option.get_inst_coverage`, which applies only when `type_option.merge_instances` set equal to 1 (by default it is set to 0).

[Table 23-2](#) offers a summary of the behavior of `get_inst_coverage()` and `get_coverage()` as they relate to `type_option.merge_instances` and `option.per_instance`:

Table 23-2. Option Settings and Coverage Results

<code>type_option.merge_instances</code>	<code>option.get_inst_coverage</code>	<code>get_inst_coverage()</code>	<code>get_coverage()</code>
0	either 0 or 1	individual instance	average of all instances
1	0	merge of all instances	merge of all instances
1	1	individual instance	merge of all instances

In this table, the default settings for the options are indicated with bolded values.

Again, the `option.get_inst_coverage` is only applicable when you set `type_option.merge_instances` to 1; when it is, the default behavior of the `get_inst_coverage()` method is such that the the per-instance data is not tracked, and thus it is not available from `get_inst_coverage()`. If desired, the tracking of per-instance data can be turned on by setting the `option.get_inst_coverage` to 1.

Example 23-2. Different Results with `get_inst_coverage` and `get_coverage`

In this example, using both 'merge_instances' and 'get_inst_coverage' you can get different results for the two methods: `get_inst_coverage` and `get_coverage`.

```
module get_inst_example;
```

```
typedef enum {red, white, blue} color;
color c1;
covergroup color_cg;
  type_option.merge_instances = 1;
  option.get_inst_coverage = 1;
  coverpoint c1;
endgroup : color_cg
color_cg cg_inst_1 = new();
color_cg cg_inst_2 = new();
initial
  begin
    c1 = red;
    cg_inst_1.sample();
    $display("Sampled 'red' in cg_inst_1 and results are...");
    $display("cg_inst_1.get_inst_coverage() == %f",
cg_inst_1.get_inst_coverage());
    $display("cg_inst_1.get_coverage() == %f", cg_inst_1.get_coverage());
    $display("");
    $display("cg_inst_2.get_inst_coverage() == %f",
cg_inst_2.get_inst_coverage());
    $display("cg_inst_2.get_coverage() == %f", cg_inst_2.get_coverage());
  end
endmodule : get_inst_example
```

The results displayed would be as follows:

```
# Sampled 'red' in cg_inst_1 and results are...
# cg_inst_1.get_inst_coverage() == 33.333333
# cg_inst_1.get_coverage() == 33.333333
#
# cg_inst_2.get_inst_coverage() == 0.000000
# cg_inst_2.get_coverage() == 33.333333
#
```

Legacy Behavior and Option Recommendations

Several changes have occurred over the past several releases of Questa. The changes are:

- 6.6 — Default settings for options were changed to be IEEE 1800-2009 compliant
- 6.4 — *type_option.merge_instances* and *option.get_inst_coverage* were added

It is possible that tests or scripts you have developed may depend on some legacy behavior. To ease the transition, use the information contained in [Table 23-1 on Questa SIM and SystemVerilog IEEE 1800-2009 Options](#).

Type-Based Coverage With Constructor Parameters

The behavior of type-based coverage with respect to parameterized covergroups was not well-defined in the IEEE Std 1800-2005 LRM, though it is in the 2009 version. This section describes the default behavior of the ModelSim tool, with a note at the end about how to achieve this in fully-compliant IEEE Std 1800-2009.

Type-based coverage is easy to understand for the case where all instances have the same number of bins, and the same values map to the same bins. However, SystemVerilog covergroups may be parameterized when they are constructed, creating implications for type-based coverage. Type-based coverage may be calculated for covergroups that have different numbers of bins because they were constructed differently. Consider [Example 23-3](#):

Example 23-3. Type-based Coverage

```

module top;
  int vbl;
  covergroup cgtype (int lhs, rhs);
    option.per_instance = 1;
    coverpoint vbl {
      bins b[] = { lhs, rhs };
    }
  endgroup
  cgtype cgvar1_2 = new(1,2);
  cgtype cgvar1_3 = new(1,3);
  initial
  begin
    vbl = 1;
    cgvar1_2.sample();
    vbl = 3;
    cgvar1_3.sample();
    $display("cgvar1_2.get_inst_coverage() == %f",
      cgvar1_2.get_inst_coverage());
    $display("cgvar1_3.get_inst_coverage() == %f",
      cgvar1_3.get_inst_coverage());
    $display("cgvar1_2.get_coverage() == %f", cgvar1_2.get_coverage());
    $display("cgvar1_3.get_coverage() == %f", cgvar1_3.get_coverage());
  end
endmodule

```

Here is the report that is generated using “[vcover report -details](#)”:

```

# COVERGROUP COVERAGE:
# -----
# Covergroup                               Metric      Goal/ Status
#                                           At Least
# -----
# TYPE /top/cgtype                          66.7%       100 Uncovered
#   Coverpoint cgtype::vbl                  66.7%       100 Uncovered
#     bin b[1]                               1           1 Covered
#     bin b[3]                               1           1 Covered
#     bin b[2]                               0           1 ZERO
#   Covergroup instance \top/cgvar1_2       50.0%       100 Uncovered
#     Coverpoint vbl                         50.0%       100 Uncovered
#       bin b[1]                             1           1 Covered
#       bin b[2]                             0           1 ZERO
#     Covergroup instance \top/cgvar1_3     50.0%       100 Uncovered
#       Coverpoint vbl                       50.0%       100 Uncovered
#         bin b[1]                           0           1 ZERO
#         bin b[3]                           1           1 Covered
#
# TOTAL COVERGROUP COVERAGE: 66.7%  COVERGROUP TYPES: 1

```

The instance coverage is clear – each covergroup instance has 50% coverage because each has two bins and only one of those is covered.

The type-based coverage is a little more complicated. The type-based coverage is 66.7%, or two out of three bins covered. The reason is that the type has three bins. The total number of bins is the union of bins of all instances. The *cgvar1_2* instance has the set of bins { *b[1]*, *b[2]* }. The *cgvar1_3* instance has the set of bins { *b[1]*, *b[3]* }. The union of these two sets of bins is { *b[1]*, *b[2]*, *b[3]* }. Of this set of bins, the set { *b[1]*, *b[3]* } is actually covered. So type-based coverage is $2 / 3 = 66.666666\%$.

Bin Names And Unions

In [Example 23-3](#), we do not count bin *b[1]* twice because it is the same in both covergroup instances. So how are bins determined to be the same?

ModelSim's approach is to consider bins to be the same if they have the same name. This relies on a naming convention that is not completely specified by the IEEE Std 1800-2009 LRM. Consider that there are three ways of specifying a bin:

```
1  bins a = { 1, 2, 3 }; // bin a
2  bins b[] = { 1, 2, 3 }; // bins b[1], b[2], b[3]
3  bins c[2] = { 1, 2, 3 }; // bins c[0] <- 1; c[1] <- 2,3
```

The first two examples clearly declare bins *a*, *b[1]*, *b[2]*, and *b[3]*. But what of the bins specified with identifier "c?" ModelSim specifies bins *c[0]* and *c[1]*. This is consistent with the constructs in the LRM where bins with an explicit size constant (as is the case for identifier "c") are taken very literally in order. For example, "bins *c[2]* = { 1, 1 };" is perfectly legal. In this case, *c[0]* is incremented when the value 1 is sampled, and so is *c[1]*.

Now, reconsider [Example 23-3](#) with an explicit size constant in the bin declaration, as shown in [Example 23-4](#):

Example 23-4. Bin Unions

```
module top;
  int vbl;
  covergroup cgtype (int lhs, rhs);
    option.per_instance = 1;
    coverpoint vbl {
      bins b[2] = { lhs, rhs }; // now with explicit size constant!
    }
  endgroup
  cgtype cgvar1_2 = new(1,2);
  cgtype cgvar1_3 = new(1,3);
```

```

initial
begin
  vbl = 1;
  cgvar1_2.sample();
  vbl = 3;
  cgvar1_3.sample();
  $display("cgvar1_2.get_inst_coverage() == %f",
    cgvar1_2.get_inst_coverage());
  $display("cgvar1_3.get_inst_coverage() == %f",
    cgvar1_3.get_inst_coverage());
  $display("cgvar1_2.get_coverage() == %f", cgvar1_2.get_coverage());
  $display("cgvar1_3.get_coverage() == %f", cgvar1_3.get_coverage());
end
endmodule

```

Here is [vcver report](#) output for this example:

```

# COVERGROUP COVERAGE:
# -----
# Covergroup                               Metric           Goal/           Status
#                                           At Least
# -----
# TYPE /top/cgtype                          100.0%           100             Covered
#   Coverpoint cgtype::vbl                  100.0%           100             Covered
#     bin b[0]                               1               1               Covered
#     bin b[1]                               1               1               Covered
#   Covergroup instance \/top/cgvar1_2      50.0%           100             Uncovered
#     Coverpoint vbl                          50.0%           100             Uncovered
#       bin b[0]                              1               1               Covered
#       bin b[1]                              0               1               ZERO
#     Covergroup instance \/top/cgvar1_3    50.0%           100             Uncovered
#       Coverpoint vbl                          50.0%           100             Uncovered
#         bin b[0]                             0               1               ZERO
#         bin b[1]                             1               1               Covered
#
# TOTAL COVERGROUP COVERAGE: 100.0%  COVERGROUP TYPES: 1

```

In this case, the union of the bins in the type is { *b[0]*, *b[1]* }. While it is true that in *cgvar1_2*, *b[1]* maps from value 2, and in *cgvar1_3*, *b[1]* maps from value 3, that does not matter for the type coverage. These are the same bin as far as the type-based coverage is concerned.

So in this case, *cgvar1_2* covers bin *b[0]* (mapped from *vbl==1*) and *cgvar1_3* covers bin *b[1]* (mapped from *vbl==3*). So the instance-based coverage is 50% for each instance, and the type-based coverage is 100% because both bins are covered for the union of bins in the type.

Viewing Functional Coverage Statistics in the GUI

SystemVerilog coverage statistics are displayed in the Covergroups Structure (sim) windows.

Note

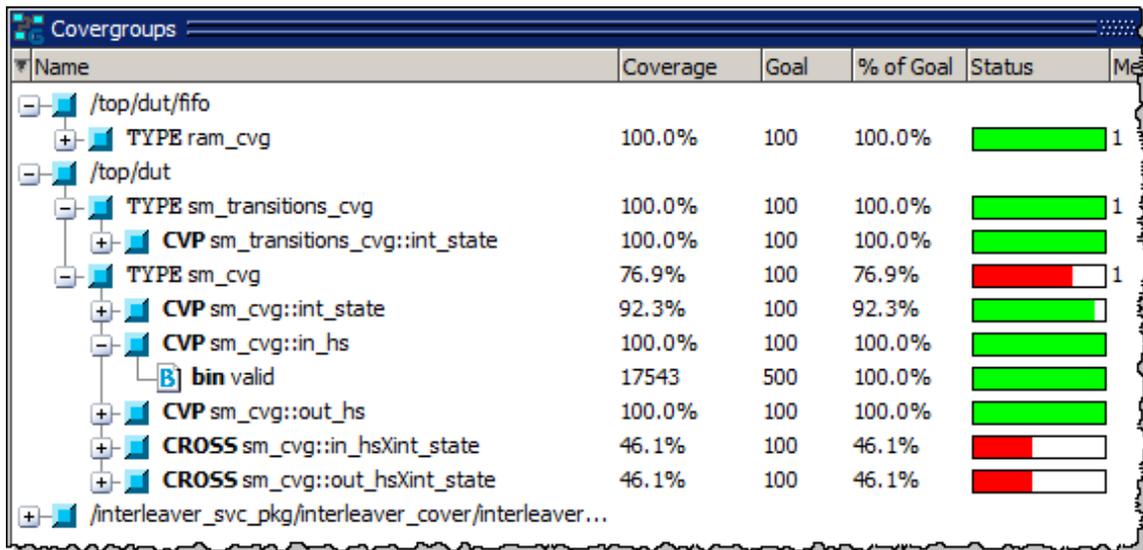
Covergroups are created dynamically during simulation. This means they will not display in the GUI until you run the simulation.

Functional Coverage Statistics in the Covergroups Window

You can view functional coverage statistics in the Covergroups window by performing the following steps:

1. Select **View > Coverage > Covergroups** from the Main window menu bar.
2. Run the simulation. (Covergroups are not displayed in the Covergroups window until you run the simulation.)

Figure 23-2. Functional Coverage Statistics in Covergroups Window



Name	Coverage	Goal	% of Goal	Status	M...
/top/dut/fifo					
+ TYPE ram_cvg	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	1
/top/dut					
- TYPE sm_transitions_cvg	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	1
+ CVP sm_transitions_cvg::int_state	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
- TYPE sm_cvg	76.9%	100	76.9%	<div style="width: 76.9%; height: 10px; background-color: red;"></div>	1
+ CVP sm_cvg::int_state	92.3%	100	92.3%	<div style="width: 92.3%; height: 10px; background-color: green;"></div>	
- CVP sm_cvg::in_hs	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
B bin valid	17543	500	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
+ CVP sm_cvg::out_hs	100.0%	100	100.0%	<div style="width: 100%; height: 10px; background-color: green;"></div>	
+ CROSS sm_cvg::in_hsXint_state	46.1%	100	46.1%	<div style="width: 46.1%; height: 10px; background-color: red;"></div>	
+ CROSS sm_cvg::out_hsXint_state	46.1%	100	46.1%	<div style="width: 46.1%; height: 10px; background-color: red;"></div>	
+ /interleaver_svc_pkg/interleaver_cover/interleaver...					

The Covergroups window displays the Coverage of each covergroup, coverpoint, cross, and bin. Covergroup coverage is a weighted average of the coverage of the constituent coverpoints and crosses. See “[Functional Coverage Computation](#)” for additional details.

For a description of the columns that can be displayed, see “[GUI Elements of the Covergroups Window](#)”.

Functional Coverage Aggregation in Structure Window

The columns specifically related to functional coverage statistics are calculated as follows:

- **Covergroups %** column shows a weighted average of all covergroup type-based coverage and cover directive coverage in the sub-tree. Covergroup coverage is calculated using the `get_coverage()` method and `type_option.weight` weighting.
- **Cover directives %** column shows a weighted average using the weights you set (see “[Weighting Cover Directives](#)”). By default, a cover directive is weighted equally with a covergroup type.
- **Assertion %** column shows a calculation using the total number of assertions and the number of assertions which are covered. The number of covered assertions is all the assertions which have never failed and where pass counts (if available) that are greater than 0. If pass counts are not available, then the number of covered assertions is the same as the number of assertions that have never failed.

For a description of the coverage summary aggregation for Total Coverage column, see “[Coverage Aggregation in the Structure Window](#)”.

Reporting on Functional Coverage

You can create functional coverage reports using dialogs accessible through the GUI or via commands entered at the command line prompt.

[Creating Text Reports Via the GUI](#)

[Creating Text Reports Via the GUI](#)

[Covergroup Bin Reporting and Timestamps](#)

[Filtering Functional Coverage Data](#)

[Reporting Via the Command Line](#)

Creating Text Reports Via the GUI

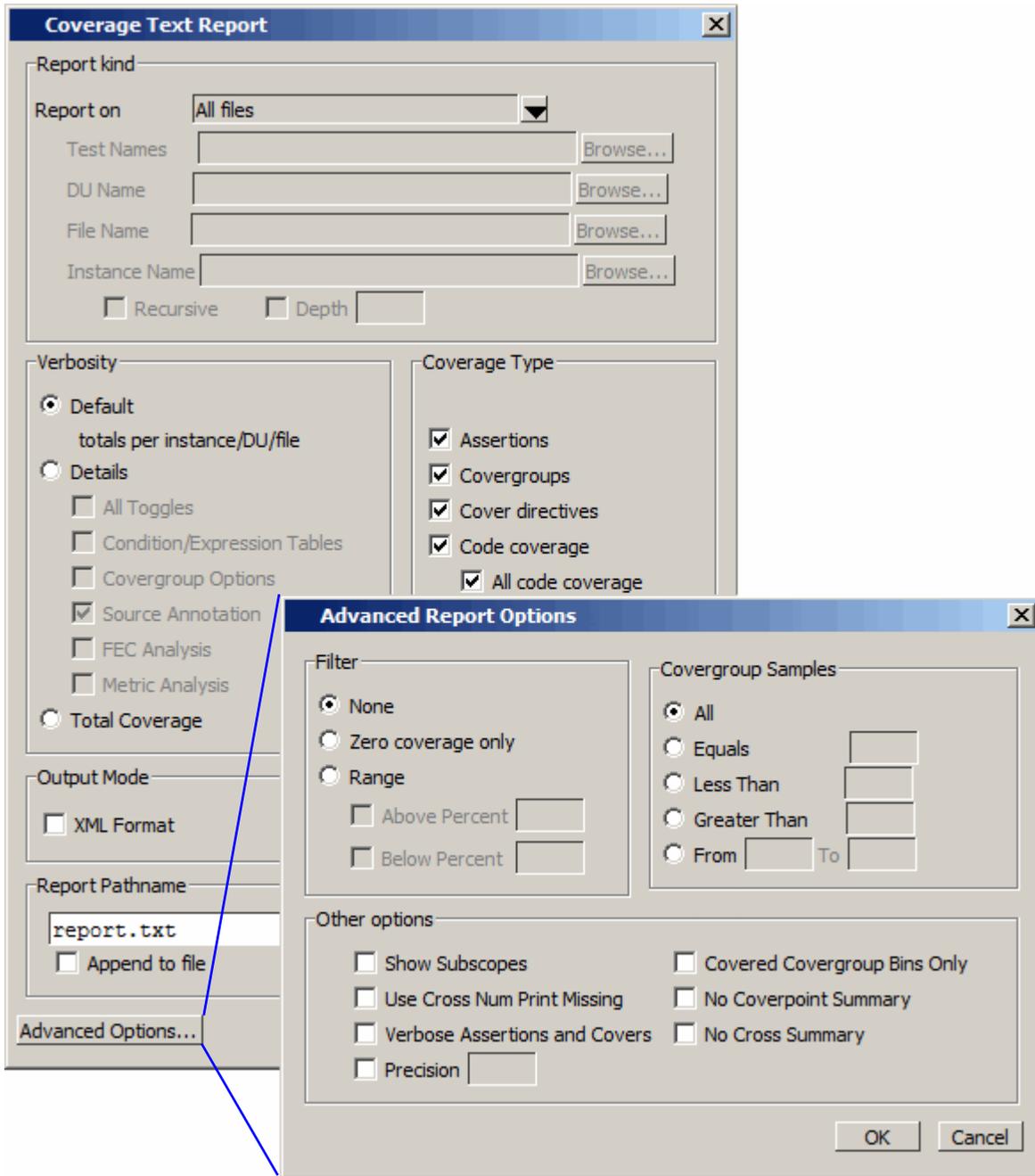
To save a functional coverage text report do any of the following:

- With the Cover Directives window active, select **Cover Directives > Report** from the Main menu.
- Right-click anywhere in the Cover Directives window and select **Report** from the popup menu.
- With the Covergroups window active, select **Covergroups > Report** from the Main menu.
- Right-click anywhere in the Covergroups window and select **Report** from the popup menu.

Any of these actions will open the Functional Coverage Report dialog.

You can create an ASCII file with the functional coverage statistics by selecting **Tools > Coverage Report > Text**. This brings up the Coverage Text Report dialog (Figure 23-3).

Figure 23-3. Creating Functional Coverage Text Reports



Use the Coverage Text Report dialog to create functional coverage reports on specific instances or on all coverage items. Options allow you to report only on covergroup coverage or on directive coverage. If covergroup coverage is selected, a functional coverage report will be created using covergroup type objects.

Here are a couple of points to keep in mind about coverage reporting:

- Filtering does not affect the calculation of aggregated statistics. It merely affects the data displayed in the report.
- A report response of "No match" indicates that the report was empty.
- The report will be sorted such that all bins with 0 counts show up as the first rows. Then, within that first section of 0-count rows, the covergroups would be the secondary sort. Thus, all 0's in covergroup A would appear next to each other, and so on for covergroup B, and others.

Creating HTML Reports Via the GUI

You can create an HTML report of the functional coverage statistics that will appear in your browser by selecting **Tools > Coverage Report > HTML**, which opens the Coverage HTML Report dialog. See "[Generating HTML Coverage Reports](#)" for further details.

Covergroup Bin Reporting and Timestamps

The **-cvgbintstamp** option for the **vsim** command enables ModelSim to record the simulation time step (timestamp) whenever a covergroup bin is covered during a simulation run.

The timestamp values for covered covergroup bins are then displayed in any coverage reports you generate (see "[Reporting on Functional Coverage](#)"). The report contains a column displaying the timestamp of each covergroup bin. When reporting only covergroups, the coverage report contains two columns — one for the timestamp value and the other for the test name.

- Timestamp values are saved in the ucdb file along with coverage information when you save coverage. Timestamp values and test names are saved with each covergroup bin.
- For reports from a simulation run (i.e. not from a ucdb file), the test name column contains the string "Current Test" instead of a test name.
- Merging UCDB files —

If you merge two ucdb files that contain timestamp values, the output (merged) ucdb file maintains the earliest timestamp value of the ucdb files, even if the merged cover items have different at_least values. The merged ucdb file also maintains the test name for the test that has the smallest timestamp value.

As a result of this fact, if you merge two different UCDB files where a timestamp value for a covergroup bin is the same in both files, that timestamp value is only assigned to one of the tests. You will lose the information that the second test also covered the bin at that timestamp.

- Editing UCDB files —

- It is not possible to make a later change to the coveritem goal recorded in the database (i.e. after it's been saved) — for example by modifying the UCDB using the [coverage edit](#) command — and automatically infer a different timestamp (see “[Timestamps and UCDB Modification or Merging](#)”).
- If a test record is removed from the UCDB, the timestamp value can be hidden from the GUI, text report and HTML report, since the timestamp value references the test where the bin was covered.

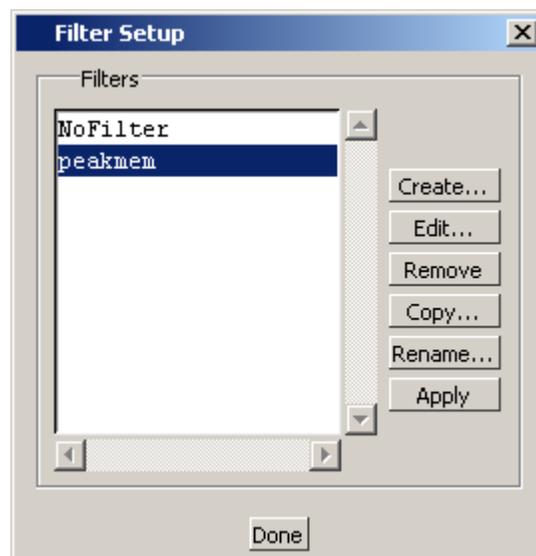
To avoid the computational overhead associated with the display of covergroup bin timestamps in HTML reports, apply the `-notimestamps` argument to the coverage report (or `vcover` report) `-html` command.

Filtering Functional Coverage Data

You can filter functional coverage data displayed in the GUI — including the Assertions, Cover Directives and Covergroups windows — as follows:

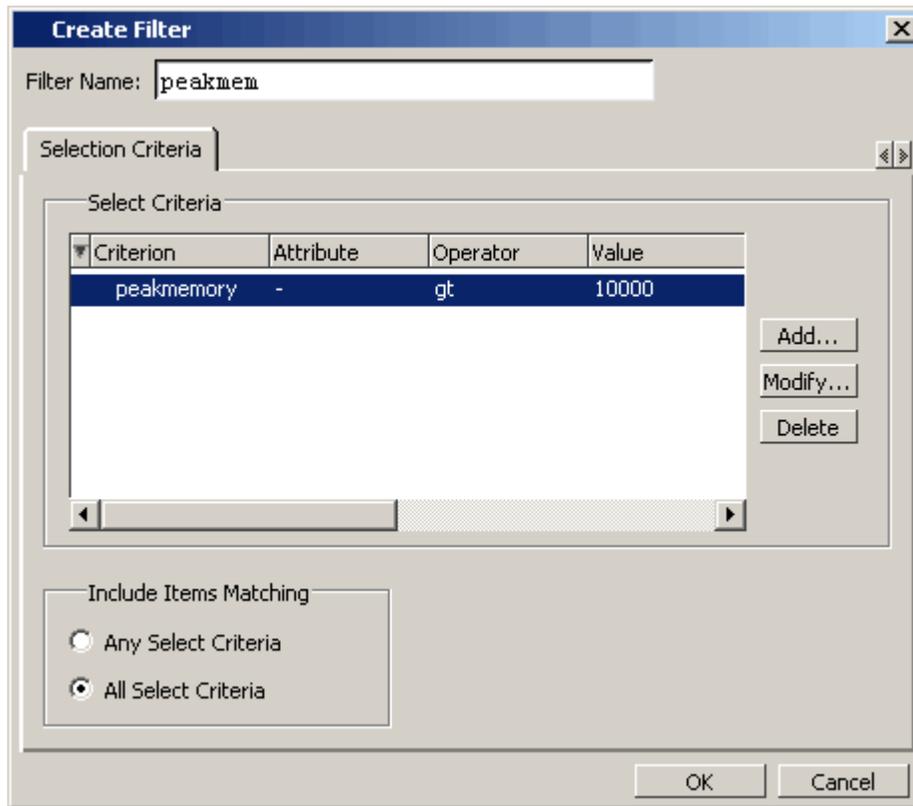
Activate the desired window and use the context sensitive menu pulldown and select Filter setup (i.e. **Covergroups** > **Filter** > **Setup**, or **Cover Directives** > **Filter** > **Setup**, or **Assertions** > **Filter** > **Setup**). This opens the Filter Setup dialog ([Figure 23-4](#)).

Figure 23-4. Filter Setup Dialog



To create a new filter, click the Create button to open the Create Filter dialog ([Figure 23-5](#)).

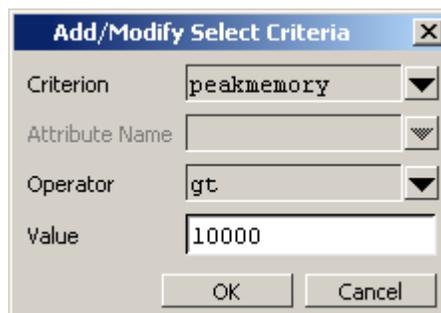
Figure 23-5. Create Filter Dialog



The Edit Filter dialog – which you open by clicking the Edit button in the Filter Setup dialog - contains all of the same functions as the Create Filter dialog.

Click the Add button to add criteria, attributes, operators, and values to the filter in the Add/Modify Select Criteria dialog (Figure 23-6).

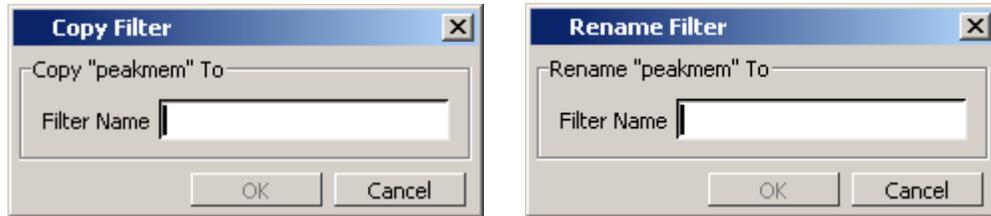
Figure 23-6. Add-Modify Select Criteria Dialog



The Criterion field includes a dropdown list that corresponds to the columns in the Assertions/Cover Directives/Covergroups window, allowing you to filter the display according to values in a specific column or columns.

You can copy the criteria from an existing filter into another by clicking the Copy button in the Filter Setup dialog, which opens the Copy Filter dialog. Or, can rename a filter by clicking the Rename button and opening the Rename Filter dialog ([Figure 23-7](#)).

Figure 23-7. Copy and Rename Filter Dialogs



The filter you just created appears in the Filters list within the Filter Setup dialog box ([Figure 23-4](#)). Either select **Apply** to filter the displayed data immediately, or select **Done** to exit the dialog box.

Reporting Via the Command Line

Two commands produce textual reports of functional coverage statistics – [coverage report](#) and [vcover report](#). The [vcover report](#) command allows you to produce textual reports from saved functional coverage statistics when a simulation is not loaded. See the Reference Manual for more details on [vcover report](#).

The following is sample report output from saved data using the [vcover report](#) command.

Example 23-5. Sample Output From vcover report Command

```

# -----
# Covergroup                               Metric           Goal/ Status
#                                           At Least
# -----
# TYPE sm_cvg                             72.5%           90 Uncovered
#   Coverpoint sm_cvg::int_state          50.0%           90 Uncovered
#     bin idle_bin                        38             500 Uncovered
#     bin load_bins                       5112           500 Covered
#     bin send_bins                       20800          500 Covered
#     bin others                          0             500 ZERO
#   Coverpoint sm_cvg::in_hs              100.0%          90 Covered
#     bin auto[0]                         21448           1 Covered
#     bin auto[1]                         4502            1 Covered
#   Coverpoint sm_cvg::out_hs             100.0%          90 Covered
#     bin auto[0]                         21449           1 Covered
#     bin auto[1]                         4501            1 Covered
#   Cross sm_cvg::in_hsXint_state         62.5%           90 Uncovered
#     bin <auto[0],idle_bin>               15             1 Covered
#     bin <auto[1],idle_bin>               23             1 Covered
#     bin <auto[0],load_bins>              633            1 Covered
#     bin <auto[1],load_bins>              4479           1 Covered
#     bin <auto[0],send_bins>              20800          1 Covered
#     bin <auto[1],send_bins>              0             1 ZERO
#     bin <auto[0],others>                 0             1 ZERO
#     bin <auto[1],others>                 0             1 ZERO
# -----
# Name           Design      Design   Lang File(Line)      Count Status
#               Unit         UnitType
# -----
#cover_intl_sm interleaver Verilog   SVA   interleaver.v(140)  375 Covered

```

By default, the report includes coverage statistics for both covergroups and cover directives. You may specify the `-covergroup` or `-directive` options for the `vcover report` command to report only covergroup coverage or only cover directive coverage, respectively.

Assertion/Cover Directive Naming Conventions

Assertion and cover directive names are important for identification in ModelSim reports and coverage windows, as well as for restoring in simulation with `$load_coverage_db()` (see [Loading a Functional Coverage Database into Simulation](#)). ModelSim assigns names to any unnamed assertion and cover directives, but these assigned names can and do change as changes are made to the source files in which they are located. You can avoid this situation by explicitly naming assertions and cover directives in the SystemVerilog source files. Consider the following properties:

```

my_assert: assert property (@(posedge clk) a ##1 b);
my_cover:  cover property (@(posedge clk) a ##1 b);

```

The assertion name `my_assert` and cover directive name `my_cover` allow you to easily identify those particular properties throughout all windows and cover reports.

Covergroup Naming Conventions

Covergroup instance names are important for identification in ModelSim reports and coverage windows, as well as for restoring in simulation with `$load_coverage_db()` (see [Loading a Functional Coverage Database into Simulation](#)). Covergroup instance names are stored in the `option.name` field of the built-in covergroup structure. You may assign these names in SystemVerilog, in which case names should be unique. This is not an absolute requirement, though it is recommended.

You can assign names in one of three ways:

- by directly assigning to `covergroupvariable.option.name`
- assigning to `option.name` within the covergroup declaration based upon a covergroup argument
- using the covergroup method `set_inst_name()`.

If you do not assign a name, the system assigns a unique name to each covergroup instance based upon the hierarchical path of the variable to which the covergroup instance was assigned with the "new" constructor. This generated name uses Verilog escaped identifier syntax because this instance name becomes part of UCDB hierarchy when a UCDB is saved. For example:

```
module top;
  covergroup ct; option.per_instance = 1; ... endgroup
  ct cv = new;
```

In this case, the instance name will be `"\top/cv "`. Note the extra space at the end to terminate the extended identifier. The report will identify the instance as `"\top/cv "`. The covergroup type name will be `"/top/ct"` and in UCDB hierarchy the covergroup instance will appear as `"/top/ct/\top/cv "`.

Covergroup in a Class

There are a couple of unexpected cases in covergroup naming conventions that occur with the embedded covergroup (the covergroup in a class):

- SystemVerilog 2009 requires that the embedded covergroup create an anonymous type, while the declaration name is considered a covergroup variable. The UCDB does not use the anonymous type name; it uses the variable name or declaration name. This has the consequence that the simulator's context tree (seen in the Structure window) is different from the context tree created in Coverage View mode. The coverage reports and user interface will be consistent between interactive simulation and Coverage View mode, but the hierarchy will not be.

Example:

```
module top;
  class c;
```

```
int i;
covergroup ct;
    coverpoint i { bins b[] = { [0:9] }; }
endgroup
```

In this case, the simulator hierarchy browser will show “/top/c/#ct#” as the covergroup type. The coverage view mode hierarchy browser will show /top/c/ct.

- **Parameterized classes** — There is no prescribed naming scheme for specializations of parameterized classes. ModelSim names these as the class name suffixed by “__” and a unique integer. These names appear in the path naming the covergroup type. For example:

```
module top;
    class child #(type T = bit, int size = 1 );
        T [size-1:0] l1;
        covergroup cg;
    ...
    endclass
    child #(logic, 3) child_inst1 = new;
    child #(bit, 4) child_inst2 = new;
```

In this case, the hierarchy browser will show “/top/child/child__1” as the scope for the first class specialization, and /top/child/child__2 as the scope for the second. In complex cases, it may not be obvious which index-suffixed name corresponds to which specialization.

Saving Functional Coverage Data

By default, no coverage information is saved into a Unified Coverage DataBase (UCDB) for later post-processing applications — regardless of whether coverage statistics were collected for a simulation — unless you explicitly save it.

To save functional coverage data (both covergroup coverage and cover directive coverage) into the UCDB:

- For all simulations:

Uncomment the [UCDBFilename](#) variable in the *modelsim.ini* file. The default filename is *vsim.ucdb*. You can also change the default name of the database by editing that variable.

- For the current simulation:

- **Command Line** — enter the [coverage save](#) -onexit command at the command line. For example:

```
coverage save -onexit [UCDBFilename <name>]
```

- **GUI** — select **Tools > Coverage Save** from the Main window

This opens the Coverage save dialog, where you can select the type of coverage you want to save, level of hierarchy to save, and output UCDB file name.

Once set using one of the above methods, the coverage data is saved at the end-of-simulation, which can occur as a result of:

- an assertion failure
- execution of explicit \$finish in Verilog
- execution of an implicit \$finish
- execution of \$stop
- execution of sc_stop()
- a fatal error

Loading a Functional Coverage Database into Simulation

To load a functional coverage database into simulation, use the \$load_coverage_db(), a standard SystemVerilog built-in system task, which loads covergroup data once it has been saved. It loads the data from a specified UCDB file into the current simulation. In cases where there are differences in design hierarchy or covergroups between the loaded design and the saved UCDB file, the \$load_coverage_db system task loads as much data as possible.

Usage Guidelines

The \$load_coverage_db() task loads the coverage data into the simulator using the following guidelines:

- Existing data values (bin counts, option, and type_option values) are replaced by the corresponding values from the database file when a data object is identified.
- A covergroup TYPE is identified first by its hierarchical path in the design. Then, a covergroup instance -- only those covergroups with option.per_instance set to 1 -- is identified by its option.name in the list of instances of a covergroup TYPE. All coverpoints, crosses, and bins will be identified subsequently by exactly matching their names. A bin count is then loaded, but only if a bin is identified properly.
- If a covergroup, coverpoint, cross, or bin is present in the loaded design but absent from the database file, then it is ignored (remains unchanged), and a non-fatal error message is issued.
- Similarly, if a covergroup, coverpoint, cross, or bin is not identified in the design (in other words, it exists in the database file, but is absent from the loaded design), it is ignored and a non-fatal error message is issued.

i **Tip:** You can suppress non-fatal error messages issued during object identification failures using the `-suppress <msgid>` argument to `vsim`, or the `"suppress = <msgid>"` directive in the `modelsim.ini` file.

- Bin identification tries to match bin RHS values when the `option.per_instance` is set. Any mismatch results in a failure to load that bin and a non-fatal error message to that effect. The bin RHS values are ignored:

- if `option.per_instance` is not set
- for automatically created cross bins

Automatically created cross bins are not identified by names; rather, they are identified by a pair of index values stored in UCDB files. If the index values are out of bound, a non-fatal error message is issued stating that the bin is not found: Otherwise, the bin is loaded.

i **Tip: Trap:** Avoid loading incorrect automatically created cross bins: If the index pair points to a different automatically generated cross bin in the simulation, you can inadvertently load the incorrect cross bins without any notification.

Loading Behavior Related to `option.per_instance`

When you load a coverage database in simulation, using `$load_coverage_db`, ModelSim requires that you set `option.per_instance` equal to 1 in all instances. Use the [SVCovergroupPerInstanceDefault](#), in the `modelsim.ini` file, to set `option.per_instance` for all covergroup instances in the design.

In all of the following cases, the covergroup information will be ignored (remains unchanged) in the loaded database and a error message is issued:

- only some instances of a covergroup have `option.per_instance` equal to 1

In this case, the task reports this as an error, and that covergroup fails to be loaded. To avoid this, you must set `option.per_instance` equal to 1 in either all, or, none of the instances of any covergroup.

- the covergroup TYPE does not have any instances in the loaded design

If `option.per_instance` is set, then the instance object is loaded, even if that instance object is no longer referenceable (in other words, the instance variable is re-assigned to another handle without destroying the previous object).

Excluding Functional Coverage

ModelSim supports several methods for excluding functional coverage from the coverage report and/or UCDB.

During live simulation, from within the source code, using either of these two methods:

- Using the SystemVerilog extension *option.no_collect* of type “bit,” which allows you to turn off the collection of coverage statistics for covergroups, coverpoints, and crosses. The default value is 0, and coverage information is collected. When set to 1, coverage information is not collected.
- Simulating with the `vsim -cvgzwnocollect` enabled, when the `option.weight` is set to 0 within the source code. The `modelsim.ini` variable `SVCovergroupZWNOCOLLECT` can be used in place of the switch to take effect for all `vsim` invocations.

Or, in post-processing (Coverage View) as well as live simulation, from a loaded coverage database (.ucdb):

- using the `coverage exclude` command in the following variations:

```
coverage exclude -cvgpath <path_to_covergroup>
```

```
coverage exclude -assertpath <path_to_assertion>
```

```
coverage exclude -dirpath <path_to_assertion>
```

See [coverage exclude](#) for full syntax details of the above listed arguments.

Sample Commands for Excluding Functional Coverage

Example 23-6. Functional Coverage in Code

```
module statemach(input bit i,
                input bit reset,
                input bit clk,
                output bit[1:0] state_out);
    enum { st0, st1, st2, st3 } state;

    always @(posedge clk or posedge reset)
    begin
        if (reset)
            state = st0;
        else
            case(state)
                st0: if (i==0) state = st2;
                    else      state = st1;
                st1: if (i==0) state = st1;
                    else      state = st2;
                st2: if (i==0) state = st1;
                    else      state = st2;
                st3: if (i==0) state = st1;
                    else      state = st3;
            endcase
    end
endmodule
```

```

        endcase
    end

    always @(state)
        state_out = state;

    covergroup machcover @(state);
        option.per_instance = 1;
        type_option.merge_instances = 1; /* this is important for excluding
                                         the type */
    st: coverpoint state;
    reset: coverpoint state {
        bins resetseq[] = ( st1,st2 => st1,st2 => st0 );
    }
    endgroup

    machcover cov;

    initial cov = new;

endmodule

```

To exclude a covergroup type:

```
coverage exclude -cvgpath /top/mach/machcover
```

To exclude a coverpoint/covercross under a covergroup type:

```
coverage exclude -cvgpath /top/mach/machcover/st
```

To exclude a cover bin under covergroup type:

```
coverage exclude -cvgpath {/top/mach/machcover/st/auto[st2]}
```

Note the braces {}, which prevent Tcl from evaluating *[st2]*.

To exclude a covergroup instance:

```
coverage exclude -cvgpath {/top/mach/machcover/\\/top/mach/cov }
```

Note the space at the end of the instance specification. It is intended.

To exclude a coverpoint/covercross under covergroup instance:

```
coverage exclude -cvgpath {/top/mach/machcover/\\/top/mach/cov /reset}
```

To exclude a cover bin under covergroup instance:

```
coverage exclude -cvgpath {/top/mach/machcover/\\/top/mach/cov
/reset/resetseq[st2=>st2=>st0]}
```

Merging Databases

When merging coverage databases offline using [vcover merge](#), the following parameters must be the same for a given scope:

- covergroup type name
- covergroup variable name
- coverpoint name
- bin name

If coverage data exists in the source database file but does not match the data in the target database, then it will be created in the target database. For more information, see “[Merging Coverage Test Data](#)”.

Chapter 24

Verification with Constrained Random Stimulus

Note



The functionality described in this chapter requires an additional license feature for ModelSim SE. Refer to the section "[License Feature Names](#)" in the Installation and Licensing Guide for more information or contact your Mentor Graphics sales representative.

In many modern electronic designs, exhaustive testing is impossible because the space of all possible inputs is too large. A simulator could not possibly reproduce all possible input vectors in any reasonable simulation time. Moreover, exhaustive testing may not be desirable because the set of interesting design states is much smaller than the set of possible inputs.

Ideally, it would be best to create targeted input vectors to test particular design states. However, this is often difficult because of the knowledge or time required to create all necessary inputs. Using constrained random stimulus allows a verification engineer to avoid the repetitive work that the simulator can perform automatically. In addition, it is possible that you obtain data on obscure corner cases that occur as consequences of the "random" input.

Functional coverage is required to evaluate which design states occurred (see [Verification with Functional Coverage](#)). Without functional coverage, there is no way of knowing what of interest actually occurred during a random test bench. It is also desirable to record the random seed with the coverage results, which the Questa UCDB takes into account.

Finally, note that fully random verification is a rarity. While some verification teams have used it exclusively, in most cases, random verification and targeted test vectors coexist because there is usually some design behavior that is, after all, easy to verify with targeted test benches.

SystemVerilog supports automated test bench development with random constraints, giving you the ability to automatically generate test benches for functional verification. SystemVerilog provides an object-oriented method for specifying constraints on random test bench values. ModelSim then processes these constraints using a constraint solver, which generates random values that meet those constraints.

The constraint solver tries to find a solution to the model of the problem by exploring the space of possible solutions and building up what is known as a "search tree." Any node in the tree represents an assignment to a variable, the depth of the tree represents the number of variables, and the width of the tree represents the number of elements in the domains of the variables. The solver explores nodes (variable values) in the search tree and rules out possible assignments that

cannot be part of a solution. For example, a solver may not visit the nodes in the search tree that cannot be part of a solution because a constraint was violated in a higher (preceding) node.

ModelSim provides the following constraint solvers (commonly referred to as “engines”), which you can choose to apply to a given verification run:

- **BDD** — Binary Decision Diagram. This solver interprets a constraint expression as a circuit representation and develops a truth table format of this expression, choosing random values for the variables based on these truth tables. While it is efficient in evaluating bit-wise operators, it proves to be much less efficient in solving complex constraint expressions where the solution space increases exponentially—thereby resulting in a large solution space. The nature of BDD implementation usually results in either enormous performance issues or inability to solve the expressions, which include:
 - Multiplication, Division, and Modulo
 - Array of objects
 - Large number of random variables with complex operators
- **ACT** — Arithmetic Constraint Technology. This solver reads a constraint expression and generates a graphical tree representation of a constraint expression, which consists of operator nodes and variable nodes. Each node is connected to two child nodes (which can be operators or variables). When this solver completes the generation of the entire graphical tree, it picks a random variable present down in the hierarchy and assigns a random value to it. It passes this value to the nodes in the upper hierarchy. The solver determines the value passed to the upper node and also the operator present in this node, and determines the valid range of values for the second variable associated with this node. From this valid range, it picks a random value. From this point, it continues to pass the values to the upper nodes until a complete solution is obtained.

Building Constrained Random Test Benches on SystemVerilog Classes

The SystemVerilog *class* data abstraction is ideally suited for building random stimulus.

- Classes are dynamically created, deleted, assigned and handled objects.
- Classes inherit properties and methods from other classes.
- Subclasses can redefine the parent’s methods explicitly.

These capabilities allow customization and randomization without breaking or rewriting known good functionality in the parent class.

Random tests are built onto the SystemVerilog class system by assigning special modifiers to class variables. The **rand** and **randc** modifiers can be used to designate a class variable as a random variable. Class variables designated with **rand** modifiers are standard random variables with values uniformly distributed over their range. Class variables designated with the **randc** modifiers are random-cyclic variables that cycle through all the values randomly in their declared range.

Values generated for random variables are controlled using constraints, as shown in [Example 24-1](#).

Example 24-1. The rand Variable

```
class Bus;
    rand bit [15:0] addr;
    rand bit [31:0] data;
    constraint word_align {addr[1:0] == 2'b0;}
endclass
```

This shows a simplified bus with the *addr* and *data* random variables, which represent the address and data values on the bus. The *word_align* constraint shows that only the *addr* random variable is constrained, the *data* variable is not constrained. The *data* variable will be assigned any value in its declared range.

ModelSim support of **randc** includes the following SystemVerilog types: integral, multi-dimension arrays, dynamic arrays, queues, and parameterized types. **randc** is not supported for associative arrays.

Generating New Random Values with `randomize()`

To generate new random values, you use the **randomize()** virtual method. [Example 24-2](#) shows how to use this method to generate random values for the bus example in [Example 24-1](#).

Example 24-2. Generating New Random Values With `randomize()`

```
Bus busA = new;
repeat (50)
    if ( busA.randomize() == 1 )
        $display("addr = %h data = %h", busA.addr, busA.data);
    else
        $display("Randomization FAILED");
end
```

Every class has a built-in **randomize()** virtual method, declared as:

```
virtual function int randomize();
```

Calling **randomize()** selects new values for all random variables in an object such that all constraints are satisfied. In [Example 24-2](#), the *busA* object is created and then randomized 50 times. The result of each randomization is checked for success. If randomization is successful,

the new random values for *addr* and *data* are displayed. If randomization fails, the “Randomization FAILED” error message is displayed.

Note  If ModelSim cannot acquire the proper license to execute the call to **randomize()**, it will display a fatal error message.

Using Attributes

The modelsim.ini file contains numerous variables whose settings control solver behavior for randomization. You can also use attributes that correspond to these variables when calling randomize() to apply their control only to that call. Using an attribute allows you to override the variable setting on a per-randomize basis.

Table 24-1 lists the attributes available for use with randomize(), along with the corresponding modelsim.ini variable.

Table 24-1. Attributes Usable with randomize()

Attribute	Variable in modelsim.ini	Description
solveactmaxtests	SolveACTMaxTests	Maximum number of tests that the ACT solver may evaluate before abandoning a solver attempt.
solvearrayresizemax	SolveArrayResizeMax	Maximum size randomize() will allow a dynamic array to be resized.
solvegraphmaxeval	SolveGraphMaxEval	Maximum number of evaluations performed on the solution graph generated during randomize().
solvegraphmaxsize	SolveGraphMaxSize	Maximum size of the solution graph that may be generated during a SystemVerilog call to randomize().
solvespeculatefirst	SolveSpeculateFirst	Specifies whether to use a BDD solution or “speculation” for a randomization scenario.
solvespeculatelevel	SolveSpeculateLevel	Specifies a level of “speculation” for randomization scenario.

Table 24-1. Attributes Usable with randomize()

Attribute	Variable in modelsim.ini	Description
solvespeculatedistfirst	SolveSpeculateDistFirst	Specifies whether to attempt speculation on solve-before constraints or dist constraints first.
solvespeculatemaxcondwidth	SolveSpeculateMaxCondWidth	Specifies the maximum bit width of a variable in a conditional expression that may be considered as the basis for “conditional” speculation.
solvespeculatemaxiterations	SolveSpeculateMaxIterations	Specifies the maximum number of attempts to make in solving a speculative set of random variables and constraints.

Syntax and Usage

Attributes for randomize() follow the definitions provided in the Verilog standard, IEEE Std 1364-2005 and IEEE Std 1364-2001.

To apply an attribute to a randomize() call, you include both as part an assert statement, according to Verilog syntax conventions. You can specify multiple attributes in a single randomize() call.

Examples:

```
assert(randomize (* solvearrayresizemax = 100 *) (a, b) with { a > b; });

assert(randomize (* solvearrayresizemax = 100 *) (* solveactmaxtests =
50000 *) (a, b) with { a > b; });
```

Improving Evaluation Performance with Solveflags Attribute

You can use the solveflags attribute with randomize() to modify the behavior of the constraint solver in order to improve the evaluation performance of some types of constraints. The “solveflags” attribute performs the same function as the **-solveflags=<flags>** option for the **vsim** command, but on a per-randomize basis (see [Example 24-3](#)).

Example 24-3. Using the solveflags attribute.

```
module top;

class TBar;
    rand int a;
endclass

TBar f = new;
int status;

initial
begin
    status = randomize (* solveflags="i" *) (f);
    $display(status);

    status = f.randomize (* solveflags="ri" *) ();
    $display(status);
end

endmodule
```

Size Constraints for Random Dynamic Arrays with randomize()

SystemVerilog size constraints for random dynamic arrays are supported by `randomize()`. A dynamic array with a size constraint may be resized by a call to `randomize()`. You can specify the maximum size `randomize()` will allow a dynamic array to be resized with the [SolveArrayResizeMax](#) variable under the `[vsim]` section of the `modelsim.ini` file (see [modelsim.ini Variables](#) in the appendix.). This variable specifies the maximum size `randomize()` will allow a dynamic array to be resized.

If `randomize()` attempts to resize a dynamic array to a value greater than specified by `SolveArrayResizeMax`, an error will be displayed and `randomize()` will fail. The default value for `SolveArrayResizeMax` is 2000. A value of 0 indicates no limit.

Debugging randomize() Failures

Conflicting constraints, such as $x > 5$ and $x < 3$, will cause the `randomize()` function to fail. You can use any one of the following three methods to debug `randomize()` failures caused by conflicting constraints:

- Set the [SolveFailDebug](#) simulator control variable to 1 in the `modelsim.ini` file. ModelSim will display the hierarchical path to the associated constraint name (except for implicit or in-line constraints).
- Use the `-solvefaildebug` argument with the `vsim` command to report any conflicting constraints with the simulation is run.
- Use the `solvefaildebug` SystemVerilog attribute in your source code. This attribute can be specified with no value (implicitly set to 1), 0, or 1. This attribute overrides the value

of the [SolveFailDebug](#) modelsim.ini variable for the associated SystemVerilog `randomize()` call, as well as the `vsim -solvefaildebug` command.

Examples:

```
status = randomize (* solvefaildebug *) (a, b) with { a > b; a < b; };
status = randomize (* solvefaildebug=1 *) (a, b) with { a > b; a < b; };
status = randomize (* solvefaildebug=0 *) (a, b) with { a > b; a < b; };
```

While evaluating constraints, you may encounter calculation overflow or underflow errors that are not critical. If you want the solver to ignore these errors while evaluating constraints, you can do so in either of the following ways:

- Use the `solveignoreoverflow` SystemVerilog attribute for the `randomize()` function in your source code. This ignores the error on a per-randomize basis.
- Set the [SolveIgnoreOverflow](#) simulator control variable to 1 in the modelsim.ini file.

Specifying a Solver Engine with solveengine Attribute

You can use the `solveengine` attribute of `randomize()` to select the best constraint solver "engine" for a particular randomization. This attribute performs the same function as the `vsim -solveengine` command or the [SolveEngine](#) variable (defined in modelsim.ini), but on a per-randomize basis.

The following example shows how to use the `solveengine` attribute to choose the Arithmetic Constraint Technology (ACT) solver for a `randomize` call:

```
module top;

class TFoo;
    rand bit[7:0] a, b, c;
endclass

TFoo f = new;
int status;

initial
begin
    status = randomize (* solveengine="act" *) (f);
    $display(status);
    status = f.randomize (* solveengine="act" *) ();
    $display(status);
end

endmodule
```

Tuning the ACT Solver Engine with solveactretrycount Attribute

You can use the `solveactretrycount` attribute of `randomize()` to choose a value for the number of attempts the ACT solver will make to solve a randomization before quitting. This attribute performs the same function as the `SolveACTRetryCount` variable (defined in `modelsim.ini`), but on a per-randomize basis.

The following example uses the `solveactretrycount` attribute to choose a nonzero value for the ACT retry count for a particular `randomize` call:

```
module top;

class TFoo;
    rand bit[7:0] a, b, c;
endclass

TFoo f = new;
int status;

initial
begin
    status = randomize (* solveactretrycount=1 *) (f);
    $display(status);
    status = f.randomize (* solveactretrycount=2 *) ();
    $display(status);
end

endmodule
```

Inheriting Constraints

SystemVerilog constraints restrict the range of random variables and allow you to specify relationships between those variables. Constraints follow the same rules of inheritance as class variables so they can be inherited from the parent class to any subclass. In this example,

```
typedef enum { low, high, other } AddrType;
class MyBus extends Bus;
    rand AddrType type;
    constraint addr_rang {
        ( type == low ) -> addr inside { [ 0 : 15] };
        ( type == high ) -> addr inside { [128 : 255] };
    }
endclass
```

the `MyBus` class inherits all random values and constraints of the `Bus` class. A random variable called `type` has been added to control the address range with the `addr_rang` constraint. The value of `type` is used by the `addr_rang` constraint to select one of the three range constraints.

As you can see here, inheritance can be used to build layered constraints, giving you the ability to develop generalized models that can be constrained to perform application-specific tasks.

Enabling/Disabling Constraints and Random Variables

The SystemVerilog `constraint_mode()` method enables or disables named constraint blocks in objects. This gives you the ability to design constraint hierarchies where the lowest level constraints can represent physical limits grouped by common properties into named constraint blocks. These blocks can then be independently enabled or disabled. (See the IEEE Std 1800-2009 for more information on constraint blocks.)

In the same way, `rand_mode()` is used to enable or disable random variables. When random variables are disabled they behave just like nonrandom variables.

Examining Solver Failures

You may encounter situations in which the solver fails to randomize or solve the specified constraints. In general, solver failures fall into the following categories:

- The set of constraints cannot be solved. This is most often due to an error in specifying constraint variables. For example:

```
a > b; b > c; a < c;
```

- If you have not specified `vsim -solvefaildebug`, the solver returns a status of 0 and does not modify any random variables.
- If you have specified `vsim -solvefaildebug`, the solver does the following:

- a. Prints the message

```
** Note: tb.sv(19): randomize() failed;
```

which indicates the file line number of the failing call to `randomize()`.

- b. Generates a Verilog testcase that includes the constraints that cannot be solved, which you can use to further investigate the constrain conflict.
- c. The solver then attempts to find the minimum set of constraints that produce the conflict. In this example, all three of the constraints are required to cause the conflict.

Note that this is a conflict in the Verilog code—the set of constraints is not solvable and the problem is independent of solver setting. Changing the solver engine or using other `vopt` switches will not correct the problem. You must fix the conflict in constraint specification in the Verilog source code (in this example, delete: `a < c;`).

- The solver encounters an internal limitation.

The solver will report a message similar to the following:

```
file.sv(line#): randomize() failed; solution graph size exceeds  
limit (SolveGraphMaxSize=value)
```

```
file.sv(line#): randomize() failed; solution graph evaluations  
exceeds limit (SolveGraphMaxEval=value)
```

```
file.sv(line#): randomize() failed; ACT test count exceeds limit  
(SolveACTMaxTests=value)
```

This type of error message indicates that the solver could not solve the constraints due to some kind of limitation.

- If the failure is due to the values for the GraphMaxSize or GraphMaxEval variables, increasing those values in the modelsim.ini file may help. Another correction that is more likely to help is to change to an ACT solver engine.
- If the error is due to the value of the SolveACTMaxTests variable, increasing this value in the modelsim.ini file may help. Another correction that is more likely to help is to change to a BDD solver engine.

Following the error report, if you have specified `vsim -solvefaildebug`, the solver also dumps a testcase containing the constraints that were run.

Setting Compatibility with a Previous Release

You can specify random sequence generation compatibility with a prior ModelSim letter release (for the SystemVerilog solver) using either of the following methods:

- Set the `SolveRev` variable in the modelsim.ini file to a previous release.
- Use the `-solverev` argument with the `vsim` command to set a previous release.

The release number you designate must consist of release number and letter, such as 6.6a, but only prior letter releases within same number release are allowed. For example, if you are using version 6.6c, you can specify 6.6b, 6.6a, or 6.6, but cannot specify 6.5f.

Note



These instructions do not apply to the SystemC/SCV solver.

Seeding the Random Number Generator (RNG)

In SystemVerilog, each thread has its own RNG state. The seed of each child thread is initialized from the parent RNG. ModelSim gives you the ability to seed the root RNG with either a specific integer or a random number.

To seed the root RNG, use the `-sv_seed` argument for the `vsim` command.

i **Tip:** If you want to change the randomization seed after elaboration, you can do so by using `vsim -load_elab` and `vsim -elab`. You can elaborate the design once using the `-elab` argument and then use the `-load_elab` argument with different seed values specified with `-sv_seed` for subsequent simulation runs.

If you do not use `-sv_seed`, the value of the `Sv_Seed` variable in the `modelsim.ini` file is used as the value for the initial seed. If `Sv_Seed` is not specified, the initial seed value defaults to 0.

Examples

- Seed the root RNG with the integer 99:

```
vsim -sv_seed 99
```

- Seed the root RNG with a random number:

```
vsim -sv_seed random
```

How to Obtain the Random Seed Value

If you want to know the initial value of the random seed you can use the `$get_initial_random_seed` system function or enter the following command in a Tcl shell window:

```
echo $Sv_Seed
```

This displays the random seed value in the shell window.

Using Program Blocks

SystemVerilog program blocks identify and encapsulate test bench code so there is a clear separation between design and test bench. Program blocks provide an entry point to test bench execution, create a scope for program-wide data, and execute in the “reactive” region of the SystemVerilog simulation model. By using program blocks you can ensure race-free interaction between your design and your test bench.

A program block is similar to a module except that it supports only **initial** blocks – it cannot contain **always** blocks, UDPs, modules, interfaces, or other programs. Furthermore, a program block’s variables are not visible to design code, so any reference to program variables will result in an error.

Program blocks can be nested in modules or interfaces, allowing multiple programs to share variables local to the scope. For example:

```
module test(...)
  int shared; // variable shared by programs test1 and test2
  program test1;
  ...
endprogram
```

```
    program test2;  
    ...  
    endprogram // test1 and test2 are implicitly instantiated once in  
module test  
endmodule
```

For more information on using program blocks, refer to IEEE Std 1800-2009.

Chapter 25

Coverage and Verification Management in the UCDB

Note



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

This chapter contains the following, basic information regarding coverage and the management of your verification environment within ModelSim.

Additional Verification Management tools and capabilities (i.e. importing a verification plan to track your verification requirements, the Verification Tracker window, Results Analysis, and Trending) are available through the use of the “qyman” license feature. See the *Questa SIM Verification Management User’s Manual* for further information.

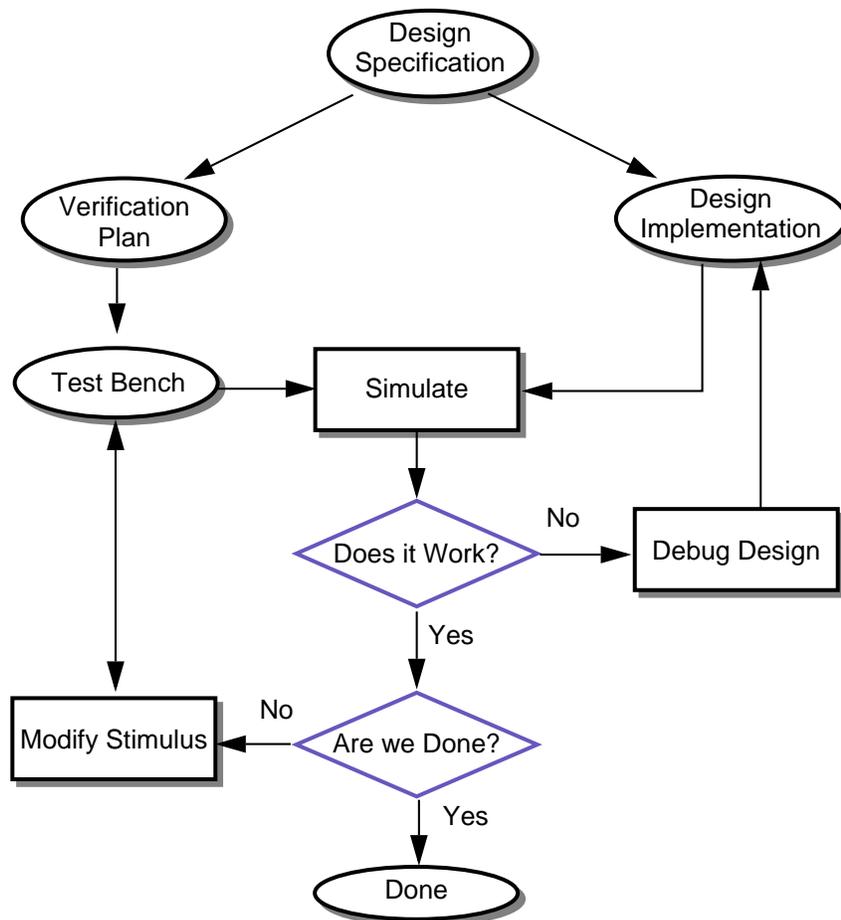
Coverage and Verification Overview	1088
What is the Unified Coverage Database?	1089
Calculation of Total Coverage	1090
Coverage and Simulator Use Modes.	1095
Coverage View Mode and the UCDB	1095
Running Tests and Collecting Data	1096
Collecting and Saving Coverage Data	1096
Understanding Stored Test Data in the UCDB.	1098
Rerunning Tests and Executing Commands.	1101
Managing Test Data in UCDBs	1104
Merging Coverage Test Data	1105
Ranking Coverage Test Data	1109
Modifying UCDBs	1111
About the Merge Algorithm	1113
Merge Usage Scenarios.	1118
Viewing and Analyzing Verification Data	1119
Understanding Stored Test Data in the UCDB.	1098
Storing User Attributes in the UCDB.	1120
Viewing Test Data in the Browser Window.	1120
Generating Coverage Reports.	1122
Filtering Data in the UCDB	1129
Filtering Results by User Attributes	1131
Storing User Attributes in the UCDB.	1120

Coverage and Verification Overview

- the merging and aggregation of coverage data
- ranking of tests
- ranking of tests within a test plan
- analysis of coverage in light of late-stage ECO's
- test and command re-runs
- various analyses of coverage data
- generation of easy-to-read HTML coverage reports

The flow described in [Figure 25-1](#) represents a typical design verification process as it can be applied in the ModelSim environment.

Figure 25-1. Verification of a Design



Every project starts with a design specification. The specification contains elaborate details on the design construction and its intent.

A verification team uses the design specification to create a verification plan. The verification plan contains a list of all questions that need to be answered by the verification process (the golden reference). The verification plan also serves as a functional spec for the test bench.

Once the test bench is built and the designers succeed in implementing the design, you simulate the design to answer the question: “Does it work?”. If the answer is no, the verification engineer gives the design back to designers to debug the design. If yes, it is time to ask the next question: “Are we done yet?”. Answering this question involves investigating how much of the design has been exercised by looking at the coverage data and comparing it against the verification plan.

What is the Unified Coverage Database?

The Unified Coverage DataBase (UCDB) is a single persistent form for various kinds of verification data, notably: coverage data of all kinds, and some other information useful for analyzing verification results. The types of coverage data collected in the database include:

- Code coverage: branch, condition, expression, statement, and toggle
- Finite State Machine (FSM) coverage
- SystemVerilog covergroup coverage
- SystemVerilog and PSL assertion coverage
- Assertion data (including immediate and concurrent assertions, and pass, non-vacuous pass, fail, attempt and other counts)
- 0-In Formal analysis results for OVL, QVL, CheckerWare, checkers and monitors, and SVA/PSL properties
- Verification Plan data
- Links between the Verification Plan and coverage data
- User-defined data
- Test data

The UCDB is used natively by ModelSim for all coverage data, deprecating previous separate file formats for code coverage and functional coverage. 0-In formal analysis data is written into the UCDB using a 0-In utility (see the 0-In “Formal User Guide”).

When created from ModelSim, the UCDB is a single “snapshot” of data in the kernel. Thus, it represents all coverage and assertion objects in the design and test bench, along with enough hierarchical environment to indicate where these objects reside. This data is sufficient to

generate complete coverage reports and can also be combined with data acquired outside ModelSim – for example, 0-In coverage and user-defined data.

For more information about the coverage data contained in the UCDB, see “[Understanding Stored Test Data in the UCDB](#)”.

Weighted Coverage

Weighting is a decision the verification engineer makes as to which coverage types are more important than others within the context of the design and the objectives of the test bench. Weightings might change based on the simulation run as specific runs could be setup with different test bench objectives. The weightings would then be a good way of filtering how close the test bench came to achieving its objectives.

For example, the likelihood that each type of bus transaction could be interrupted in a general test is very low as interrupted transactions are normally rare. You would probably want to ensure that the design handles the interrupt of all types of transactions and recovers properly from them. Therefore, you might construct a test bench such that the stimulus is constrained to ensure that all types of transactions are generated and that the probability of transactions being interrupted is relatively high. For that test bench, the weighting of the interrupted transaction cover points would probably be higher than the weightings of uninterrupted transactions (or other coverage criteria).

Calculation of Total Coverage

A summary of all coverage types in one aggregated total is available in:

- **Structure** (sim) window: **Total coverage** column
- **Verification Tracker** window: **Coverage** column
- **Verification Browser** window: **Total Coverage** column
- HTML coverage report ([coverage report](#) -html): **Design Coverage Summary** section
- [coverage analyze](#) command output

The coverage aggregation depends on the whether the scope of interest is a design unit or an instance:

- **Design Units** — aggregated coverage for a specific design unit is the weighted average of all kinds of coverage found within it. For coverage summary statistics viewable in the above listed locations, the coverage number is pre-aggregated into the design unit. This pre-aggregation behaves like a merge operation, where the coverage of the design unit is the union of coverage in all the instances of that design unit. This pre-aggregation occurs for all code coverage types, functional coverage (both covergroups and cover directives), and assertions which have succeeded or have been formally proven to succeed. The [coverage weight](#) command allows these to be weighted independently —

but globally — in the aggregation computation. This is equivalent to averaging together the numbers reported with “coverage report -bydu” for that particular design unit -- weighted by the coverage weights shown with “coverage weight -bydu”.

- Instances — aggregated coverage for an instance is computed from the weighted average of all different types of coverage found within the entire subtree rooted at that instance (recursive view, which is the default) or within the particular instance (local view). View coverage locally in the Structure window by deselecting **Code Coverage > Enable Recursive Coverage Sums**. In the Verification Tracker window, testplan sections are treated similarly to instances in the Structure window. See “[Coverage Calculation in the Tracker Window](#)” for specific information.

Aggregation totals include the following coverage:

- all code coverage types - statements, branches, conditions, expressions, FSM, and toggles
- covergroups, coverpoints, and cover directives —

The crux of SystemVerilog functional coverage reporting is that coverage for a bin is binary – a bin is either covered or not covered – while coverage statistics are aggregated within a coverpoint and within covergroups as a percentage of desired coverage. Coverage statistics may be aggregated among all instances of a covergroup or per instance, as desired by the user.

- assertions which have succeeded or have been formally proven to succeed. A successful assertion is one that passed — if pass counts are available (use `vsim -assertdebug` to enable pass counts) — and never failed.

Coverage Binning and Calculation

The general algorithm for coverage aggregation can be expressed in the following formula

$$\frac{\sum cov(t) \times wt(t)}{\sum wt(t)}$$

where:

$cov(t) = \text{\#covered bins} / \text{\#bins}$

$t = \text{the set of all coverage types}$

The coverage types (t) are as shown in [Table 25-1](#). Each type of coverage has its own definition of how bins are created and how many bins exist.

Aggregation is performed across different scopes depending on the UI command issued (i.e. you can aggregate across a single design unit, or you can aggregate across the entire design, or various ranges between those extremes). The region in which coverage is calculated is known as

the “scope of aggregation”. For each coverage type, cov(t) is calculated across the complete set of bins visible in the scope of aggregation.

Table 25-1. Coverage Calculation for each Coverage Type

Coverage Type	Binning and calculation methods for cov(t)	Weighting
Assertion	There is one bin per assertion. The bin count is incremented when an assertion never fails (==0) AND it passes non-vacuously at least once. If pass counts are not available, the non-vacuous pass requirement is dropped. Use <code>vsim -assertdebug</code> to enable pass counts.	Each assertion is weighted equally — there is no way to individually weight assertions through the ModelSim user interface ¹
Branch	All True branches and “AllFalse” branches in the scope of aggregation form the complete set of bins. Calculation follows the general algorithm.	Weighted equally
Condition	All FEC and UDP table rows in the scope of aggregation form the complete set of bins. Rows that contain X and Z values are excluded. FEC and UDP numbers are calculated separately, then each contributes 50% to cov(t) for Condition coverage.	Weighted equally
Covergroup	Performed as if <code>\$get_coverage()</code> was called on the scope of aggregation. <code>\$get_coverage()</code> is described in SV1800-2009, Clause 19.11, Coverage Computation	Controlled by <code>type_option.weight</code>
Cover Directive	There is one bin per cover directive. The bin count is incremented when the cover directive passes.	Weighted equally, unless <code>-weight</code> is used with <code>fcover configure</code> command
Expression	All FEC and UDP table rows in the scope of aggregation form the complete set of bins. Rows that contain X and Z values are excluded. FEC and UDP numbers are calculated separately, then each contributes 50% to cov(t) for Expression coverage.	Weighted equally
FSM	The complete set of states in the scope of aggregation forms the state bins, similar for transitions. State coverage and Transition coverage are calculated separately according to the general formula. Then each contributes 50% to cov(t) for FSM coverage.	Weighted equally
Statement	There is one bin per statement. The bin count is incremented when the statement executes.	Weighted equally

Table 25-1. Coverage Calculation for each Coverage Type (cont.)

Coverage Type	Binning and calculation methods for cov(t)	Weighting
Toggle	Binning varies based on the type of togglenode. Enum, integer, and real types are binned specially. Extended toggle mode binning is different than regular toggle mode binning. See Toggle Coverage and Understanding Toggle Counts for further description. The complete set of all toggle bins in the scope of aggregation is used when calculating cov(t).	Toggle nodes are effectively weighted according to how many bins there are for the type of togglenode

1. Assertions can be weighted through the UCDB API.

Note



Weights in the [coverage weight](#) command for assertion counts (other than non-vacuous passes) are not used for any purpose.

The weights, listed by the different kinds of coverage, would be shown by entering:

coverage weight -byinstance

You can find out exactly what the coverage was for each coverage type using either of the following commands:

coverage analyze -path <instance> -summary

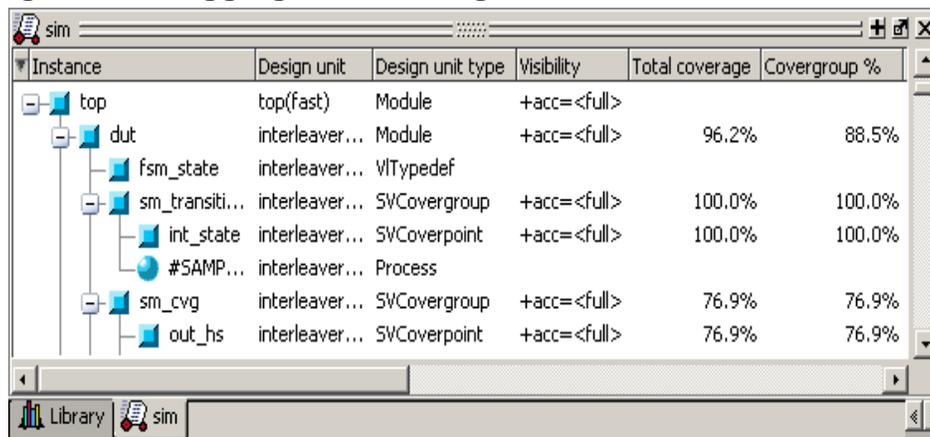
coverage analyze -du <du_name> -summary

The information from these commands, along with [Table 25-1](#), can help you understand the Total Coverage number. See [coverage analyze](#) for further details.

Coverage Aggregation in the Structure Window

Aggregated coverage data is displayed in Total Coverage column of the Structure (sim) window. Coverage statistics are calculated either for the local instance, or recursively. The data comprises covergroup, cover directive, assertion coverage, and code coverage on a per design region basis. Each sub-tree of design hierarchy has its own set of metrics.

Figure 25-2. Aggregated Coverage Data in the Structure Window



Instance	Design unit	Design unit type	Visibility	Total coverage	Covergroup %
top	top(fast)	Module	+acc=<full>		
dut	interleaver...	Module	+acc=<full>	96.2%	88.5%
fsm_state	interleaver...	VTypeDef			
sm_transiti...	interleaver...	SVCovergroup	+acc=<full>	100.0%	100.0%
int_state	interleaver...	SVCoverpoint	+acc=<full>	100.0%	100.0%
#SAMP...	interleaver...	Process			
sm_cvg	interleaver...	SVCovergroup	+acc=<full>	76.9%	76.9%
out_hs	interleaver...	SVCoverpoint	+acc=<full>	76.9%	76.9%

The Structure window includes the **Total Coverage** column, which by default shows a weighted average of all coverage types in the sub-tree recursively, including covergroup type-based coverage, and cover directive coverage, code coverage, assertion coverage, method and type_option.weight weighting. Cover directives are weighted using the weights you set (see “[Weighting Cover Directives](#)”). By default, a cover directive is weighted equally with a covergroup type.

When you disable the default selection of **Structure -> Code Coverage -> Enable Recursive Coverage Sums**, only constructs local to the current design instance contribute to the Total Coverage number. In this mode, the Total Coverage column displays coverage information for each design instance in isolation, and no contributions from child instances are taken into account.

Coverage Calculation in the Browser Window

The Browser window's Total Coverage column is calculated similarly to the Total Coverage column in the Structure window. However, in the Browser window, all design roots are taken into account in the calculation. This includes packages, top level modules, and other top level design units. Furthermore, the Browser's calculation is always done recursively, which means that all hierarchy underneath all design roots is taken into account.

Coverage Calculation in the Tracker Window

The Tracker window's Coverage column is calculated similarly to the Total Coverage column in the Structure window. Two distinct cases of coverage calculations are presented that column: one occurs when the section is a leaf testplan section, another when the section is a non-leaf testplan section.

- A leaf testplan section is a section which only has individual coverage links. The numbers from those links are combined using the usual per-type global weighting.

- A non-leaf testplan section is a section which has one or more child testplan sections. The calculation for a non-leaf testplan section is always performed recursively. Weighting is performed across all immediate child Testplan sections as usual, according to the weight assigned to each section. In addition, if there are any coverage links in a non-leaf testplan section, those are combined together and treated as a single child testplan section with a weight of 1.

Coverage and Simulator Use Modes

Most commands related to coverage are used in one of three simulation use modes that correspond to the type of coverage analysis required.

Table 25-2. Coverage Modes

Mode	Type of Coverage Analysis	Commands to Use
Simulation Mode	Interactive simulation	coverage , toggle , and vcover commands (such as clear , merge , report , save , tag , unlinked...)
Coverage View Mode	Post-process	coverage open <file>.ucdb or vsim -viewcov <file>.ucdb brings up separate GUI All coverage related commands are available
Batch Mode	Batch simulation	vcover commands (such as merge , report , stats , testnames...)

Each of these modes of analysis act upon a single, universal database that stores your coverage data, the Unified Coverage Database.

Coverage View Mode and the UCDB

Raw UCDB coverage data can be saved, merged with coverage statistics from the current simulation or from previously saved coverage data, and viewed at a later date using Coverage View mode.

Coverage View mode allows all ModelSim coverage data saved in the UCDB format – code coverage, covergroup coverage, directive coverage, and assertion data – to be called up and displayed in the same coverage GUIs used for simulation. The coverage view invocation of the tool is separate from that of the simulation. You can view coverage data in the Coverage View mode using the [coverage open](#) command or `vsim -viewcov`.

Running Tests and Collecting Data

Collecting and Saving Coverage Data	1096
Understanding Stored Test Data in the UCDB	1098
Rerunning Tests and Executing Commands	1101

Collecting and Saving Coverage Data

When you run tests, the coverage data which you instruct the simulator to save is placed in the Unified Coverage DataBase (UCDB). The UCDB is a single persistent form for various kinds of verification data, notably: coverage data of all kinds, and some other information useful for analyzing verification results. For more information on the UCDB, see “[What is the Unified Coverage Database?](#)”.

Prerequisites

Before coverage data can be saved, you must first:

1. Select the type of code coverage to be collected (vlog +cover). See “[Specifying Coverage Types for Collection](#)”.
2. Enable the coverage collection mechanism for the simulation run. See “[Enabling Simulation for Code Coverage Collection](#)”.
3. Run the simulation.

Naming the Test UCDB Files

By default, the test name given to a test is the same as the UCDB file base name, however you can name the test before saving the UCDB using a command such as:

```
coverage attribute -test mytestname
```

i **Tip:** If you are saving test data for later test-associated merging and ranking, it is important that the test name for each test be unique. Otherwise, you will not be able to distinguish between tests when they are reported in per-test analysis.

Saving Coverage Data

Optionally, you can save the coverage data to a UCDB for post-process viewing and analysis.

Methods

The coverage data you have collected can be saved either on demand, or at the end of simulation.

Saving Data On Demand

Options for saving coverage data dynamically (during simulation) or in coverage view mode are:

- GUI: **Tools > Coverage Save**

This brings up the Coverage Save dialog box, where you can specify coverage types to save, select the hierarchy, and output UCDB filename.

- [coverage save](#) CLI command

During simulation, the following command saves data from the current simulation into a UCDB file called *myfile1.ucdb*:

```
coverage save myfile1.ucdb
```

While viewing results in coverage view mode, you can make changes to the data (using the [coverage attribute](#) command, for example). You can then save the changed data to a new file using the following command:

```
coverage save myfile2.ucdb
```

- `$coverage_save` or `$coverage_save_mti` system tasks (not recommended)

The non-standard SystemVerilog `$coverage_save_mti` system task saves code coverage data only. It is not recommended for that reason. The `$coverage_save` system function is defined in the IEEE Std 1800; current non-compliant behavior is deprecated and also, therefore, not recommended. For more information, see “[Simulator-Specific System Tasks and Functions](#).”

Saving Data at End of Simulation

By default, coverage data is not automatically saved at the end of simulation. To enable the auto-save of coverage data, set a legal filename for the data using any of the following methods:

- GUI: **Tools > Coverage Save**. Enable the “Save on exit” radio button.

This brings up the Coverage Save dialog box, where you can also specify coverage types to save, select the hierarchy, and the output UCDB filename.

- `UCDBFilename=“<filename>”`, set in *modelsim.ini*

By default, `<filename>` is an empty string (“”).

- **coverage save -onexit** command, specified at `Vsim>` prompt

The **coverage save** command preserves instance-specific information. For example:

```
coverage save -onexit myoutput.ucdb
```

- `$set_coverage_db_name(<filename>)`, executed in SystemVerilog code

If more than one method is used for a given simulation, the last command encountered takes precedence. For example, if you issue the command **coverage save -onexit vsim.ucdb** before simulation, but your SystemVerilog code also contains a **\$set_coverage_db_name()** task, with no name specified, coverage data is not saved for the simulation.

Related Topics

- [Running Tests and Collecting Data](#)
- [Merging Coverage Test Data](#)
- [Ranking Coverage Test Data](#)

Understanding Stored Test Data in the UCDB

When you save a set of coverage data into the UCDB, that data is written into individual *test data records*, one for each test that is run. When you perform a merge on multiple UCDBs, all test data records with unique testnames are concatenated into the merged UCDB file. If you merge two tests that have identical names but different data contents, a warning is issued. For information on making test names unique, see “[Multiple Test Data Records with Same Name](#)”.

A merged file contains one test data record for each of the different tests that were merged into the file. For example, if you merged three UCDBs saved from simulation (vsim), you get three test attribute records. If you merge that file with another one saved from vsim, you get $3 + 1 = 4$ test data records, and so on.

Test Attribute Records in the UCDB

The test record contains “layers” of information. Each test record contains test record *attributes* (fields) which, in turn, contain two sub-sets of attributes. Specifically, these are:

- predefined attributes —
These contain information about the test, such as the tool specific arguments and switches, the date and time that the simulation was run, the amount of CPU time taken, the name of user that ran the test, and so on. These predefined attributes define the columns that appear by default in the Browser and Tracker windows when you view a UCDB. See [Table 25-3](#) for a list of predefined attributes which appear as columns.
- user-defined attributes (as name/value pairs) —
The values of these attributes define the columns that you can select to appear in the Browser window, and that appear by default in the Tracker window. See “[Storing User Attributes in the UCDB](#)” for instructions on creating user-defined attributes.

Test attribute records are stored in the UCDB when you save your coverage information. One test attribute record exists for each simulation (test). Each test attribute record contains name-value pairs — which are attributes themselves — representing information about that particular test. Many of these attributes within the test attribute record are predefined, however, you can also create your own using the [coverage attribute](#) command.

Several methods are available which allow you to interface with the test record and its attributes. These include:

- the UCDB database API (Application Programming Language). See the UCDB API User’s Manual for further details.
- the CLI (Command Line Interface) or directly within SystemVerilog using. See the ModelSim Reference Manual for command syntax.
- the DPI (Direct Programming Interface). See the appendix entitled “[Verilog Interfaces to C](#)”.

Using one of these methods it is possible to set values to the default fields or add any number of user defined fields to carry other interesting information about the verification run.

Predefined Attribute Data

Each field has a default value based on your simulation. You can override some values with the “[coverage attribute](#)” command while in simulation mode, and before saving the UCDB file with the “[coverage save](#)” command.

Caution



On Overloading Predefined Attributes:
 Some risk is inherent in overloading a predefined attribute (such as TESTSTATUS). If you change the setting of a predefined attribute to outside the set of expected values, unintended behavior may result.

[Table 25-3](#) lists fields in the test attribute record (in the UCDB) that are predefined for users.

Table 25-3. Predefined Fields in UCDB Test Attribute Record

Field / Attribute Name	Override with “coverage attribute” CLI Command	Setting/Description
TESTNAME	-name <testname> -value <string>	Name of the coverage test. This is also the name of the test record. If not set through CLI, default name is base name of the file when database is saved.
SIMTIME		Simulation time at completion of the test. (In ModelSim, the \$Now TCL variable contains the current simulation time in a string of the form: <i>simtime simtime_units</i> .)
TIMEUNIT		Units for simulation time: "fs", "ps", "ns", "us", "ms", "sec", "min", "hr".

Table 25-3. Predefined Fields in UCDB Test Attribute Record (cont.)

Field / Attribute Name	Override with "coverage attribute" CLI Command	Setting/Description
CPUTIME		CPU time for completion of the test. (In QuestaSim, the <i>simstats</i> command returns the CPU time.)
DATE		Timestamp is at the start of simulation. If created by ModelSim, this is a string of the format: yyyymmddhhmmss for example: 20060105160030 which represents 4:00:30 PM January 5, 2006
VSIMARGS	-name VSIMARGS -value <string>	Simulator command line arguments.
USERNAME	-name USERNAME -value <string>	User ID of user who ran the test.
TESTSTATUS	-name TESTSTATUS -value <int>	Status of the test, where the values ¹ are either an integer (transcript) or an enumerated value (GUI): 0 (OK) - this is the default setting 1 (WARNING) - severity level 2 (ERROR) - severity level 3 (FATAL) - severity level 4 (MISSING) - as a result of merge operation, indicates a directed test missing in UCDB that is referenced in the test plan 5 (MERGE_ERROR) - indicates error during merge process
ORIGFILENAME		Name of the test file.
SEED	-seed <int>	Randomization seed for the test. (Same as the seed value provided by the "-sv_seed" vsim option.)
COMPULSORY	-compulsory <0 1>	Whether (1) or not (0) this test should be considered compulsory (that is, a "must-run" test).
RUNCWD		Current working directory for the test.
HOSTNAME		Computer identification
HOSTOS		Operating System
WLFNAME		Associated WLF file for the test.

Table 25-3. Predefined Fields in UCDB Test Attribute Record (cont.)

Field / Attribute Name	Override with “coverage attribute” CLI Command	Setting/Description
LOGNAME		Associated transcript or log file for the test.
TESTCOMMENT	-comment	String (description) saved by the user associated with the test. This field will only appear when explicitly specified.
TESTCMD	-command	Test script arguments. Used to capture "knob settings" for parameterizable tests, as well as the name of the test script. This field will only appear when explicitly specified.

1. The listed TESTSTATUS values (integer) are reserved within ModelSim to indicate severity levels and other messages. If creating a new status indicator, it is strongly suggested that you use integers 6 and above.

Related Topics

- [coverage analyze Command](#)
- [Storing User Attributes in the UCDB](#)

Rerunning Tests and Executing Commands

You can rerun tests and execute commands from the **Verification Browser > Command Execution** functionality.

Requirements

The following requirement applies only if:

- you are running multiple simulations using the same UCDB filename and you have used the same UCDB name in different directories (fred/cov.ucdb, george/cov.ucdb, and so forth)

OR

- you are loading multiple UCDBs from the same basic test (that is, *fred.ucdb* is the basic test and you want to create multiple runs of that test)

If either of these cases is true, your **initial** simulation run (the one you intend to re-run) must include a command to set the TESTNAME attribute. Failure to set the TESTNAME attribute in these cases may result in otherwise unique tests being identified as duplicates (and therefore not executed) by the re-run algorithm and in the merge/rank output files. See the Tip below for further information.

To explicitly set the TESTNAME attribute in simulation, include a command such as:

```
coverage attribute -name TESTNAME -value <unique_test1>
```

See “[coverage attribute](#)” for command syntax.

i **Tip:** When you rerun a test, the simulator uses an attribute called TESTNAME, saved in each test record, to build a list of unique files selected for re-run of that test. By default, the TESTNAME is the pathless basename of the UCDB file into which the coverage results of a given test were stored from the initial run. See “[Multiple Test Data Records with Same Name](#)” for further details on ensuring unique test data records for subsequent runs.

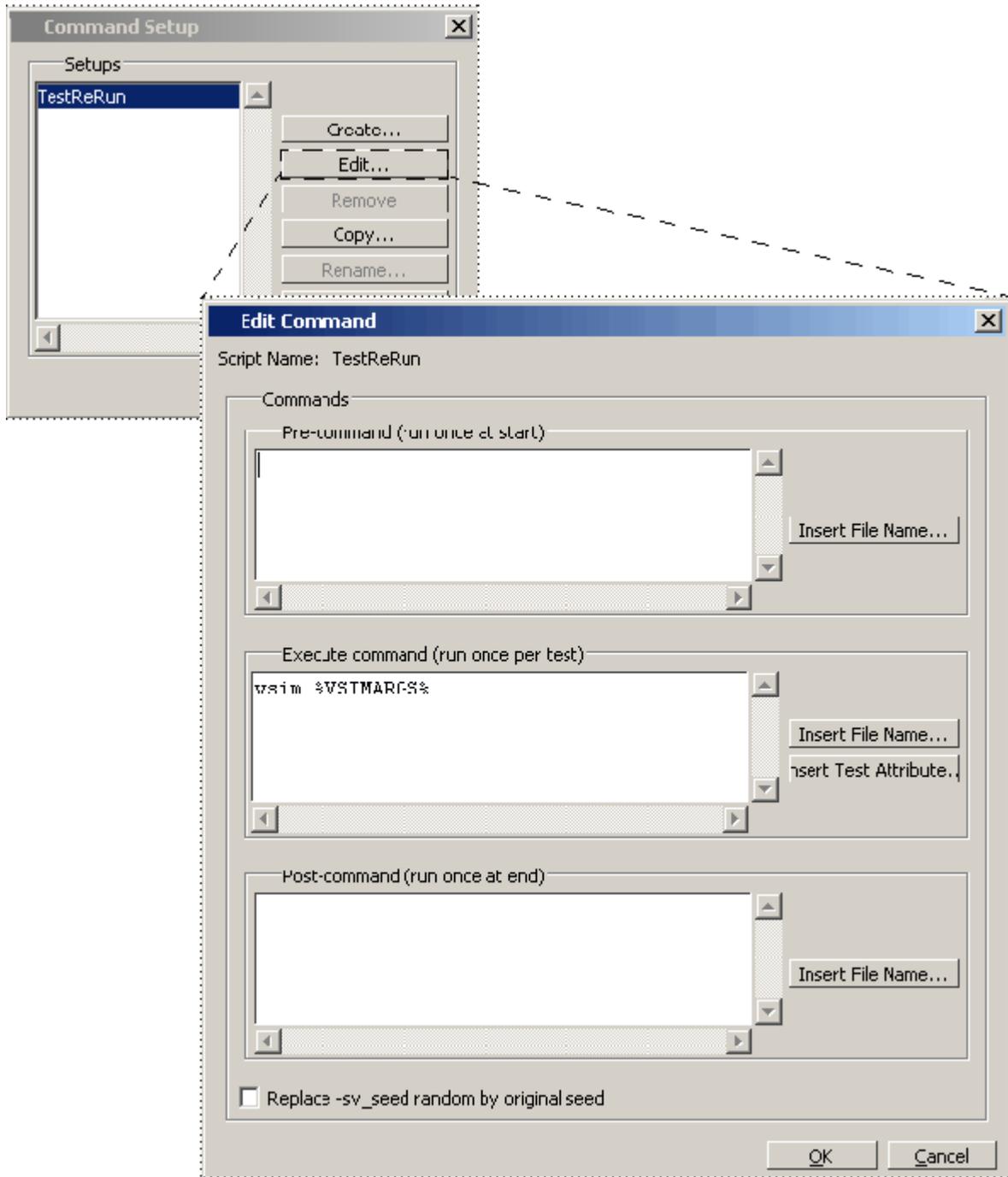
Procedure

To rerun a test or execute a command from the Browser:

1. Enter the re-run setup:
 - a. Select one or more UCDB files.
 - b. Right-click and select **Command Execution > Setup**.

This displays the Command Setup dialog box, shown in [Figure 25-3](#).

Figure 25-3. Command Setup Dialog Box



The Command Execution Setup dialog box allows you to select and view user-defined command setups, save new setups, and remove run setups previously saved. You can either select an existing test to re-run, or enter the following commands to run individually:

- Pre-command — a script you may need to run once at startup, prior to the run
- Execute Command — commands to execute: This field is pre-populated with the command(s) necessary to rerun the test(s) selected when you opened the dialog.
- Post-command — a script you may need to run at the end (for example, a cleanup script)

You can also select a radio button to apply the original seed (defined by the last run wherein the vsim -sv_seed random argument was used) to the current execution.

c. Select OK to apply the setup as edited.

2. Rerun the test: Right-click in the Browser, and select:

- **Command Execution > Execute on All** to re run all tests listed in browser, or
- **Command Execution > Execute on Selected** to run only those tests associated with the currently selected UCDB files.

Related Topics

- [Running Tests and Collecting Data](#)
- [Rerunning Tests and Executing Commands](#)
- [Ranking Coverage Test Data](#)
- [Merging Coverage Test Data](#)

Managing Test Data in UCDBs

Merging Coverage Test Data	1105
Warnings During Merge	1108
Ranking Coverage Test Data	1109
Modifying UCDBs	1111
About the Merge Algorithm	1113
Merge Usage Scenarios	1118

Once you have run tests and collected coverage, the real task of analyzing the coverage can begin. Managing the coverage data which has been collected and stored in one or more UCDBs is critical to the task. For example, you might have a coverage on a design which includes both the test bench and the design under test. But, you would like to separate out coverage of the TB from that of the DUT to examine them separately.

Another example scenario would be if you've run 100 test runs, all using different stimulus. Next, you would want to analyze the data in those UCDBs to determine the coverage redundancy, and eliminate extraneous tests. You can merge and rank the data for just this purpose.

You can also merge a verification plan (or “test plan”) with the actual coverage test data contained in the UCDB(s). If you are merging a verification plan with UCDB test data, you must have an imported test plan in UCDB format.

You can also edit a UCDB, modifying its contents, using the [coverage edit](#) command. See “[Modifying UCDBs](#)”.

Merging Coverage Test Data

When you have multiple sets of coverage data, from multiple tests of the same design, you can combine the results into a single UCDB by merging the UCDB files. The merge utility supports:

- instance-specific toggles
- summing the instances of each design unit
- source code annotation
- printing of condition and expression truth tables
- cumulative / concurrent merges
- a “master” merge, whereby you can identify a UCDB to use as the superset of test data to merge with other tests.

The coverage data contained in the merged UCDB is a union of the items in the UCDBs being merged. For more information, see “[About the Merge Algorithm](#)”.

A file locking feature of the merge allows for cumulative merging on a farm — “vclover merge out out in” — such that the “out” file is not corrupted with multiple concurrent merges. It recovers from crashing merges, crashing hosts, and allows time-out of merges, as well as backups of the previous output.

The tool allows you to merge test data using the:

- GUI: Verification Browser window
- Command Line: See the [vclover merge](#) command for syntax

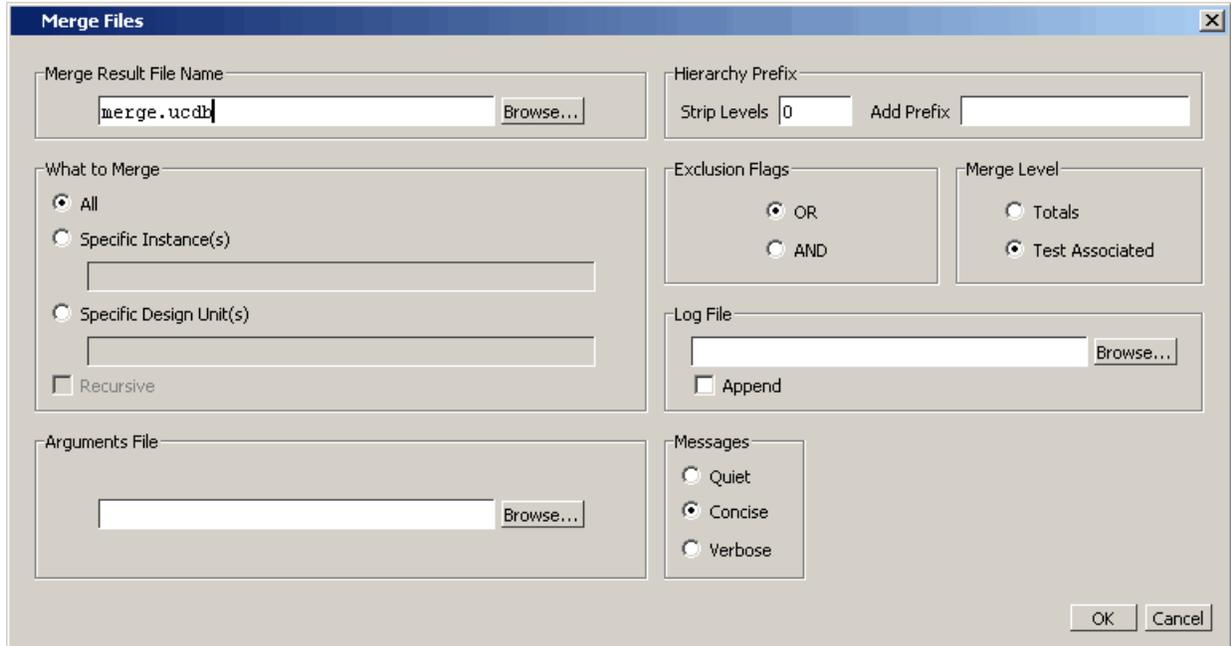
Merging within Verification Browser Window

You can merge multiple .ucdb files (including a verification plan) from the Verification Browser window as follows:

1. Select the .ucdb file(s) to merge.
2. Right-click over the file names and select Merge.

This displays the Merge Files Dialog Box, as shown in [Figure 25-4](#). The various options within the dialog box correspond to the arguments available with the `vcover merge` command.

Figure 25-4. File Merge Dialog



3. Fill in fields in the Merge Files dialog box, as required. A few of the more important, less intuitive fields are highlighted here. For full details related to these fields, see [“vcover merge”](#).
 - Set the Hierarchy Prefix: Strip Level / Add Prefix to add or remove levels of hierarchy from the specified instance or design unit.
 - For Exclusion Flags, select AND when you want to exclude statements in the output file *only* if they are excluded in all input files. When OR is selected (default) a statement is excluded in the output merge file if the statement is excluded in any of the input files.
 - Test Associated merge is the default Merge Level. Before selecting Totals as the Merge Level, be sure you understand the difference between them. If you do not, refer to [“Test-Associated Merge versus Totals Merge Algorithm”](#) and [“Limitations of Merge for Coverage Analysis”](#) for further information.
4. Select OK

This creates the merged file and loads the merged file into the Verification Browser window.

Validation

If the merge was successful, a transcript message such as the following appears:

```
# Merge in Process.....  
#  
# Merging file C:/QuestaSim_6.6/examples/ucdb/testplan/DataTest.ucdb  
# Merging file C:/QuestaSim_6.6/examples/ucdb/testplan/FifoTest.ucdb  
# Merging file C:/QuestaSim_6.6/examples/ucdb/testplan/IntialTest.ucdb  
# Merging file C:/QuestaSim_6.6/examples/ucdb/testplan/ModeTwoTest.ucdb  
# Writing merged result to merge.ucdb
```

Merging with the `vcover merge` Command

The merge command, `vcover merge`, allows you to merge multiple coverage data files offline, without first loading a design. The merge utility is a standard ModelSim utility that can be invoked from the command line. For example,

```
vcover merge output inputA.ucdb inputB.ucdb
```

merges coverage statistics in UCDB files `inputA.ucdb` and `inputB.ucdb` and writes them to a new UCDB file called `output`. See “`vcover merge`” for a complete list and description of available arguments.

Merging Using a Master UCDB

Often, a design changes over a period of time, and you may want to merge UCDB files generated earlier with newer UCDBs generated from the same design. Let’s say you made changes in the DUT or in testbench, wherein you change a regular covergroup bin to an `ignore_bin`, or add or delete various scopes and bins, etc.. You could merge those changes into the new UCDB using a “master” UCDB. The “master” UCDB is the file which reflects the latest state of your design and contains all the items you want to see. In essence, the master merge provides you with a mechanism by which to ignore anything which is not present in the master UCDB file, while merging that UCDB’s data with other regular UCDB files. The command “`vcover merge -master`” provides a method of filtering out stale data.

For example, you could enter the following command:

```
vcover merge out.ucdb in1.ucdb in2.ucdb -master master.ucdb
```

For specific syntax details, [vcover merge](#).

Specifically, the master merge does the following:

- Merges the content of the master UCDB, which means bin counts from the master UCDB are added up in the final merged file (`output.ucdb`).
- Merges any scope of a coveritem from the other input files, **if and only if** the corresponding scope or coveritem is found in the master database.

- Merges any attribute, tag, and comment from the other input files **if and only if** the corresponding item is found in the master database. However, this restriction does not apply to test data records and other data which are generated at run time. The following is a list of items which are merged from non-master UCDB files even when those items are not present in the master UCDB file:
 - Test data records
 - Memory statistics
 - Covergroup bin first hit timestamp data
 - Any count, like a branch scope's count coming in, etc.
 - Information on whether a covergroup is sampled or not

Warnings During Merge

Several types of warnings can occur during a merge operation. The following sections are intended to guide you in understanding of the meaning of and resolution for a few of the more common warnings.

Multiple Test Data Records with Same Name

ModelSim requires that test data records created within a merged UCDB are unique. In a situation such as the merging of UCDB files that exist in different directories, but which have test records whose names are identical, you can get a warning such as:

```
** Warning: (vcover-6854) Multiple test data records with the same name
encountered during the merge of file 'xyz.ucdb'
These test data records contain conflicting data....
```

To handle this situation and establish unique names for the test data records, you can:

- Add a unique test name for each run prior to saving the UCDB. You would do this by entering the following command for each test:
 - coverage attr -name TESTNAME -value test_1**
- Alternatively, you could open the already created UCDBs in Viewcov mode and assign different TESTNAME attributes for each, by entering the following commands at the command prompt:

```
coverage attribute -ucdb -name TESTNAME -value run_1
coverage save run_1.ucdb
```

```
coverage attribute -ucdb -name TESTNAME -value run_2
coverage save run_2.ucdb;
```

Merging and Source Code Mismatches

By default ModelSim performs checks on source code stability when performing operations like merging statement coverage, or other source-based data.

If one UCDB is generated with version X of file *t.v*, and then another UCDB is generated with version Y of file *t.v*, it doesn't make sense to merge the source-based coverage metrics in those two UCDBs. This kind of checking is known as design unit signature checking. When a merge such as this is attempted, warning #6820 is issued, stating that the merge of the instance in which those lines occur is being skipped.

One cause of legitimate difference between the design source in two different UCDBs is the use of ``ifdef` to conditionally define code. Another cause is a difference in the UCDB release version for certain specific VHDL designs. However legitimate the cause, these difference may not be substantive. In cases such as these, you might want to work around this check. To bypass the ModelSim DU signature check, use the `-ignoredusig` argument to the `vcover merge` command.

Caution



You should not use `-ignoredusig` lightly, without validating that the differences in source code are OK. Misuse of `-ignoredusig` can lead to very confusing coverage results, because code coverage results are merged based on line numbers, and merged results can be significantly wrong if line numbers have changed between versions of the source.

Related Topics

- [About the Merge Algorithm](#)
- [Merge Usage Scenarios](#)
- [Modifying UCDBs](#)
- [Ranking Coverage Test Data](#)
- [Rerunning Tests and Executing Commands](#)

Ranking Coverage Test Data

Ranking seeks to order with respect to their contribution to the coverage metric, such as Total Coverage. The ordering of the tests within the ranking report is from most (top) to least (bottom). All tests which do not contribute to increased coverage numbers are not included in the ranked results.

You can rank your tests by any number of criteria using either the **Verification Browser** > **Rank** menu selection, or the `vcover ranktest` command with various parameters and UCDB files as input. The ranking result files are based on the selected `.ucdb` files. You can rank normal UCDB files, as well as merged UCDB files.

Procedure

1. Select one or more .ucdb files.
2. Right-click and select **Rank**.

This displays the Rank Files Dialog Box. The various options within the dialog box correspond to arguments available with the -du and -path arguments to `vcover ranktest` command.

3. Fill in What to Rank, Rank By, and Stop Ranking When and Messages, as desired. (The selection of Rank By > Fewest means to rank the files by the fewest number of tests.)
4. Select Advanced Options to open the dialog box to set your coverage metrics, arguments file, and ranked results file names.
5. **OK**

This creates the .rank file and loads it into the Test Browser; it also outputs ranking data to the transcript window.

Validation

If the rank was successful, a transcript message such as the following appears:

```
#
#           Metric           Bins   Covered%   Inc%
#
# Cover Groups/Points       5/18     0.0000    0.0000
#   CoverDirectives         10     0.0000    0.0000
#     Statements           2605    47.0633   47.0633   *****
#       Branches           1978    29.6764   29.6764   *****
#         Expressions        711    18.2841   18.2841   ***
#           Conditions       1315    15.9696   15.9696   ***
#             ToggleNodes    2214     1.1743    1.1743
#               States        17     17.6471   17.6471   ***
#                 Transitions  45     6.6667    6.6667   *
#                   AssertPasses 12     0.0000    0.0000
```

Merged Results vs. Rank Report Results

In most cases, when you rank a merged UCDB, the coverage numbers for each test within the ranking report will not match the numbers within that same test, as they are listed in a merged .ucdb file. This is due to the fact that the coverage numbers listed in a merged UCDB are raw coverage numbers for that particular test. Whereas, in the ranked report, the numbers within each column for a particular test are a cumulative total of all the data in the columns from tests listed above it.

Another reason there can be a difference in these numbers is due to assertions. Assertions are not coverage numbers, and as such, they are not included in the ranked results.

Ranking Most Effective Tests

You can rank the most effective tests from the **Verification Tracker** as follows:

1. Select test plan or test plan section containing the items to be ranked.
2. Right-click and select **Test Analysis > Rank Most Effective Tests**.

This displays the Rank Selected Item dialog box where you can specify test plan section, criteria for ranking, and when to stop ranking, rank results filename, and so forth.

The various options within the dialog box correspond to the arguments available with `vcover ranktest -plansection`.

Test-Associated vs. Iterative Ranking

Test-associated ranking performs a merge and proceeds to rank based upon a test-associated merged result held in memory, whereas iterative ranking ranks each individual test by performing an iteration of merges on the file system.

The test-associated ranking method (default) is superior to the iterative method in two important ways:

- significantly better performance
- ranking is performed with respect to the entire coverage space

However, there are some cases where the test-associated ranking does not always function intuitively: Because it only records what test covered a particular bin, the test-associated algorithm may underestimate coverage for test subsets where “at_least” values are greater than 1. This is the same limitation as explained in “[Limitations of Merge for Coverage Analysis](#)”.

The iterative ranking option is available to work around cases of covergroups and cover directives where `at_least > 1`, however it is considerably less efficient (slower). Iterative ranking ranks each individual test by performing an iteration of merges on the file system.

Related Topics

- [About the Merge Algorithm](#)
- [Test-Associated Merge versus Totals Merge Algorithm](#)
- [Merge Usage Scenarios](#)
- [Modifying UCDBs](#)

Modifying UCDBs

You can edit the contents of a UCDB using the `coverage edit` command. Specifically, you can:

- remove coverage data from UCDB

As a consequence, it is not possible to make a later change to the coveritem goal recorded in the database (i.e. after it's been saved) — for example by modifying the UCDB using the [coverage edit](#) command — and automatically infer a different timestamp. You would have to rerun simulation for the new timestamp to appear.

A similar confusion can arise with timestamps during merges. When coveritems with multiple timestamps are merged, the earliest timestamp is retained, even if the merged coveritems have different `at_least` (goal) values.

Related Topics

- [vsim -cvgbintstamp](#)
- [Covergroup Bin Reporting and Timestamps](#)
- [coverage goal](#)
-

About the Merge Algorithm

- When you perform a merge on multiple UCDBs, all test data records with unique testnames are concatenated into the merged UCDB file. If you merge two tests that have identical names but different data contents, a warning is issued.
- A merged file contains one test data record for each of the different tests that were merged into the file. For example, if you merged three UCDBs saved from simulation (`vsim`), you get three test attribute records. If you merge that file with another one saved from `vsim`, you get $3 + 1 = 4$ test data records, and so on. See “[Understanding Stored Test Data in the UCDB](#)” for more information on the content of test data records.
- The merge algorithm that ModelSim uses is a union merge. Line numbers from the files are merged together, so that if different UCDBs have different sets of coverage source lines, the resulting merged database contains a union of the set of source lines in the inputs.
- Toggles and FSMs, which have no source information, are merged as follows:
 - Toggles are merged with a union operation: objects with the same name are always merged together, regardless of how many are present in each UCDB input file.
 - FSMs are merged together by the order in which they appear in a given module instance or design unit.
- Assertion and functional coverage objects in the UCDB are always merged as a union algorithm: objects of the same name are merged together, but the result contains the union set of differently named objects from all inputs. (This has implications if class specializations are used; see “[Covergroup Naming Conventions](#)” for more information.) Covergroup instances — those for which `option.per_instance = 1` — are merged together based on the “`option.name`” string for each instance. Instances with the same

name are considered the same, and bins are merged together as a union, regardless of parameterization.

There are some exceptions for other kinds of data (besides coverage counts):

- Coverage “at_least” values are taken as the maximum of all inputs.
- Covergroup “goal” and “auto_bin_max” options are taken as the maximum of all inputs.
- Other covergroup options are taken as “first one wins” — that is, values from the first input file are taken.
- Exclusions flags are configurable: with the `-and` switch to `vcover merge`, they are ANDed together; otherwise ORed together.
- User-defined attributes is a union (attributes with the same name are “first one wins” — that is, value in the first input file survives).
- Assertion and cover directive limits are taken as the maximum of all inputs.
- User-defined flag values are ORed together.

Test-Associated Merge versus Totals Merge Algorithm

You can choose one of two levels of information that is preserved in the merged database by using either the `-testassociated` (default), or the `-totals` argument to the `vcover merge` command. The default merge provides the tool with the level of data required for `coverage analyze`, which you can use to analyze your results on a per-test basis. The merge options are:

- **-totals** merge — Creates a basic merge: sums the coverage.

Merges together the coverage scopes, design scopes, and test plan scopes. The counts are incremented together (ORed together in the case of vector bin counts). The final merge is a union of objects from the input files. Information about which test contributed what coverage into the merge is lost. Information about tests themselves are not lost — test data records are added together from all merge inputs. While the list of tests can be known, it cannot be known what tests might have incremented particular bins.

- **-testassociated** merge — This is the default merge. Includes all data in totals merge and additionally marks covered bins with the test that covered them.

Includes the basic information obtained with `-totals` as well as the associated tests and bins. When tests and bins are associated, each coverage count is marked with the test that caused it to be covered.

- For functional coverage, this means that the bin count should be greater than or equal to the `at_least` parameter.

- For code coverage and assertion data, any non-zero count for a test causes the bin to be marked with the test.

While the test-associated merge can not tell you which test incremented a bin by exactly how much, it can tell you which test caused a bin to be covered.

Limitations of Merge for Coverage Analysis

Because a test-associated merge does not perfectly preserve information, it can be misleading in some circumstances. These circumstances are detected during the merge, and a warning is issued as follows:

```
Information has not been perfectly preserved during the merge of
file 'test.ucdb'.
If you use 'coverage analyze -test', test filtering in the Tracker
GUI, or test ranking, results may be inaccurate based on this merge.
For more information issue the command 'verror 6846'.
For more details, rerun merge with the '-verbose' option set.
```

You only need to be concerned about this warning if you are using any of the following verification management features:

- “coverage analyze -test” at the command line
- “coverage analyze -coverage” at the command line
- Test plan analysis in the GUI, using either:
 - **Verification Tracker > Filter > Setup** (or **Apply**)
 - **Verification Tracker > Test Analysis** menu items
- The difference between test-associated and totals merge may also be significant if you do test ranking in its “test-associated merging” ranking method (which is the default; the alternative is “iterative merging”).)

If you do not plan to use any of the above mentioned features, you can safely ignore the “verror 6846” error message. However, some features that rely on a certain level of preservation — test ranking and Coverage View (vsim -viewcov) mode CLI features, for example — issue the warning if a test-associated merge file is used in any of the following cases:

- For functional coverage: Coverage thresholds (at_least values) are different in separate merge input files. See [Example 25-2](#).

In these cases, it would be possible for a bin to be covered after the merge but in none of the inputs. In this case, test-associated analysis will be correct with respect to the individual tests but incorrect regarding merged coverage; ranking in particular will be inaccurate because of the discrepancy in merged coverage.

- Weights (for example, covergroup weights) are different in separate merge input files.

In these cases, because coverage (for example, covergroup coverage) can depend on weighting, it will be impossible to recreate the original coverage of some of the input files. During the merge, the maximum weight is chosen; conflicting weights are not preserved.

- Differing sets of coverage objects in merge input files. This most commonly occurs due to parameterization. See [Example 25-3](#).

For these cases, your best option may be to preserve the original UCDBs and analyze or rank them individually

i **Tip: Important:** In the case of different sets of coverage objects in different merge input files — test ranking is actually more accurate with the test-associated merge, because ranking should reasonably be done with respect to the union of all coverage.

Example 25-2. Coverage Threshold Difference

For example, suppose `at_least == 3` and you have 2 testcases each with counts of 2 in a cover directive.

The following command results in 100% coverage:

```
coverage analyze -total -path /path/to/cover
```

This is due to the fact that the merged result is $2 + 2 = 4 > 3$.

The following command results in 0% coverage, as it should:

```
coverage analyze -total -path /path/to/cover -test test1
```

However, the following command results in an incorrect result of 0% coverage:

```
coverage analyze -total -path /path/to/cover -test test1 test2
```

The result for “-test test1 test2” (generating test-associated merge data) is not the same as without -test (a totals merged data), because the test-associated database is missing the covercount information that is contained in the totals database.

Example 25-3. Coverage Object Differences with Parameters

Here's a typical example of code which, when merged with the default (test-associated) merge, provokes error 6846. It contains a parameterized array:

```
module top;
  parameter int size = 2;
  bottom #(size) inst();
endmodule
module bottom;
  parameter int size = 2;
  reg[size-1:0] tog;
```

```

if (size==2) begin
  initial begin
    #1 tog[0] = 0;
    #1 tog[0] = 1;
    #1 tog[0] = 0;
  end
end else begin
  initial begin
    #1 tog[1] = 0;
    #1 tog[1] = 1;
  end
end
endmodule

```

Imagine you compile this for toggle coverage, creating two different UCDB files with two different array sizes, then merge them together, like so:

```

vlog +cover=t test.sv
vsim top -novopt -coverage -c -G/top/size=2 -do "run -all; coverage save test2.ucdb; quit"
vsim top -novopt -coverage -c -G/top/size=3 -do "run -all; coverage save test3.ucdb; quit"
vcover merge test.ucdb test2.ucdb test3.ucdb

```

This provokes warning 6846. What is the potential problem? Look at the results of these two different summary reports, the first issued during simulation on the active database, and the second during post-processing on a saved UCDB:

```

> vcover stats test2.ucdb

SUMMARY FOR FILE "test2.ucdb":
Coverage Summary BY INSTANCES: Number of Instances 2
  Enabled Coverage      Active      Hits      Percent
  -----
Toggle Nodes           2          1       50.0

> vsim -viewcov test.ucdb -c -do "coverage analyze -test test2 -summary;
quit"

# Hierarchical Summary Report For Design Instances
# Filtered by Tests: test2
# SUMMARY FOR SCOPE "/top":
# Coverage Summary BY INSTANCES: Number of Instances 2
#   Enabled Coverage      Active      Hits      Percent
#   -----
#   Toggle Nodes           3          1       33.33

```

The difference between these two summary reports is that the “test-associated” merge loses some data: in particular, it loses knowledge of what coverage objects were in what file. The knowledge of what was covered is accurate (in this case), namely “tog[0]”, but as the merged result has 3 toggles, that is used as the denominator of the coverage fraction.

Related Topics

- [Ranking Coverage Test Data](#)
- [Merge Usage Scenarios](#)

Merge Usage Scenarios

The decision on how to merge a set of UCDB files depends upon where and how the data being merged is stored in the databases. As a way of understanding your options, consider three basic merge scenarios, as follows:

Scenario 1: Two UCDBs, same scope

You have data from two or more UCDB files, at the same level of hierarchy (scope) in the design. Example commands:

```
vcover merge output.ucdb file1.ucdb file2.ucdb
```

Scenario 2: Two UCDBs, different scopes

You have data from two or more UCDB files, at different levels of hierarchy. For example: */top/des* instance in *filea.ucdb*, and *top/i/des* instance in *fileb.ucdb*.

- Option 1: Strip top levels of hierarchy from both and then merge the stripped files. Example commands:

```
vcover merge -strip 1 filea_stripped.ucdb filea.ucdb  
vcover merge -strip 2 fileb_stripped.ucdb fileb.ucdb  
vcover merge output.ucdb filea_stripped.ucdb fileb_stripped.ucdb
```

- Option 2: Strip levels off instance in one UCDB file, and install to match the hierarchy in the other. In this example, strip */top/* off the */top/des* and then add the */top/i* hierarchy to it. Example commands:

```
vcover merge -strip 1 filea_stripped.ucdb filea.ucdb  
vcover merge -install /top/i filea_installed.ucdb filea_stripped.ucdb  
vcover merge output.ucdb filea_installed.ucdb fileb.ucdb
```

Scenario 3: Single UCDB, two sets of data

You have two sets of data from a *single* UCDB file, at different levels of hierarchy. Because they are instantiated at different levels within the same file, the tool cannot merge both of them into the same database. In this scenario, it is best to merge by design unit type using the [vcover merge -du](#) command.

For example, */top/designinst1* and */top/other/designinst2* are two separate instantiations of the same design unit within a single UCDB file. An example command for merging all instances in *file3.ucdb* would be:

- for Verilog with a module name of *design*
`vcover merge -du design -recursive output.ucdb file3.ucdb`
- for VHDL with an entity name of *design* and an architecture name of *arch1* would be
`vcover merge -du design(arch1) -recursive output.ucdb file3.ucdb`

Scenario 4: Concurrent merge jobs

You can have concurrent merge jobs running on different machines which are simultaneously writing to the same target merge file. A lock file is created which prevents any conflicts. The utility can recover from crashing merges and crashing hosts. It also allows a configurable time-out of merges, as well as backups of the previous output. See the [vcover merge](#) command for syntax details.

Use the `vcover merge` command as follows:

```
vcover merge out.ucdb out.ucdb in.ucdb
```

Note



If this is the very first merge, the input file “out.ucdb” will not exist yet, so the simulator issues a warning. In this case, specify the appropriate <input>.ucdb file.

This command takes the output UCDB and merges it with a second input UCDB.

```
vcover merge out.ucdb out.ucdb in2.ucdb -timeout 10 -backup
```

Then, another machine can take the output of the first merge command and third input UCDB, and so on.

Related Topics

- [Merging Coverage Test Data](#)
- [About the Merge Algorithm](#)

Viewing and Analyzing Verification Data

Storing User Attributes in the UCDB	1120
Viewing Test Data in the Browser Window	1120
Generating Coverage Reports	1122
Filtering Data in the UCDB	1129
Filtering Results by User Attributes	1131
Analysis for Late-stage ECO Changes	1133
Retrieving Test Attribute Record Content	1133

Storing User Attributes in the UCDB

You can add your own attributes to a specified test record using [coverage attribute](#), with a command such as:

```
coverage attribute -test testname -name Responsible -value Joe
```

This command adds the “Responsible” attribute to the list of attributes and values displayed when you create a coverage report on *testname.UCDB*. This shows up as a column when the UCDB is viewed in the Tracker pane.

Viewing Test Data in the GUI

Data related to the management of your verification data can be viewed in the Verification Browser window (**Layout > VMgmt** or **View > Verification Management > Browser**). See [“Viewing Test Data in the Browser Window”](#).

Viewing Test Data in the Browser Window

You can view both UCDB (.ucdb) and rank result (.rank) files in the Verification Browser window.

Prerequisites

- You must be located in a directory containing .ucdb or .rank files.

Procedure

1. Open the Verification Management window:
View > Verification Management > Browser
2. Add files to the Browser using one of the following three methods:
 - Right-click in the window and select **Add File**. Select desired .ucdb files from the list that appears in the **Add File(s)** dialog box.
 - When the window is active, select **Verification Browser > Add File** from the menu bar of the Main window. Select desired .ucdb files from the list that appears in the **Add File(s)** dialog box.
 - At the vsim command prompt, execute the **add testbrowser** command, which accepts UCDB and rank result files as arguments. For example,
add testbrowser test.ucdb

The Verification Browser window appears, similar to [Figure 25-6](#).

Figure 25-6. Test Data in Verification Browser Window

FileName	TestName	TotalCoverage	Statements	Branches	Expressions	Conditions	ToggleNodes	States
U CPURegisterTest.ucdb	CPURegister...	51.77	78.81	65.67	69.90	34.36	47.32	100.
U DataTest.ucdb	DataTest	39.77	70.33	58.24	59.22	30.84	37.36	52.
U FifoTest.ucdb	FifoTest	48.77	73.81	64.67	67.12	34.14	45.41	76.
U IntialTest.ucdb	IntialTest	46.01	72.59	64.12	64.21	34.00	42.19	76.
U ModeTwoTest.ucdb	ModeTwoTe...	47.94	73.05	64.57	65.60	34.79	41.47	76.
+ M results.ucdb	-	74.26	94.84	90.85	77.53	84.64	73.45	100.
U TxDataTest.ucdb	TxDataTest	48.13	73.05	64.57	66.43	34.79	42.45	76.
U VariableTest.ucdb	VariableTest	46.12	72.59	64.12	64.63	34.00	43.04	76.

The coverage numbers in the Browser window are based on the Total Coverage calculations described in “[Calculation of Total Coverage](#)”, however all design roots are taken into account and include all hierarchy underneath all design roots. See “[Coverage Calculation in the Browser Window](#)”.

Deleting UCDB Files from the Browser Window

To remove UCDB files from Browser, highlight the test(s) you want to delete from view and select or <Ctrl-Del> (while the Verification Browser window is active). A popup appears for you to confirm if you want to delete the selected test(s) from the Verification Browser window.

Invoking Coverage View Mode

UCDB files from previously saved simulations are only viewable in Coverage View mode (post-processing). You can invoke Coverage View Mode on any of your .ucdb files in the Test Browser or at the command line. This allows you to view saved and/or merged coverage results from earlier simulations.

Procedure

- GUI:
 - a. Right-click to select .ucdb file. This functionality does not work on .rank files.
 - b. Select **Invoke CoverageView Mode**.

The tool then opens the selected .ucdb file and reformats the Main window into the coverage layout. A new dataset is created.

- Command Line:

Enter `vsim` with the `-viewcov <ucdb_filename>` argument. Multiple `-viewcov` arguments are allowed. For example, the Coverage View mode is invoked with:

```
vsim -viewcov myresult.ucdb
```

where *myresult.ucdb* is the coverage data saved in the UCDB format. The design hierarchy and coverage data is imported from a UCDB.

Related Topics

- [Coverage View Mode and the UCDB](#)

Customizing the Column Views

You can customize the display of columns in the Verification Browser or Tracker windows, and then save these views for later use.

Procedure

1. Select [**Create/Edit/Remove ColumnLayout...**] from the pull down list.
This displays the Create/Edit/Remove Column Layout dialog box.
2. Enter a new name in the Layout Name text entry box.
3. Click **OK**.

You can also add or modify pre-defined column arrangement from the Create/Edit/Remove Column Layout dialog box by adding columns to or removing them from the Visible Columns box as desired.

After applying your selections, the rearranged columns and custom layouts are saved and appear when you next open that column view in the Verification Browser or Tracker windows.

Related Topics

- [Test Attribute Records in the UCDB](#)

Generating Coverage Reports

You can use the GUI or [coverage report](#) command to create three types of reports to display the coverage metrics in your design:

- ASCII Text Reports (see “[Generating ASCII Text Reports](#)”)
- HTML Reports (see “[Generating HTML Coverage Reports](#)”)
- Exclusion Reports (see “[Generating Coverage Exclusion Reports](#)”)

Prerequisites

- Run a simulation with the various coverage types enabled to collect coverage metrics.

- If opening from the Browser window, the UCDB must be opened in Coverage View mode by right clicking on the UCDB and selecting **Invoking CoverageView Mode**.

You can access any of these Coverage Report dialogs by:

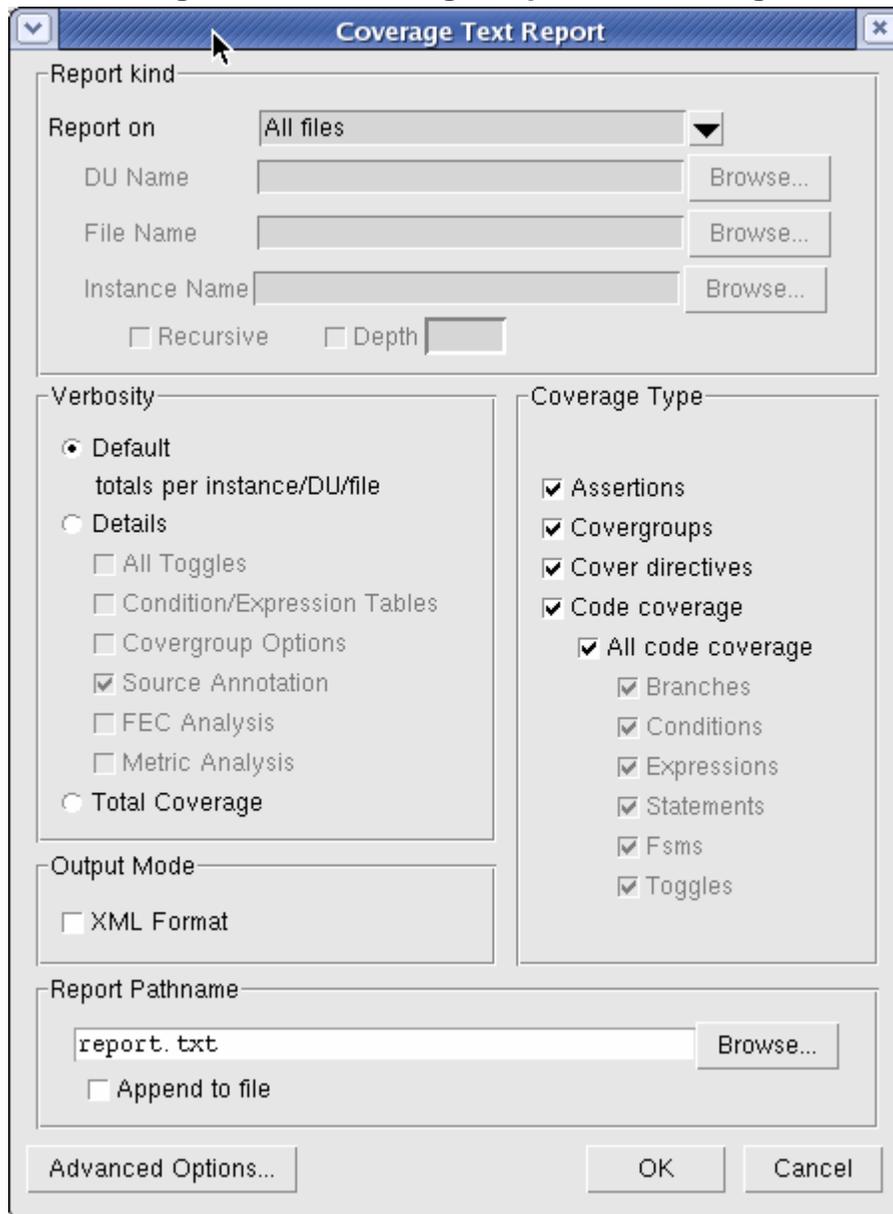
- Right-clicking on any object in the Files or Structure (sim) windows and selecting **Code Coverage > Code Coverage Reports**; or, select **Tools > Coverage Report > Text** or **HTML** or **Exclusions**.
- Selecting the UCDB in the Browser window and Select **Tools > Coverage Report > Text** or **HTML** or **Exclusions**.

These actions display the Coverage Text Report, Coverage HTML Report, or Coverage Exclusions Report dialog boxes.

Generating ASCII Text Reports

1. Access the Coverage Text Report dialog as described in “[Generating Coverage Reports](#)”.

Figure 25-7. Coverage Report Text Dialog



2. From the **Report on** dropdown, select one of the following:
 - All files — reports data for all design units defined in each file. (-byfile switch with coverage report).
 - All instances — reports data in each instance, merged together. (-byinst with coverage report).
 - All design unit — reports data in all instances of each design unit, merged together. (-bydu with coverage report).
3. In the Coverage Type pane, ensure that the desired coverage types are selected.

4. Alter any of the other options as needed. All options in this dialog correspond to [coverage report](#) and [vcover report](#) options.
5. Click **OK** to create coverage report.

Results

- Writes the report (*report.txt*) to the current working directory.
- Opens a notepad window containing the *report.txt* file.

Related Topics

[FSM Coverage](#)
[Code Coverage](#)
[Verification with Functional Coverage](#)
[Generating Coverage Reports](#)
[Coverage Reports](#)

Generating HTML Coverage Reports

You can create on-screen, static coverage reports in HTML that are easy to use and understand.

Requirements and Recommendations

- For best results, the report should be viewed with a browser that supports the following:
 - JavaScript — Without this support, your browser will work, but the report is not as aesthetically pleasing.
 - cookies — For convenience of viewing coverage items in the HTML pages, you should enable cookies.
 - frames and Cascading Stylesheets (CSS) — Though support is recommended for frames, reports can still be displayed on browsers without this support. The report writer uses CSS to control the presentation, and will be best viewed with browsers that support frames and CSS.

Procedure

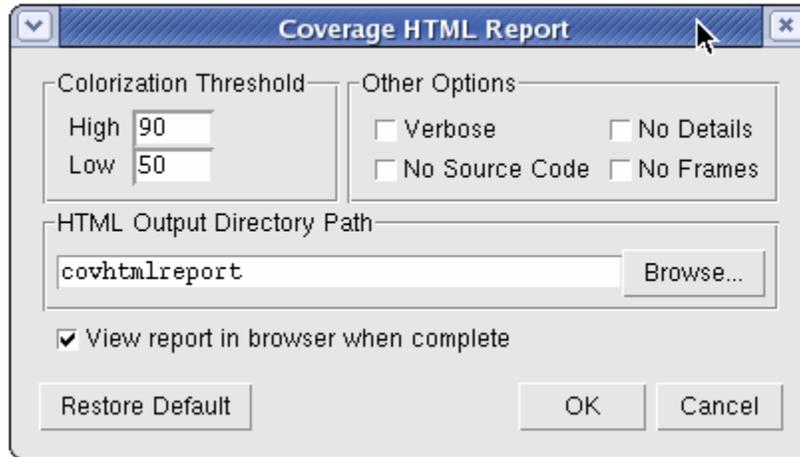
From the command line, use the `-html` switch with the [coverage report](#) or [vcover report](#) command.

To generate an HTML report from the GUI:

1. Select a single UCDB file from the Browser.
2. From the main menu, select `Browser > HTML Report` or in the Browser, select `Right click > HTML Report`.

This brings up the Coverage HTML Report dialog box that allows you to control the generation and subsequent viewing of the report.

Figure 25-8. Coverage HTML Report Dialog



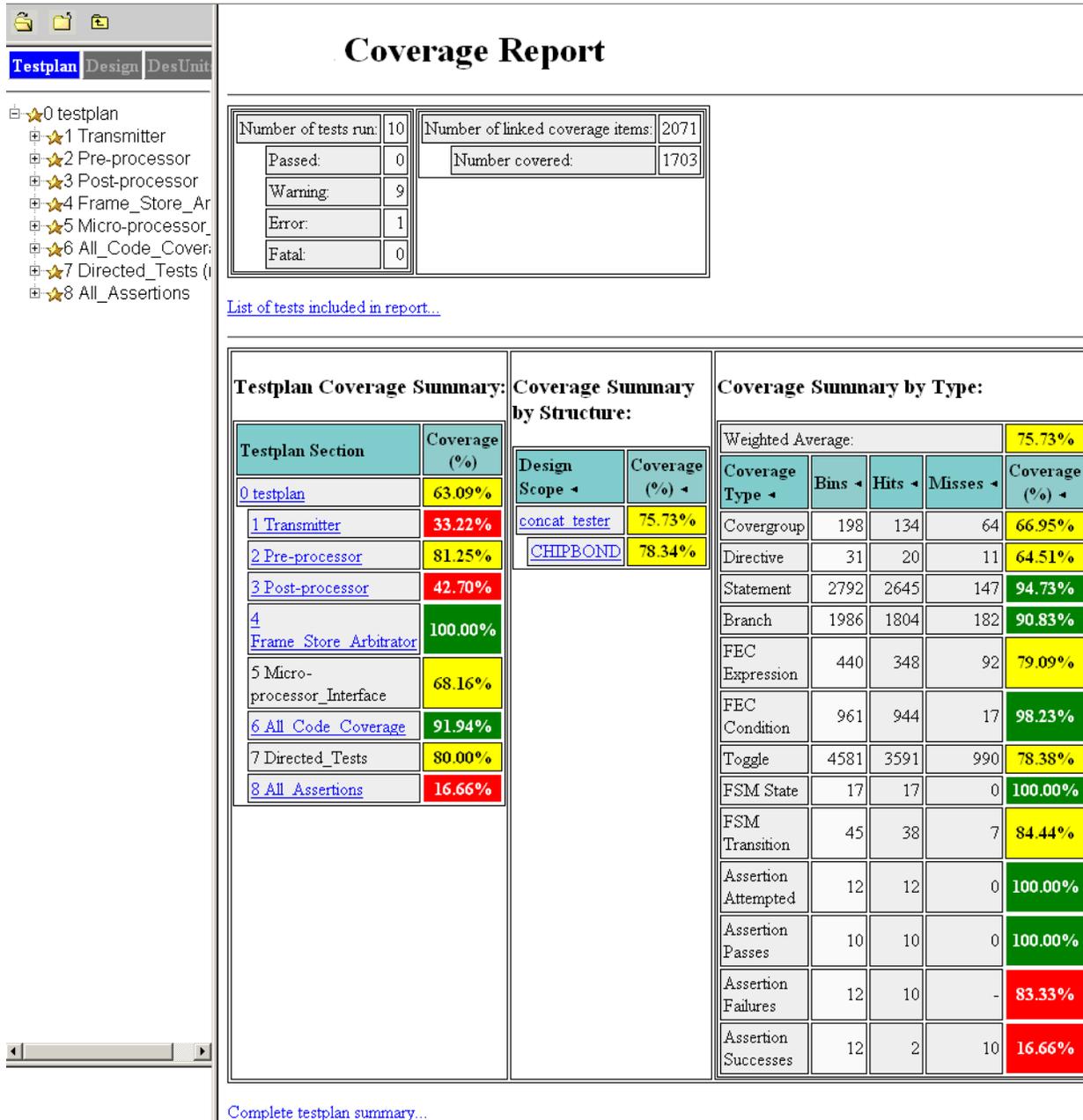
3. Set the color for both the high and low threshold to define the number above which the coverage number displays in green, and below which it is displayed in red.
4. Select No Frames to save on report generation time and disk space for larger designs (see [HTML Generation for Large Designs](#)).

Results

- Writes the report (*index.html*) to the specified directory (default is */covhtmlreport*).

The HTML file is viewable with any reasonably modern web browser, an example which is shown in [Figure 25-9](#).

Figure 25-9. HTML Coverage Report



The web browser allows you to explore the hierarchy of the design, much like you might browse a file system. Colorized copies of the design source code are generated and linked into the report at the appropriate places.

The coverage statistics displayed in the **Coverage Summary by Structure** section on the HTML report are calculated in accordance with the algorithms shown in “[Calculation of Total Coverage](#)”, and the **Coverage Summary by Type** statistics are calculated using the algorithms and weightings as described in [Table 25-1](#).

Compared to other types of coverage reports, HTML report generation can cause machines to be particularly sensitive to issues of disk space, memory usage and slowness. Many of the HTML reporting options, both through the radio buttons in the Coverage HTML Report dialog box and the coverage report -html options, are geared toward improving the speed and performance of report generation. Additionally, you may want to target the reports by excluding specific coverage types and/or reducing the scope of items in the report. See [coverage report -html](#) for full details on these options.

Canonical Toggle Nodes in HTML Reports

Some toggle nodes in the report appear with a notation of “[canonical]”. This denotes that the togglenode is an alias of a canonical node elsewhere in the design. Canonical means a common name for the overall net. All aliases point to the canonical name, which is equivalent to all aliases.

option or type_option Values

For all covergroups, coverpoints and crosses, the value of option/type_option is displayed in the HTML report whenever it differs from the default value specified in the SystemVerilog LRM.

HTML Generation for Large Designs

The No Frames radio button in the GUI corresponds to the -noframes switch used with the coverage report -html command. Selecting this functionality disables generation of JavaScript-based tree which has known performance problems for designs with a large number of design scopes. With this option, the report comes up as a single frame containing the top-level summary page and an HTML-only design scope index page is available as a link from the top-level page.

Additionally, the No Details option omits coverage detail pages. This can save report generation time and disk space for HTML generation for very large designs.

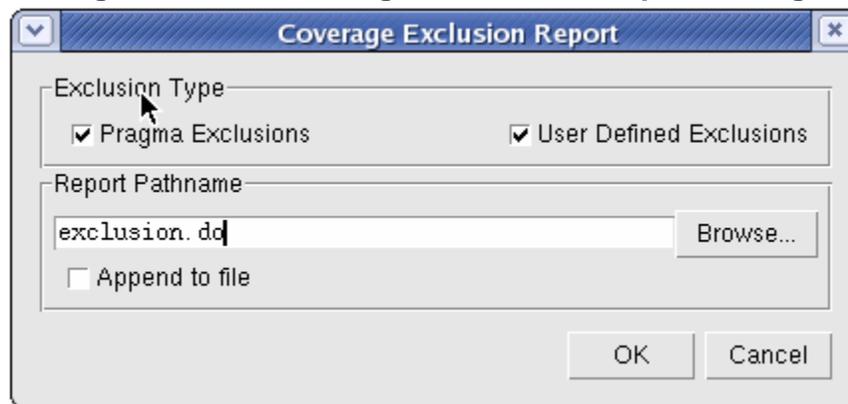
Related Topics

[Code Coverage](#)
[Generating Coverage Reports](#)
[Coverage Reports](#)

Generating Coverage Exclusion Reports

1. Access the Coverage Exclusion Report dialog as described in “[Generating Coverage Reports](#)”.

Figure 25-10. Coverage Exclusions Report Dialog



2. Select Pragma and/or User Defined Exclusions to report.
3. Save the pathname.
4. Click **OK**.

Related Topics

[Code Coverage](#)
[Coverage Exclusions](#)
[Generating Coverage Reports](#)
[Coverage Reports](#)

Filtering Data in the UCDB

You can use a filter to display only the desired coverage information in various Verification Management and coverage windows:

- Filter UCDB data in the Verification Management Browser window
- Filter UCDB data in the Verification Management Tracker window
- Filter covergroup related coverage data from Covergroup window
- Filter assertion related data from Assertion window
- Filter cover directive data from Cover Directive window

For details on filtering assertion, cover directive, or covergroup data, see “[Filtering Functional Coverage Data](#)”.

The filter operation is a “selection” filter. In other words, you are selecting criteria used for the inclusion of the specified information, filtering out everything else.

Setting up or Modifying a Filter for UCDB Data

To display only those items you wish to view in a UCDB:

1. From the context sensitive window menu, select **Filter > Setup**.

This opens the Filter Setup dialog box.

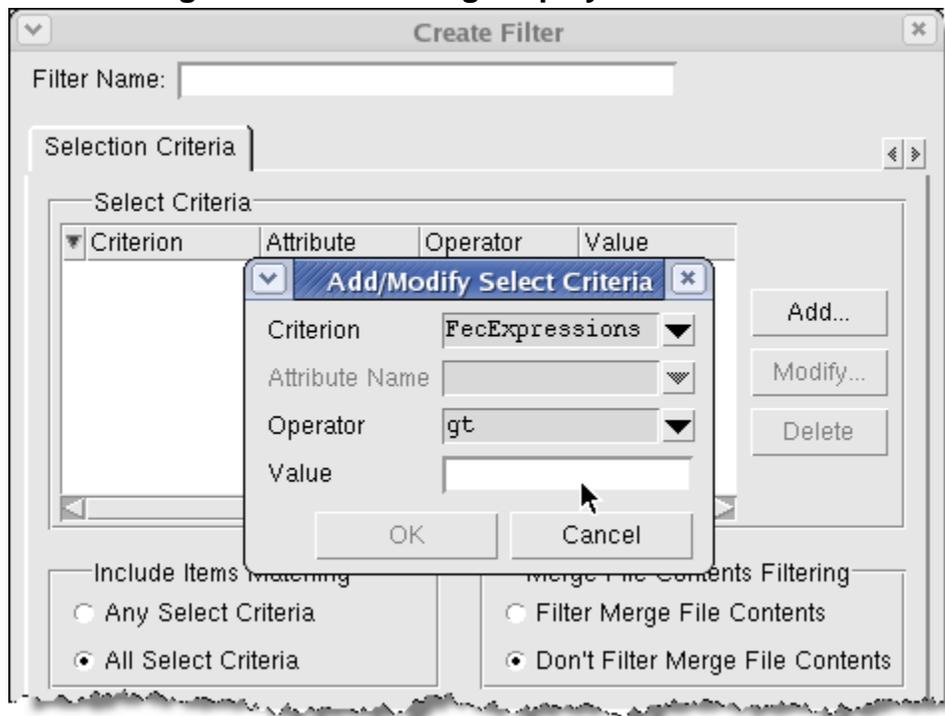
2. Select Create to create a new filter.

This opens the Create Filter dialog box.

3. Select Add to specify criteria.

This opens the Add/Modify/Select Criteria dialog box.

Figure 25-11. Filtering Displayed UCDB Data



4. Select Criterion and choose the type of coverage you wish to use as a filter.
 - a. Select Operator. See [coverage analyze](#) -select for definitions.
 - b. Enter Value of item to match.
 - c. Click **OK**.

The criterion you just entered appears in the Select Criteria list.

5. Enter a Filter Name and select OK to save that filter.
6. Either select **Apply** to filter the UCDB data, or select Done to exit the dialog box.

Applying a Filter

- From the Filter Setup dialog, select the desired Filter from the list and select **Apply**.
- From the Verification Management Browser:
 - a. Right click on UCDB(s) to filter.
 - b. Click on **Filter>Apply**, and then select a filter from the list.

UCDBs with matching criteria are included in the data now displayed in the Browser.

Filtering Results by User Attributes

One powerful feature for tracing verification requirements is the ability to filter your coverage results by attributes, which can be added to a test plan for that purpose. For example, if your original test plan included such data fields as the engineer responsible for tests, or the priority level assigned a specific section of the plan, you can add these as attributes to the imported and merged test plan UCDB and use them for filtering the coverage results.

Prerequisites

In order to successfully filter by user attributes:

- The user attribute used to filter the data must already exist in the original plan, or you must add the user attribute to the original plan before importing it (see “[Storing User Attributes in the UCDB](#)”).
- Plan must be imported.
- Plan must be merged with test results.

Procedure

To filter items for display on a specific column in the UCDB verification plan:

1. Select all tests to which you wish to apply selection criteria.
2. Right-click in the Tracker or Browser window and select **Filter > Setup**.

This opens the Filter Setup dialog box.

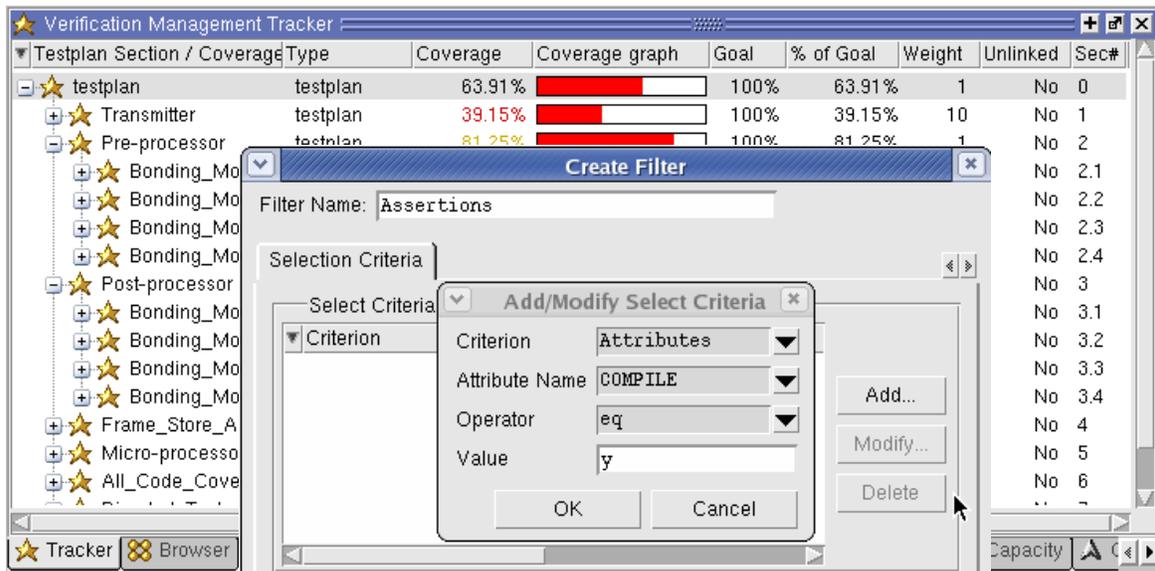
3. Select **Create**.

This opens the Create Filter dialog box to the Selection Criteria tab.

4. Select **Add**.

This opens the Add/Modify/Select Criteria dialog box.

Figure 25-12. Filtering on User Attributes



5. For **Criterion**, select **Attributes** from the pull-down list. Without selecting Attributes, the Attribute Name field is grayed out.
 - a. For **Attribute Name**, select desired attribute name(s) from pull-down list. The list contains all pre-defined attributes and any user attributes you added.
 - b. For **Operator**, select one of the options. See [coverage analyze](#) -select for definitions.
 - c. Enter Value of item to match.
 - d. Click **OK**.

The criterion you just entered appears in the Selection Criteria list.

6. Select the **Specify Tests** tab to select specific tests. By default, all tests are subject to the filter.
7. Enter a **Filter Name** and select **OK** to save the filter with the specified selection criteria. The filter you just created appears in the Filters list within the Filter Setup dialog box.
8. Either select **Apply** to filter the UCDB data, or select **Done** to exit the dialog box.

Applying a Filter

- From the Filter Setup dialog, select the desired Filter from the list and select **Apply**.
- From the Verification Management Browser:
 - a. Right click on UCDB(s) to filter.
 - b. Click on **Filter>Apply**, and then select a filter from the list.

UCDBs with matching criteria are included in the data now displayed in the Browser.

Related Topics

[Understanding Stored Test Data in the UCDB](#) `coverage analyze` Command

[Storing User Attributes in the UCDB](#)

Retrieving Test Attribute Record Content

Two commands can be used to retrieve the content of test attributes: [coverage attribute](#) and [vcover attribute](#), depending on which of the simulation modes you are in.

To retrieve test attribute record contents from:

- the simulation database during simulation (vsim), use [coverage attribute](#). For example:
coverage attribute
- a UCDB file during simulation, use [vcover attribute](#). For example:
vcover attribute <file>.ucdb
- a UCDB file loaded with -viewcov, use [coverage attribute](#). For example:
coverage attribute -test <testname>

The Verification Browser and Tracker windows display columns which correspond to the individual test data record contents, including name/value pairs created by the user. The pre-defined attributes that appear as columns are listed in [Table 25-3](#).

See “[Customizing the Column Views](#)” for more information on customizing the column view.

Analysis for Late-stage ECO Changes

Often ECOs (Engineering Change Orders) can occur late in the design cycle, when a design is highly stable. Only small sections of the design are affected by changes. You can use various Verification Management tools to analyze which tests most effectively cover those few areas and re-run those specific tests to demonstrate satisfactory coverage numbers.

- Rank tests (see “[Ranking Coverage Test Data](#)”).
- Re-run tests (see “[Rerunning Tests and Executing Commands](#)”).

C Debug allows you to interactively debug FLI/PLI/VPI/DPI/SystemC/C/C++ source code with the open-source gdb debugger. Even though C Debug does not provide access to all gdb features, you may wish to read gdb documentation for additional information. For debugging memory errors in C source files, refer to the application note titled *Using the Valgrind Tool with ModelSim*, available via SupportNet.

Note



To use C Debug, you must have a cdebug license feature in your ModelSim license file. Refer to the section "[License Feature Names](#)" in the Installation and Licensing Guide for more information or contact your Mentor Graphics sales representative.



Tip: Before you use C Debug, please note the following qualifications and requirements:

- C Debug is an interface to the open-source gdb debugger. ModelSim contains no customized gdb source code, and C Debug does not remove any limitations or problems of gdb.
- You should have some experience and competence with C or C++ coding, and C debugging in general.
- Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.
- Generally, you should not have an existing `.gdbinit` file. If you do, make certain you have not done any of the following within it: defined your own commands or renamed existing commands; used 'set annotate...', 'set height...', 'set width...', or 'set print...'; set breakpoints or watchpoints.
- To use C Debug on Windows platforms, you must compile your source code with gcc/g++. Refer to [Running C Debug on Windows Platforms](#), below.

Supported Platforms and gdb Versions

ModelSim ships with the gdb 6.3 or 6.6 debugger. Testing has shown these versions to be the most reliable for SystemC applications. However, for FLI/PLI/VPI applications, you can also use a current installation of gdb if you prefer.

For gcc versions 4.0 and above, gdb version 6.1 (or later) is required.

C Debug has been tested on these platforms with these versions of gdb:

Table 26-1. Supported Platforms and gdb Versions

Platform	Required gdb version
32-bit Redhat Linux 7.2 or later ¹	<code>/usr/bin/gdb</code> 5.2 or later
32-bit Windows 2000 and XP	<code>gdb</code> 6.3 from MinGW-32
Opteron / SuSE Linux 9.0 or Redhat EWS 3.0 (32-bit mode only) ¹	<code>gdb</code> 6.0 or later
x86 / Redhat Linux 6.0 to 7.1 ¹	<code>/usr/bin/gdb</code> 5.2 or later
Opteron & Athlon 64 / Redhat EWS 3.0	<code>gdb</code> 5.3.92 or 6.1.1

1. ModelSim ships gdb 6.6 for Linux platforms.

To invoke C Debug, you must have the following:

- A `cdebug` license feature.
- The correct gdb debugger version for your platform.

Running C Debug on Windows Platforms

To use C Debug on Windows, you must compile your C/C++ source code using the gcc/g++ compiler installed separately from ModelSim. Source compiled with Microsoft Visual C++ is not debuggable using C Debug.

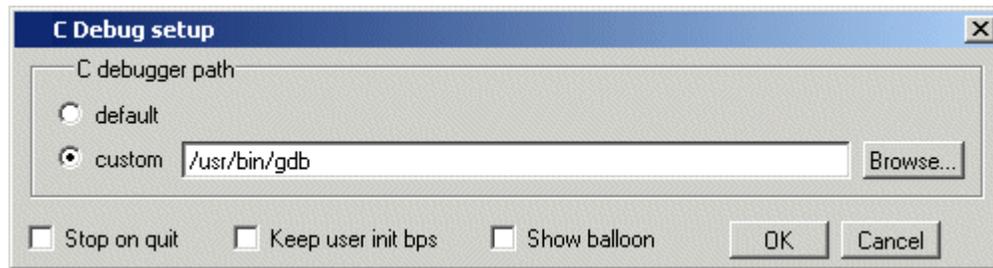
You should install the g++ compiler at the same directory level as your product install

Setting Up C Debug

Before viewing your SystemC/C/C++ source code, you must set up the C Debug path and options. To set up C Debug, follow these steps:

1. Compile and link your C code with the `-g` switch (to create debug symbols) and without `-O` (or any other optimization switches you normally use). See [SystemC Simulation](#) for information on compiling and linking SystemC code. Refer to the chapter [Verilog Interfaces to C](#) for information on compiling and linking C code.
2. Specify the path to the gdb debugger by selecting **Tools > C Debug > C Debug Setup**

Figure 26-1. Specifying Path in C Debug setup Dialog



Select "default" to point at the supplied version of gdb or "custom" to point at a separate installation.

3. Start the debugger by selecting **Tools > C Debug > Start C Debug**. ModelSim will start the debugger automatically if you set a breakpoint in a SystemC file.
4. If you are not using **gcc**, or otherwise have not specified a source directory, specify a source directory for your C code with the following command:

```
ModelSim> gdb dir <srcdirpath1>[:<srcdirpath2>[...]]
```

Running C Debug from a DO File

You can run C Debug from a DO file but there is a configuration issue of which you should be aware. It takes C Debug a few moments to start-up. If you try to execute a run command before C Debug is fully loaded, you may see an error like the following:

```
# ** Error: Stopped in C debugger, unable to real_run mti_run 10us
# Error in macro ./do_file line 8
# Stopped in C debugger, unable to real_run mti_run 10us
#   while executing
# "run 10us
```

In your DO file, add the command **cdbg_wait_for_starting** to alleviate this problem. For example:

```
cdbg enable_auto_step on
cdbg set_debugger /modelsim/5.8c_32/common/linux
cdbg debug_on
cdbg_wait_for_starting
run 10us
```

Setting Breakpoints

Breakpoints in C Debug work much like normal HDL breakpoints. You can create and edit them with ModelSim commands ([bp](#), [bd](#), [enablebp](#), [disablebp](#)) or within a Source window in the GUI (see [File-Line Breakpoints](#)). Some differences do exist:

- The Modify Breakpoints dialog, accessed by selecting **Tools > Breakpoints**, in the ModelSim GUI does not list C breakpoints.

- C breakpoint id numbers require a "c." prefix when referenced in a command.
- When using the `bp` command to set a breakpoint in a C file, you must use the `-c` argument.
- You can set a SystemC breakpoint so it applies only to the specified instance using the `-inst` argument to the `bp` command.
- If you set a breakpoint inside an export function call that was initiated from an `SC_METHOD`, you must use the `-scdpidebug` argument to the `vsim` command. This will enable you to single-step through the code across the SystemC/SystemVerilog boundary.

Here are some example commands:

```
bp -c *0x400188d4
```

Sets a C breakpoint at the hex address 400188d4. Note the '*' prefix for the hex address.

```
bp -c or_checktf
```

Sets a C breakpoint at the entry to function `or_checktf`.

```
bp -c or.c 91
```

Sets a C breakpoint at line 91 of `or.c`.

```
bp -c -cond "x < 5" foo.c 10
```

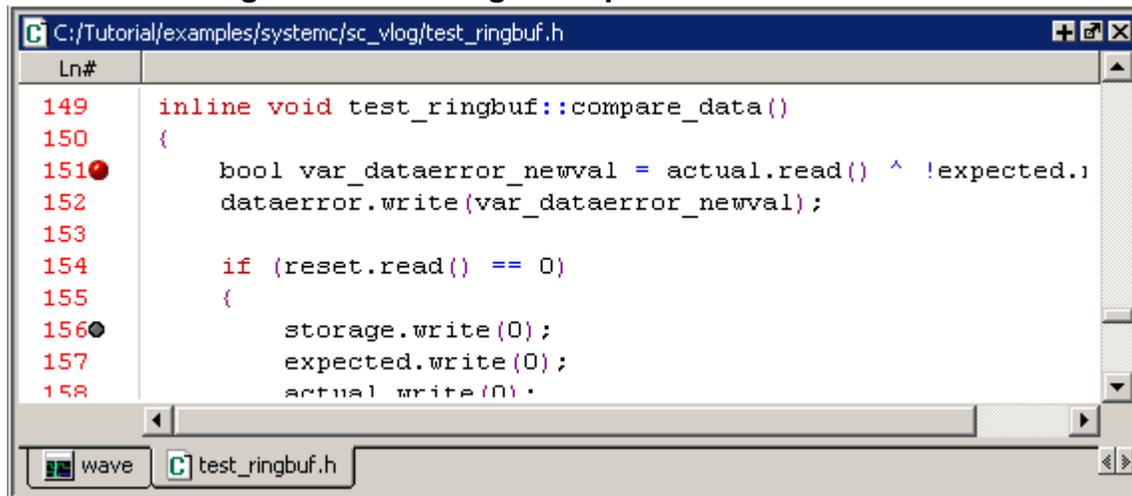
Sets a C breakpoint at line 10 of source file `foo.c` for the condition expression "x < 5".

```
enablebp c.1
```

Enables C breakpoint number 1.

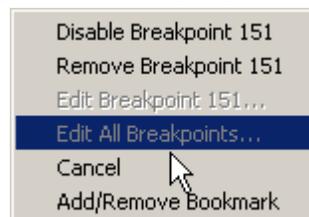
The graphic below shows a C file with one enabled breakpoint (indicated by a red ball on line 151) and one disabled breakpoint (indicated by a gray ball on line 154).

Figure 26-2. Setting Breakpoints in Source Code



Clicking the red ball with your right (third) mouse button pops up a menu with commands for removing or enabling/disabling the breakpoints.

Figure 26-3. Right Click Pop-up Menu on Breakpoint



Note

The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Do not set breakpoints in constructors of SystemC objects; it may crash the debugger.

Stepping in C Debug

Stepping in C Debug works much like you would expect. You use the same buttons and commands that you use when working with an HDL-only design.

Table 26-2. Simulation Stepping Options in C Debug

Button	Menu equivalent	Other equivalents
--------	-----------------	-------------------

Table 26-2. Simulation Stepping Options in C Debug

 <p>Step Into steps the current simulation to the next statement; if the next statement is a call to a C function that was compiled with debug info, ModelSim will step into the function</p>	<p>Tools > C Debug > Run > Step</p>	<p>use the step command at the CDBG> prompt see: step command</p>
 <p>Step Over statements are executed but treated as simple statements instead of entered and traced line-by-line; C functions are not stepped into unless you have an enabled breakpoint in the C file</p>	<p>Tools > C Debug > Run > Step -Over</p>	<p>use the step -over command at the CDBG> prompt see: step command</p>
 <p>Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event</p>	<p>Tools > C Debug > Run > Continue</p>	<p>use the run -continue command at the CDBG> prompt see: run command</p>

Debugging Active or Suspended Threads

You can use C Debug to debug either an active or a suspended thread and then view its contents (such as call stack, local variables, and source code).

You can debug a suspended thread in either of the following situations:

- While the simulation is running
- When the simulation is stopped at a breakpoint in an active SystemC/HDL thread

Known Problems With Stepping in C Debug

The following are known limitations which relate to problems with gdb:

- With some platform and compiler versions, **step** may actually behave like **run -continue** when in a C file. This is a gdb limitation that results from not having any debugging information when in an internal function to VSIM (that is, any FLI or VPI function). In these situations, use **step -over** to move line-by-line.
- Single-stepping into DPI-SC import function occasionally does not work if the SystemC source files are compiled with gcc-3.2.3 on linux platform. This is due to a gcc/gdb

interaction issue. It is recommended that you use gcc-4.3.3 (the default gcc compiler for linux platform in current release) to compile the SystemC source code.

Quitting C Debug

To end a debugging session, you can do one of the following.

- From the GUI:
Select **Tools > C Debug > Quit C Debug**.
- From the command line, enter the following in the Transcript window:

```
cgdb quit
```

Note



Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.

Finding Function Entry Points with Auto Find bp

ModelSim can automatically locate and set breakpoints at all currently known function entry points (that is, PLI/VPI/DPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Select **Tools > C Debug > Auto find bp** to invoke this feature.

The **Auto find bp** command provides a "snapshot" of your design when you invoke the command. If additional callbacks get registered later in the simulation, ModelSim will not identify these new function entry points *unless* you re-execute the **Auto find bp** command. If you want functions to be identified regardless of when they are registered, use [Identifying All Registered Function Calls](#) instead.

The **Auto find bp** command sets breakpoints in an enabled state and does not toggle that state to account for **step -over** or **run -continue** commands. This may result in unexpected behavior. For example, say you have invoked the **Auto find bp** command and you are currently stopped on a line of code that calls a C function. If you execute a **step -over** or **run -continue** command, ModelSim will stop on the breakpoint set in the called C file.

Identifying All Registered Function Calls

Auto step mode automatically identifies and sets breakpoints at registered function calls (that is, PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Auto step mode is helpful when you are not entirely familiar with a design and its associated C routines. As you step through the design,

ModelSim steps into and displays the associated C file when you hit a C function call in your HDL code. If you execute a **step -over** or **run -continue** command, ModelSim does not step into the C code.

When you first enable Auto step mode, ModelSim scans your design and sets enabled breakpoints at all currently known function entry points. As you step through the simulation, Auto step continues looking for newly registered callbacks and sets enabled breakpoints at any new entry points it identifies. Once you execute a **step -over** or **run -continue** command, Auto step disables the breakpoints it set, and the simulation continues running. The next time you execute a step command, the automatic breakpoints are re-enabled and Auto step sets breakpoints on any new entry points it identifies.

Note that Auto step does not disable user-set breakpoints.

Enabling Auto Step Mode

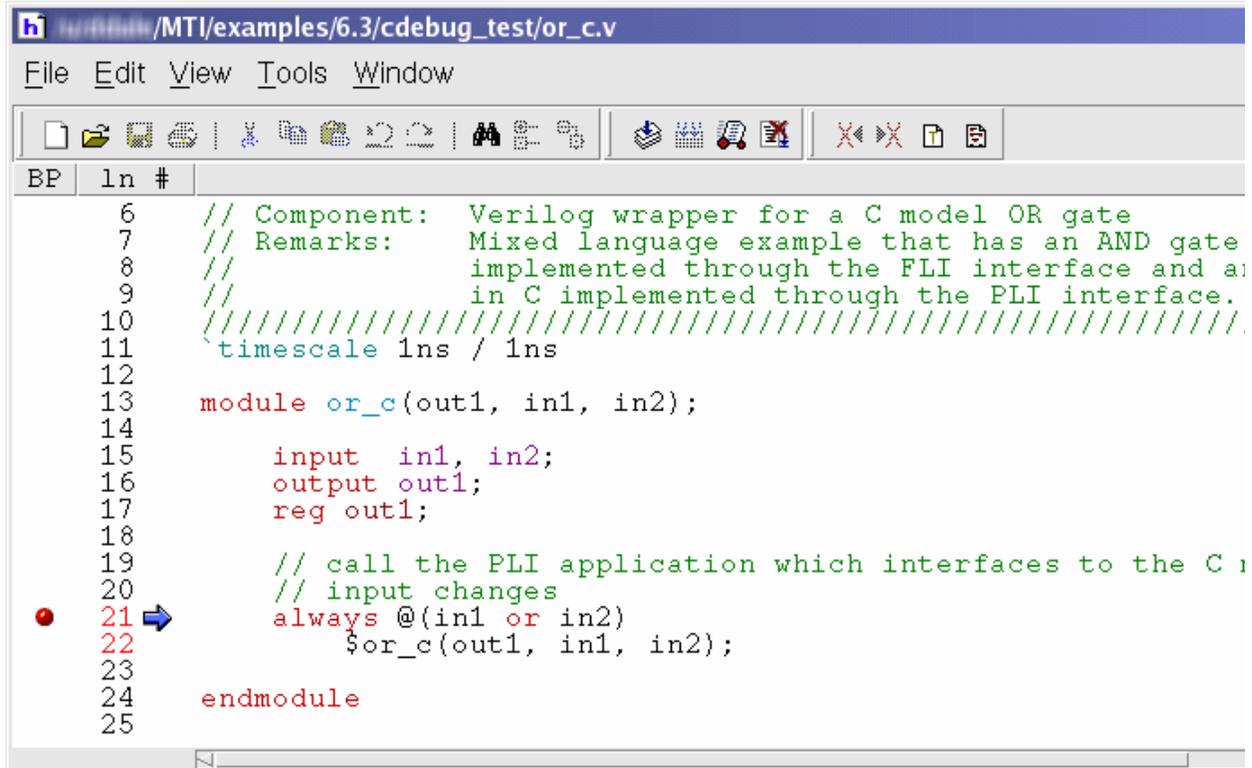
To enable Auto step mode, follow these steps:

1. Configure C Debug as described in [Setting Up C Debug](#).
2. Select **Tools > C Debug > Enable auto step**.
3. Load and run your design.

Example

The graphic below shows a simulation that has stopped at a user-set breakpoint on a PLI system task.

Figure 26-4. Simulation Stopped at Breakpoint on PLI Task



Because Auto step mode is enabled, ModelSim automatically sets a breakpoint in the underlying `xor_gate.c` file. If you click the step button at this point, ModelSim will step into that file.

Figure 26-5. Stepping into Next File

```
BP | ln # |
---|---|
11 | #include "veriusr.h"
12 |
13 | #define OR_RESULT 1 /* system task arg 1 is OR result out
14 | #define OR_VAL1 2 /* system task arg 2 is OR val1 input
15 | #define OR_VAL2 3 /* system task arg 3 is OR val2 input
16 | extern void or_gate(int *, int, int);
17 | /*****
18 | * calltf routine - Serves as an interface between ModelSim
19 | * It is called whenever the inputs to the OR gate change
20 | * the input values, passes these values to the C model, and
21 | * model's output value back into ModelSim.
22 | *****/
23 | int or_calltf()
24 | {
25 |     int val1, val2, result;
26 |
27 |     /* Read current values of C model inputs from Verilog
28 |     val1 = tf_getp(OR_VAL1);
29 |     val2 = tf_getp(OR_VAL2);
30 | }
```

Auto Find bp Versus Auto Step Mode

As noted in [Finding Function Entry Points with Auto Find bp](#), the **Auto find bp** command also locates and sets breakpoints at function entry points. Note the following differences between Auto find bp and Auto step mode:

- Auto find bp provides a "snapshot" of currently known function entry points at the time you invoke the command. Auto step mode continues to locate and set automatic breakpoints in newly registered function calls as the simulation continues. In other words, Auto find bp is static while Auto step mode is dynamic.
- Auto find bp sets automatic breakpoints in an enabled state and does not change that state to account for step-over or run-continue commands. Auto step mode enables and disables automatic breakpoints depending on how you step through the design. In cases where you invoke both features, Auto step mode takes precedence over Auto find bp. In other words, even if Auto find bp has set enabled breakpoints, if you then invoke Auto step mode, it will toggle those breakpoints to account for step-over and run-continue commands.

Debugging Functions During Elaboration

Initialization mode allows you to examine and debug functions that are called during elaboration (that is, while your design is in the process of loading). When you select this mode, ModelSim sets special breakpoints for foreign architectures and PLI/VPI modules that allow

you to set breakpoints in the initialization functions. When the design finishes loading, the special breakpoints are automatically deleted, and any breakpoints that you set are disabled (unless you specify **Keep user init bps** in the C debug setup dialog).

To run C Debug in initialization mode, follow these steps:

1. Start C Debug by selecting **Tools > C Debug > Start C Debug** *before* loading your design.
2. Select **Tools > C Debug > Init mode**.
3. Load your design.

As the design loads, ModelSim prints to the Transcript the names and/or hex addresses of called functions. For example the Transcript below shows a function pointer to a foreign architecture:

Figure 26-6. Function Pointer to Foreign Architecture



```
Transcript
# Loading work.or_c
# Loading work.mux4 1(struct)
# Loading work.and_c(wrapper)
# Loading ./and_gate.sl
# Shared object file './and_gate.sl'
#   Function name 'and_gate_init'
#   Function ptr '0x4001b571'. Foreign architecture.
# C breakpoint c.1
# 0x0819efaa in mti_cdbg_shared_objects_loaded ()

CDBG> |
```

To set a breakpoint on that function, you would type:

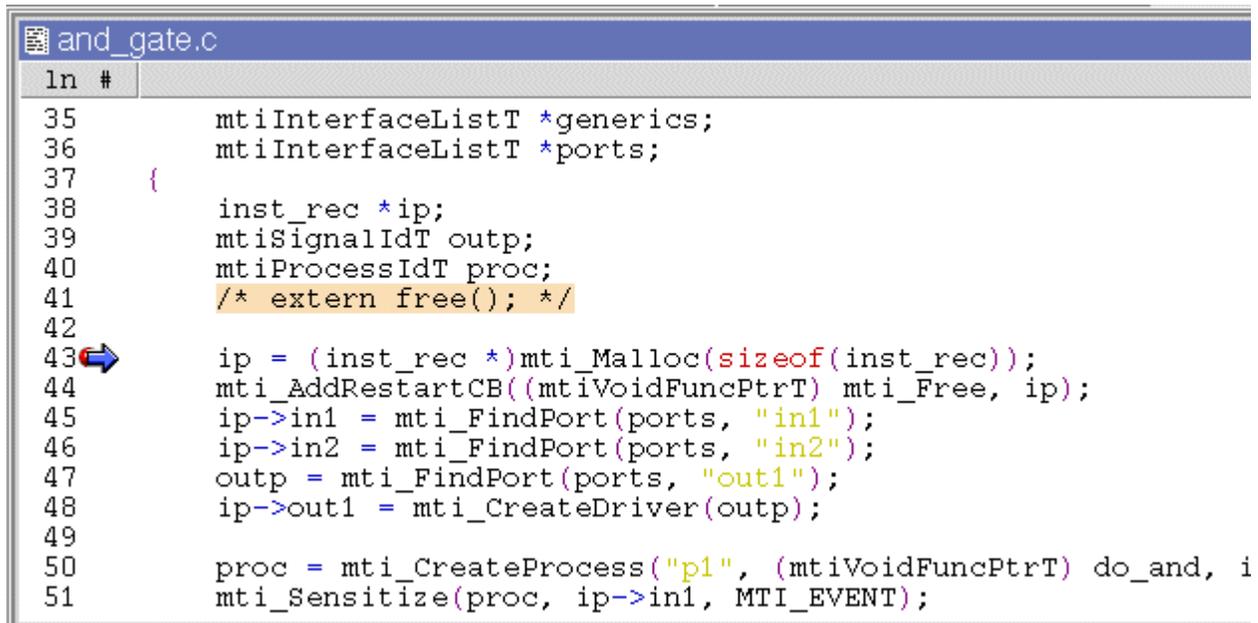
```
bp -c *0x4001b571
```

or

```
bp -c and_gate_init
```

ModelSim in turn reports that it has set a breakpoint at line 37 of the *and_gate.c* file. As you continue through the design load using **run -continue**, ModelSim hits that breakpoint and displays the file and associated line in a Source window.

Figure 26-7. Highlighted Line in Associated File



```
and_gate.c
ln #
35     mtiInterfaceListT *generics;
36     mtiInterfaceListT *ports;
37     {
38         inst_rec *ip;
39         mtiSignalIdT outp;
40         mtiProcessIdT proc;
41         /* extern free(); */
42
43     ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
44     mti_AddRestartCB((mtiVoidFuncPtrT) mti_Free, ip);
45     ip->in1 = mti_FindPort(ports, "in1");
46     ip->in2 = mti_FindPort(ports, "in2");
47     outp = mti_FindPort(ports, "out1");
48     ip->out1 = mti_CreateDriver(outp);
49
50     proc = mti_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, i
51     mti_Sensitize(proc, ip->in1, MTI_EVENT);
```

FLI Functions in Initialization Mode

There are two kinds of FLI functions that you may encounter in initialization mode. The first is a foreign architecture which was shown above. The second is a foreign function. ModelSim produces a Transcript message like the following when it encounters a foreign function during initialization:

```
# Shared object file './all.sl'
#   Function name 'in_params'
#   Function ptr '0x4001a950'. Foreign function.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (for example, `bp -c in_params`) or the function pointer (for example, `bp -c *0x4001a950`). Note, however, that foreign functions aren't called during initialization. You would hit the breakpoint only during runtime and then only if you enabled the breakpoint after initialization was complete or had specified **Keep user init bps** in the C debug setup dialog.

PLI Functions in Initialization Mode

There are two methods for registering callback functions in the PLI: 1) using a `veriusertfs` array to define all `usertf` entries; and 2) adding an `init_usertfs` function to explicitly register each `usertfs` entry (see [Registering PLI Applications](#) for more details). The messages ModelSim produces in initialization mode vary depending on which method you use.

ModelSim produces a Transcript message like the following when it encounters a `veriusertfs` array during initialization:

```
# vsim -pli ./veriusertfs mux_tb
# Loading ./veriusertfs
# Shared object file './veriusertfs'
#   veriusertfs array - registering calltf
#   Function ptr '0x40019518'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriusertfs'
#   veriusertfs array - registering checktf
#   Function ptr '0x40019570'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriusertfs'
#   veriusertfs array - registering sizetf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriusertfs'
#   veriusertfs array - registering misctf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set breakpoints on non-null callbacks using the function pointer (for example, `bp -c *0x40019570`). You cannot set breakpoints on null functions. The `sizetf` and `misctf` entries in the example above are null (the function pointer is `'0x0'`).

ModelSim reports the entries in multiples of four with at least one entry each for `calltf`, `checktf`, `sizetf`, and `misctf`. `checktf` and `sizetf` functions are called during initialization but `calltf` and `misctf` are not called until runtime.

The second registration method uses `init_usertfs` functions for each `usertfs` entry. ModelSim produces a Transcript message like the following when it encounters an `init_usertfs` function during initialization:

```
# Shared object file './veriusertfs'
#   Function name 'init_usertfs'
#   Function ptr '0x40019bec'. Before first call of init_usertfs.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (for example, `bp -c init_usertfs`) or the function pointer (for example, `bp -c *0x40019bec`). ModelSim will hit this breakpoint as you continue through initialization.

VPI Functions in Initialization Mode

VPI functions are registered via routines placed in a table named `vlog_startup_routines` (see [Registering PLI Applications](#) for more details). ModelSim produces a Transcript message like the following when it encounters a `vlog_startup_routines` table during initialization:

```
# Shared object file './vpi_test.sl'  
#   vlog_startup_routines array  
#   Function ptr '0x4001d310'. Before first call using function pointer.  
# C breakpoint c.1  
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using the function pointer (for example, `bp -c *0x4001d310`). ModelSim will hit this breakpoint as you continue through initialization.

Completing Design Load

If you are through looking at the initialization code you can select **Tools > C Debug > Complete load** at any time, and ModelSim will continue loading the design without stopping. The one exception to this is if you have set a breakpoint in a `LoadDone` callback and also specified **Keep user init bps** in the C Debug setup dialog.

Debugging Functions when Quitting Simulation

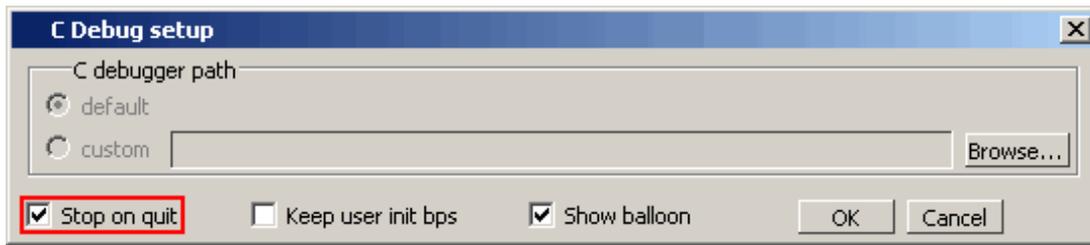
Stop on quit mode allows you to debug functions that are called when the simulator exits. Such functions include those referenced by one of the following:

- **mti_AddQuitCB** function in FLI code
- **miscf** function called by a `quit` or `$finish` in PLI code
- **cbEndofSimulation** function called by a `quit` or `$finish` in VPI code.

To enable **Stop on quit** mode, follow these steps:

1. Start C Debug by choosing **Tools > C Debug > Start C Debug** from the main menu.
2. Choose **Select Tools > C Debug > C Debug Setup** from the main menu.
3. Select **Stop on quit** in the C Debug setup dialog box ([Figure 26-8](#)).
4. Click OK.

Figure 26-8. Stop on quit Button in Dialog



With this mode enabled, if you have set a breakpoint in a quit callback function, C Debug will stop at the breakpoint after you issue the quit command in ModelSim. This allows you to step and examine the code in the quit callback function.

Invoke **run -continue** when you are done looking at the C code. When simulation completes, ModelSim automatically quits C-debugger and the GUI (whether or not a C breakpoint was hit and you return to the VSIM> prompt).

C Debug Command Reference

Table 26-3 provides a brief description of the commands that you can invoke when C Debug is running. Follow the links to the Reference Manual for complete command syntax.

Table 26-3. Command Reference for C Debug

Command	Description	Corresponding menu command
bd	Deletes a previously set C breakpoint	Right-click breakpoint in Source window and choose Remove Breakpoint.
bp -c	Sets a C breakpoint	Click in the line number column next to the desired line number in the Source window.
change	Changes the value of a C variable	none
describe	Prints the type information of a C variable	Select the C variable name in the Source window and choose Tools > Describe , or right-click and choose Describe.
disablebp	Disables a previously set C breakpoint	Right-click breakpoint in Source window and choose Disable Breakpoint.
enablebp	Enables a previously disabled C breakpoint	Right-click breakpoint in Source window and select Enable Breakpoint.

Table 26-3. Command Reference for C Debug

Command	Description	Corresponding menu command
<code>examine</code>	Prints the value of a C variable	Select the C variable name in the Source window and choose Tools > Examine , or right-click and choose Examine.
<code>gdb dir</code>	Sets the source directory search path for the C debugger	none
<code>pop</code>	Moves the specified number of call frames up the C callstack	none
<code>push</code>	Moves the specified number of call frames down the C callstack	none
<code>run -continue</code>	Continues running the simulation after stopping	Click the run -continue button on the Main or Source window toolbar.
<code>run -finish</code>	Continues running the simulation until control returns to the calling function	Tools > C Debug > Run > Finish
<code>show</code>	Displays the names and types of the local variables and arguments of the current C function	Tools > C Debug > Show
<code>step</code>	c step in the C debugger to the next executable line of C code; step goes into function calls, whereas step -over does not	Click the step or step -over button on the Main or Source window toolbar.
<code>tb</code>	Displays a stack trace of the C call stack	Tools > C Debug > Traceback

Chapter 27

Profiling Performance and Memory Use

The ModelSim profiler combines a statistical sampling profiler with a memory allocation profiler to provide instance specific execution and memory allocation data. It allows you to quickly determine how your memory is being allocated and easily identify areas in your simulation where performance can be improved. The profiler 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 functionality described in this chapter requires a profiler license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Introducing Performance and Memory Profiling

The profiler provides an interactive graphical representation of both memory and CPU usage on a per instance basis. It shows you what part of your design is consuming resources (CPU cycles or memory), allowing you to more quickly find problem areas in your code.

The profiler enables those familiar with the design and validation environment to find first-level improvements in a matter of minutes. For example, the statistical sampling profiler might show the following:

- non-accelerated VITAL library cells that are impacting simulation run time
- objects in the sensitivity list that are not required, resulting in a process that consumes more simulation time than necessary
- a test bench process that is active even though it is not needed
- an inefficient C module
- random number processes that are consuming simulation resources in a test bench 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.

The memory allocation profiler provides insight into how much memory different parts of the design are consuming. The two major areas of concern are typically: 1) memory usage during elaboration, and 2) during simulation. If memory is exhausted during elaboration, for example,

memory profiling may provide insights into what part(s) of the design are memory intensive. Or, if your HDL or PLI/FLI code is allocating memory and not freeing it when appropriate, the memory profiler will indicate excessive memory use in particular portions of the design.

Statistical Sampling Profiler

The profiler's statistical sampling profiler samples the current simulation at a user-determined rate (every <n> milliseconds of real or "wall-clock" time, not simulation time) and records what is executing at each sample point. The advantage of statistical sampling is that an entire simulation need not be run to get good information about what parts of your design are using the most simulation time. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

The statistical profiler reports only on the samples that it can attribute to user code. For example, if you use the `-nodebug` argument to `vcom` or `vlog`, it cannot report sample results.

Memory Allocation Profiler

The profiler's memory allocation profiler records every memory allocation and deallocation that takes place in the context of elaborating and simulating the design. It makes a record of the design element that is active at the time of allocation so memory resources can be attributed to appropriate parts of the design. This provides insights into memory usage that can help you re-code designs to, for example, minimize memory use, correct memory leaks, and change optimization parameters used at compile time.

Getting Started with the Profiler

Memory allocation profiling and statistical sampling are enabled separately.

Note



It is suggested that you not run the memory allocation and statistical sampling profilers at the same time. The analysis of the memory allocation can skew the results of the statistical sampling profiler.

Enabling the Memory Allocation Profiler

To record memory usage during elaboration and simulation, enable memory allocation profiling when the design is loaded with the `-memprof` argument to the `vsim` command.

```
vsim -memprof <design_unit>
```

Note that profile-data collection for the call tree is off by default. See [Calltree Window](#) for additional information on collecting call-stack data.

You can use the graphic user interface as follows to perform the same task.

1. Select **Simulate > Start Simulation** or the Simulate icon, to open the Start Simulation dialog box.
2. Select the Others tab.
3. Click the **Enable memory profiling** checkbox to select it.
4. Click **OK** to load the design with memory allocation profiling enabled.

If memory allocation during elaboration is not a concern, the memory allocation profiler can be enabled at any time after the design is loaded by doing any one of the following:

- select **Tools > Profile > Memory**
- use the -m argument with the [profile on](#) command

```
profile on -m
```

- click the Memory Profiling icon 

Handling Large Files

To allow memory allocation profiling of large designs, where the design itself plus the data required to keep track of memory allocation exceeds the memory available on the machine, the memory profiler allows you to route raw memory allocation data to an external file. This allows you to save the memory profile with minimal memory impact on the simulator, regardless of the size of your design.

The external data file is created during elaboration by using either the `-memprof+file=<filename>` or the `-memprof+fileonly=<filename>` argument with the [vsim](#) command.

The `-memprof+file=<filename>` option will collect memory profile data during both elaboration and simulation and save it to the named external file *and* makes the data available for viewing and reporting during the current simulation.

The `-memprof+fileonly=<filename>` option will collect memory profile data during both elaboration and simulation and save it to *only* the named external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

Alternatively, you can save memory profile data from the simulation only by using either the `-m -file <filename>` or the `-m -fileonly <filename>` argument with the [profile on](#) command.

The `-m -file <filename>` option saves memory profile data from simulation to the designated external file *and* makes the data available for viewing and reporting during the current simulation.

The `-m -fileonly <filename>` option saves memory profile data from simulation to *only* the designated external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

After elaboration and/or simulation is complete, a separate session can be invoked and the profile data can be read in with the `profile reload` command for analysis. It should be noted, however, that this command will clear all performance and memory profiling data collected to that point (implicit `profile clear`). Any currently loaded design will be unloaded (implicit `quit -sim`), and run-time profiling will be turned off (implicit `profile off -m -p`). If a new design is loaded after you have read the raw profile data, then all internal profile data is cleared (implicit profile clear), but run-time profiling is not turned back on.

Enabling the Statistical Sampling Profiler

To enable the profiler's statistical sampling profiler prior to a simulation run, do any one of the following:

- select **Tools > Profile > Performance**
- use the `profile on` command
- click the Performance Profiling icon 

Collecting Memory Allocation and Performance Data

Both memory allocation profiling and statistical sampling occur during the execution of a ModelSim `run` command. With profiling enabled, all subsequent `run` commands will collect memory allocation data and performance statistics. Profiling results are cumulative – each `run` command performed with profiling enabled will add new information to the data already gathered. To clear this data, select **Tools > Profile > Clear Profile Data** or use the `profile clear` command.

With the profiler enabled and a `run` command initiated, the simulator will provide a "Profiling" message in the transcript to indicate that profiling has started.

If the statistical sampling profiler and the memory allocation profiler are on, the status bar will display the number of Profile Samples collected and the amount of memory allocated, as shown below. Each profile sample will become a data point in the simulation's performance profile.

Figure 27-1. Status Bar: Profile Samples



```
Now: 100 ps Delta: 0 Profile Samples: 18 Memory: 65.4KB
```

Turning Profiling Off

You can turn off the statistical sampling profiler or the memory allocation profiler by doing any one of the following:

- deselect the **Performance** and/or **Memory** options in the **Tools > Profile menu**
- deselect the Performance Profiling and Memory Profiling icons in the toolbar
- use the [profile off](#) command with the -p or -m arguments.

Any ModelSim [run](#) commands that follow will not be profiled.

Running the Profiler on Windows with PLI/VPI Code

If you need to run the profiler under Windows on a design that contains FLI/PLI/VPI code, add these two switches to the compiling/linking command:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* file that the profiler can use in its report.

Interpreting Profiler Data

The utility of the data supplied by the profiler depends in large part on how your code is written. In cases where a single model or instance consumes a high percentage of simulation time or requires a high percentage of memory, the statistical sampling profiler or the memory allocation profiler quickly identifies that object, allowing you to implement a change that runs faster or requires less memory.

More commonly, simulation time or memory allocation will be spread among a handful of modules or entities – for example, 30% of simulation time split between models X, Y, and Z; or 20% of memory allocation going to models A, B, C and D. In such situations, careful examination and improvement of each model may result in overall speed improvement or more efficient memory allocation.

There are times, however, when the statistical sampling and memory allocation profilers tell you nothing more than that simulation time or memory allocation is fairly equally distributed throughout your design. In such situations, the profiler provides little helpful information and improvement must come from a higher level examination of how the design can be changed or optimized.

Viewing Profiler Results

The profiler provides four views of the collected data – *Ranked*, *Design Units*, *Call Tree* and *Structural*. All four views are enabled by selecting **View > Profiling**.

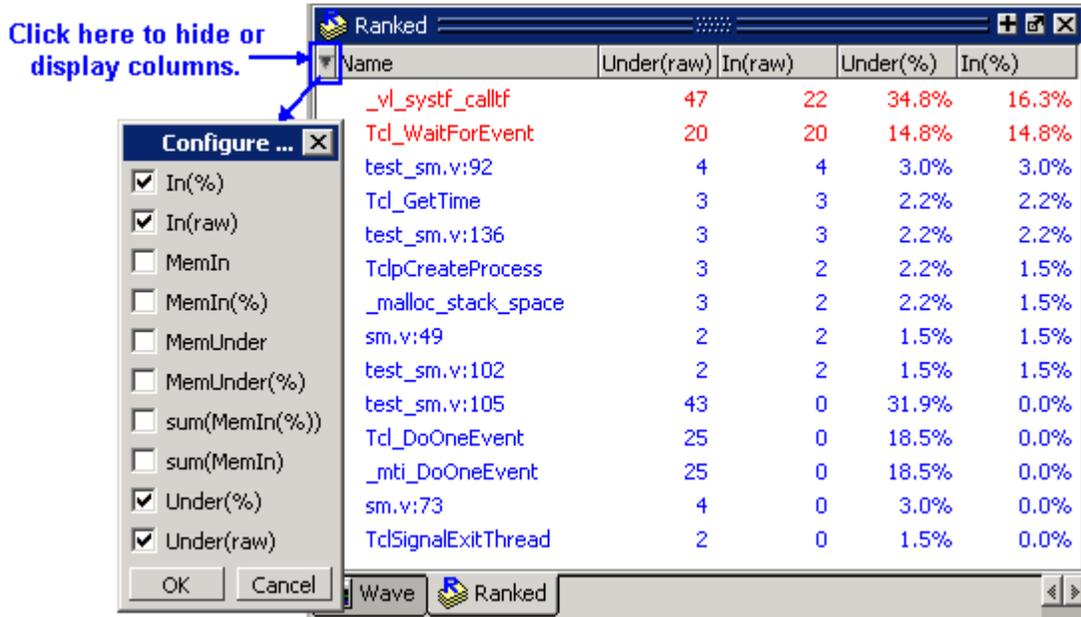
Note

The Ranked, Design Units, Calltree, and Structural windows, by default, only show performance and memory profile data equal to or greater than 1 percent. You can change this with the Profile Cutoff tool in the profile toolbar group.

Ranked Window

The Ranked window displays the results of the statistical performance profiler and the memory allocation profiler for each function or instance. By default, ranked profiler results are sorted by values in the *In%* column, which shows the percentage of the total samples collected for each function or instance. Click the down arrow to the left of the Name column to open a list of available columns and allows you to select which columns are to be hidden or displayed (Figure 27-2).

Figure 27-2. Ranked Window



You can sort ranked results by any other column by simply clicking the column heading.

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

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

Design Units Window

The Design Units window displays the profiler results, aggregated for the different design units. This window provides information similar to the Structural window, but organized by design unit, rather than hierarchically.

Figure 27-3. Design Units Window

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

Calltree Window

Data collection for the call tree is off by default, due to the fact that it will increase the simulation time and resource usage. Collection can be turned on from the VSIM command prompt with **profile option collect_calltrees on** and off with **profile option collect_calltrees off**. Call stack data collection can also be turned on with the `-memprof+call` argument to the `vsim` command.

By default, profiler results in the Call Tree window are sorted according to the *Under(%)* column, which shows the percentage of the total samples collected for each function or instance and all supporting routines or instances. Sort results by any other column by clicking the column heading. As in the Ranked window, red object names indicate functions or instances that, by default, are consuming 5% or more of simulation time.

The Call Tree window differs from the Ranked window in two important respects.

- Entries in the Name column of the Calltree window are indented in hierarchical order to indicate which functions or routines call which others.

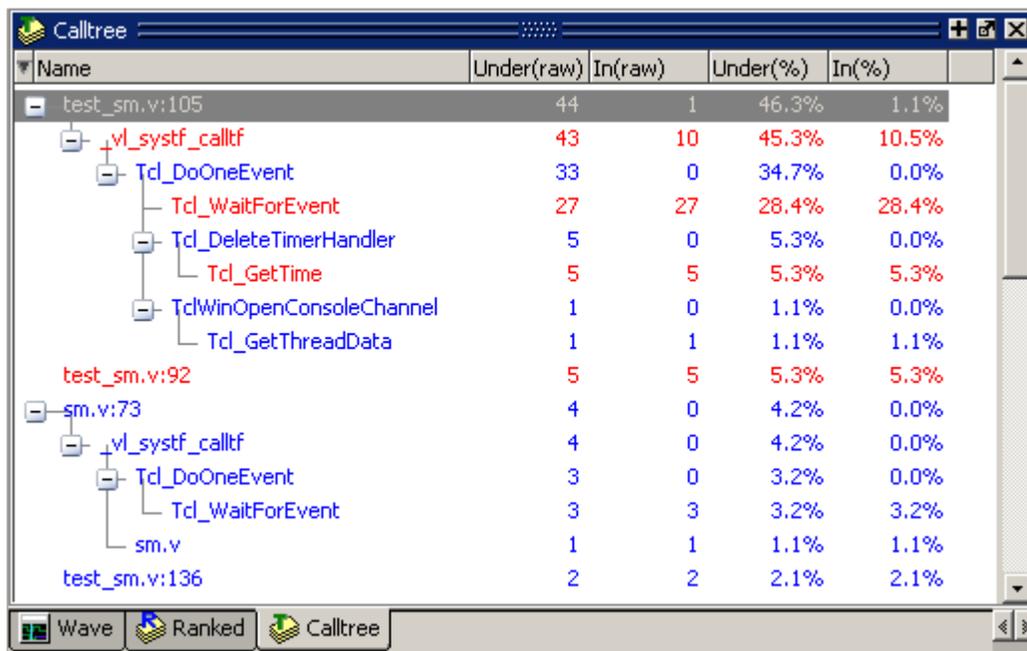
- A *%Parent* column in the Calltree window allows you to see what percentage of a parent routine’s simulation time is used in which subroutines.

The Calltree window presents data in a call-stack format that provides more context than does the Ranked window about where simulation time is spent. For example, your models may contain several instances of a utility function that computes the maximum of 3-delay values. A Ranked window 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 Calltree window 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 in the Calltree window shows the percent of simulation time or allocated memory a given function or instance is using of its parent’s total simulation time or available memory. From this column, you can calculate the percentage of total simulation time or memory taken up by any function. For example, if a particular parent entry used 10% of the total simulation time or allocated memory, and it called a routine that used 80% of its simulation time or memory, then the percentage of total simulation time spent in, or memory allocated to, that routine would be 80% of 10%, or 8%.

In addition to these differences, the Ranked window displays any particular function only once, regardless of where it was used. In the Calltree window, the function can appear multiple times – each time in the context of where it was used.

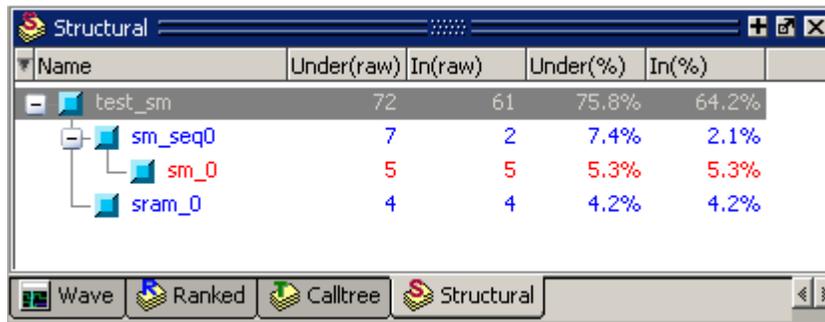
Figure 27-4. Calltree Window



Structural Window

The Structural profile window displays instance-specific performance and memory profile information in a hierarchical structure format identical to the Structure window. It contains the same information found in the Calltree window but adds an additional dimension with which to categorize performance samples and memory allocation. It shows how call stacks are associated with different instances in the design.

Figure 27-5. Structural Window

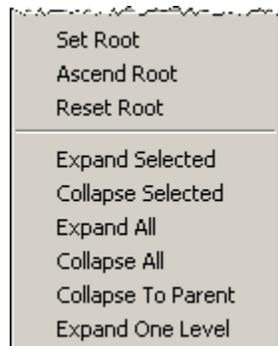


Name	Under(raw)	In(raw)	Under(%)	In(%)
test_sm	72	61	75.8%	64.2%
sm_seq0	7	2	7.4%	2.1%
sm_0	5	5	5.3%	5.3%
sram_0	4	4	4.2%	4.2%

In the Calltree and Structural profile windows, you can expand and collapse the various levels to hide data that is not useful to the current analysis and/or is cluttering the display. Click on the '+' box next to an object name to expand the hierarchy and show supporting functions and/or instances beneath it. Click the '-' box to collapse all levels beneath the entry.

You can also right click any function or instance in the Calltree and Structural windows to obtain popup menu selections for rooting the display to the currently selected item, to ascend the displayed root one level, or to expand and collapse the hierarchy (Figure 27-6).

Figure 27-6. Expand and Collapse Selections in Popup Menu



Toggling Display of Call Stack Entries

By default call stack entries do not display in the Structural window. To display call stack entries, right-click in the window and select **Show Calls**.

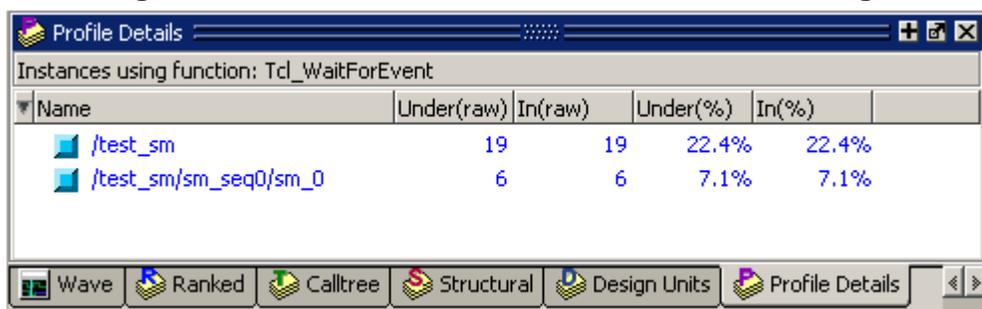
Viewing Profile Details

The Profiler increases visibility into simulation performance and memory usage with dynamic links to the Source window and the Profile Details window. The Profile Details window is enabled by selecting **View > Profiling > Profile Details** or by entering the **view profiledetails** command at the VSIM prompt. You can also right-click any function or instance in the Ranked, Call Tree, or Structural windows to open a popup menu that includes options for viewing profile details. The following options are available:

- View Source — opens the Source window to the location of the selected function.
- View Instantiation — opens the Source window to the location of the instantiation.
- Function Usage — opens the Profile Details window and displays all instances using the selected function.

In the Profile Details window shown below, all the instances using function *Tcl_WaitForEvent* are displayed. The statistical performance data shows how much simulation time is used by *Tcl_WaitForEvent* in each instance.

Figure 27-7. Profile Details Window: Function Usage



The screenshot shows the Profile Details window with the title bar 'Profile Details'. The main area displays 'Instances using function: Tcl_WaitForEvent'. Below this is a table with the following data:

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

At the bottom of the window, there is a toolbar with icons for Wave, Ranked, Calltree, Structural, Design Units, and Profile Details.

- Instance Usage — opens the Profile Details window and displays all instances with the same definition as the selected instance.

Figure 27-8. Profile Details Window: Instance Usage



The screenshot shows the Profile Details window with the title bar 'Profile Details'. The main area displays 'Instances with same definition as /test_sm/sm_seq0/sm_0'. Below this is a table with the following data:

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

At the bottom of the window, there is a toolbar with icons for Wave, Ranked, Calltree, Structural, Design Units, and Profile Details.

- View Instantiation — opens the Source window to the point in the source code where the selected instance is instantiated.

- Callers and Callees — opens the Profile Details window and displays the callers and callees for the selected function. Items above the selected function are callers; items below are callees.

The selected function is distinguished with an arrow on the left and in 'hotForeground' color as shown below.

Figure 27-9. Profile Details Window: Callers and Callees

The screenshot shows a window titled 'Profile Details' with the subtitle 'Callers and callees of 'TclSignalExitThread''. It contains a table with the following data:

Name	Under(raw)	In(raw)	Under(%)	In(%)
Tcl_DoOneEvent	8	0	100.0%	0.0%
➔ TclSignalExitThread	8	1	100.0%	12.5%
Tcl_GetTime	6	6	85.7%	85.7%
TclpGetClicks	1	1	14.3%	14.3%

At the bottom of the window, there are several tabs: Wave, Ranked, Calltree, Structural, Design Units, and Profile Detail.

- Display in Call Tree — expands the Calltree window and displays all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

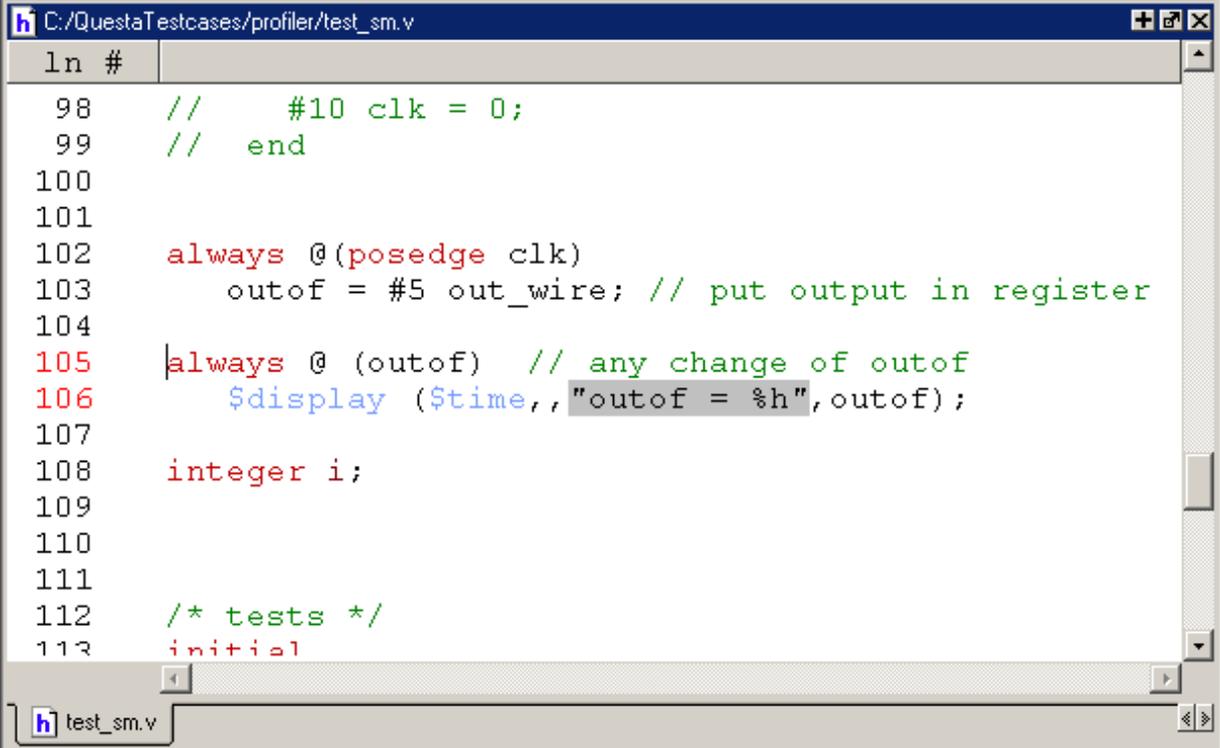
Note that profile-data collection for the call tree is off by default. See [Calltree Window](#) for additional information on collecting call-stack data.

- Display in Structural — expands the Structural window and displays all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

Integration with Source Windows

The Ranked, Design Unit, Call Tree, and Structural windows are all dynamically linked to Source window. You can double-click any function or instance in these windows to bring up that object in a Source window with the selected line displayed.

Figure 27-10. Accessing Source from Profile Views



```
C:/QuestatTestcases/profiler/test_sm.v
ln #
98 // #10 clk = 0;
99 // end
100
101
102 always @(posedge clk)
103     outof = #5 out_wire; // put output in register
104
105 always @ (outof) // any change of outof
106     $display ($time, "outof = %h", outof);
107
108 integer i;
109
110
111
112 /* tests */
113 initial
```

You can perform the same task by right-clicking any function or instance in any one of the four Profile views and selecting View Source from the popup menu that opens.

When you right-click an instance in the Structural window, the View Instantiation selection will become active in the popup menu. Selecting this option opens the instantiation in a Source window and highlights it.

The right-click popup menu also allows you to change the root instance of the display, ascend to the next highest root instance, or reset the root instance to the top level instance.

The selection of a context in the structure window will cause the root display to be set in the Structural window.

Analyzing C Code Performance

You can include C code in your design via SystemC, the Verilog PLI/VPI, or the ModelSim FLI. The profiler can be used to determine the impact of these C modules on simulator performance. Factors that can affect simulator performance when a design includes C code are as follows:

- PLI/FLI applications with large sensitivity lists
- Calling operating system functions from C code

- Calling the simulator's command interpreter from C code
- Inefficient C code

In addition, the Verilog PLI/VPI requires maintenance of the simulator's internal data structures as well as the PLI/VPI data structures for portability. (VHDL does not have this problem in ModelSim because the FLI gets information directly from the simulator.)

Searching Profiler Results

Each of the Profiler windows provides find and filter functions to assist you in isolating and examining specific results. A "Find" toolbar will appear along the bottom edge of the active window when you do either of the following:

- Select **Edit > Find** in the menu bar.
- Click the **Find** icon in the Standard toolbar. 

All of the above actions are toggles - repeat the action and the Find toolbar will close.

The Find or Filter entry fields prefill as you type, based on the context of the current window selection. The find or filter action begins as you type.

For additional details on the find and filter functions, see [Using the Find and Filter Functions](#).

Reporting Profiler Results

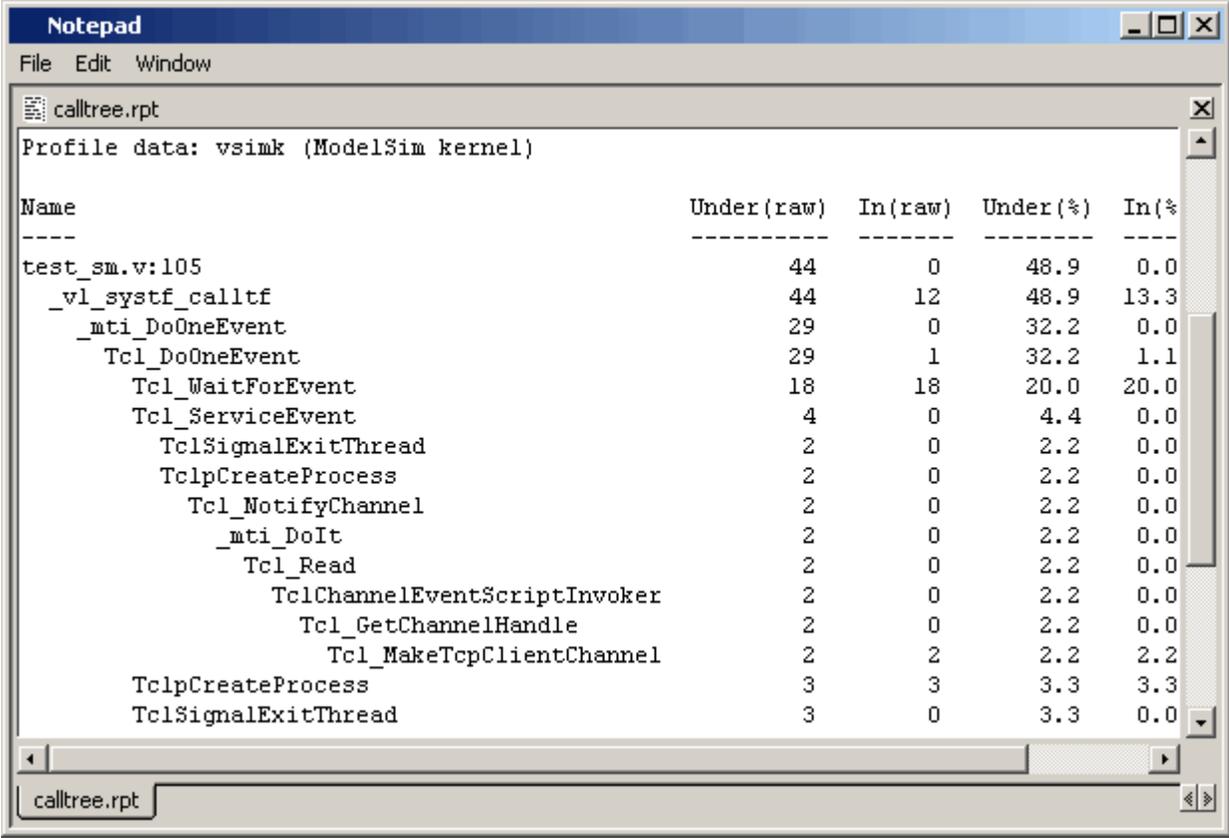
You can create performance and memory profile reports using the Profile Report dialog or the [profile report](#) command.

For example, the command

```
profile report -calltree -file calltree.rpt -cutoff 2
```

will produce a Call Tree profile report in a text file called *calltree.rpt*, as shown here.

Figure 27-11. Profile Report Example



```
Profile data: vsimk (ModelSim kernel)
```

Name	Under (raw)	In (raw)	Under (%)	In (%)
test_sm.v:105	44	0	48.9	0.0
_vl_systf_calltf	44	12	48.9	13.3
_mti_DoOneEvent	29	0	32.2	0.0
Tcl_DoOneEvent	29	1	32.2	1.1
Tcl_WaitForEvent	18	18	20.0	20.0
Tcl_ServiceEvent	4	0	4.4	0.0
TclSignalExitThread	2	0	2.2	0.0
TclpCreateProcess	2	0	2.2	0.0
Tcl_NotifyChannel	2	0	2.2	0.0
_mti_DoIt	2	0	2.2	0.0
Tcl_Read	2	0	2.2	0.0
TclChannelEventScriptInvoker	2	0	2.2	0.0
Tcl_GetChannelHandle	2	0	2.2	0.0
Tcl_MakeTcpClientChannel	2	2	2.2	2.2
TclpCreateProcess	3	3	3.3	3.3
TclSignalExitThread	3	0	3.3	0.0

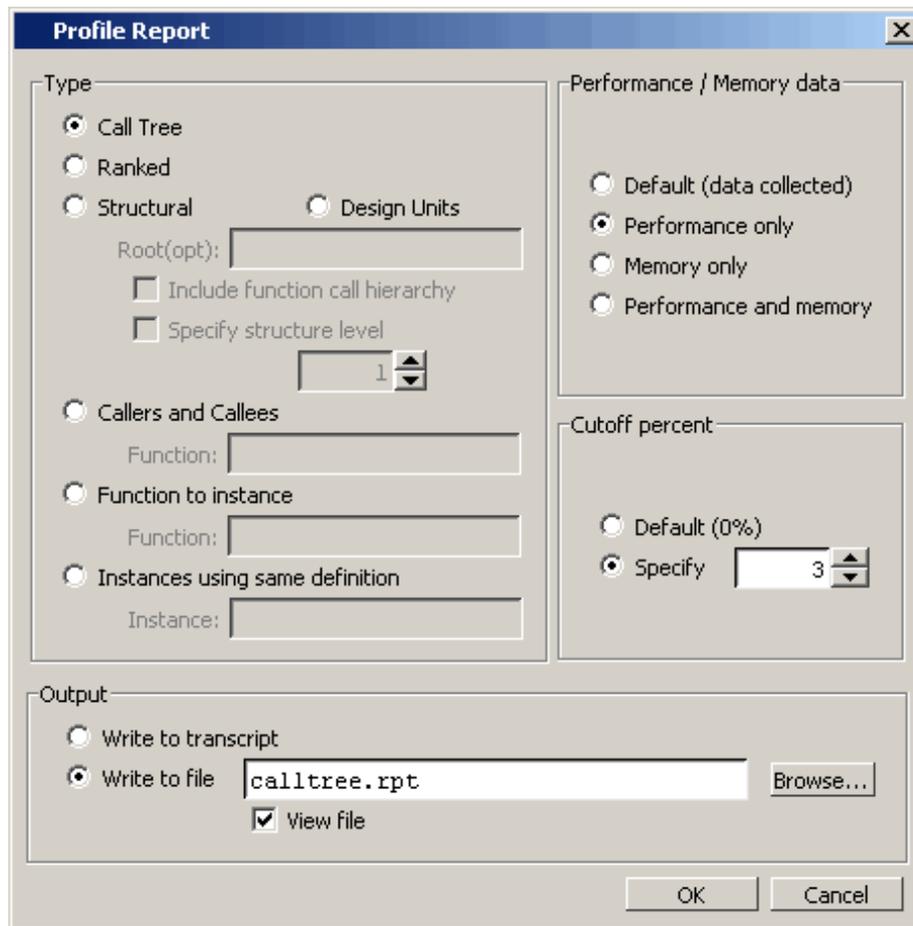
Select **Tools > Profile > Profile Report** to open the Profile Report dialog. The Profile Report dialog allows you to select the following performance profile type for reporting: Calltree, Ranked, Structural, Callers and Callees, Function to Instance, and Instances using the same definition. When the Structural profile type is selected, you can designate the root instance pathname, include function call hierarchy, and specify the structure level to be reported.

You can elect to report performance information only, memory information only, or a both. By default, all data collected will be reported.

Both performance and memory data will be displayed with a default cutoff of 0% - meaning, the report will contain any functions or instances that use simulation time or memory - unless you specify a different cutoff percentage.

You may elect to write the report directly to the Transcript window or to a file. If the "View file" box is selected, the profile report will be generated and immediately displayed in Notepad when the OK button is clicked.

Figure 27-12. Profile Report Dialog Box



Capacity Analysis

This section describes how to display memory usage (capacity) data that ModelSim collects for the following types of SystemVerilog constructs in the current design:

- Classes
- Queues, dynamic arrays, and associative arrays (QDAS)
- Assertions and cover directives
- Covergroups
- Solver (calls to randomize ())

ModelSim updates memory usage data at the end of every time step of the simulation and collects the number of objects allocated, the current memory allocated for the class object, the peak memory allocated, and the time at which peak memory occurred.

You can display this data in column format in the Capacity window of the user interface ([Obtaining a Graphical Interface \(GUI\) Display](#)) or as a text-based report in the Transcript window ([Writing a Text-Based Report](#)), which you can also write to a text file.

Enabling or Disabling Capacity Analysis

When you invoke ModelSim, you can use arguments to the `vsim` command to select one of the following levels of capacity analysis before loading your design:

- No analysis. Specify `-nocapacity`, along with any other `vsim` arguments you want to use.
- Coarse-grain analysis. Enabled by default (no additional `vsim` argument required).
- Fine-grain analysis: Specify `-capacity`, along with any other `vsim` arguments you want to use.

Note



Coarse-level and fine-level analyses are described in [Levels of Capacity Analysis](#).

In addition, you can use various other commands to enable collection of memory capacity data, along with viewing and reporting that data. [Table 27-1](#) summarizes the different ways to enable, view, and report memory capacity data.

Refer to the [ModelSim Reference Manual](#) for more information on using the commands listed in [Table 27-1](#).

Table 27-1. How to Enable and View Capacity Analysis

Action	Result	Description
<code>vsim <filename></code>	Collects coarse-grain analysis data.	No need to explicitly specify a coarse-grain analysis; enabled by default.
<code>vsim -capacity <filename></code>	Collects fine-grain analysis data.	Overrides default coarse-grain analysis.
<code>vsim -nocapacity <filename></code>	Disables capacity analysis.	No capacity data is collected.
<code>view capacity</code>	Displays the Capacity window containing capacity data.	Same as choosing View > Capacity from main menu.
<code>write report</code> { <code>[-capacity [-l -s] [-assertions -classes -cvg -qdas -solver]]</code> }	Reports data on memory capacity in either the Transcript window or to a file.	Use the <code>-capacity</code> switch along with other switches for object types to display memory data.
<code>profile on</code> { <code>-solver -qdas -classes -cvg -assertions</code> }	Enables fine-grain analysis.	Use this command after you have already loaded the design; you can specify multiple command switches.
<code>profile off</code> { <code>-solver -qdas -classes -cvg -assertions</code> }	Disables fine-grain analysis.	Use this command after you have already loaded the design; you can specify multiple command switches.
<code>coverage report -memory</code>	Reports coarse-grain data in either the Transcript window or to a file.	Use with <code>-cvg</code> and <code>-details</code> switches to obtain fine-grain data for covergroups.
<code>vcover report -memory</code>	Reports coarse-grain data from a previously saved code or functional coverage run in either the Transcript window or to a file.	Use with <code>-cvg</code> and <code>-details</code> switches to obtain fine-grain data for covergroups.
<code>vcover stats -memory</code>	Reports coarse-grain data from a previously saved code or functional coverage run in either the Transcript window or to a file.	No fine-grain analysis available.

Table 27-1. How to Enable and View Capacity Analysis (cont.)

Action	Result	Description
Choose View > Capacity from main menu	Displays the Capacity window containing capacity data.	Same as entering the view capacity command.

Levels of Capacity Analysis

ModelSim collects data as either a coarse-grain analysis or a fine-grain analysis of memory capacity. The main difference between the two levels is the amount of capacity data collected.

Coarse-grain Analysis

The coarse-grain analysis data is enabled by default when you run the vsim command.

The purpose of this analysis is to provide a simple display of the number of objects, the memory allocated for each class of design objects, the peak memory allocated, and the time at which peak memory occurred.

You can display the results of a coarse-grain analysis as either a graphical display in the user interface (see [Obtaining a Graphical Interface \(GUI\) Display](#)) or as a text report (see [Writing a Text-Based Report](#)).

Fine-grain Analysis

When you enable a fine-grain analysis, ModelSim collects more capacity data that you can use to dig deeper into the area where memory consumption is problematic. The details about each type of object are further quantified.

The display of the Capacity window expands the coarse-grain categories and shows the count, current and peak memory allocation per class type, per assertions, per coverage groups/bins and per dynamic array objects.

Classes — displays aggregate information for each class type, including the current filename and line number where the allocation occurred.

QDAS — displays aggregate information for each type (queues, dynamic, associative), including the current filename and line number where the allocation has occurred.

Assertions — displays the assertion full name, number of threads allocated, current memory, peak memory and peak time.

Covergroups — displays the covergroup full name, number of coverpoints and crosses allocated, current memory, peak memory and peak time.

Solver — displays file name and line number of randmize() calls grouped by file name, line number, current memory, peak memory and peak time.

To enable fine-grain analysis, use either of the following commands:

```
vsim -capacity
profile on {-solver | -qdas | -classes | -cvg | -assertions}
```

Note that turning off fine-grain analysis reverts to coarse-grain analysis.

Obtaining a Graphical Interface (GUI) Display

To display a tabular listing of memory capacity data, choose View > Capacity from the main menu or enter the view command with “capacity” as the window type:

```
view capacity
```

This creates the Capacity window that displays memory data for the current design.

Refer to the section “Capacity Window” for more information.

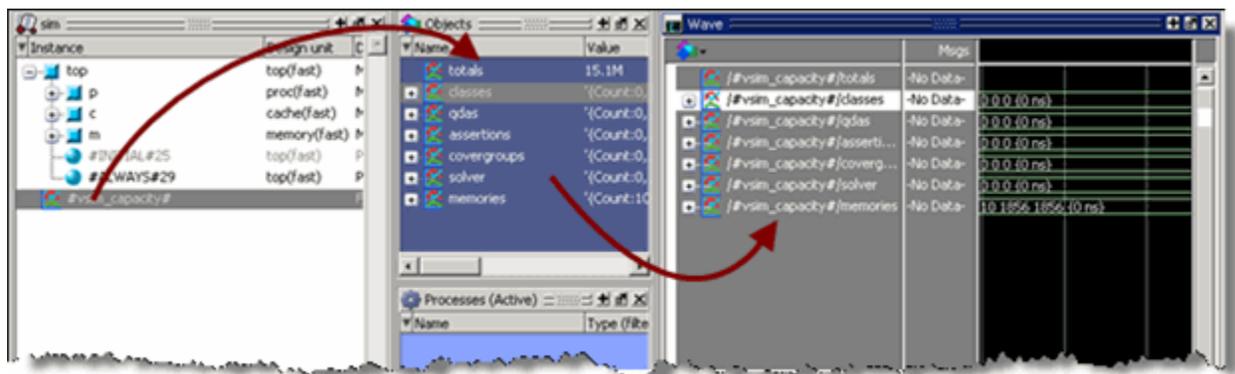
Displaying Capacity Data in the Wave Window

In addition, you can add the signals from the Capacity window to the Wave window like any other signal, either by drag-and-drop in the Main window or from the command line.

To drag-and-drop, do the following:

1. Click the Structure (sim) window tab.
2. Select the Instance labeled #vsim_capacity#. Selecting this instance displays a set of capacity types in the Objects window (see Figure 27-13).
3. Select one or more objects in the Objects window. Note that you can click on the [+] indicator to expand the listing of data below any type.
4. Drag and drop the selected objects to the Wave window or click the middle mouse button when the cursor is over an object.

Figure 27-13. Displaying Capacity Objects in the Wave Window



You can also display capacity objects in the Wave window from the command line, according to the following:

```
add wave /#vsim_capacity#{assertions | classes | covergroups | qdas | solver}.  
[{{Count | Current | Peak | Time}}
```

For example to display the count for classes in the Wave window, enter the following:

```
add wave /#vsim_capacity#/classes.Count
```

Writing a Text-Based Report

To generate a text-based report of the capacity data, you can use the write report command:

```
write report -capacity [-l | -s] [-classes | -qdas | -assertions | -cvg | -solver]
```

When you specify `-s` or no other switch, it reports coarse-grain analysis. When you specify `-l`, it reports the fine-grain analysis.

For example:

```
write report -capacity -l
```

produces a report with the following format (all memory numbers are in bytes):

```
# write report -capacity -l  
# CAPACITY REPORT Generated on Wed Dec 31 16:00:00 1969  
#  
# Total Memory Allocated:3866920  
# TYPE: (COUNT, CURRENT MEM, PEAK MEM, PEAK MEM TIME)  
# Classes: (159, 16136, 16136, 4450 ns)  
#   /std::semaphore: (18, 504, 504, 1650 ns)  
#     verilog_src/std/std.sv(25): (10, 280, 280, 1650 ns)  
#     verilog_src/std/std.sv(27): (8, 224, 224, 1650 ns)  
#   /std::process: (8, 256, 320, 1658 ns)  
#     c:/dev/mainline/modeltech/win32/./verilog_src/std/std.sv: (8,  
256, 320, 1658 ns)  
#   /defs::Packet: (97, 12416, 12416, 4450 ns)  
#     src/test_router.sv(309): (97, 12416, 12416, 4450 ns)  
#   /test_router_sv_unit::scoreboard: (1, 168, 168, 1650 ns)
```

```

#           src/test_router.sv(355): (1, 168, 168, 1650 ns)
#   /test_router_sv_unit::monitor: (8, 1024, 1024, 1650 ns)
#           src/test_router.sv(362): (8, 1024, 1024, 1650 ns)
#   /test_router_sv_unit::driver: (8, 608, 608, 1650 ns)
#           src/test_router.sv(361): (8, 608, 608, 1650 ns)
#   /test_router_sv_unit::stimulus: (8, 416, 416, 1650 ns)
#           src/test_router.sv(360): (8, 416, 416, 1650 ns)
#   /test_router_sv_unit::test_env: (1, 144, 144, 1650 ns)
#           src/test_router.sv(408): (1, 144, 144, 1650 ns)
#   /std::mailbox::mailbox__1: (8, 480, 480, 1650 ns)
#           src/test_router.sv(363): (8, 480, 480, 1650 ns)
#   /std::mailbox::mailbox__1: (2, 120, 120, 1650 ns)
#           src/test_router.sv(353): (1, 60, 60, 1650 ns)
#           src/test_router.sv(354): (1, 60, 60, 1650 ns)
# QDAS: (310, 3078, 3091, 4450 ns)
#   Arrays: (300, 2170, 2183, 4450 ns)
#           src/test_router.sv(355): (1, 40, 40, 1650 ns)
#   c:/dev/mainline/modeltech/win32/./verilog_src/std/std.sv: (9,
117, 130, 1657 ns)
#           src/test_router.sv(309): (89, 1157, 1170, 4450 ns)
#           src/test_router.sv(355): (3, 26, 36, 1650 ns)
#           src/defs.sv(24): (97, 194, 194, 4450 ns)
#           src/test_router.sv(295): (91, 456, 458, 1657 ns)
#   <NOFILE>: (15, 26, 26, 4450 ns)
#           src/test_router.sv(386): (174, 300, 300, 4450 ns)
#           src/test_router.sv(351): (1, 32, 32, 1650 ns)
#           src/test_router.sv(348): (1, 32, 32, 1650 ns)
#           src/test_router.sv(349): (1, 32, 32, 1650 ns)
#           src/test_router.sv(350): (1, 32, 32, 1650 ns)

```

```
# Queues: (9, 888, 888, 1657 ns)
#       c:/dev/mainline/modeltech/win32/./verilog_src/std/std.sv(39):
#       (9, 888, 888, 1657 ns)
# Associative: (1, 20, 20, 1650 ns)
#       src/test_router.sv(355): (1, 20, 20, 1650 ns)
# Assertions/Cover Directives: (0, 0, 0, 0 ns)
# Covergroups: (3, 1696, 1696, 1650 ns)
#       /test_router_sv_unit::scoreboard::cov1: (3, 1696, 1696, 1650 ns)
# Solver: (97, 2072816, 2081036, 1651 ns)
#       src/test_router.sv(311): (97, 2891148, 2891148, 4450 ns)
```

Reporting Capacity Analysis Data From a UCDB File

By default, coarse-grain analysis data is saved into a UCDB file, along with the simulation coverage data using the coverage save command.

You can report this data from UCDB file using `vcover report` or the `vcover stats` commands in the following forms:

```
vcover report -memory <UCDB_filename>
```

```
vcover stats -memory <UCDB_filename>
```

Currently the fine-grain analysis data is not available from this report except as details related to covergroup memory usage. To report the covergroup memory usage details, you can use the `vcover report` command with the following arguments:

```
vcover report -cvg -details -memory
```

Example

```
vcover report -memory test.ucdb
```

```
COVERGROUP MEMORY USAGE: Total 13.3 KBytes, Peak 13.3 KBytes at time 0 ns
for total 4 coverpoints/crosses.
```

```
ASSERT/COVER MEMORY USAGE: Total Memory 0 Bytes.
```

```
CONSTRAINT SOLVER MEMORY USAGE: Total 1.1 MBytes, Peak 1.1 MBytes at time
0 ns for total 100 randomize() calls.
```

```
CLASS OBJECTS MEMORY USAGE: Total Memory 68 Bytes and Peak Memory 68 Bytes
used at time 0 ns for total 1 class objects.
```

DYNAMIC OBJECTS MEMORY USAGE: Total Memory 35 Bytes and Peak Memory 35 Bytes used at time 0 ns for total 2 dynamic objects.

Chapter 28

Signal Spy

The Verilog language allows access to any signal from any other hierarchical block without having to route it through the interface. This means you can use hierarchical notation to either write or read the value of a signal in the design hierarchy from a test bench. Verilog can also reference a signal in a VHDL block or reference a signal in a Verilog block through a level of VHDL hierarchy.

With the VHDL-2008 standard, VHDL supports hierarchical referencing as well. However, you cannot reference from VHDL to Verilog. The Signal Spy procedures and system tasks provide hierarchical referencing across any mix of Verilog, VHDL and/or SystemC, allowing you to monitor (spy), drive, force, or release hierarchical objects in mixed designs. While not strictly required for references beginning in Verilog, it does allow references to be consistent across all languages.

Signal Spy procedures for VHDL are provided in the [VHDL Utilities Package \(util\)](#) within the *modelsim_lib* library. To access these procedures, you would add lines like the following to your VHDL code:

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

The Verilog tasks and SystemC functions are available as built-in [System Tasks and Functions](#).

Table 28-1. Signal Spy Reference Comparison

Refer to:	VHDL procedures	Verilog system tasks	SystemC function
disable_signal_spy	disable_signal_spy()	\$disable_signal_spy()	disable_signal_spy()
enable_signal_spy	enable_signal_spy()	\$enable_signal_spy()	enable_signal_spy()
init_signal_driver	init_signal_driver()	\$init_signal_driver()	init_signal_driver()
init_signal_spy	init_signal_spy()	\$init_signal_spy()	init_signal_spy()
signal_force	signal_force()	\$signal_force()	signal_force()
signal_release	signal_release()	\$signal_release()	signal_release()

Designed for Test Benches

Note that using Signal Spy procedures limits the portability of your code—HDL code with Signal Spy procedures or tasks works only in Questa and Modelsim. Consequently, you should use Signal Spy only in test benches, where portability is less of a concern and the need for such procedures and tasks is more applicable.

Signal Spy Formatting Syntax

Strings that you pass to Signal Spy commands are not language-specific and should be formatted as if you were referring to the object from the command line of the simulator. Thus, you use the simulator's path separator. For example, the Verilog LRM specifies that a Verilog hierarchical reference to an object always has a period (.) as the hierarchical separator, but the reference does not begin with a period.

Signal Spy Supported Types

Signal Spy supports the following SystemVerilog types and user-defined SystemC types.

- SystemVerilog types
 - All scalar and integer SV types (bit, logic, int, shortint, longint, integer, byte, both signed and unsigned variations of these types)
 - Real and Shortreal
 - User defined types (packed/unpacked structures including nested structures, packed/unpacked unions, enums)
 - Arrays and Multi-D arrays of all supported types.
- SystemC types
 - Primitive C floating point types (double, float)
 - User defined types (structures including nested structures, unions, enums)

Cross-language type-checks and mappings are included to support these types across all the possible language combinations:

- SystemC-SystemVerilog
- SystemC-SystemC
- SystemC-VHDL
- VHDL-SystemVerilog
- SystemVerilog-SystemVerilog

In addition to referring to the complete signal, you can also address the bit-selects, field-selects and part-selects of the supported types. For example:

```
/top/myInst/my_record[2].my_field1[4].my_vector[8]
```

disable_signal_spy

This reference section describes the following:

- VHDL Procedure — `disable_signal_spy()`
- Verilog Task — `$disable_signal_spy()`
- SystemC Function— `disable_signal_spy()`

The `disable_signal_spy` call disables the associated `init_signal_spy`. The association between the `disable_signal_spy` call and the `init_signal_spy` call is based on specifying the same *src_object* and *dest_object* arguments to both. The `disable_signal_spy` call can only affect `init_signal_spy` calls that had their *control_state* argument set to "0" or "1".

By default this command uses a forward slash (/) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the *modelsim.ini* file.

VHDL Syntax

```
disable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

Verilog Syntax

```
$disable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

SystemC Syntax

```
disable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

Returns

Nothing

Arguments

- `src_object`
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the `init_signal_spy` call that you want to disable.
- `dest_object`
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the `init_signal_spy` call that you want to disable.
- `verbose`
Optional integer. Specifies whether you want a message reported in the transcript stating that a disable occurred and the simulation time that it occurred.

0 — Does not report a message. Default.

1 — Reports a message.

Related procedures

[init_signal_spy](#), [enable_signal_spy](#)

Example

See [init_signal_spy Example](#) or [\\$init_signal_spy Example](#)

enable_signal_spy

This reference section describes the following:

- VHDL Procedure — enable_signal_spy()
- Verilog Task — \$enable_signal_spy()
- SystemC Function— enable_signal_spy()

The enable_signal_spy() call enables the associated init_signal_spy call. The association between the enable_signal_spy call and the init_signal_spy call is based on specifying the same src_object and dest_object arguments to both. The enable_signal_spy call can only affect init_signal_spy calls that had their control_state argument set to "0" or "1".

By default this command uses a forward slash (/) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the *modelsim.ini* file.

VHDL Syntax

```
enable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

Verilog Syntax

```
$enable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

SystemC Syntax

```
enable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

Returns

Nothing

Arguments

- src_object
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the init_signal_spy call that you want to enable.
- dest_object
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the init_signal_spy call that you want to enable.
- verbose
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that an enable occurred and the simulation time that it occurred.
0 — Does not report a message. Default.

1 — Reports a message.

Related tasks

[init_signal_spy](#), [disable_signal_spy](#)

Example

See [\\$init_signal_spy Example](#) or [init_signal_spy Example](#)

init_signal_driver

This reference section describes the following:

- VHDL Procedure — `init_signal_driver()`
- Verilog Task — `$init_signal_driver()`
- SystemC Function— `init_signal_driver()`

The `init_signal_driver()` call drives the value of a VHDL signal, Verilog net, or SystemC (called the `src_object`) onto an existing VHDL signal or Verilog net (called the `dest_object`). This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (for example, a test bench).

Note



Destination SystemC signals are not supported.

The `init_signal_driver` procedure drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the `init_signal_driver` value in the resolution of the signal.

By default this command uses a forward slash (/) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the `modelsim.ini` file.

Call only once

The `init_signal_driver` procedure creates a persistent relationship between the source and destination signals. Hence, you need to call `init_signal_driver` only once for a particular pair of signals. Once `init_signal_driver` is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

For VHDL, you should place all `init_signal_driver` calls in a VHDL process and code this VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only `init_signal_driver` calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

For Verilog, you should place all `$init_signal_driver` calls in a Verilog initial block. See the example below.

VHDL Syntax

```
init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)
```

Verilog Syntax

```
$init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)
```

SystemC Syntax

`init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)`

Returns

Nothing

Arguments

- **src_object**
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, Verilog net, or SystemC signal. Use the path separator to which your simulation is set (for example, "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **dest_object**
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (for example, "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **delay**
Optional time value. Specifies a delay relative to the time at which the `src_object` changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
- **delay_type**
Optional `del_mode` or integer. Specifies the type of delay that will be applied.
For the VHDL `init_signal_driver` Procedure, The value must be either:
 - `mti_inertial` (default)
 - `mti_transport`For the Verilog `$init_signal_driver` Task, The value must be either:
 - 0 — inertial (default)
 - 1 — transportFor the SystemC `init_signal_driver` Function, The value must be either:
 - 0 — inertial (default)
 - 1 — transport
- **verbose**
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the `src_object` is driving the `dest_object`.
 - 0 — Does not report a message. Default.
 - 1 — Reports a message.

Related procedures

[init_signal_spy](#), [signal_force](#), [signal_release](#)

Limitations

- For the VHDL `init_signal_driver` procedure, when driving a Verilog net, the only *delay_type* allowed is `inertial`. If you set the delay type to `mti_transport`, the setting will be ignored and the delay type will be `mti_inertial`.
- For the Verilog `$init_signal_driver` task, when driving a Verilog net, the only *delay_type* allowed is `inertial`. If you set the delay type to `1` (transport), the setting will be ignored, and the delay type will be `inertial`.
- For the SystemC `init_signal_driver` function, when driving a Verilog net, the only *delay_type* allowed is `inertial`. If you set the delay type to `1` (transport), the setting will be ignored, and the delay type will be `inertial`.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.
- Verilog memories (arrays of registers) are not supported.

`$init_signal_driver` Example

This example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The `.../blk1/clk` will match local *clk0* and a message will be displayed. The `.../blk2/clk` will match the local *clk0* but be delayed by 100 ps. For the second call to work, the `.../blk2/clk` must be a VHDL based signal, because if it were a Verilog net a 100 ps inertial delay would consume the 40 ps clock period. Verilog nets are limited to only inertial delays and thus the setting of `1` (transport delay) would be ignored.

```
`timescale 1 ps / 1 ps

module testbench;

reg clk0;

initial begin
    clk0 = 1;
    forever begin
        #20 clk0 = ~clk0;
    end
end

initial begin
    $init_signal_driver("clk0", "/testbench/uut/blk1/clk", , , 1);
    $init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100, 1);
end

...

endmodule
```

init_signal_driver Example

This example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay_type while setting the verbose parameter to a 1. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps.

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
    signal clk0 : std_logic;
begin
    gen_clk0 : process
    begin
        clk0 <= '1' after 0 ps, '0' after 20 ps;
        wait for 40 ps;
    end process gen_clk0;

    drive_sig_process : process
    begin
        init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
        init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps,
            mti_transport);

        wait;
    end process drive_sig_process;
    ...
end;
```

init_signal_spy

This reference section describes the following:

- VHDL Procedure — `init_signal_spy()`
- Verilog Task — `$init_signal_spy()`
- SystemC Function— `init_signal_spy()`

The `init_signal_spy()` call mirrors the value of a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal (called the `src_object`) onto an existing VHDL signal, Verilog register, or SystemC signal (called the `dest_object`). This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture or Verilog or SystemC module (for example, a test bench).

The `init_signal_spy` call only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value that was set by `init_signal_spy`.

By default this command uses a forward slash (/) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the `modelsim.ini` file.

Call only once

The `init_signal_spy` call creates a persistent relationship between the source and destination signals. Hence, you need to call `init_signal_spy` once for a particular pair of signals. Once `init_signal_spy` is called, any change on the source signal will mirror on the destination signal until the end of the simulation unless the `control_state` is set.

However, you can place simultaneous read/write calls on the same signal using multiple `init_signal_spy` calls, for example:

```
init_signal_spy ("/sc_top/sc_sig", "/top/hdl_INST/hdl_sig");  
init_signal_spy ("/top/hdl_INST/hdl_sig", "/sc_top/sc_sig");
```

The `control_state` determines whether the mirroring of values can be enabled/disabled and what the initial state is. Subsequent control of whether the mirroring of values is enabled/disabled is handled by the `enable_signal_spy` and `disable_signal_spy` calls.

For VHDL procedures, you should place all `init_signal_spy` calls in a VHDL process and code this VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only `init_signal_spy` calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

For Verilog tasks, you should place all `$init_signal_spy` tasks in a Verilog initial block. See the example below.

VHDL Syntax

```
init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)
```

Verilog Syntax

`$init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)`

SystemC Syntax

`init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)`

Returns

Nothing

Arguments

- **src_object**
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or SystemVerilog or Verilog register/net. Use the path separator to which your simulation is set (for example, "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **dest_object**
Required string. A full hierarchical path (or relative downward 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 (for example, "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **verbose**
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the `src_object`'s value is mirrored onto the `dest_object`.
 - 0 — Does not report a message. Default.
 - 1 — Reports a message.
- **control_state**
Optional integer. Possible values are -1, 0, or 1. Specifies whether or not you want the ability to enable/disable mirroring of values and, if so, specifies the initial state.
 - 1 — no ability to enable/disable and mirroring is enabled. (default)
 - 0 — turns on the ability to enable/disable and initially disables mirroring.
 - 1 — turns on the ability to enable/disable and initially enables mirroring.

Related procedures

[init_signal_driver](#), [signal_force](#), [signal_release](#), [enable_signal_spy](#), [disable_signal_spy](#)

Limitations

- When mirroring the value of a SystemVerilog or Verilog register/net onto a VHDL signal, the VHDL signal must be of type `bit`, `bit_vector`, `std_logic`, or `std_logic_vector`.
- Verilog memories (arrays of registers) are not supported.

init_signal_spy Example

In this example, the value of */top/uut/inst1/sig1* is mirrored onto */top/top_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the `init_signal_spy` is initially enabled.

The mirroring of values will be disabled when `enable_sig` transitions to a '0' and enable when `enable_sig` transitions to a '1'.

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use modelsim_lib.util.all;
entity top is
end;
architecture only of top is
    signal top_sig1 : std_logic;
begin
    ...
    spy_process : process
    begin
        init_signal_spy("/top/uut/inst1/sig1", "/top/top_sig1", 1, 1);
        wait;
    end process spy_process;
    ...
    spy_enable_disable : process(enable_sig)
    begin
        if (enable_sig = '1') then
            enable_signal_spy("/top/uut/inst1/sig1", "/top/top_sig1", 0);
        elsif (enable_sig = '0')
            disable_signal_spy("/top/uut/inst1/sig1", "/top/top_sig1", 0);
        end if;
    end process spy_enable_disable;
    ...
end;
```

\$init_signal_spy Example

In this example, the value of *.top.uut.inst1.sig1* is mirrored onto *.top.top_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the `init_signal_spy` is initially enabled.

The mirroring of values will be disabled when `enable_reg` transitions to a '0' and enabled when `enable_reg` transitions to a '1'.

```
module top;
    ...
    reg top_sig1;
    reg enable_reg;
    ...
    initial
    begin
        $init_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 1, 1);
    end
```

Signal Spy **init_signal_spy**

```
    always @ (posedge enable_reg)
    begin
    $enable_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 0);
    end
    always @ (negedge enable_reg)
    begin
    $disable_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 0);
    end
    ...
endmodule
```

signal_force

This reference section describes the following:

- VHDL Procedure — `signal_force()`
- Verilog Task — `$signal_force()`
- SystemC Function— `signal_force()`

The `signal_force()` call forces the value specified onto an existing VHDL signal, Verilog register or net, or SystemC signal (called the `dest_object`). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (for example, a test bench).

A `signal_force` works the same as the `force` command with the exception that you cannot issue a repeating force. The force will remain on the signal until a `signal_release`, a force or release command, or a subsequent `signal_force` is issued. `Signal_force` can be called concurrently or sequentially in a process.

This command displays any signals using your `radix` setting (either the default, or as you specify) unless you specify the radix in the *value* you set.

By default this command uses a forward slash (/) as a path separator. You can change this behavior with the `SignalSpyPathSeparator` variable in the `modelsim.ini` file.

VHDL Syntax

```
signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>, <verbose>)
```

Verilog Syntax

```
$signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>,  
<verbose>)
```

SystemC Syntax

```
signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>, <verbose>)
```

Returns

Nothing

Arguments

- `dest_object`
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal, SystemVerilog or Verilog register/net or SystemC signal. Use the path separator to which your simulation is set (for example, "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- value

Required string. Specifies the value to which the `dest_object` is to be forced. The specified value must be appropriate for the type.

Where *value* can be:

- a sequence of character literals or as a based number with a radix of 2, 8, 10 or 16. For example, the following values are equivalent for a signal of type `bit_vector (0 to 3)`:
 - 1111 — character literal sequence
 - 2#1111 — binary radix
 - 10#15 — decimal radix
 - 16#F — hexadecimal radix
- a reference to a Verilog object by name. This is a direct reference or hierarchical reference, and is not enclosed in quotation marks. The syntax for this named object should follow standard Verilog syntax rules.

- rel_time

Optional time. Specifies a time relative to the current simulation time for the force to occur. The default is 0.

- force_type

Optional forcetype or integer. Specifies the type of force that will be applied.

For the VHDL procedure, the value must be one of the following;

default — which is "freeze" for unresolved objects or "drive" for resolved objects
deposit
drive
freeze

For the Verilog task, the value must be one of the following;

0 — default, which is "freeze" for unresolved objects or "drive" for resolved objects
1 — deposit
2 — drive
3 — freeze

For the SystemC function, the value must be one of the following;

0 — default, which is "freeze" for unresolved objects or "drive" for resolved objects
1 — deposit
2 — drive
3 — freeze

See the force command for further details on force type.

- **cancel_period**

Optional time or integer. Cancels the `signal_force` command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit.

For the VHDL procedure, a value of zero cancels the force at the end of the current time period. Default is -1 ms. A negative value means that the force will not be cancelled.

For the Verilog task, A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.

For the SystemC function, A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.

- **verbose**

Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the `dest_object` at the specified time.

0 — Does not report a message. Default.

1 — Reports a message.

Related procedures

[init_signal_driver](#), [init_signal_spy](#), [signal_release](#)

Limitations

- You cannot force bits or slices of a register; you can force only the entire register.
- Verilog memories (arrays of registers) are not supported.

\$signal_force Example

This example forces `reset` to a "1" from time 0 ns to 40 ns. At 40 ns, `reset` is forced to a "0", 200000 ns after the second `$signal_force` call was executed.

```
`timescale 1 ns / 1 ns

module testbench;

initial
begin
    $signal_force("/testbench/uut/blk1/reset", "1", 0, 3, , 1);
    $signal_force("/testbench/uut/blk1/reset", "0", 40, 3, 200000, 1);
end

...

endmodule
```

signal_force Example

This example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 2 ms after the second *signal_force* call was executed.

If you want to skip parameters so that you can specify subsequent parameters, you need to use the keyword "open" as a placeholder for the skipped parameter(s). The first *signal_force* procedure illustrates this, where an "open" for the *cancel_period* parameter means that the default value of -1 ms is used.

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
begin

    force_process : process
    begin
        signal_force("/testbench/uut/blk1/reset", "1", 0 ns, freeze, open, 1);
        signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms,
1);
        wait;
    end process force_process;

    ...

end;
```

signal_release

This reference section describes the following:

- VHDL Procedure — `signal_release()`
- Verilog Task — `$signal_release()`
- SystemC Function— `signal_release()`

The `signal_release()` call releases any force that was applied to an existing VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal (called the `dest_object`). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (for example, a test bench).

A `signal_release` works the same as the `noforce` command. `Signal_release` can be called concurrently or sequentially in a process.

By default this command uses a forward slash (/) as a path separator. You can change this behavior with the `SignalSpyPathSeparator` variable in the `modelsim.ini` file.

VHDL Syntax

```
signal_release(<dest_object>, <verbose>)
```

Verilog Syntax

```
$signal_release(<dest_object>, <verbose>)
```

SystemC Syntax

```
signal_release(<dest_object>, <verbose>)
```

Returns

Nothing

Arguments

- `dest_object`
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. Use the path separator to which your simulation is set (for example, "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- `verbose`
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release.
 - 0 — Does not report a message. Default.
 - 1 — Reports a message.

Related procedures

[init_signal_driver](#), [init_signal_spy](#), [signal_force](#)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

signal_release Example

This example releases any forces on the signals *data* and *clk* when the signal *release_flag* is a "1". Both calls will send a message to the transcript stating which signal was released and when.

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is

    signal release_flag : std_logic;

begin

    stim_design : process
    begin
        ...
        wait until release_flag = '1';
        signal_release("/testbench/dut/blk1/data", 1);
        signal_release("/testbench/dut/blk1/clk", 1);
        ...
    end process stim_design;

    ...

end;
```

\$signal_release Example

This example releases any forces on the signals *data* and *clk* when the register *release_flag* transitions to a "1". Both calls will send a message to the transcript stating which signal was released and when.

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
    $signal_release("/testbench/dut/blk1/data", 1);
    $signal_release("/testbench/dut/blk1/clk", 1);
end

...

endmodule
```

Chapter 29

Monitoring Simulations with JobSpy

This chapter describes JobSpy™, a tool for monitoring and controlling batch simulations and simulation farms.

Designers frequently run multiple simulation jobs in batch mode once verification reaches the regression testing stage. They face the problem that simulation farms and batch-mode runs offer little visibility into and control over simulation jobs. JobSpy helps alleviate this problem by allowing you to interact with batch jobs. By creating a process external to the running simulator, JobSpy can send and receive information about the running jobs.

Some applications of JobSpy include the following:

- Checking the progress of a simulation.
- Examining internal signal values to check if the design is functioning correctly, without stopping the simulation.
- Suspending one job to release a license for a more important job, also allowing you to restart the suspended job later.
- Instructing the running batch job to do a checkpoint of the job and then continue the run. If the workstation that was running a batch job were to fail at sometime in the future, you would could restart the job again from the saved checkpoint file.

You can run JobSpy from the command line, from within the ModelSim GUI, or from a standalone GUI. The actual commands that are sent and received across the communication pipe are the same for all modes of operation. The standalone GUI simply provides a dialog box where you can see all the running jobs.

Basic JobSpy Flow

The basic steps for setting up and using JobSpy are as follows:

1. Set JOBSPY_DAEMON environment variable.
 - port@host— Refer to the section “[Starting the JobSpy Daemon](#)”
 - directory — Refer to the section “[Setting the JOBSPY_DAEMON Variable as a Directory](#)”
2. Start JobSpy daemon.
 - Command line: **jobspsy -startd**

You do not need to specify `-startd` if you set the `JOBSPY_DAEMON` to a directory.

- GUI: **Tools > JobSpy > Daemon > Start Daemon**
3. Start simulation jobs as you normally would. The tool will communicate with the JobSpy daemon through the use of the `JOBSPY_DAEMON` environment variable.
 4. Use **jobspsy** command or Job Manager GUI to monitor results.

Starting the JobSpy Daemon

You must start the JobSpy daemon prior to launching any simulation jobs. The daemon tracks jobs by setting up a communication pipe with each running simulation. When a simulation job starts, the daemon opens a TCP/IP port for the job and then records to a file:

- port number
- host name that the job was started on
- working directory

With a connection to the job established, you can invoke various commands via the command line or GUI to monitor or control the job. There are two steps to starting the daemon:

1. Set the `JOBSPY_DAEMON` environment variable.

The environment variable is set with the following syntax:

```
JOBSPY_DAEMON=<port_NUMBER>@<host>
```

For example,

```
JOBSPY_DAEMON=1301@mymachine
```

Every user who will run JobSpy must set this environment variable. You will typically set this in a start-up script, such as your `.cshrc` file, so that every new shell has access to the daemon.

2. Invoke the daemon using the **jobspsy -startd** command or by selecting **Tools > JobSpy > Daemon > Start Daemon** from within ModelSim.

You do not need to specify `-startd` if you set the `JOBSPY_DAEMON` to a directory.

Any person who knows what `port@host` to set their `JOBSPY_DAEMON` variable to can control jobs submitted to that host. The intended use is that a person would set their `JOBSPY_DAEMON` variable, start the daemon, and then only they could control their jobs (unless they told somebody what `port@host` to use). Each user can use his/her own port id to monitor only their jobs.

Setting the JOBSPY_DAEMON Variable as a Directory

As an alternative to using a TCP/IP port, you can instruct the JobSpy Daemon to communicate with simulation jobs via a directory and file structure. Although a directory location is not technically a Daemon, for ease of use we will be referring to it as one in this document.

To specify a directory as your JobSpy Daemon, you would use the JOBSPY_DAEMON environment variable similar to the following:

```
JOBSPY_DAEMON=/server/directory/subdirectory
```

This instructs any simulation job invoked with the same \$JOBSPY_DAEMON to create files containing communication and run information in the specified directory, which enables communication between JobSpy and the simulation jobs.

The jobspy command behaves similarly regardless of your using a TCP/IP port or a directory name for your JobSpy Daemon.

Running JobSpy from the Command Line

The JobSpy command-line interface is accessible from a shell prompt or within the ModelSim GUI, where the syntax is:

```
jobspy [-gui] [-killid] [-startd] | jobs | status | <jobid> <command>
```

See the [jobspy](#) command for complete syntax. The most common invocations are:

- **jobspy -startd** — invokes the daemon
 You do not need to specify -startd if you set the JOBSPY_DAEMON to a directory.
- **jobspy jobs** — lists all jobs and their id numbers; you need the ids in order to execute commands on the jobs
- **<jobid> <command>** — allows you to issue commands to a job; only certain commands can be used, as noted below

Simulation Commands Available to JobSpy

You can perform a select number of simulator commands on jobs via JobSpy. The table below lists the available commands with a brief description.

Table 29-1. Simulation Commands You can Issue from JobSpy

Command	Description
stop	stops a simulation
go	resumes a stopped job

Table 29-1. Simulation Commands You can Issue from JobSpy

Command	Description
checkpoint or check	checkpoints a simulation
savewlf	saves simulation results to a WLF file; see Viewing Results During Active Simulation ; by default this command uses the pathname from the remote machine
examine	prints the value of a signal in the remote job
force	forces signal values in the remote job
log	logs signals in the waveform log file (.wlf)
nolog	removes logged signals from the waveform log file (.wlf)
now	prints job's current simulation time
profile on	enable profiling of remote job
profile off	disable profiling of remote job
profile save [<filename>]	save a profile of remote job. Default <filename> is <i>job<jobid>.prof</i>
pwd	prints the job's current working directory
quit	exits a simulation (terminates job)
savecov [<filename>]	writes out a coverage data UCDB file, equivalent to the coverage save command. Default <filename> is <i>Job_<gridtype>_<jobid>.ucdb</i> where <gridtype> is mti, sge, lsf or vov.
set	sets a TCL variable in the remote job's interpreter
simstatus	shows current status of the simulation
suspend	suspends job (releases license)
unsuspend	un-suspends job (reacquires license)

Example Session

The following example illustrates a session of JobSpy:

```

$ JOBSPY_DAEMON=1300@time //sets the daemon to a port@host
$ export JOBSPY_DAEMON //exports the environment variable

$ jobspy -startd //start the daemon

$ jobspy jobs // print list of jobs
JobID Type Sim Status Sim Time user Host PID Start Time Directory
5 mti Running 1,200ns alla time 24710 Mon Dec 27. /u/alla/z
11 mti Running 3,433ns mcar larg 24915 Tue Dec 28. /u/mcar/x

$ jobspy 11 checkpoint //checkpoint job 11

```

```
Checkpointing Job

$ jobspy 11 cont //resume job 11
continuing

$ jobspy 5 savewlf snap.wlf // saving waveforms from job 5
Dataset "sim" exported as WLF file: snap.wlf. @ 1,200ns

$ vsim -view snap.wlf // viewing waveforms from job 5
```

Running the JobSpy GUI

JobSpy includes a GUI called Job Manager that you can invoke from within ModelSim or separately as a stand-alone tool. The Job Manager shows all active simulations in real time and provides convenient access to JobSpy commands.

Starting Job Manager

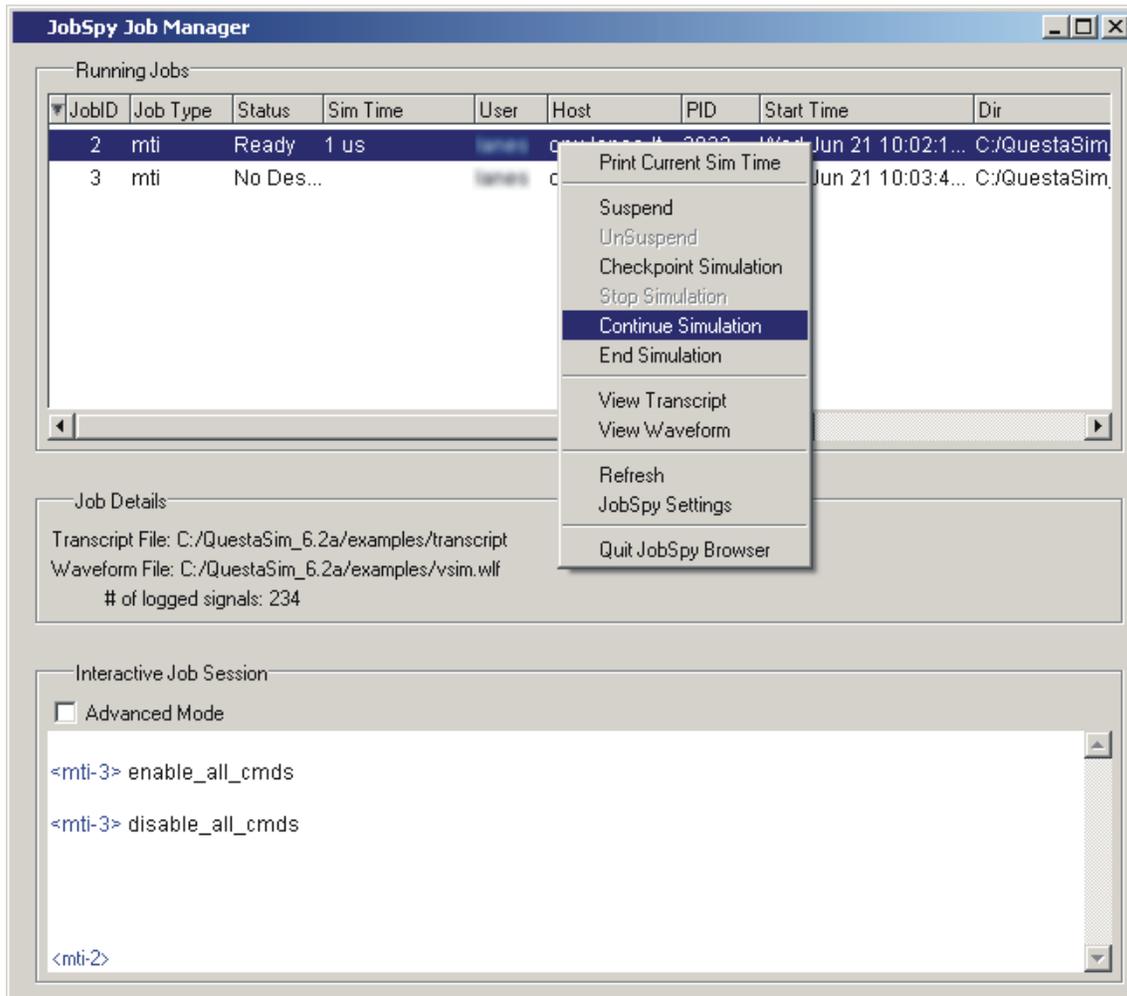
You can start Job Manager from a shell prompt or from within ModelSim:

- Shell Prompt — **jobspy -gui**
- GUI— **Tools > JobSpy > JobSpy Job Manager**

Invoking Simulation Commands in Job Manager

You can invoke simulation commands in Job Manager via a right-click menu or from the prompt in the Interactive Job Session window. Right-click a job in the list to access menu commands:

Figure 29-1. JobSpy Job Manager



Interactive Job Session Pane

The Interactive Job Session pane provides a command line for interacting with jobs. Commands you enter affect the job currently selected in the Running Jobs portion of the dialog box. See [Table 29-1](#) for a list of commands you can enter in the Interactive Job Session pane.

Note



If you check Advanced Mode, you can enter any ModelSim command at the prompt. However, you need to be careful as many ModelSim commands will not function properly with JobSpy.

View Commands and Pathnames

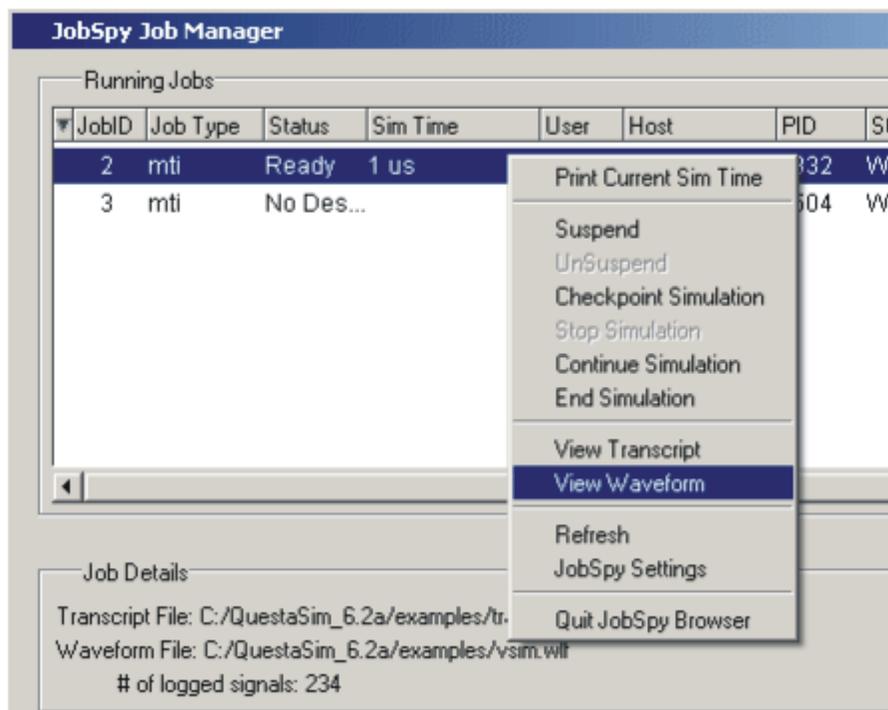
The **View Transcript** and **View Waveform** commands display files (*transcript* and *<name>.wlf*, respectively) that are output by the simulator. These commands use the pathname

from the remote machine to locate the required file. Depending on how your network is organized, the pathname may be different or inaccessible from the machine which is running JobSpy. In such cases, these commands will not work. The work around is to use **jobspy savewlf** to specify a known location for the WLF file or **cp** to copy the Transcript file to a known location.

Viewing Results During Active Simulation

You may want to check simulation results while your simulation jobs are still running. You can do this via the GUI or by using the **savewlf** command. To view waveforms from the JobSpy GUI, right-click a job and select **View Waveform**.

Figure 29-2. Job Manager View Waveform



Here are two important points to remember about viewing waveforms from the GUI:

- You must first log signals before you can view them as waveforms. If you haven't logged any signals, the View Waveform command in the GUI will be disabled.
- View Waveform uses the pathname from the remote machine to access a WLF file. The command may not work on some networks. See [View Commands and Pathnames](#) for details.

Viewing Waveforms from the Command Line

From the command line, there are three steps to viewing waveforms:

1. Log the appropriate signals

```
add log *
```

2. Save a dataset

```
$ jobspy 1204 savewlf snap.wlf  
Dataset "sim" exported as WLF file: snap.wlf. @ 84,785,547 ns
```

3. View the dataset

```
vsim -view snap.wlf
```

Licensing and Job Suspension

When you suspend a job via JobSpy, the simulation license is released by default. You can change this behavior by modifying the `MTI_RELEASE_ON_SUSPEND` environment variable. By default the variable is set to 10 (in seconds), which releases the license 10 seconds after receiving a suspend signal. If you change the value to 0 (off), simulation licenses will not be released upon job suspension.

Checkpointing Jobs

Checkpointing allows you to save the state of a simulation and restore it at a later time. There are three primary reasons for checkpointing jobs:

- Free up a license for a more important job
- Migrate a job from one machine to another
- Backup a job in case of a hardware crash or failure

In the case of freeing up a license, you should use the `suspend` command instead. Job suspension does not have the restrictions that checkpointing does.

If you need to checkpoint a job for migration or backup, keep in mind the following restrictions:

- The job must be restored on the same platform and exact OS on which the job was checkpointed.
- If your job includes any foreign C code (such as PLI or FLI), the foreign application must be written to support checkpointing. See [The PLI Callback reason Argument](#) for more information on checkpointing with PLI applications. See the Foreign Language Interface Reference Manual for information on checkpointing with FLI applications.
- Checkpoint is not supported once a SystemC design has been loaded.

Connecting to Load-Sharing Software

Load-sharing software, such as Platform Computing's LSF or Sun's Grid Engine, centralize management of distributed computing resources. JobSpy can access and monitor simulation runs that were submitted to these load-managing products.

With the exception of checkpointing (discussed below), the only requirement for connecting to load-sharing software is that the JobSpy daemon be running prior to submitting the jobs. If the daemon is running, the jobs will show up in JobSpy automatically.

JobSpy supports Sun Grid Engine's task arrays, where the simulation jobs use the `JOB_ID` and the `SGE_TASK_ID` environment variables. The `jobspsy` command can reference these jobs as "`<taskId>.<jobId>`".

Checkpointing with Load-Sharing Software

Some additional steps are required to configure load-sharing software for checkpointing **vsim** jobs. The configuration depends on which load-sharing software you run.

Configuring LSF for Checkpointing

Do the following to enable **vsim** checkpointing with LSF:

- Set the environment variable `LSB_ECHKPNT_METHOD_DIR` to point to `<install_dir>/modeltech/<platform>`
`<platform>` refers to the VCO for the ModelSim installation (for example, `linux`, `sunos5`, and so forth). See the *Installation Guide* for a complete list.
- Set the environment variable `LSB_ECHKPNT_METHOD` to "modelsim"

With these environment variables set, you can use standard LSF commands to checkpoint **vsim** jobs. Consult LSF documentation for information on those commands.

Configuring Flowtracer for Checkpointing

Flowtracer does not support checkpointing of **vsim** jobs.

Configuring Grid Engine for Checkpointing

To checkpoint **vsim** jobs with Grid Engine, you must create a Checkpoint Object, taking note of the following settings:

- Set Interface to:
`APPLICAITON-LEVEL.`
- Set the Checkpoint Command field to:

```
<install_dir>/modeltech/<platform>/jobspy -check
```

where <platform> refers to the VCO for the ModelSim installation (for example, linux, sunos5, and so forth). See the *Installation Guide* for a complete list.

- Set the Migration Command field to:

```
<install_dir>/modeltech/<platform>/jobspy -check k
```

Consult the Grid Engine documentation for additional information.

Chapter 30

Generating Stimulus with Waveform Editor

The ModelSim Waveform Editor offers a simple method for creating design stimulus. You can generate and edit waveforms in a graphical manner and then drive the simulation with those waveforms. With Waveform Editor you can do the following:

- Create waveforms using four predefined patterns: clock, random, repeater, and counter. See [Creating Waveforms from Patterns](#).
- Edit waveforms with numerous functions including inserting, deleting, and stretching edges; mirroring, inverting, and copying waveform sections; and changing waveform values on-the-fly. See [Editing Waveforms](#).
- Drive the simulation directly from the created waveforms
- Save created waveforms to four stimulus file formats: Tcl force format, extended VCD format, Verilog module, or VHDL architecture. The HDL formats include code that matches the created waveforms and can be used in test benches to drive a simulation. See [Exporting Waveforms to a Stimulus File](#)

Limitations

The current version does not support the following:

- Enumerated signals, records, multi-dimensional arrays, and memories
- User-defined types
- SystemC or SystemVerilog

Getting Started with the Waveform Editor

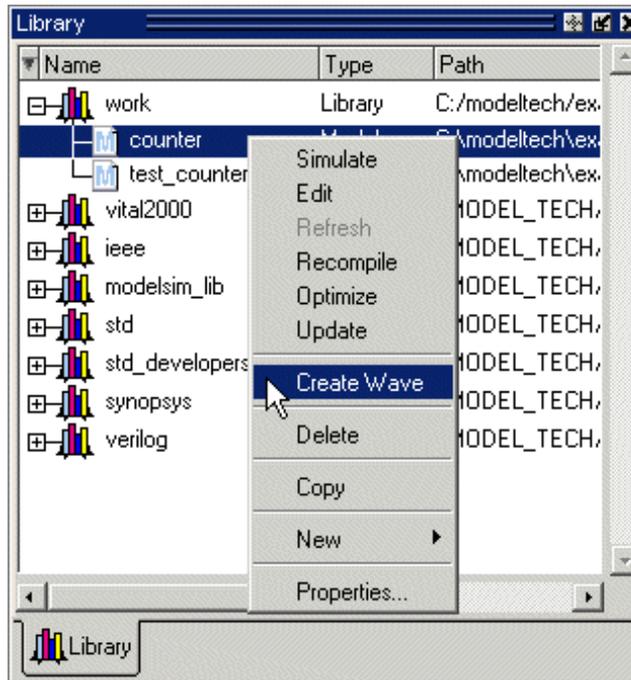
You can use Waveform Editor before or after loading a design. Regardless of which method you choose, you will select design objects and use them as the basis for created waveforms.

Using Waveform Editor Prior to Loading a Design

Here are the basic steps for using waveform editor prior to loading a design:

1. Right-click a design unit on the Library Window and select Create Wave.

Figure 30-1. Waveform Editor: Library Window



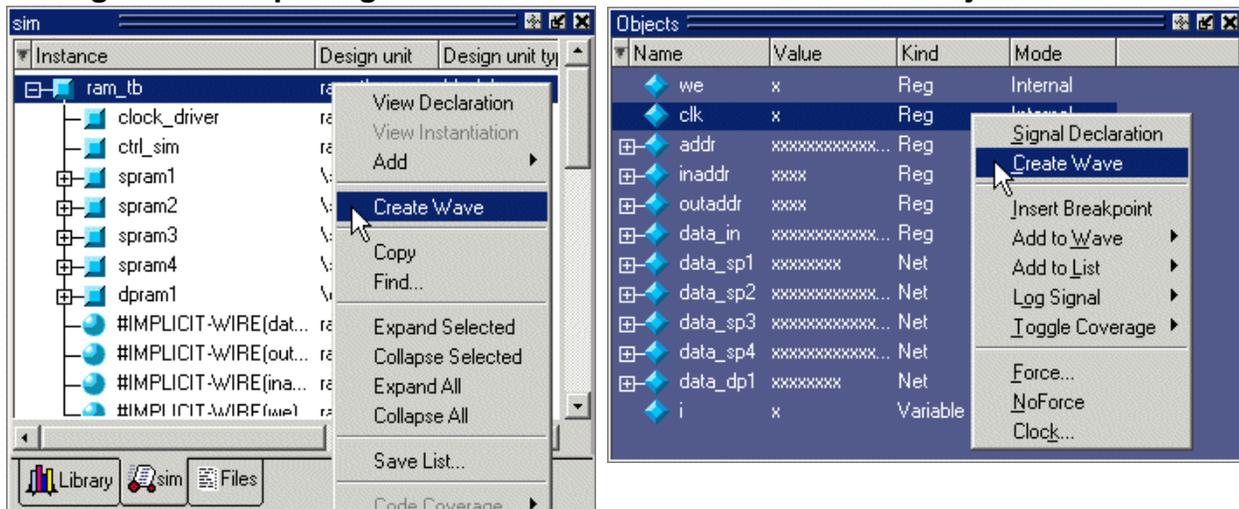
2. Edit the waveforms in the Wave window. See [Editing Waveforms](#) for more details.
3. Run the simulation (see [Simulating Directly from Waveform Editor](#)) or save the created waveforms to a stimulus file (see [Exporting Waveforms to a Stimulus File](#)).

Using Waveform Editor After Loading a Design

Here are the basic steps for using waveform editor after loading a design:

1. Right-click a block in the structure window or an object in the Object pane and select **Create Wave**.

Figure 30-2. Opening Waveform Editor from Structure or Objects Windows



2. Use the Create Pattern wizard to create the waveforms (see [Creating Waveforms from Patterns](#)).
3. Edit the waveforms as required (see [Editing Waveforms](#)).
4. Run the simulation (see [Simulating Directly from Waveform Editor](#)) or save the created waveforms to a stimulus file (see [Exporting Waveforms to a Stimulus File](#)).

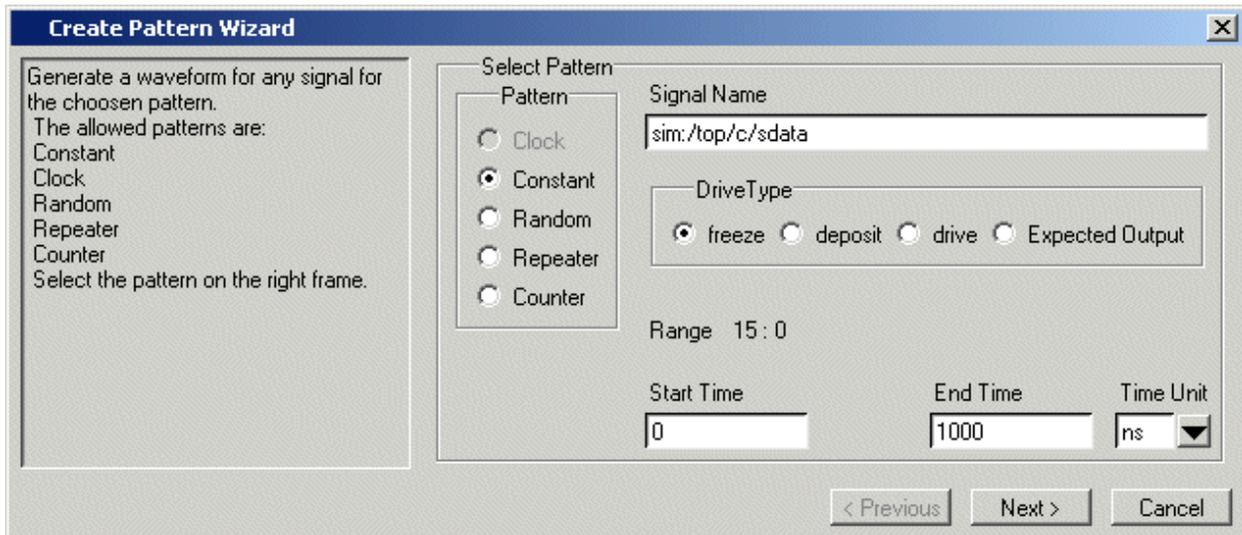
Creating Waveforms from Patterns

Waveform Editor includes a Create Pattern wizard that walks you through the process of creating waveforms. To access the wizard:

- Right-click an object in the Objects pane or structure pane (that is, sim tab of the Workspace pane) and select **Create Wave**.
- Right-click a signal already in the Wave window and select Create/Modify Waveform. (Only possible before simulation is run.)

The graphic below shows the initial dialog in the wizard. Note that the Drive Type field is not present for input and output signals.

Figure 30-3. Create Pattern Wizard



In this dialog you specify the signal that the waveform will be based upon, the Drive Type (if applicable), the start and end time for the waveform, and the pattern for the waveform.

The second dialog in the wizard lets you specify the appropriate attributes based on the pattern you select. The table below shows the five available patterns and their attributes:

Table 30-1. Signal Attributes in Create Pattern Wizard

Pattern	Description
Clock	Specify an initial value, duty cycle, and clock period for the waveform.
Constant	Specify a value.
Random	Generates different patterns depending upon the seed value. Specify the type (normal or uniform), an initial value, and a seed value. If you don't specify a seed value, ModelSim uses a default value of 5.
Repeater	Specify an initial value and pattern that repeats. You can also specify how many times the pattern repeats.
Counter	Specify start and end values, time period, type (Range, Binary, Gray, One Hot, Zero Hot, Johnson), counter direction, step count, and repeat number.

Creating Waveforms with Wave Create Command

The `wave create` command gives you the ability to generate clock, constant, random, repeater, and counter waveform patterns from the command line. You can then modify the waveform

interactively in the GUI and use the results to drive simulation. See the [wave create](#) command in the Command Reference for correct syntax, argument descriptions, and examples.

Editing Waveforms

You can edit waveforms interactively with menu commands, mouse actions, or by using the [wave edit](#) command.

To edit waveforms in the Wave window, follow these steps:

1. Create an editable pattern as described under [Creating Waveforms from Patterns](#).
2. Enter editing mode by right-clicking a blank area of the toolbar and selecting **Wave_edit** from the toolbar popup menu.

This will open the Wave Edit toolbar. For details about the Wave Edit toolbar, please refer to [Wave Edit Toolbar](#).

Figure 30-4. Wave Edit Toolbar



3. Select an edge or a section of the waveform with your mouse. See [Selecting Parts of the Waveform](#) for more details.
4. Select a command from the **Wave > Wave Editor** menu when the Wave window is docked, from the **Edit > Wave** menu when the Wave window is undocked, or right-click on the waveform and select a command from the **Wave** context menu.

The table below summarizes the editing commands that are available.

Table 30-2. Waveform Editing Commands

Operation	Description
Cut	Cut the selected portion of the waveform to the clipboard
Copy	Copy the selected portion of the waveform to the clipboard
Paste	Paste the contents of the clipboard over the selected section or at the active cursor location
Insert Pulse	Insert a pulse at the location of the active cursor
Delete Edge	Delete the edge at the active cursor
Invert	Invert the selected waveform section
Mirror	Mirror the selected waveform section
Value	Change the value of the selected portion of the waveform

Table 30-2. Waveform Editing Commands (cont.)

Operation	Description
Stretch Edge	Move an edge forward/backward by "stretching" the waveform; see Stretching and Moving Edges for more information
Move Edge	Move an edge forward/backward without changing other edges; see Stretching and Moving Edges for more information
Extend All Waves	Extend all created waveforms by the specified amount or to the specified simulation time; ModelSim cannot undo this edit or any edits done prior to an extend command
Change Drive Type	Change the drive type of the selected portion of the waveform
Undo	Undo waveform edits (except changing drive type and extending all waves)
Redo	Redo previously undone waveform edits

These commands can also be accessed via toolbar buttons. See [Wave Edit Toolbar](#) for more information.

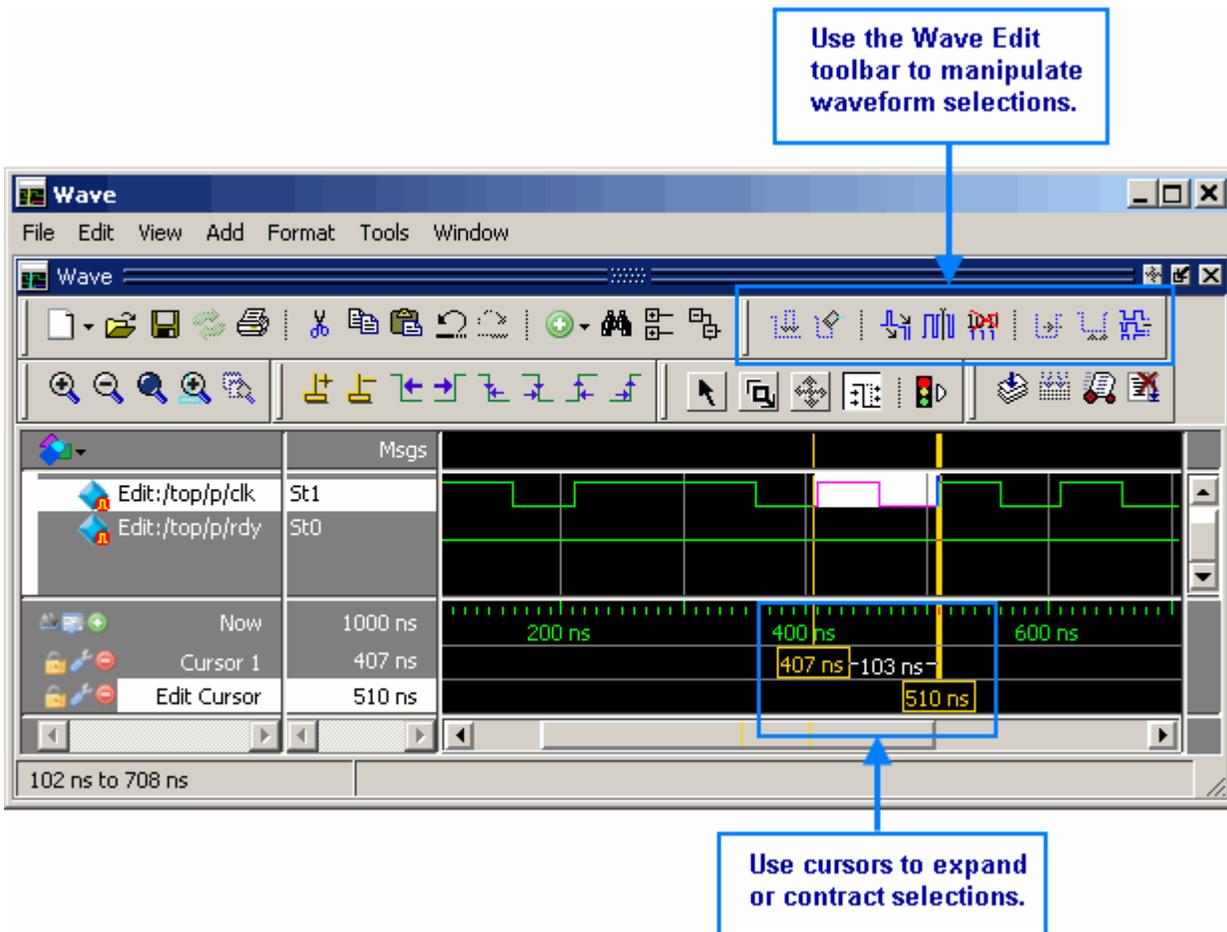
Selecting Parts of the Waveform

There are several methods for selecting edges or sections of a waveform. The table and graphic below describe the various options.

Table 30-3. Selecting Parts of the Waveform

Action	Method
Select a waveform edge	Click on or just to the right of the waveform edge
Select a section of the waveform	Click-and-drag the mouse pointer in the waveform pane
Select a section of multiple waveforms	Click-and-drag the mouse pointer while holding the <Shift> key
Extend/contract the selection size	Drag a cursor in the cursor pane
Extend/contract selection from edge-to-edge	Click Next Transition/Previous Transition icons after selecting section

Figure 30-5. Manipulating Waveforms with the Wave Edit Toolbar and Cursors



Selection and Zoom Percentage

You may find that you cannot select the exact range you want because the mouse moves more than one unit of simulation time (for example, 228 ns to 230 ns). If this happens, zoom in on the Wave display (see [Zooming the Wave Window Display](#)), and you should be able to select the range you want.

Auto Snapping of the Cursor

When you click just to the right of a waveform edge in the waveform pane, the cursor automatically "snaps" to the nearest edge. This behavior is controlled by the Snap Distance setting in the Wave window preferences dialog.

Stretching and Moving Edges

There are mouse and keyboard shortcuts for moving and stretching edges:

Table 30-4. Wave Editor Mouse/Keyboard Shortcuts

Action	Mouse/keyboard shortcut
Stretch an edge	Hold the <Ctrl> key and drag the edge
Move an edge	Hold the <Ctrl> key and drag the edge with the 2nd (middle) mouse button

Here are some points to keep in mind about stretching and moving edges:

- If you stretch an edge forward, more waveform is inserted at the beginning of simulation time.
- If you stretch an edge backward, waveform is deleted at the beginning of simulation time.
- If you move an edge past another edge, either forward or backward, the edge you moved past is deleted.

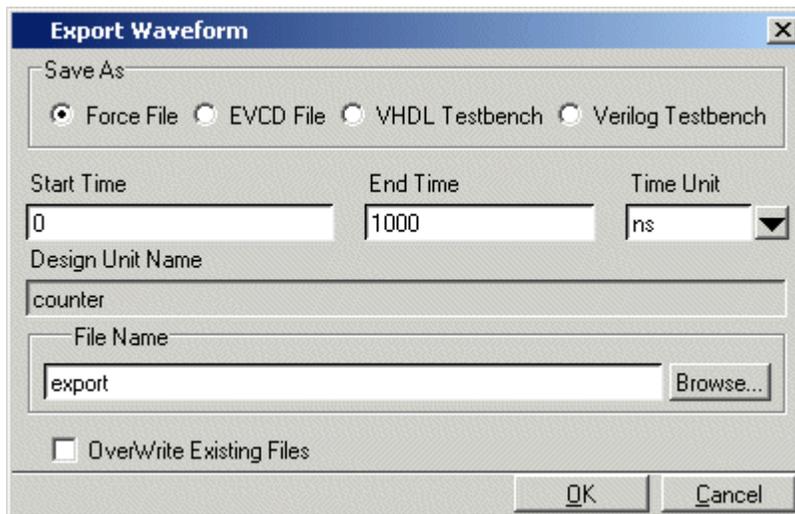
Simulating Directly from Waveform Editor

You need not save the waveforms in order to use them as stimulus for a simulation. Once you have configured all the waveforms, you can run the simulation as normal by selecting **Simulate > Start Simulation** in the Main window or using the `vsim` command. ModelSim automatically uses the created waveforms as stimulus for the simulation. Furthermore, while running the simulation you can continue editing the waveforms to modify the stimulus for the part of the simulation yet to be completed.

Exporting Waveforms to a Stimulus File

Once you have created and edited the waveforms, you can save the data to a stimulus file that can be used to drive a simulation now or at a later time. To save the waveform data, select **File > Export > Waveform** or use the `wave export` command.

Figure 30-6. Export Waveform Dialog



You can save the waveforms in four different formats:

Table 30-5. Formats for Saving Waveforms

Format	Description
Force format	Creates a Tcl script that contains force commands necessary to recreate the waveforms; source the file when loading the simulation as described under Driving Simulation with the Saved Stimulus File
EVCD format	Creates an extended VCD file which can be reloaded using the Import > EVCD File command or can be used with the -vcdstim argument to vsim to simulate the design
VHDL Testbench	Creates a VHDL architecture that you load as the top-level design unit
Verilog Testbench	Creates a Verilog module that you load as the top-level design unit

Driving Simulation with the Saved Stimulus File

The method for loading the stimulus file depends upon what type of format you saved. In each of the following examples, assume that the top-level of your block is named "top" and you saved the waveforms to a stimulus file named "mywaves" with the default extension.

Table 30-6. Examples for Loading a Stimulus File

Format	Loading example
Force format	<code>vsim top -do mywaves.do</code>

Table 30-6. Examples for Loading a Stimulus File (cont.)

Format	Loading example
Extended VCD format ¹	<code>vsim top -vcdstim mywaves.vcd</code>
VHDL Testbench	<code>vcom mywaves.vhd</code> <code>vsim mywaves</code>
Verilog Testbench	<code>vlog mywaves.v</code> <code>vsim mywaves</code>

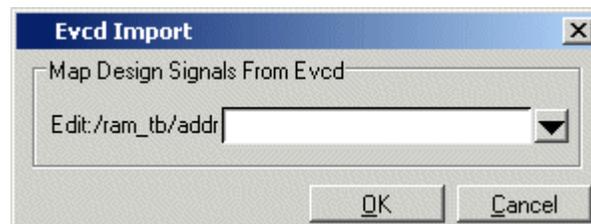
1. You can also use the **Import > EVCD** command from the Wave window. See below for more details on working with EVCD files.

Signal Mapping and Importing EVCD Files

When you import a previously saved EVCD file, ModelSim attempts to map the signals in the EVCD file to the signals in the loaded design by matching signals based on name and width.

If ModelSim can not map the signals automatically, you can do the mapping yourself by selecting a signal, right-clicking the selected signal, then selecting **Map to Design Signal** from the popup menu. This opens the Evcd Import dialog.

Figure 30-7. Evcd Import Dialog



Select a signal from the drop-down arrow and click OK.

Note  This command works only with extended VCD files created with ModelSim.

Using Waveform Compare with Created Waveforms

The Waveform Compare feature compares two or more waveforms and displays the differences in the Wave window (see [Waveform Compare](#) for details). This feature can be used in tandem with Waveform Editor. The combination is most useful in situations where you know the expected output of a signal and want to compare visually the differences between expected output and simulated output.

The basic procedure for using the two features together is as follows:

- Create a waveform based on the signal of interest with a drive type of expected output
- Add the design signal of interest to the Wave window and then run the design
- Start a comparison and use the created waveform as the reference dataset for the comparison. Use the text "Edit" to designate a create waveform as the reference dataset. For example:

```
compare start Edit sim  
compare add -wave /test_counter/count  
compare run
```

Saving the Waveform Editor Commands

When you create and edit waveforms in the Wave window, ModelSim tracks the underlying Tcl commands and reports them to the transcript. You can save those commands to a DO file that can be run at a later time to recreate the waveforms.

To save your waveform editor commands, select **File > Save**.

Chapter 31

Standard Delay Format (SDF) Timing Annotation

This chapter covers the ModelSim implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator.

ASIC and FPGA vendors usually provide tools that create SDF files for use with their cell libraries. Refer to your vendor's documentation for details on creating SDF files for your library. Many vendors also provide instructions on using their SDF files and libraries with ModelSim.

The SDF specification was originally created for Verilog designs, but it has also been adopted for VHDL VITAL designs. In general, the designer does not need to be familiar with the details of the SDF specification because the cell library provider has already supplied tools that create SDF files that match their libraries.

Note



ModelSim can read SDF files that were compressed using gzip. Other compression formats (for example, Unix zip) are not supported.

Specifying SDF Files for Simulation

ModelSim supports SDF versions 1.0 through 4.0 (IEEE 1497), except the NETDELAY and LABEL statements. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following `vsim` 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 test bench. 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
```

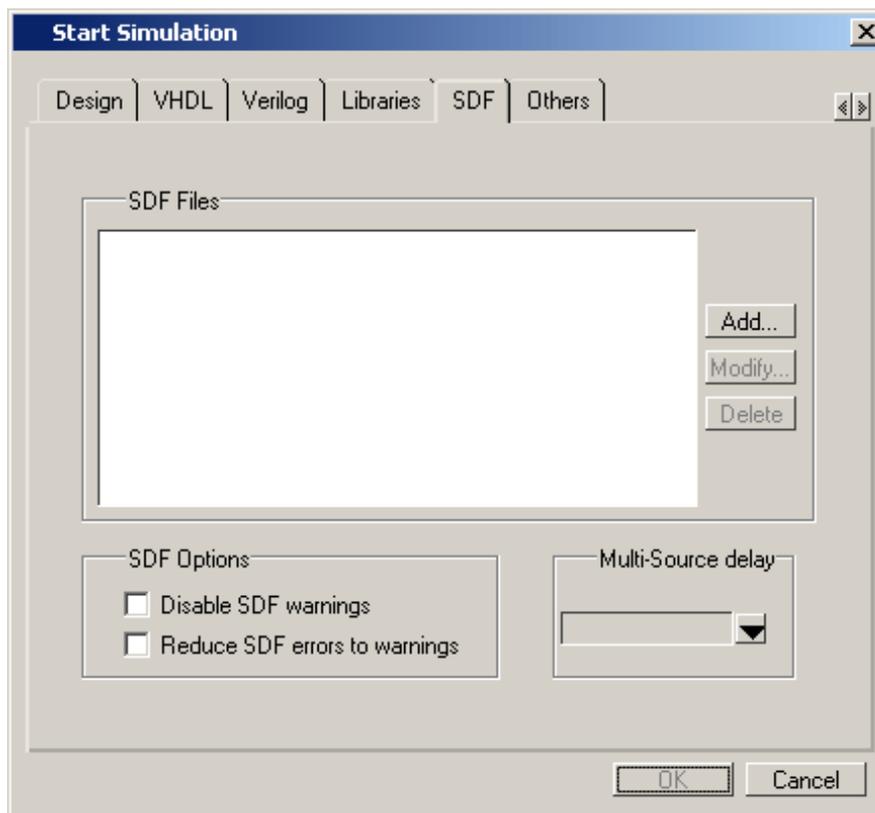
If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a test bench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

SDF Specification with the GUI

As an alternative to the command line options, you can specify SDF files in the **Start Simulation** dialog box under the SDF tab.

Figure 31-1. SDF Tab in Start Simulation Dialog



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Start Simulation**.

For Verilog designs, you can also specify SDF files by using the `$sdf_annotate` system task. See [\\$sdf_annotate](#) 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 either the `-sdfnoerror` or the `+nosdferror` option with `vsim` to change SDF errors to warnings so that the simulation can continue.
- Use either the `-sdfnowarn` or the `+nosdfwarn` option with `vsim` to suppress warning messages.

Another option is to use the **SDF** tab from the **Start Simulation** dialog box (Figure 31-1). Select **Disable SDF warnings** (`-sdfnowarn +nosdfwarn`) to disable warnings, or select **Reduce SDF errors to warnings** (`-sdfnoerror`) to change errors to warnings.

See [Troubleshooting](#) for more information on errors and warnings and how to avoid them.

Compiling SDF Files

The `sdfcom` command compiles SDF files. Compiled SDF can be annotated to Verilog and VHDL regions, including those regions hierarchically underneath SystemC modules. However, compiled SDF cannot be targeted to a SystemC node directly (even if the intended annotation objects underneath the SystemC node are Verilog and/or VHDL).

In situations where the same SDF file is used for multiple simulation runs, the elaboration time will be reduced significantly.

Note



When compiled SDF files are used, the annotator behaves as if the `-v2k_int_delays` switch for the `vsim` command has been specified.

Simulating with Compiled SDF Files

Compiled SDF files may be specified on the `vsim` line with the `-sdfmin`, `-sdftyp`, and `-sdfmax` arguments. Alternatively, they may be specified as the filename in a `$sdf_annotate()` system task in the Verilog source.

Using \$sdf_annotate() with Compiled SDF

The following limitations exist when using compiled SDF files with \$sdf_annotate():

- The \$sdf_annotate() call cannot be made from a delayed initial block:

```
initial #10 $sdf_annotate(...); // Not allowed
```

- The \$sdf_annotate() call cannot be made from an if statement:

```
reg doSdf = 1'b1;
initial begin
    if (doSdf) $sdf_annotate(...); // Not allowed
end
```

- If the annotation order of multiple \$sdf_annotate() calls is important, you must have all of them in a single initial block.

VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE Std 1076.4-2000, *IEEE Standard for VITAL ASIC Modeling Specification* describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see [VITAL Usage and Compliance](#).

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:

Table 31-1. Matching SDF to VHDL Generics

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

Table 31-1. Matching SDF to VHDL Generics (cont.)

SDF construct	Matching VHDL generic name
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk
(WIDTH (COND (reset==1'b0) clk) (5))	tpw_clk_reset_eq_0
(DEVICE y (1))	tdevice_c1_y ¹

1. c1 is the instance name of the module containing the previous generic(tdevice_c1_y).

The SDF statement CONDELSE, when targeted for Vital cells, is annotated to a **tpd** generic of the form **tpd_<inputPort>_<outputPort>**.

Resolving Errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
** Error (vsim-SDF-3240) myasic.sdf(18):
Instance '/testbench/dut/u1' does not have a generic named 'tpd_a_y'
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files provided by the cell library vendor.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don't match the names expected by the annotator, then there are several possibilities:

- The vendor's tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor's library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke **vsim** with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see [Troubleshooting](#).

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.

\$sdf_annotate

Syntax

```
$sdf_annotate  
  ["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"], ["<mtm_spec>"],  
  ["<scale_factor>"], ["<scale_type>"]];
```

Arguments

- "<sdffile>"
String that specifies the SDF file. Required.
- <instance>
Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the \$sdf_annotate call is made.
- "<config_file>"
String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.
- "<log_file>"
String that specifies the logfile. Optional. Currently not supported, this argument is ignored.
- "<mtm_spec>"
String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is "tool_control". The "tool_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).
- "<scale_factor>"
String that specifies delay scaling factors. Optional. The format is "<min_mult>:<typ_mult>:<max_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.
- "<scale_type>"
String that overrides the <mtm_spec> delay selection. Optional. The <mtm_spec> delay selection is always used to select the delay scaling factor, but if a <scale_type> is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the <mtm_spec> value.

Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

SDF to Verilog Construct Matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows.

- **IOPATH** is matched to specify path delays or primitives:

Table 31-2. Matching SDF IOPATH to Verilog

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 ModelSim can't locate a corresponding specify path delay, it returns an error unless you use the +sdf_iopath_to_prim_ok argument to **vsim**. If you specify that argument and the module contains no path delays, then all primitives that drive the specified output port are annotated.

- **INTERCONNECT** and **PORT** are matched to input ports:

Table 31-3. Matching SDF INTERCONNECT and PORT to Verilog

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:

Table 31-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog

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:

Table 31-5. Matching SDF DEVICE to Verilog

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:

Table 31-6. Matching SDF SETUP to Verilog

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:

Table 31-7. Matching SDF HOLD to Verilog

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:

Table 31-8. Matching SDF SETUPHOLD to Verilog

SDF	Verilog
(SETPHOLD d (posedge clk) (5) (5))	\$setup(d, posedge clk, 0);
(SETPHOLD d (posedge clk) (5) (5))	\$hold(posedge clk, d, 0);
(SETPHOLD d (posedge clk) (5) (5))	\$setuphold(posedge clk, d, 0, 0);

- **RECOVERY** is matched to \$recovery:

Table 31-9. Matching SDF RECOVERY to Verilog

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	\$recovery(negedge reset, posedge clk, 0);

- **REMOVAL** is matched to \$removal:

Table 31-10. Matching SDF REMOVAL to Verilog

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	\$removal(negedge reset, posedge clk, 0);

- **RECREM** is matched to \$recovery, \$removal, and \$crem:

Table 31-11. Matching SDF RECREM to Verilog

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))	\$crem(negedge reset, posedge clk, 0);

- **SKEW** is matched to \$skew:

Table 31-12. Matching SDF SKEW to Verilog

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	\$skew(posedge clk1, posedge clk2, 0);

- **WIDTH** is matched to \$width:

Table 31-13. Matching SDF WIDTH to Verilog

SDF	Verilog
(WIDTH (posedge clk) (5))	\$width(posedge clk, 0);

- **PERIOD** is matched to \$period:

Table 31-14. Matching SDF PERIOD to Verilog

SDF	Verilog
(PERIOD (posedge clk) (5))	\$period(posedge clk, 0);

- **NOCHANGE** is matched to \$nochange:

Table 31-15. Matching SDF NOCHANGE to Verilog

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	\$nochange(negedge write, addr, 0, 0);

To see complete mappings of SDF and Verilog constructs, please consult IEEE Std 1364-2005, Chapter 16 - Back Annotation Using the Standard Delay Format (SDF).

Retain Delay Behavior

The simulator processes RETAIN delays in SDF files as described in this section. A RETAIN delay can appear as:

```
(IOPATH addr[13:0] dout[7:0]
  (RETAIN (rval1) (rval2) (rval3)) // RETAIN delays
  (dval1) (dval2) ...           // IOPATH delays
)
```

Because *rval2* and *rval3* on the RETAIN line are optional, the simulator makes the following assumptions:

- Only *rval1* is specified — *rval1* is used as the value of *rval2* and *rval3*.
- *rval1* and *rval2* are specified — the smaller of *rval1* and *rval2* is used as the value of *rval3*.

During simulation, if any *rval* that would apply is larger than or equal to the applicable path delay, then RETAIN delay is not applied.

You can specify that RETAIN delays should not be processed by using `+vlog_retain_off` on the `vsim` command line.

Retain delays apply to an IOPATH for any transition on the input of the PATH unless the IOPATH specifies a particular edge for the input of the IOPATH. This means that for an IOPATH such as RCLK -> DOUT, RETAIN delay should apply for a negedge on RCLK even though a Verilog model is coded only to change DOUT in response to a posedge of RCLK. If (posedge RCLK) -> DOUT is specified in the SDF then an associated RETAIN delay applies only for posedge RCLK. If a path is conditioned, then RETAIN delays do not apply if a delay path is not enabled.

Table 31-16 defines which delay is used depending on the transitions:

Table 31-16. RETAIN Delay Usage (default)

Path Transition	Retain Transition	Retain Delay Used	Path Delay Used	Note
0->1	0->x->1	rval1 (0->x)	0->1	
1->0	1->x->0	rval2 (1->x)	1->0	
z->0	z->x->0	rval3 (z->x)	z->0	
z->1	z->x->1	rval3 (z->x)	z->1	
0->z	0->x->z	rval1 (0->x)	0->z	
1->z	1->x->z	rval2 (1->x)	1->z	
x->0	x->x->0	n/a	x->0	use PATH delay, no RETAIN delay is applicable
x->1	x->x->1	n/a	x->1	
x->z	x->x->z	n/a	x->z	
0->x	0->x->x	rval1 (0->x)	0->x	use RETAIN delay for PATH delay if it is smaller
1->x	1->x->x	rval2 (1->x)	1->x	
z->x	z->x->x	rval3 (z->x)	z->x	

You can specify that X insertion on outputs that do not change except when the causal inputs change by using +vlog_retain_same2same_on on the vsim command line. An example is when CLK changes but bit DOUT[0] does not change from its current value of 0, but you want it to go through the transition 0 -> X -> 0.

Table 31-17. RETAIN Delay Usage (with +vlog_retain_same2same_on)

Path Transition	Retain Transition	Retain Delay Used	Path Delay Used	Note
0->0	0->x->0	rval1 (0->x)	1->0	
1->1	1->x->1	rval2 (1->x)	0->1	
z->z	z->x->z	rval3 (z->x)	max(0->z,1->z)	
x->x	x->x->x			No output transition

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:

Table 31-18. Matching Verilog Timing Checks to SDF SETUP

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.

Table 31-19. SDF Data May Be More Accurate Than Model

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	\$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	\$setup(data, posedge clk, 0);

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers can also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. For example,

Table 31-20. Matching Explicit Verilog Edge Transitions to Verilog

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(data, edge[01, 0x] clk, 0);

The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port.

Optional Conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

Table 31-21. SDF Timing Check Conditions

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,

Table 31-22. SDF Path Delay Conditions

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 for more information on SDF command line options.

Interconnect Delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the [vsim](#) command for more information on the relevant command line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Disabling Timing Checks

ModelSim offers a number of options for disabling timing checks on a global or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the Reference Manual for more details.

Table 31-23. Disabling Timing Checks

Command and argument	Effect
tcheck_set ¹	modifies reporting or X generation status on one or more timing checks
tcheck_status ¹	prints to the Transcript the current status of one or more timing checks
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vopt +notimingchecks	removes all timing check entries from the design as it is parsed; fixes the TimingChecksOn generic for all Vital models to FALSE; As a consequence, using vsim +notimingchecks at simulation may not have any effect on the simulation depending on the optimization of the model.

Table 31-23. Disabling Timing Checks (cont.)

Command and argument	Effect
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	disables Verilog and VITAL timing checks for all instances in the specified design; sets generic TimingChecksOn to FALSE for all VHDL Vital models with the Vital_level0 or Vital_level1 attribute. Setting this generic to FALSE disables the actual calls to the timing checks along with anything else that is present in the model's timing check block.
vsim +nospecify	disables specify path delays and timing checks for all instances in the specified design

1. tcheck_set and tcheck_status commands will not operate on a module instance (and the underlying hierarchy) that has been compiled with "-nodebug" option. A suppressible warning message will be issued.

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 test bench. See [Instance Specification](#) for an example.

Simple examples for both a VHDL and a Verilog test bench are provided below. For simplicity, these test bench examples do nothing more than instantiate a model that has no ports.

VHDL Test Bench

```
entity testbench is end;
```

```
architecture only of testbench is
  component myasic
  end component;
begin
  dut : myasic;
end;
```

Verilog Test Bench

```
module testbench;
  myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either test bench, 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, view the structure pane, navigate to the model instance, select it, and enter the [environment](#) command. This command displays the instance name that should be used in the SDF command line option.

Matching a Single Timing Check

SDF annotation of RECREM or SETUPHOLD matching only a single setup, hold, recovery, or removal timing check will result in a Warning message.

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 test benches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/myasic'.
```

Forgetting to Specify the Instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

```
vsim -sdfmax myasic.sdf testbench
```

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u1'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u2'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u3'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u4'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u5'  
** Warning (vsim-SDF-3432) myasic.sdf:  
This file is probably applied to the wrong instance.  
** Warning (vsim-SDF-3432) myasic.sdf:  
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:  
Failed to find any of the 358 instances from this file.  
** Warning (vsim-SDF-3442) myasic.sdf:  
Try instance '/testbench/dut'. It contains all instance paths from this  
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see [Resolving Errors](#) for specific VHDL VITAL SDF troubleshooting.

Chapter 32

Value Change Dump (VCD) Files

The Value Change Dump (VCD) file format is supported for use by ModelSim and is specified in the IEEE 1364-2005 standard. A VCD file is an ASCII file that contains information about value changes on selected variables in the design stored by VCD system tasks. This includes header information, variable definitions, and variable value changes.

VCD is in common use for Verilog designs and is controlled by VCD system task calls in the Verilog source code. ModelSim provides equivalent commands for these system tasks and extends VCD support to SystemC and VHDL designs. You can use these ModelSim VCD commands on Verilog, VHDL, SystemC, or mixed designs.

Extended VCD supports Verilog and VHDL ports in a mixed-language design containing SystemC. However, extended VCD does not support SystemC ports in a mixed-language design.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, contact your ASIC vendor.

Creating a VCD File

ModelSim provides two general methods for creating a VCD file:

- **Four-State VCD File** — produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information.
- **Extended VCD File** — produces an extended VCD (EVCD) file with variable changes in all states and strength information and port driver data.

Both methods also capture port driver changes unless you filter them out with optional command-line arguments.

Four-State VCD File

First, compile and load the design. For example:

```
% cd <installDir>/examples/tutorials/verilog/basicSimulation
% vlib work
% vlog counter.v tcounter.v
% vopt test_counter +acc -o test_counter_opt
% vsim test_counter_opt
```

Next, with the design loaded, specify the VCD file name with the `vcd file` command and add objects to the file with the `vcd add` command:

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

Upon quitting the simulation, there will be a VCD file in the working directory.

Extended VCD File

First, compile and load the design. For example:

```
% cd <installDir>/examples/tutorials/verilog/basicSimulation
% vlib work
% vlog counter.v tcounter.v
% vopt test_counter +acc -o test_counter_opt
% vsim test_counter_opt
```

Next, with the design loaded, specify the VCD file name and objects to add with the `vcd dumpports` command:

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

Upon quitting the simulation, there will be an extended VCD file called *myvcdfile.vcd* in the working directory.

Note



There is an internal limit to the number of ports that can be listed with the `vcd dumpports` command. If that limit is reached, use the `vcd add` command with the `-dumpports` option to name additional ports.

VCD Case Sensitivity

Verilog designs are case-sensitive, so ModelSim maintains case when it produces a VCD file. However, VHDL is not case-sensitive, so ModelSim converts all signal names to lower case when it produces a VCD file.

Checkpoint/Restore and Writing VCD Files

If a checkpoint occurs while ModelSim is writing a VCD file, the entire VCD file is copied into the checkpoint file. Since VCD files can be very large, it is possible that disk space problems may occur. Consequently, ModelSim issues a warning in this situation.

Using Extended VCD as Stimulus

You can use an extended VCD file as stimulus to re-simulate your design. There are two ways to do this:

1. Simulate the top level of a design unit with the input values from an extended VCD file.
2. Specify one or more instances in a design to be replaced with the output values from the associated VCD file.

Simulating with Input Values from a VCD File

When simulating with inputs from an extended VCD file, you can simulate only one design unit at a time. In other words, you can apply the VCD file inputs only to the top level of the design unit for which you captured port data.

The general procedure includes two steps:

1. Create a VCD file for a single design unit using the `vcd dumpports` command.
2. Resimulate the single design unit using the `-vcdstim` argument with the `vsim` command. Note that `-vcdstim` works only with VCD files that were created by a ModelSim simulation.

Example 32-1. Verilog Counter

First, create the VCD file for the single instance using `vcd dumpports`:

```
% cd <installDir>/examples/tutorials/verilog/basicSimulation
% vlib work
% vlog counter.v tcounter.v
% vopt test_counter +acc -o test_counter_opt
% vsim test_counter_opt +dumpports+nocollapse
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the test bench, using the `-vcdstim` argument:

```
% vopt counter -o counter_replay
% vsim counter_replay -vcdstim counter.vcd
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 32-2. VHDL Adder

First, create the VCD file using `vcd dumpports`:

```
% cd <installDir>/examples/vcd
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vopt testbench2 +acc -o testbench2_opt
% vsim testbench2_opt +dumpports+nocollapse
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the test bench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 32-3. Mixed-HDL Design

First, create three VCD files, one for each module:

```
% cd <installDir>/examples/tutorials/mixed/projects
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vopt top +acc -o top_opt
% vsim top_opt +dumpports+nocollapse
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

Note



When using VCD files as stimulus, the VCD file format does not support recording of delta delay changes – delta delays are not captured and any delta delay ordering of signal changes is lost. Designs relying on this ordering may produce unexpected results.

Replacing Instances with Output Values from a VCD File

Replacing instances with output values from a VCD file lets you simulate without the instance's source or even the compiled object. The general procedure includes two steps:

1. Create VCD files for one or more instances in your design using the [vcd dumpports](#) command. If necessary, use the **-vcdstim** switch to handle port order problems (see below).

2. Re-simulate your design using the `-vcdstim <instance>=<filename>` argument to `vsim`. Note that this works only with VCD files that were created by a ModelSim simulation.

Example 32-4. Replacing Instances

In the following example, the three instances `/top/p`, `/top/c`, and `/top/m` are replaced in simulation by the output values found in the corresponding VCD files.

First, create VCD files for all instances you want to replace:

```
vcd dumpports -vcdstim -file proc.vcd /top/p/*
vcd dumpports -vcdstim -file cache.vcd /top/c/*
vcd dumpports -vcdstim -file memory.vcd /top/m/*
run 1000
```

Next, simulate your design and map the instances to the VCD files you created:

```
vsim top_opt -vcdstim /top/p=proc.vcd -vcdstim /top/c=cache.vcd
-vcdstim /top/m=memory.vcd
quit -f
```

Note



When using VCD files as stimulus, the VCD file format does not support recording of delta delay changes – delta delays are not captured and any delta delay ordering of signal changes is lost. Designs relying on this ordering may produce unexpected results.

Port Order Issues

The `-vcdstim` argument to the `vcd dumpports` command ensures the order that port names appear in the VCD file matches the order that they are declared in the instance's module or entity declaration. Consider the following module declaration:

```
module proc(clk, addr, data, rw, strb, rdy);
  input  clk, rdy;
  output addr, rw, strb;
  inout data;
```

The order of the ports in the module line (`clk, addr, data, ...`) does not match the order of those ports in the input, output, and inout lines (`clk, rdy, addr, ...`). In this case the `-vcdstim` argument to the `vcd dumpports` command needs to be used.

In cases where the order is the same, you do not need to use the `-vcdstim` argument to `vcd dumpports`. Also, module declarations of the form:

```
module proc(input clk, output addr, inout data, ...)
```

do not require use of the argument.

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.

Table 32-1. VCD Commands and SystemTasks

VCD commands	VCD system tasks
vcd add	\$dumpvars
vcd checkpoint	\$dumpall
vcd file	\$dumpfile
vcd flush	\$dumpflush
vcd limit	\$dumplimit
vcd off	\$dumpoff
vcd on	\$dumpon

ModelSim also supports extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

Table 32-2. VCD Dumpport Commands and System Tasks

VCD dumpports commands	VCD system tasks
vcd dumpports	\$dumpports
vcd dumpportsall	\$dumpportsall
vcd dumpportsflush	\$dumpportsflush
vcd dumpportslimit	\$dumpportslimit
vcd dumpportsoff	\$dumpportsoff
vcd dumpportson	\$dumpportson

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364-2005 specification. The tasks behave the same as the IEEE equivalent tasks such as \$dumpfile, \$dumpvar, and so forth. 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. [Table 32-3](#) maps the VCD commands to their associated tasks. For additional details, please see the Verilog IEEE Std 1364-2005 specification.

Table 32-3. VCD Commands and System Tasks for Multiple VCD Files

VCD commands	VCD system tasks
<code>vcd add -file <filename></code>	<code>\$fdumpvars(levels, { , module_or_variable }¹, filename)</code>
<code>vcd checkpoint <filename></code>	<code>\$fdumpall(filename)</code>
<code>vcd files <filename></code>	<code>\$fdumpfile(filename)</code>
<code>vcd flush <filename></code>	<code>\$fdumpflush(filename)</code>
<code>vcd limit <filename></code>	<code>\$fdumplimit(filename)</code>
<code>vcd off <filename></code>	<code>\$fdumpoff(filename)</code>
<code>vcd on <filename></code>	<code>\$dumpon(filename)</code>

1. denotes an optional, comma-separated list of 0 or more modules or variables

Using VCD Commands with SystemC

VCD commands are supported for the following SystemC signals:

```
sc_signal<T>
sc_signal_resolved
sc_signal_rv<N>
```

VCD commands are supported for the following SystemC signal ports:

```
sc_in<T>
sc_out<T>
sc_inout<T>
sc_in_resolved
sc_out_resolved
sc_inout_resolved
sc_in_rv<N>
sc_out_rv<N>
sc_inout_rv<N>
```

<T> can be any of types shown in [Table 32-4](#).

Table 32-4. SystemC Types

unsigned char	char	sc_int
unsigned short	short	sc_uint
unsigned int	int	sc_bigint
unsigned long	float	sc_biguint
unsigned long long	double	sc_signed
	enum	sc_unsigned
		sc_logic
		sc_bit
		sc_bv
		sc_lv

Unsupported types are the SystemC fixed point types, class, structures and unions.

Compressing Files with VCD Tasks

ModelSim can produce compressed VCD files using the **gzip** compression algorithm. Since we cannot change the syntax of the system tasks, we act on the extension of the output file name. If you specify a .gz extension on the filename, ModelSim will compress the output.

VCD File from Source to Output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL Source Code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
  port (CLK, RESET, data_in  : IN STD_LOGIC;
        Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
  process (CLK,RESET)
  begin
    if (RESET = '1') then
      Q <= (others => '0') ;
    elsif (CLK'event and CLK = '1') then
      Q <= Q(Q'left - 1 downto 0) & data_in ;
    end if ;
  end process ;
end ;
```

VCD Simulator Commands

At simulator time zero, the designer executes the following commands:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
quit -sim
```

VCD Output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

```

$date                               $end                               #700
  Thu Sep 18                         #100                              1!
11:07:43 2003                       1!                                1(
$end                                  #150                              #750
$version                              0!                                0!
  <Tool> Version                     #200                              #800
<version>                             1!                                1!
$end                                  $dumpoff                          1'
$timescale                           x!                                #850
  1ns                                 x"                                0!
$end                                  x#                                #900
$scope module                         x$                                1!
shifter_mod $end                     x%                                1&
$var wire 1 ! clk                     x&                                #950
$end                                  x'                                0!
$var wire 1 " reset                   x(                                #1000
$end                                  x)                                1!
$var wire 1 # data_in                 x*                                1%
$end                                  x+                                #1050
$var wire 1 $ q [8]                   x,                                0!
$end                                  $end                              #1100
$var wire 1 % q [7]                   #300                              1!
$end                                  $dumpon                          1$
$var wire 1 & q [6]                   1!                                #1150
$end                                  0"                                0!
$var wire 1 ' q [5]                   1#                                1"
$end                                  0$                                0,
$var wire 1 ( q [4]                   0%                                0+
$end                                  0&                                0*
$var wire 1 ) q [3]                   0'                                0)
$end                                  0(                                0(
$var wire 1 * q [2]                   0)                                0'
$end                                  0*                                0&
$var wire 1 + q [1]                   0+                                0%
$end                                  1,                                0$
$var wire 1 , q [0]                   $end                              #1200
$end                                  #350                              1!
$upscope $end                        0!                                $dumpall
$enddefinitions $end                 #400                              1!
#0                                    1!                                1"
$dumpvars                             1+                                1#
0!                                    #450                              0$
1"                                    0!                                0%
0#                                    #500                              0&
0$                                    1!                                0'
0%                                    1*                                0(
0&                                    #550                              0)
0'                                    0!                                0*
0(                                    #600                              0+
0)                                    1!                                0,
0*                                    1)                                $end
0+                                    #650
0,                                    0!

```

VCD to WLF

The ModelSim `vcd2wlf` command is a utility that translates a `.vcd` file into a `.wlf` file that can be displayed in ModelSim using the `vsim -view` argument. This command only works on VCD files containing positive time values.

Capturing Port Driver Data

Some ASIC vendors' toolkits read a VCD file format that provides details on port drivers. This information can be used, for example, to drive a tester. For more information on a specific toolkit, refer to the ASIC vendor's documentation.

In ModelSim, use the `vcd dumpports` command to create a VCD file that captures port driver data. 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<state> <0 strength> <1 strength> <identifier_code>
```

Driver States

Table 32-5 shows the driver states recorded as TSSI states if the direction is known.

Table 32-5. Driver States

Input (testfixture)	Output (dut)
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state
d low (two or more drivers active)	l low (two or more drivers active)
u high (two or more drivers active)	h high (two or more drivers active)

If the direction is unknown, the state will be recorded as one of the following:

Table 32-6. State When Direction is Unknown

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)

Table 32-6. State When Direction is Unknown (cont.)

Unknown direction	
F	three-state (input and output unconnected)
A	unknown (input driving low and output driving high)
a	unknown (input driving low and output driving unknown)
B	unknown (input driving high and output driving low)
b	unknown (input driving high and output driving unknown)
C	unknown (input driving unknown and output driving low)
c	unknown (input driving unknown and output driving high)
f	unknown (input and output three-stated)

Driver Strength

The recorded 0 and 1 strength values are based on Verilog strengths:

Table 32-7. Driver Strength

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	

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".

Resolving Values

The resolved values written to the VCD file depend on which options you specify when creating the file.

Default Behavior

By default, ModelSim generates VCD output according to the IEEE Std 1364TM-2005, *IEEE Standard for Verilog[®] Hardware Description Language*. This standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:

- Strong: strengths 7, 6, and 5
- Weak: strengths 4, 3, 2, 1

The rules for resolving values are as follows:

- If the input and output are driving the same value with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- If the input is driving a strong strength and the output is driving a weak strength, the resolved value is D, d, U or u, and the strength is the strength of the input.
- If the input is driving a weak strength and the output is driving a strong strength, the resolved value is L, l, H or h, and the strength is the strength of the output.

When force Command is Used

If you **force** a value on a net that does not have a driver associated with it, ModelSim uses the port direction as shown in [Table 32-8](#) to dump values to the VCD file. When the port is an inout, the direction cannot be determined.

Table 32-8. VCD Values When Force Command is Used

Value forced on net	Port Direction		
	input	output	inout
0	D	L	0
1	U	H	1
X	N	X	?
Z	Z	T	F

Extended Data Type for VHDL (vl_logic)

Mentor Graphics has created an additional VHDL data type for use in mixed-language designs, in case you need access to the full Verilog state set. The `vl_logic` type is an enumeration that defines the full set of VHDL values for Verilog nets, as defined for Logic Strength Modeling in IEEE 1364™-2005.

This specification defines the following driving strengths for signals propagated from gate outputs and continuous assignment outputs:

Supply, Strong, Pull, Weak, HiZ

This specification also defines three charge storage strengths for signals originating in the `trireg` net type:

Large, Medium, Small

Each of these strengths can assume a strength level ranging from 0 to 7 (expressed as a binary value from 000 to 111), combined with the standard four-state values of 0, 1, X, and Z. This results in a set of 256 strength values, which preserves Verilog strength values going through the VHDL portion of the design and allows a VCD in extended format for any downstream application.

The `vl_logic` type is defined in the following file installed with ModelSim, where you can view the 256 strength values:

```
<install_dir>/vhdl_src/verilog/vltypes.vhd
```

This location is a pre-compiled **verilog** library provided in your installation directory, along with the other pre-compiled libraries (**std** and **ieee**).

Note



The Wave window display and WLF do not support the full range of `vl_logic` values for VHDL signals.

Ignoring Strength Ranges

You may wish to ignore strength ranges and have ModelSim handle each strength separately. Any of the following options will produce this behavior:

- Use the `-no_strength_range` argument to the [vcd dumptports](#) command
- Use an optional argument to `$dumptports` (see [Extended \\$dumptports Syntax](#) below)
- Use the `+dumptports+no_strength_range` argument to [vsim](#) command

In this situation, ModelSim reports strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, ModelSim reports only the “winning”

strength. In other words, the two strength values either match (for example, pA 5 5 !) or the winning strength is shown and the other is zero (for instance, pH 0 5 !).

Extended \$dumpports Syntax

ModelSim extends the \$dumpports system task in order to support exclusion of strength ranges. The extended syntax is as follows:

```
$dumpports (scope_list, file_pathname, ncsim_file_index, file_format)
```

The `nc_sim_index` argument is required yet ignored by ModelSim. It is required only to be compatible with NCSim's argument list.

The `file_format` argument accepts the following values or an ORed combination thereof (see examples below):

Table 32-9. Values for file_format Argument

File_format value	Meaning
0	Ignore strength range
2	Use strength ranges; produces IEEE 1364-compliant behavior
4	Compress the EVCD output
8	Include port direction information in the EVCD file header; same as using <code>-direction</code> argument to <code>vcd dumpports</code>

Here are some examples:

```
// ignore strength range
$dumpports(top, "filename", 0, 0)
// compress and ignore strength range
$dumpports(top, "filename", 0, 4)
// print direction and ignore strength range
$dumpports(top, "filename", 0, 8)
// compress, print direction, and ignore strength range
$dumpports(top, "filename", 0, 12)
```

Example 32-5. VCD Output from vcd dumpports

This example demonstrates how **vcd dumpports** resolves values based on certain combinations of driver values and strengths and whether or not you use strength ranges. [Table 32-10](#) is sample driver data.

Table 32-10. Sample Driver Data

time	in value	out value	in strength value (range)	out strength value (range)
0	0	0	7 (strong)	7 (strong)
100	0	0	6 (strong)	7 (strong)
200	0	0	5 (strong)	7 (strong)
300	0	0	4 (weak)	7 (strong)
900	1	0	6 (strong)	7 (strong)
27400	1	1	5 (strong)	4 (weak)
27500	1	1	4 (weak)	4 (weak)
27600	1	1	3 (weak)	4 (weak)

Given the driver data above and use of 1364 strength ranges, here is what the VCD file output would look like:

```
#0
p0 7 0 <0
#100
p0 7 0 <0
#200
p0 7 0 <0
#300
pL 7 0 <0
#900
pB 7 6 <0
#27400
pU 0 5 <0
#27500
p1 0 4 <0
#27600
p1 0 4 <0
```

Chapter 33

Tcl and Macros (DO Files)

Tcl is a scripting language for controlling and extending ModelSim. Within ModelSim you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts “on the fly” without stopping to recompile or restart ModelSim. In addition, if ModelSim does not provide the command you need, you can use Tcl to create your own commands.

Tcl Features

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

Tcl References

For quick reference information on Tcl, choose the following from the ModelSim main menu:

Help > Tcl Man Pages

In addition, the following books provide more comprehensive usage information on Tcl:

- *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc.
- *Practical Programming in Tcl and Tk* by Brent Welch, published by Prentice Hall.

Tcl Commands

For complete information on Tcl commands, select **Help > Tcl Man Pages**. Also see [Simulator GUI Preferences](#) for information on Tcl preference variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands, as shown in [Table 33-1](#).

Table 33-1. Changes to ModelSim Commands

Previous ModelSim command	Command changed to (or replaced by)
continue	run with the <code>-continue</code> option
format list wave	write format with either list or wave specified
if	replaced by the Tcl if command, see If Command Syntax for more information
list	add list
nolist nowave	delete with either list or wave specified
set	replaced by the Tcl set command, see set Command Syntax for more information
source	vsource
wave	add wave

Tcl Command Syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on [If Command Syntax](#) and [set Command Syntax](#) follow.

1. A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.
2. A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
3. Words of a command are separated by white space (except for newlines, which are command separators).

4. If the first character of a word is a double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.
5. If the first character of a word is an open brace ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
6. If a word contains an open bracket ([) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (]). The result of the script (that is, 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. Table 33-2 lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

Table 33-2. Tcl Backslash Sequences

Sequence	Value
\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 pound sign (#) appears at a point where Tcl is expecting the first character of the first word of a command, then the pound sign and the characters that follow it, up through the next newline, are treated as a comment and ignored. The # character denotes a comment only 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 question mark (?) 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 returns or sets the values of variables.

Syntax

```
set <varName> [<value>]
```

Arguments

- <varName> — (required) The name of a Tcl variable. The variable name relates to the following:
 - GUI preference variables. You can view a complete list of these variables within the GUI from the **Tools > Edit Preferences** menu selection.
 - Simulator control variables.

UserTimeUnit

IgnoreNote

CheckpointCompressMode

RunLength

IgnoreNote

NumericStdNoWarnings

IterationLimit

IgnoreError

StdArithNoWarnings

BreakOnAssertion

IgnoreFailure

PathSeparator

DefaultForceKind

DefaultRadix

WLFFilename

DelayFileOpen

WLFTimeLimit

WLFSizeLimit

If you do not specify a <value> this command will return the value of the <varName> you specify.

- <value> — (optional) The value to be assigned to the variable.

When you specify <value> you will change the current state of the <varName> you specify.

Description

Returns the value of variable *varName*. If you specify *value*, the command sets the value of *varName* to *value*, creating a new variable if one does not already exist, and returns its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the global command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare *varName* to be a namespace variable.

Command Substitution

Placing a command in square brackets ([]) will cause that command to be evaluated first and its results returned in place of the command. For example:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

This generates the following output:

```
"the result is 12"
```

Substitution allows you to obtain VHDL variables and signals, and Verilog nets and registers using the following construct:

```
[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 braces ({}) is not evaluated immediately. This is important for if-then-else statements, procedures, loops, and so forth.

Tcl Relational Expression Evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

```
if {[exa var_1] == 345}...
```

The following will also work:

```
if {[exa var_1] == "345"}...
```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

```
if {[exa var_2] == 001Z}...
```

will give an error.

```
if {[exa var_2] == "001Z"}...
```

will work okay.

- Do not quote single characters between apostrophes; use quotation marks instead. For example:

```
if {[exa var_3] == 'X'}...
```

will produce an error. However, the following:

```
if {[exa var_3] == "X"}...
```

will work.

- For the equal operator, you must use the C operator (==). For not-equal, you must use the C operator (!=).

Variable Substitution

When a `$<var_name>` is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

Note



Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)  
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See [modelsim.ini Variables](#) for more information about ModelSim-defined variables.

System Commands

To pass commands to the UNIX shell or DOS window, use the Tcl **exec** command:

```
echo The date is [exec date]
```

Simulator State Variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros). The variables are referenced in commands by prefixing the name with a dollar sign (\$).

argc

This variable returns the total number of parameters passed to the current macro.

architecture

This variable returns the name of the top-level architecture currently being simulated; for an optimized Verilog module, returns architecture name; for a configuration or non-optimized Verilog module, this variable returns an empty string.

configuration

This variable returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration.

delta

This variable returns the number of the current simulator iteration.

entity

This variable returns the name of the top-level VHDL entity or Verilog module currently being simulated.

library

This variable returns the library name for the current region.

MacroNestingLevel

This variable returns the current depth of macro call nesting.

n

This variable represents a macro parameter, where n can be an integer in the range 1-9.

Now

This variable always returns the current simulation time with time units (for example, 110,000 ns). Note: the returned value contains a comma inserted between thousands.

now

This variable returns the current simulation time with or without time units—depending on the setting for time resolution, as follows:

- When time resolution is a unary unit (such as 1ns, 1ps, 1fs), this variable returns the current simulation time without time units (for example, 100000).
- When time resolution is a multiple of the unary unit (such as 10ns, 100ps, 10fs), this variable returns the current simulation time with time units (for example, 110000 ns).

Note: the returned value does not contain a comma inserted between thousands.

resolution

This variable returns the current simulation time resolution.

Referencing Simulator State Variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign (\$). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 ps 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a backslash (\). For example, `\$now` will not be interpreted as the current simulator time.

Special Considerations for the now Variable

For the **when** command, special processing is performed on comparisons involving the **now** variable. If you specify "when {\$now=100}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

```
if { [gtTime $now 2us] } {  
.  
.  
.
```

See [Simulator Tcl Time Commands](#) for details on 64-bit time operators.

List Processing

In Tcl, a "list" is a set of strings in 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, as shown in [Table 33-3](#).

Table 33-3. Tcl List Commands

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, ..., 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, ..., just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, ...
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, ...

Two other commands, **lsearch** and **lsort**, are also available for list manipulation. See the Tcl man pages ([Help > Tcl Man Pages](#)) for more information on these commands.

See also the ModelSim Tcl command: [lecho](#)

Reading Variable Values From the INI File

You can read values from the *modelsim.ini* file with the following function:

```
GetPrivateProfileString <section> <key> <defaultValue>
```

Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetPrivateProfileString vsim
                               CheckpointCompressMode 1 modelsim.ini ]

set PrefMain(file) [GetPrivateProfileString vsim TranscriptFile ""
                                             modelsim.ini]
```

Simulator Tcl Commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided in [Table 33-4](#). For more information and command syntax see [Commands](#).

Table 33-4. Simulator-Specific Tcl Commands

Command	Description
alias	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find	locates incrTcl classes and objects
lecho	takes one or more Tcl lists as arguments and pretty-prints them to the Transcript pane
lshift	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv	echoes to the Transcript pane the current names and values of all environment variables

Simulator Tcl Time Commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (for example, 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. When values are smaller than the current Time Scale, the values are truncated to 0 and a warning is issued.

Conversions

Table 33-5. Tcl Time Conversion Commands

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Table 33-6. Tcl Time Relation Commands

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

Table 33-7. Tcl Time Arithmetic Commands

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 33-1](#) uses the Tcl **while** loop to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

Example 33-1. Tcl while Loop

```
set b [list]
set i [expr {[length $a] - 1}]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

[Example 33-2](#) uses the Tcl **for** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

Example 33-2. Tcl for Command

```
set b [list]
for {set i [expr {[length $a] - 1}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

[Example 33-3](#) 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):

Example 33-3. Tcl foreach Command

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

[Example 33-4](#) shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

Example 33-4. Tcl break Command

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

[Example 33-5](#) is a list reversal that skips a particular element by using the Tcl **continue** command:

Example 33-5. Tcl continue Command

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

[Example 33-6](#) works in UNIX only. In a Windows environment, the Tcl **exec** command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

Example 33-6. Access and Transfer System Information

(in VHDL source):

```
signal datetime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in macro):

```
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [clock format [clock seconds]]
    force -deposit datetime $s
    if {do_the_echo} {
        echo "New time is [examine -value datetime]"
    }
}

bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date
```

[Example 33-7](#) specifies the compiler arguments and lets you compile any number of files.

Example 33-7. Tcl Used to Specify Compiler Arguments

```
set Files [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    set lappend Files $1
    shift
}
eval vcom -93 -explicit -noaccel $Files
```

[Example 33-8](#) is an enhanced version of the last one. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a *.vhd* file extension.

Example 33-8. Tcl Used to Specify Compiler Arguments—Enhanced

```
set vhdFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    if {[string match *.vhd $1]} {
        lappend vhdFiles $1
    } else {
        lappend vFiles $1
    }
    shift
}
if {[llength $vhdFiles] > 0} {
    eval vcom -93 -explicit -noaccel $vhdFiles
}
if {[llength $vFiles] > 0} {
    eval vlog $vFiles
}
```

Macros (DO Files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools > TCL > Execute Macro** menu selection or the `do` command.

Creating DO Files

You can create DO files, like any other Tcl script, by doing one of the following:

- Type the required commands in any editor and save the file with the extension *.do*.
- Save the transcript as a DO file (refer to [Saving a Transcript File as a Macro \(DO file\)](#)).
- Use the [write format](#) restart command to create a *.do* file that will recreate all debug windows, all file/line breakpoints, and all signal breakpoints created with the [when](#) command.

All "event watching" commands (for example, [onbreak](#), [onerror](#), and so forth) must be placed before [run](#) commands within the macros in order to take effect.

The following is a simple DO file that was saved from the 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
onerror {cont}
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO Files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters \$1 through \$9 in the macro file. For example say the macro "*testfile*" contains the line **bp** \$1 \$2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do testfile design.vhd 127
```

There is a limit of 20 parameters that can be passed to macros, but only nine values are visible at one time. You can use the [shift](#) command to see the other parameters.

Deleting a File from a .do Script

To delete a file from a *.do* script, use the Tcl **file** command as follows:

```
file delete myfile.log
```

This will delete the file "*myfile.log*."

You can also use the **transcript file** command to perform a deletion:

```
transcript file ()  
transcript file my file.log
```

The first line will close the current log file. The second will open a new log file. If it has the same name as an existing file, it will replace the previous one.

Making Macro Parameters Optional

If you want to make macro parameters optional (that is, be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the **argc** simulator state variable. The **argc** simulator state variable returns the number of parameters passed. The examples below show several ways of using **argc**.

Example 33-9. Specifying Files to Compile With **argc** Macro

This macro specifies the files to compile and handles 0-2 compiler arguments as parameters. If you supply more arguments, ModelSim generates a message.

```
switch $argc {
  0 {vcom file1.vhd file2.vhd file3.vhd }
  1 {vcom $1 file1.vhd file2.vhd file3.vhd }
  2 {vcom $1 $2 file1.vhd file2.vhd file3.vhd }
  default {echo Too many arguments. The macro accepts 0-2 args. }
}
```

Example 33-10. Specifying Compiler Arguments With Macro

This macro specifies the compiler arguments and lets you compile any number of files.

```
variable Files ""
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set Files [concat $Files $1]
  shift
}
eval vcom -93 -explicit -noaccel $Files
```

Example 33-11. Specifying Compiler Arguments With Macro—Enhanced

This macro is an enhanced version of the one shown in example 2. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a *.vhd* file extension.

```

variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist 0
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    set vhdFiles [concat $vhdFiles $1]
    set vhdFilesExist 1
  } else {
    set vFiles [concat $vFiles $1]
    set vFilesExist 1
  }
  shift
}
if {$vhdFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
  eval vlog $vFiles
}

```

Useful Commands for Handling Breakpoints and Errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The commands in [Table 33-8](#) may be useful for handling such events. (Any other legal command may be executed as well.)

Table 33-8. Commands for Handling Breakpoints and Errors in Macros

command	result
run -continue	continue as if the breakpoint had not been executed, completes the run that was interrupted
resume	continue running the macro
onbreak	specify a command to run when you hit a breakpoint within a macro
onElabError	specify a command to run when an error is encountered during elaboration
onerror	specify a command to run when an error is encountered within a macro
status	get a traceback of nested macro calls when a macro is interrupted
abort	terminate a macro once the macro has been interrupted or paused

Table 33-8. Commands for Handling Breakpoints and Errors in Macros

command	result
pause	cause the macro to be interrupted; the macro can be resumed by entering a resume command via the command line
transcript	control echoing of macro commands to the Transcript pane

You can also set the `OnErrorDefaultAction` Tcl variable to determine what action ModelSim takes when an error occurs. To set the variable on a permanent basis, you must define the variable in a *modelsim.tcl* file (see [The modelsim.tcl File](#) for details).

Error Action in DO Files

If a command in a macro returns an error, ModelSim does the following:

1. If an [onerror](#) command has been set in the macro script, ModelSim executes that command. The `onerror` command must be placed prior to the run command in the DO file to take effect.
2. If no [onerror](#) command has been specified in the script, ModelSim checks the `OnErrorDefaultAction` variable. If the variable is defined, its action will be invoked.
3. If neither 1 or 2 is true, the macro aborts.

Using the Tcl Source Command with DO Files

Either the `do` command or Tcl source command can execute a DO file, but they behave differently.

With the Tcl 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.

The Tcl Debugger

Note



Mentor Graphics would like to acknowledge the contribution from Gregor Schmid for making TDebug available for use in the public domain.

The TDebug program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of FITNESS FOR A PARTICULAR PURPOSE.

Starting the Debugger

Select **Tools > TCL > Tcl Debugger** to run the debugger. Make sure you use the ModelSim and TDebug menu selections to invoke and close the debugger. If you would like more information on the configuration of TDebug see **Help > Technotes > tdebug**.

The following text is an edited summary of the README file distributed with TDebug.

How the debugger 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 and so forth. The advantages are that TDebug knows which statement in which procedure is currently being executed and can give visual feedback by highlighting it. All currently accessible variables and their values are displayed as well. Code can be evaluated in the context of the current procedure. Breakpoints can be set and deleted with the mouse.

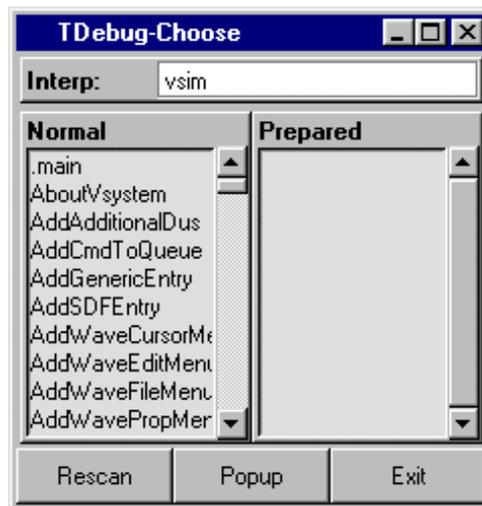
Unfortunately there are drawbacks to this approach. Preparation of large procedures is slow and due to Tcl's dynamic nature there is no guarantee that a procedure can be prepared at all. This problem has been alleviated somewhat with the introduction of partial preparation of procedures. There is still no possibility to get at code running in the global context.

The Chooser

Select **Tools > TCL > Tcl Debugger** 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.

Figure 33-1. TDebug Choose Dialog



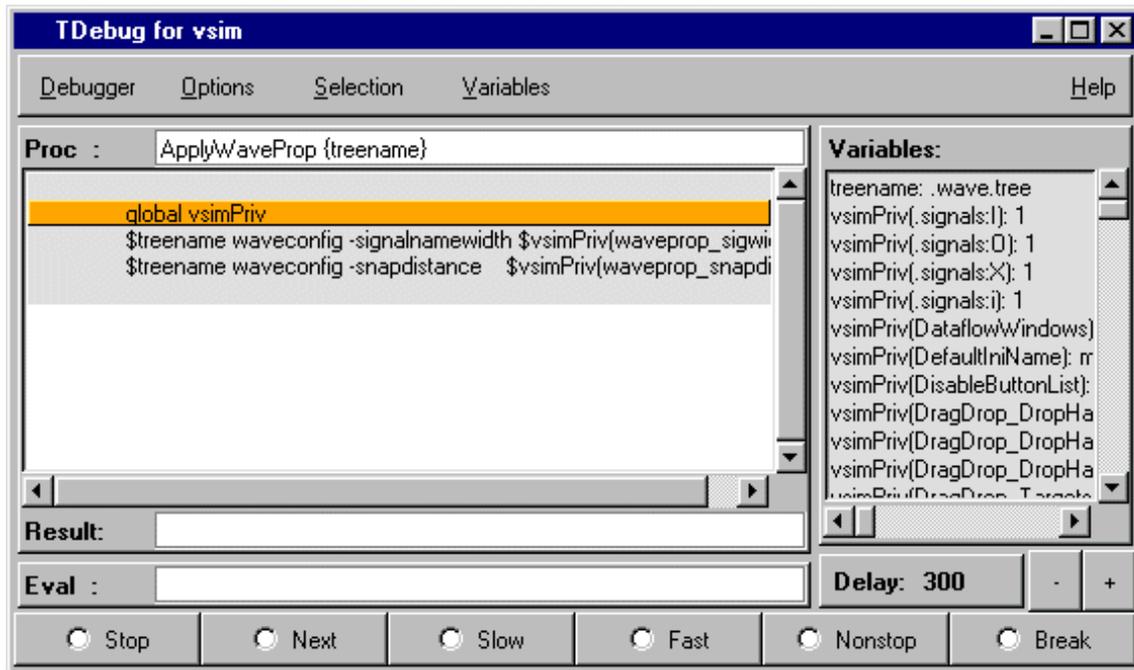
Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.

The three buttons at the bottom let you force a **Rescan** of the available procedures, **Popup** the debugger window or **Exit** TDebug. Exiting from TDebug doesn't terminate ModelSim, it merely detaches from *vsim.op_*, restoring all prepared procedures to their unmodified state.

The Debugger

Select the **Popup** button in the Chooser to open the debugger window ([Figure 33-2](#)).

Figure 33-2. Tcl Debugger for vsim



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 shown in [Table 33-9](#).

Table 33-9. Tcl Debug States

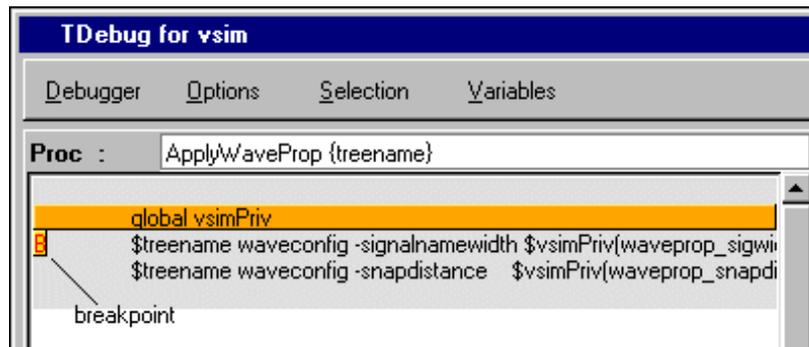
Button	Description
Stop	stop after next expression, used to get out of slow/fast/nonstop mode
Next	execute one expression, then revert to idle
Slow	execute until end of procedure, stopping at breakpoints or when the state changes to stop; after each execution, stop for `delay' milliseconds; the delay can be changed with the `+' and `-' buttons
Fast	execute until end of procedure, stopping at breakpoints
Nonstop	execute until end of procedure without stopping at breakpoints or updating the display
Break	terminate execution of current procedure

Closing the debugger doesn't quit it, it only does `wm withdraw'. The debugger window will pop up the next time a prepared procedure is called. Make sure you close the debugger with **Debugger > Close**.

Breakpoints

To set/unset a breakpoint, double-click inside the listing. The breakpoint will be set at the innermost available expression that contains the position of the click. Conditional or counted breakpoints aren't supported.

Figure 33-3. Setting a Breakpoint in the Debugger

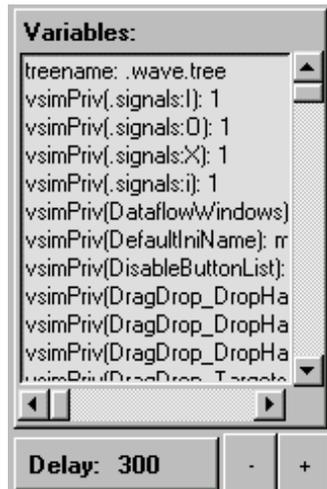


The **Eval** entry supports a simple history mechanism available via the <Up_arrow> and <Down_arrow> keys. If you evaluate a command while stepping through a procedure, the command will be evaluated in the context of the procedure; otherwise it will be evaluated at the

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).

Figure 33-4. Variables Dialog Box



Configuration

You can customize TDebug by setting up a file named `.tdebugrc` in your home directory. See the TDebug README at **Help > Technotes > tdebug** for more information on the configuration of TDebug.

TclPro Debugger

The Tools menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger and any available documentation can be acquired from Scriptics. Once acquired, do the following steps to use the TclPro Debugger:

1. Make sure the TclPro bin directory is in your PATH.
2. In TclPro Debugger, create a new project with Remote Debugging enabled.
3. Start ModelSim and select **Tools > TclPro Debugger**.
4. Press the Stop button in the debugger in order to set breakpoints, and so forth.

Note



TclPro Debugger version 1.4 does not work with ModelSim.

Appendix A

modelsim.ini Variables

This chapter covers the contents and modification of the *modelsim.ini* file.

- [Organization of the modelsim.ini File](#) — A list of the different sections of the *modelsim.ini* file.
- [Making Changes to the modelsim.ini File](#) — How to modify variable settings in the *modelsim.ini* file.
- [Variables](#) — An alphabetized list of *modelsim.ini* variables and their properties.
- [Commonly Used modelsim.ini Variables](#) — A discussion of the most frequently used variables and their settings.

Organization of the modelsim.ini File

The *modelsim.ini* file is the default initialization file and contains control variables that specify reference library paths, optimization, compiler and simulator settings, and various other functions. It is located in your install directory and is organized into the following sections.

- **The [library]** section contains variables that specify paths to various libraries used by ModelSim.
- **The [vcom]** section contains variables that control the compilation of VHDL files.
- **The [vlog]** section contains variables that control the compilation of Verilog files.
- **The [sccom]** section contains variables that control the compilation of SystemC files.
- **The [vopt]** section contains variables that control optimization.
- **The [vsim]** section contains variables that control the simulator.
- **The [lmc] section** contains variables that control the interface between the simulator and Logic Modeling's SmartModel SWIFT software.
- **The [msg_system]** section contains variables that control the severity of notes, warnings, and errors that come from **vcom**, **vlog** and **vsim**.

The [sccom], [vcom], and [vlog] sections contain **compiler control variables**.

The [vopt] section contains **optimization variables**.

The [vsim] section contains **simulation control variables**.

The System Initialization chapter contains [Environment Variables](#).

Making Changes to the modelsim.ini File

Modify *modelsim.ini* variables by:

- Changing the settings in the [The Runtime Options Dialog](#).
- [Editing modelsim.ini Variables](#).

The Read-only attribute must be turned off to save changes to the *modelsim.ini* file.

Changing the modelsim.ini Read-Only Attribute

When first installed, the *modelsim.ini* file is protected as a Read-only file. In order to make and save changes to the file the Read-only attribute must first be turned off in the *modelsim.ini* Properties dialog box.

Procedure

1. Navigate to the location of the *modelsim.ini* file.
2. <install directory>/modelsim.ini
3. Right-click on the *modelsim.ini* file and choose **Properties** from the popup menu.
4. This displays the *modelsim.ini* Properties dialog box.
5. Uncheck the Attribute: **Read-only**.
6. **Click OK**

To protect the *modelsim.ini* file after making changes, follow the above steps and at step 5, check the **Read-only** attribute.

The Runtime Options Dialog

To access, select **Simulate > Runtime Options** in the Main window. The dialog contains three tabs - Defaults, Severity, and WLF Files.

The **Runtime Options** dialog writes changes to the active *modelsim.ini* file that affect the current session. If the read-only attribute for the *modelsim.ini* file is turned off, the changes are saved, and affect all future sessions. See [Changing the modelsim.ini Read-Only Attribute](#).

Figure A-1. Runtime Options Dialog: Defaults Tab

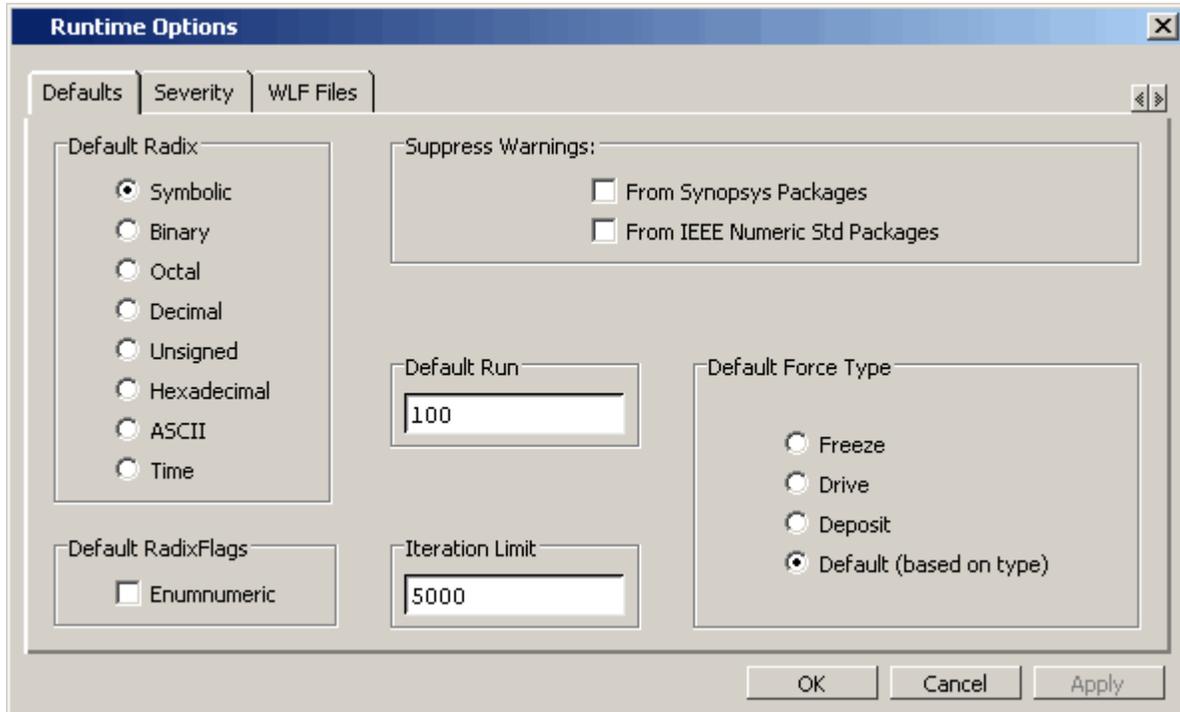


Table A-1. Runtime Option Dialog: Defaults Tab Contents

Option	Description
Default Radix	Sets the default radix for the current simulation run. The chosen radix is used for all commands (force , examine , change are examples) and for displayed values in the Objects, Locals, Dataflow, Schematic, List, and Wave windows, as well as the Source window in the source annotation view. The corresponding <i>modelsim.ini</i> variable is DefaultRadix . You can override this variable with the radix command.
Default Radix Flags	Displays SystemVerilog and SystemC enums as numbers rather than strings. This option overrides the global setting of the default radix. You can override this variable with the add list -radixenumsymbolic .

Table A-1. Runtime Option Dialog: Defaults Tab Contents

Option	Description
Suppress Warnings	<p>From Synopsys Packages suppresses warnings generated within the accelerated Synopsys std_arith packages. The corresponding <i>modelsim.ini</i> variable is StdArithNoWarnings.</p> <p>From IEEE Numeric Std Packages suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. The corresponding <i>modelsim.ini</i> variable is NumericStdNoWarnings.</p>
Default Run	Sets the default run length for the current simulation. The corresponding <i>modelsim.ini</i> variable is RunLength . You can override this variable by specifying the run command.
Iteration Limit	Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. The corresponding <i>modelsim.ini</i> variable is IterationLimit .
Default Force Type	Selects the default force type for the current simulation. The corresponding <i>modelsim.ini</i> variable is DefaultForceKind . You can override this variable by specifying the force command argument -default , -deposit , -drive , or -freeze .

Figure A-2. Runtime Options Dialog Box: Severity Tab



Table A-2. Runtime Option Dialog: Severity Tab Contents

Option	Description
Immediate Assertion Break Severity	Selects the Verilog and VHDL immediate assertion severity level that will stop simulation. The corresponding <i>modelsim.ini</i> variable is BreakOnAssertion . 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.
No Message Display For - Verilog	Selects the SystemVerilog concurrent assertion severity for which messages will not be displayed. Multiple selections are possible. Corresponding <i>modelsim.ini</i> variables are IgnoreSVAFatal , IgnoreSVAError , IgnoreSVAWarning , and IgnoreSVAInfo .
No Message Display For -VHDL	Selects the VHDL assertion severity for which messages will not be displayed (even if break on assertion is set for that severity). Multiple selections are possible. The corresponding <i>modelsim.ini</i> variables are IgnoreFailure , IgnoreError , IgnoreWarning , and IgnoreNote .

Figure A-3. Runtime Options Dialog Box: WLF Files Tab

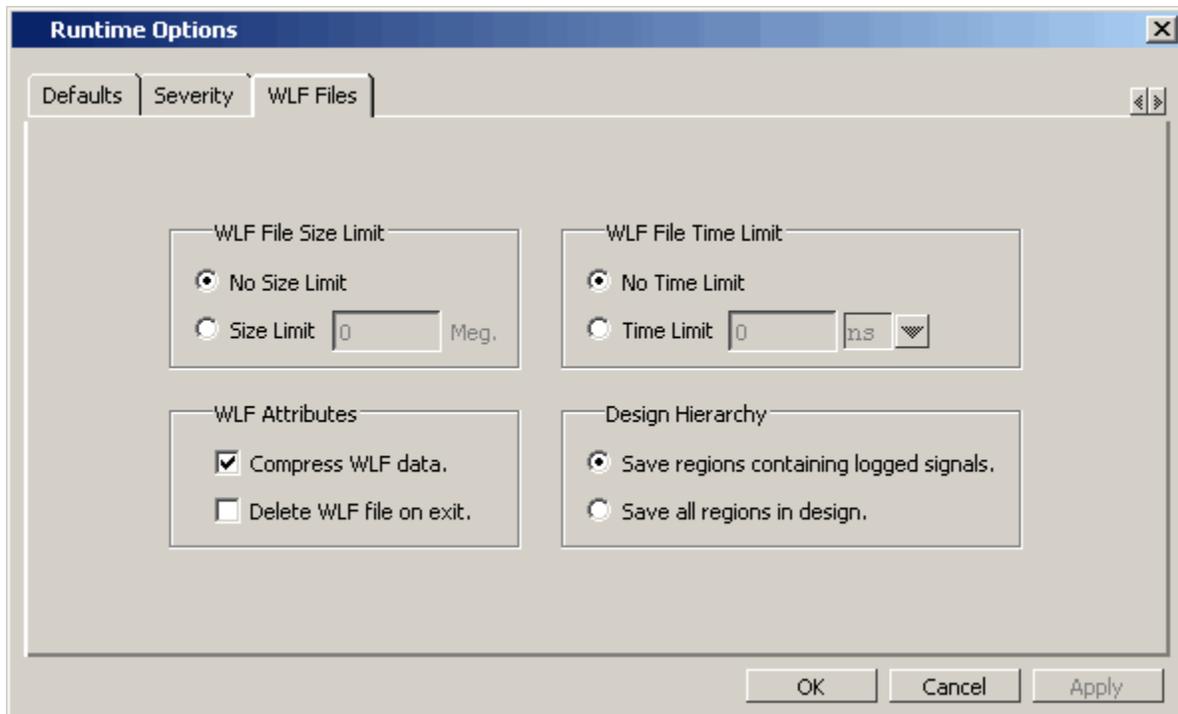


Table A-3. Runtime Option Dialog: WLF Files Tab Contents

Option	Description
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. The corresponding <i>modelsim.ini</i> variable is WLFSizeLimit .
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. The corresponding <i>modelsim.ini</i> variable is WLFTimeLimit .
WLF Attributes	Specifies whether to compress WLF files and whether to delete the WLF file when the simulation ends. You would typically only disable compression for troubleshooting purposes. The corresponding <i>modelsim.ini</i> variables are WLFCompress for compression and WLFDeleteOnQuit for WLF file deletion.
Design Hierarchy	Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. The corresponding <i>modelsim.ini</i> variable is WLFSaveAllRegions .

Editing modelsim.ini Variables

The syntax for variables in the file is:

<variable> = <value>

Procedure

1. Open the *modelsim.ini* file with a text editor.
2. Find the variable you want to edit in the appropriate section of the file.
3. Type the new value for the variable after the equal (=) sign.
4. If the variable is commented out with a semicolon (;) remove the semicolon.
5. Save.

Overriding the Default Initialization File

You can make changes to the working environment during a work session by loading an alternate initialization file that replaces the default *modelsim.ini* file. This file overrides the file and path specified by the MODELSIM environment variable.

Procedure

1. Open the *modelsim.ini* file with a text editor.
2. Make changes to the *modelsim.ini* variables.
3. Save the file with an alternate name to any directory.
4. After start up of the tool, specify the `-modelsimini <ini_filepath>` switch with one of the following commands:

Table A-4. Commands for Overriding the Default Initialization File

Simulator Commands	Compiler Commands	Utility Commands
<code>vsim</code>	<code>scom</code> <code>vcom</code> <code>vlog</code> <code>vopt</code>	<code>scgenmod</code> <code>vcover attribute</code> <code>vcover merge</code> <code>vcover ranktest</code> <code>vcover report</code> <code>vcover stats</code> <code>vcover testnames</code> <code>vdel</code> <code>vdir</code> <code>vgencomp</code> <code>vmake</code>

See the <command> **-modelsimini** argument description for further information.

Variables

The *modelsim.ini* variables are listed in order alphabetically. The following information is given for each variable.

- A short description of how the variable functions.
- The location of the variable, by section, in the *modelsim.ini* file.
- The syntax for the variable.
- A listing of all values and the default value where applicable.
- Related arguments that are entered on the command line to override variable settings. Commands entered at the command line always take precedence over *modelsim.ini* settings. Not all variables have related command arguments.
- Related topics and links to further information about the variable.

AcceptLowerCasePragmaOnly

This variable instructs the Verilog compiler to accept only lower case pragmas in Verilog source files.

Section [vlog]

Syntax

AcceptLowerCasePragmaOnly = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vlog -lowercasepragma`.

AddPragmaPrefix

This variable enables recognition of synthesis and coverage pragmas with a user specified prefix. If this argument is not specified, pragmas are treated as comments and the previously excluded statements included in the synthesized design. All regular synthesis and coverage pragmas are honored.

Section [vcom], [vlog]

Syntax

AddPragmaPrefix = <prefix>

<prefix> — Specifies a user defined string where the default is no string, indicated by quotation marks (“”).

AmsStandard

This variable specifies whether `vcom` adds the declaration of `REAL_VECTOR` to the `STANDARD` package. This is useful for designers using VHDL-AMS to test digital parts of their model.

Section [vcom]

Syntax

AmsStandard = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vcom {-amsstd | -noamsstd}`.

Related Topics

[MGC_AMS_HOME](#)

AssertFile

This variable specifies an alternative file for storing VHDL/PSL/SVA assertion messages. By default, assertion messages are output to the file specified by the [TranscriptFile](#) variable in the *modelsim.ini* file (refer to “[Creating a Transcript File](#)”). If the `AssertFile` variable is specified, all assertion messages will be stored in the specified file, not in the transcript.

Section [vsim]

Syntax

`AssertFile = <filename>`

`<filename>` — Any valid file name containing assertion messages, where the default name is *assert.log*.

You can override this variable by specifying `vsim -assertfile`.

AssertionActiveThreadMonitor

This variable enables tracking of currently active assertion threads for a given instance.

Section [vsim]

Syntax

`AssertionActiveThreadMonitor = {0 | 1}`

0 — Tracking is disabled.

1 — Tracking is enabled. (default)

Related Topics

[Using the Assertion Active Thread Monitor](#)

AssertionActiveThreadMonitorLimit

This variable limits the number of active assertion threads displayed for a given instance.

Section [vsim]

Syntax

`AssertionActiveThreadMonitorLimit = <n>`

`<n>` — Any positive integer, where 5 is the default.

Related Topics

[Using the Assertion Active Thread Monitor](#)

AssertionCover

This variable enables extended count information for assertions.

Section [vsim]

Syntax

AssertionCover = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying **vsim -assertcover** or **-noassertcover** at the command line.

AssertionDebug

This variable specifies that assertion passes are reported and enables debug options such as assertion thread viewing (ATV), HDL failed expression analysis, extended count information, and causality traceback.

Section [vsim]

Syntax

AssertionDebug = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying **vsim -assertdebug** or **-noassertdebug** at the command line.

Related Topics

AssertionEnable

This variable enables VHDL/PSL/SVA assertions.

Section [vsim]

Syntax

AssertionEnable = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [assertion enable -off](#).

Passes and failures cannot be enabled or disabled independently. So if **AssertionEnable** is used, both passes and failures are enabled or disabled.

AssertionEnableVacuousPassActionBlock

This variable enables execution of assertion pass actions for vacuous passes in action blocks.

Section [vsim]

Syntax

AssertionEnableVacuousPassActionBlock = {0 | 1 }

0 — Off (default)

1 — On

You may override this variable, when it is turned on (1), by specifying [assertion action -actionblock vacuousoff](#).

AssertionFailAction

This variable sets an action for a PSL/SVA failure event.

Section [vsim]

Syntax

AssertionFailAction = {0 | 1 | 2 }

0 — Continue (default)

1 — Break

2 — Exit

You can override this variable by specifying [assertion fail -action](#).

AssertionFailLocalVarLog

This variable prints SVA concurrent assertion local variable values corresponding to failed assertion threads when you run [vsim -assertdebug](#).

Section [vsim]

Syntax

AssertionFailLocalVarLog = {0 | 1 }

0 — Off

1 — On (default)

You can override this variable by specifying [assertion fail -lvlog](#).

AssertionFailLog

This variable enables transcript logging for PSL assertion failure events.

Section [vsim]

Syntax

AssertionFailLog = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [assertion fail -log](#).

AssertionLimit

This variable sets a limit for the number of times ModelSim responds to a VHDL/PSL/SVA assertion failure event. ModelSim disables an assertion after reaching the limit.

Section [vsim]

Syntax

AssertionLimit = <n>

<n> — Any positive integer where the default is -1 (unlimited).

You can override this variable by specifying [assertion fail -limit](#).

AssertionPassLog

This variable enables logging of SystemVerilog and PSL assertion pass events.

Section [vsim]

Syntax

AssertionPassLog = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [assertion pass -log](#).

AssertionThreadLimit

This variable sets a limit on the number of threads logged for each assertion. If the number of threads logged for an assert directive exceeds the limit, the assertion is either killed or switched off as specified by the [AssertionThreadLimitAction](#) variable.

Section [vsim]

Syntax

AssertionThreadLimit = <n>

<n> — Any positive integer where the default is -1 (unlimited hits)

AssertionThreadLimitAction

This variable controls the action taken once the assert limit set by the [AssertionThreadLimit](#) variable has been reached.

Section [vsim]

Syntax

AssertionThreadLimitAction = {kill | off}

kill — (default) All existing threads for an assertion are terminated and no new instances of the assertion are started.

off — All current assertions and threads are kept but no new instances of the assertion are started. Existing instances of the assertion continue to be evaluated and generate threads.

Related Topics

[assertion enable -off](#)

ATVStartTimeKeepCount

This variable controls how many thread start times will be preserved for ATV viewing for a given assertion instance.

Section [vsim]

Syntax

ATVStartTimeKeepCount = <n>

<n> — Any non-negative number where the default is -1 (all).

AutoExclusionsDisable

This variable is used to control automatic code coverage exclusions. By default, assertions and FSMs are excluded from the code coverage. For FSMs, all transitions to and from excluded states are also automatically excluded. When “all” is selected, code coverage is enabled for both assertions and FSMs.

Section [vsim]

Syntax

AutoExclusionsDisable = {assertions | fsm | all}

assertions — Enable code coverage for assertions.

fsm — Enable code coverage for FSMs.

all — Enable code coverage for all automatic exclusions.

To enable multiple values, use a comma or space separated list.

You can override this variable by specifying [vsim -autoexclusionsdisable](#).

BindAtCompile

This variable instructs ModelSim to perform VHDL default binding at compile time rather than load time.

Section [vcom]

Syntax

BindAtCompile = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vcom {-bindAtCompile | -bindAtLoad}](#).

Related Topics

[Default Binding](#)

[RequireConfigForAllDefaultBinding](#)

BreakOnAssertion

This variable stops the simulator when the severity of a VHDL assertion message or a SystemVerilog severity system task is equal to or higher than the value set for the variable. It also controls any messages in the source code that use *assertion_failure_**. For example, since most runtime messages use some form of *assertion_failure_**, any runtime error will cause the simulation to break if the user sets BreakOnAssertion = 2 (error).

Section [vsim]

Syntax

BreakOnAssertion = {0 | 1 | 2 | 3 | 4}

0 — Note

1 — Warning

2 — Error

3 — Failure (default)

4 — Fatal

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl Command Syntax](#).

CheckPlusargs

This variable defines the simulator's behavior when encountering unrecognized plusargs. The simulator checks the syntax of all system-defined plusargs to ensure they conform to the syntax defined in the Reference Manual. By default, the simulator does not check syntax or issue warnings for unrecognized plusargs (including accidentally misspelled, system-defined plusargs), because there is no way to distinguish them from a user-defined plusarg.

Section [vsim]

Syntax

CheckPlusargs = {0 | 1 | 2}

0 — Ignore (default)

1 — Issues a warning and simulates while ignoring.

2 — Issues an error and exits.

CheckpointCompressMode

This variable specifies that checkpoint files are written in compressed format.

Section [vsim]

Syntax

CheckpointCompressMode = {0 | 1}

0 — Off

1 — On (default)

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

CheckSynthesis

This variable turns on limited synthesis rule compliance checking, which includes checking only signals used (read) by a process and understanding only combinational logic, not clocked logic.

Section [vcom]

Syntax

CheckSynthesis = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vcom -check_synthesis](#).

ClassDebug

This variable enables visibility into and tracking of class instances.

Section [vsim]

Syntax

ClassDebug = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vsim -classdebug](#).

CodeCoverage

This variable enables code coverage.

Section [vsim]

Syntax

CodeCoverage = {0 | 1}

0 — Off

1 — On (default)

CommandHistory

This variable specifies the name of a file in which to store the Main window command history.

Section [vsim]

Syntax

CommandHistory = <filename>

<filename> — Any string representing a valid filename where the default is *cmdhist.log*.

The default setting for this variable is to comment it out with a semicolon (;).

CompilerTempDir

This variable specifies a directory for compiler temporary files instead of “work/_temp.”

Section [vcom]

Syntax

CompilerTempDir = <directory>

<directory> — Any user defined directory where the default is work/_temp.

ConcurrentFileLimit

This variable controls the number of VHDL files open concurrently. This number should be less than the current limit setting for maximum file descriptors.

Section [vsim]

Syntax

ConcurrentFileLimit = <n>

<n> — Any non-negative integer where 0 is unlimited and 40 is the default.

Related Topics

[Syntax for File Declaration](#)

Coverage

This variable enables coverage statistic collection.

Section [vcom], [vlog], [vopt]

Syntax

Coverage = {0 | s | b | c | e | f | t}

0 — Off (default)

s — statement
b — branch
c — condition
e — expression
f — fsm
t — toggle

CoverAtLeast

This variable specifies the minimum number of times a functional coverage directive must evaluate to true.

Section [vsim]

Syntax

CoverAtLeast = <n>

<n> — Any positive integer where the default is 1.

CoverCells

This variable enables code coverage of Verilog modules defined by 'celldefine and 'endcelldefine compiler directives.

Section [vlog]

Syntax

CoverCells = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vlog {-covercells|-nocovercells}` or `vopt {-covercells|-nocovercells}`.

Related Topics

[Verilog-XL Compatible Compiler Arguments](#)

CoverClkOptBuiltins

This variable enables clkOpt optimization builtins for code coverage.

Section [vcom]

Syntax

CoverClkOptBuiltins = {0 | 1}

0 — Off

1 — On (default)

CoverCountAll

This variable applies to condition and expression coverage UDP tables. Thus, it has no effect unless UDP is enabled for coverage with vcom/vlog/vopt -coverudp. If this variable is turned off (0) and a match occurs in more than one row, none of the counts for all matching rows is incremented. By default, counts are incremented for all matching rows.

Section [vsim]

Syntax

CoverCountAll = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vsim -covercountnone](#).

Related Topics

[Verilog-XL Compatible Compiler Arguments](#)

CoverEnable

This variable specifies that all PSL/SVA coverage directives in the current simulation are enabled.

Section [vsim]

Syntax

CoverEnable = {0 | 1}

0 — Off

1 — On (default)

CoverExcludeDefault

This variable excludes VHDL code coverage data collection from the OTHERS branch in both Case statements and Selected Signal Assignment statements.

Sections [vcom], [vlog]

Syntax

CoverExcludeDefault = {0 | 1}
0 — Off (default)
1 — On

CoverFEC

This variable controls the collection of code coverage for focused expression and condition coverage statistics.

Sections [vcom], [vlog]

Syntax

CoverFEC = {0 | 1}
0 — Off
1 — On (default)

You can override this variable by specifying [vcom](#), [vlog](#), or [vopt -nocoverfec](#).

CoverLimit

This variable specifies the number of cover directive hits before the directive is auto disabled.

Section [vsim]

Syntax

CoverLimit = <n>
<n> — Any positive integer where the default is -1 (unlimited hits).

CoverLog

This variable enables transcript logging for functional coverage directive messages.

Section [vsim]

Syntax

CoverLog = {0 | 1}
0 — Off (default)
1 — On

CoverOpt

This variable controls the default level of optimizations for compilations with code coverage.

Sections [vcom], [vlog], [vopt]

Syntax

CoverOpt = {0 | 1 | 2 | 3 | 4 | 5}

- 0 — Turns off Verilog module inlining and VHDL arch lining.
- 1 — Turns off continuous assignment optimizations and clock suppression.
- 2 — Turn off expression optimization, convertint primitives to continuous assignment, VHDL subprogram inlining and VHDL clkOpt (converting FF's to builtins).
- 3 — (default) Turns off process, always block and if statement merging.
- 4 — Turns off removal of unreferenced code.
- 5 — Turns on all allowable optimizations, 0 - 4.

You can override this variable by specifying the **vcom**, **vlog**, or **vopt** command with the **-coveropt** argument.

Note



If fsm coverage is turned on, optimizations are forced to level 3 and conversion of primitives to continuous assigns is turned off.

Related Topics

[vlog +cover](#)

CoverRespectHandL

This variable specifies whether you want the VHDL 'H' and 'L' input values on conditions and expressions to be automatically converted to '1' and '0', respectively.

This variable controls the default level of optimizations for compilations with code coverage.

Section: [vcom]

Syntax

CoverRespectHandL = {0 | 1}

- 0 — On
- 1 — Off (default) H and L values are not automatically converted.

If you are not using 'H' and 'L' values and do not want the additional UDP rows that are difficult to cover — you can:

Variables

- Change your VHDL expressions of the form (a = '1') to (to_x01(a) = '1') or to std_match(a,'1'). These functions are recognized and used to simplify the UDP tables.
- Override this variable by specifying **vcom -nocoverrespecthandl**.

CoverReportCancelled

This variable Enables code coverage reporting of branch conditions that have been optimized away due to a static or null condition. The line of code is labeled EA in the Source Window and EBCS in the hits column in a Coverage Report.

Sections [vcom], [vlog], [vopt]

Syntax

CoverReportCancelled = {0 | 1}

0 — (default) Do not report code that has been optimized away.

1— Enable code coverage reporting of code that has been optimized away.

Related Topics

[vcom -coverreportcancelled](#)

[vlog -coverreportcancelled](#)

[vopt -coverreportcancelled](#)

[coverage report](#)

CoverShortCircuit

This variable enables short-circuiting of expressions when coverage is enabled.

Sections [vcom], [vlog]

Syntax

CoverShortCircuit = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying either the **vcom** or **vlog** command with the **-nocovershort** argument.

CoverSub

This variable controls the collection of code coverage statistics in VHDL subprograms.

Section [vcom]

Syntax

CoverSub = {0 | 1}

0 — Off

1 — On (default)

CoverThreadLimit

This variable sets a limit on the number of threads logged for each cover directive. If the number of threads logged for a cover directive exceeds the limit, the assertion is either killed or switched off as specified by the [CoverThreadLimitAction](#) variable.

Section [vsim]

Syntax

CoverThreadLimit = <n>

<n> — Any positive integer where the default is -1 (unlimited hits)

CoverThreadLimitAction

This variable controls the action taken once the cover directive limit set by the [CoverThreadLimit](#) variable has been reached.

Section [vsim]

Syntax

AssertionThreadLimitAction = {kill | off}

kill — (default) All existing threads for an assertion are terminated and no new instances of the assertion are started.

off — All current assertions and threads are kept but no new instances of the assertion are started. Existing instances of the assertion continue to be evaluated and generate threads.

Related Topics

[assertion enable](#) -off

CoverUDP

This variable controls the collection of code coverage for UDP expression and condition coverage statistics. By default, UDP coverage is not collected when expression and/or condition coverage is active, and can be enabled on a select basis using the vcom/vlog/vopt -coverudp argument.

Sections [vcom], [vlog]

Syntax

CoverUDP = {0 | 1}

0 — Off (default)

1 — On

CoverWeight

This variable specifies the relative weighting for functional coverage directives.

Section [vsim]

Syntax

CoverWeight = <n>

<n> — Any non-negative integer, where the default is 1.

CppOptions

This variable adds any specified C++ compiler options to the **sccom** command line at the time of invocation.

Section [sccom]

Syntax

CppOptions = <options>

<options> — Any normal C++ compiler options where the default is -g (enable source debugging).

You turn this variable off by commenting the variable line in the *modelsim.ini* file.

Related Topics

[sccom <CPP compiler options>](#)

CppPath

This variable should point directly to the location of the g++ executable, such as:

```
CppPath = /usr/bin/g++
```

This variable is not required when running SystemC designs. By default, you should install and use the built-in g++ compiler that comes with ModelSim.

Section [sccom]

Syntax

CppPath = <path>

<path> — The path to the g++ executable.

CreateDirForFileAccess

This variable controls whether the Verilog system task \$fopen or vpi_mcd_open() will create a non-existent directory when opening a file in append (a), or write (w) modes.

Section [vsim]

Syntax

CreateDirForFileAccess = {0 | 1}

0 — Off (default)

1 — On

Related Topics

[New Directory Path With \\$fopen](#)

CvgZWNNoCollect

This variable controls coverage collection for any coverage item (coverpoint, cross, or the entire covergroup) when 0 is assigned as its *option.weight*.

Section [vsim]

Syntax

CvgZWNNoCollect = {0 | 1}

0 — Off. (default) Collect coverage data for zero-weight coverage items as normal.

1 — On. Disables collection of coverage data for the zero-weight coverage item. Zero-weight coverage items will not be displayed in any coverage report or contribute to any coverage score computation.

You can override this variable with the -cvgzwnocollect argument to [vsim](#)

DatasetSeparator

This variable specifies the dataset separator for fully-rooted contexts, for example:

```
sim:/top
```

The variable for DatasetSeparator must not be the same character as the [PathSeparator](#) variable, or the [SignalSpyPathSeparator](#) variable.

Section [vsim]

Syntax

DatasetSeparator = <character>

<character> — Any character except special characters, such as backslash (\), brackets ({}), and so forth, where the default is a colon (:).

DefaultForceKind

This variable defines the kind of force used when not otherwise specified.

Section [vsim]

Syntax

DefaultForceKind = {default | deposit | drive | freeze}

default — Uses the signal kind to determine the force kind.

deposit — Sets the object to the specified value.

drive — **Default for resolved signals.**

freeze — Default for unresolved signals.

You can override this variable by specifying **force** {-default | -deposit | -drive | -freeze}.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the Tcl [set Command Syntax](#).

DefaultRadix

This variable allows a numeric radix to be specified as a name or number. For example, you can specify binary as “binary” or “2” or octal as “octal” or “8”.

Section [vsim]

Syntax

DefaultRadix = {ascii | binary | decimal | hexadecimal | octal | symbolic | unsigned}

ascii — Display values in 8-bit character encoding.

binary— Display values in binary format. You can also specify 2.

decimal or 10 — Display values in decimal format. You can also specify 10.

hexadecimal— Display values in hexadecimal format. You can also specify 16.

octal— Display values in octal format. You can also specify 8.

symbolic — (*default*) Display values in a form closest to their natural format.

unsigned — Display values in unsigned decimal format.

You can override this variable by specifying **radix** {**ascii** | **binary** | **decimal** | **hexadecimal** | **octal** | **symbolic** | **unsigned**}.

Related Topics

You can set this variable in the [The Runtime Changing Radix \(base\) for the Wave Window Options Dialog](#).

You can set this variable interactively with the Tcl [set Command Syntax](#).

DefaultRestartOptions

This variable sets the default behavior for the restart command.

Section [vsim]

Syntax

```
DefaultRestartOptions = {-force | -noassertions | -nobreakpoint | -nofcovers | -nolist | -nolog | -nowave}
```

-force — Restart simulation without requiring confirmation in a popup window.

-noassertions — Restart simulation without maintaining the current assert directive configurations.

-nobreakpoint — Restart simulation with all breakpoints removed.

-nofcovers — Restart without maintaining the current cover directive configurations.

-nolist — Restart without maintaining the current List window environment.

-nolog — Restart without maintaining the current logging environment.

-nowave — Restart without maintaining the current Wave window environment.

semicolon (;) — Default is to prevent initiation of the variable by commenting the variable line.

You can specify one or more value in a space separated list.

You can override this variable by specifying **restart** {**-force** | **-noassertions** | **-nobreakpoint** | **-nofcovers** | **-nolist** | **-nolog** | **-nowave**}.

Related Topics

[checkpoint](#) command

[vsim -restore](#)

[Checkpointing and Restoring Simulations](#)

DelayFileOpen

This variable instructs ModelSim to open VHDL87 files on first read or write, else open files when elaborated.

Section [vsim]

Syntax

DelayFileOpen = {0 | 1}

0 — On (default)

1 — Off

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

displaymsgmode

This variable controls where the simulator outputs system task messages. The display system tasks displayed with this functionality include: \$display, \$strobe, \$monitor, \$write as well as the analogous file I/O tasks that write to STDOUT, such as \$fwrite or \$fdisplay.

Section [msg_system]

Syntax

displaymsgmode = {both | tran | wlf}

both — Outputs messages to both the transcript and the WLF file.

tran — (default) Outputs messages only to the transcript, therefore they are unavailable in the Message Viewer.

wlf — Outputs messages only to the WLF file/Message Viewer, therefore they are unavailable in the transcript.

You can override this variable by specifying [vsim -displaymsgmode](#).

Related Topics

[Message Viewer Window](#)

DpiCppPath

This variable specifies an explicit location to a gcc compiler for use with automatically generated DPI exportwrappers.

Section [vsim], [vlog]

Syntax

DpiCppPath = <gcc_installation_directory>/bin/gcc

Enusre that the argument points directly to the compiler executable.

DpiOutOfTheBlue

This variable enables DPI out-of-the-blue Verilog function calls. It is also used to enable debugging support for a SystemC thread. The C functions must not be declared as import tasks or functions.

Section [vsim]

Syntax

DpiOutOfTheBlue = {0 | 1 | 2}

- 0 — (default) Support for DPI out-of-the-blue calls is disabled.
- 1 — Support for DPI out-of-the-blue calls is enabled, but debugging support is not available.
- 2 — Support for DPI out-of-the-blue calls is enabled with debugging support for a SystemC thread.

To turn on debugging support in a SystemC method, set DpiOutOfTheBlue = 2 and specify **vsim -scdpidebug**.

You can override this variable using vsim **-dpioutoftheblue**.

Related Topics

[vsim -dpioutoftheblue](#)

[vsim -scdpidebug](#)

[Making Verilog Function Calls from non-DPI C Models](#)

DumpportsCollapse

This variable collapses vectors (VCD id entries) in dumpports output.

Section [vsim]

Syntax

DumpportsCollapse = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vsim {+dumpports+collapse | +dumpports+nocollapse}`.

EmbeddedPsl

This variable enables the parsing of embedded PSL statements in VHDL files.

Sections [vcom], [vlog]

Syntax

EmbeddedPsl = {0 | 1}

0 — Off

1 — On (default)

EnableSVCoverpointExprVariable

This variable, used in conjunction with the [SVCoverpointExprVariablePrefix](#), creates variables containing the effective values of Coverpoint expressions. The current settings for both expression variables is displayed in the Object view.

Note



You must re-compile your design after any change in the setting of either this variable, or the [SVCoverpointExprVariablePrefix](#) variable.

Section [vlog]

Syntax

EnableSVCoverpointExprVariable = {0 | 1}

0 — Off (default)

1 — On

EnableTypeOf

This variable enables support of SystemVerilog 3.1a \$typeof() function. This variable has no impact on SystemVerilog 1364-2005 designs.

Section [vlog]

Syntax

EnableTypeOf = {0 | 1}

0 — Off (default)

1 — On

EnumBaseInit

This variable initializes enum variables in SystemVerilog using either the default value of the base type or the leftmost value.

Section [vsim]

Syntax

EnumBaseInit= {0 | 1}

0 — Initialize to leftmost value

1 — (default) Initialize to default value of base type

error

This variable changes the severity of the listed message numbers to "error".

Section [msg_system]

Syntax

error = <msg_number>...

<msg_number>...— An unlimited list of message numbers, comma separated.

You can override this variable by specifying the [sccom](#), [vcom](#), [vlog](#), [vopt](#), or [vsim](#) command with the **-error** argument.

Related Topics

[verror <msg number>](#) prints a detailed description about a message number.

[Changing Message Severity Level](#)

[fatal](#), [note](#), [suppress](#), [warning](#)

ErrorFile

This variable specifies an alternative file for storing error messages. By default, error messages are output to the file specified by the [TranscriptFile](#) variable in the *modelsim.ini* file. If the ErrorFile variable is specified, all error messages will be stored in the specified file, not in the transcript.

Section [vsim]

Syntax

ErrorFile = <filename>

<filename> — Any valid filename where the default is *error.log*.

You can override this variable by specifying [vsim -errorfile](#).

Related Topics

[Creating a Transcript File](#)

Explicit

This variable enables the resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration). Using this variable makes QuestaSim compatible with common industry practice.

Section [vcom]

Syntax

Explicit = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vcom -explicit](#).

ExtendedToggleMode

This variable specifies one of three modes for extended toggle coverage.

Section [vsim]

Syntax

ExtendedToggleMode = {1 | 2 | 3}

where:

1 — 0L->1H & 1H->0L & any one 'Z' transition (to/from 'Z')

2 — 0L->1H & 1H->0L & one transition to 'Z' & one transition from 'Z'

3 — (default) 0L->1H & 1H->0L & all 'Z' transitions

You can override this variable by specifying **-extendedtogglemode {1|2|3}** to the [vcom/vlog/vopt](#) or [toggle add](#) commands.

Related Topics

[Understanding Toggle Counts](#)

fatal

This variable changes the severity of the listed message numbers to "fatal".

Section [msg_system]

Syntax

fatal = <msg_number>...

<msg_number>...— An unlimited list of message numbers, comma separated.

You can override this variable by specifying the [sccom](#), [vcom](#), [vlog](#), [vopt](#), or [vsim](#) command with the -fatal argument.

Related Topics

[verror](#) <msg number> prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [note](#), [suppress](#), [warning](#)

FecEffort

This variable increases or decreases the maximum number of rows allowed in an FEC table when implementing a condition coverage or expression coverage expression. A higher value will increase time for compile however more expressions are covered.

Section [vcom], [vlog], [vopt]

Syntax

FecEffort = {1 | 2 | 3}

Variables

- 1 — Low FecEffort. Covers only small expressions or conditions and skips larger ones. Covers up to 8 inputs per expression.
- 2 — (default) Medium FecEffort. Covers between 9 and 16 inputs per expression.
- 3 — High FecEffort. covers large expressions and conditions. Covers up to 20 inputs per expression, however some complex expressions may be limited to 12 inputs.

Note

The numbers associated with each effort level are approximate and will vary with the complexity of the expression.

floatfixlib

This variable sets the path to the library containing VHDL floating and fixed point packages.

Section [library]

Syntax

floatfixlib = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./floatfixlib. May include environment variables.

ForceSigNextIter

This variable controls the iteration of events when a VHDL signal is forced to a value.

Section [vsim]

Syntax

ForceSigNextIter = {0 | 1}

0 — Off (default) Update and propagate in the same iteration.

1 — On Update and propagate in the next iteration.

ForceUnsignedIntegerToVHDLInteger

This variable controls whether untyped Verilog parameters in mixed-language designs that are initialized with unsigned values between $2^{31}-1$ and 2^{32} are converted to VHDL generics of type INTEGER or ignored. If mapped to VHDL Integers, Verilog values greater than $2^{31}-1$ (2147483647) are mapped to negative values. Default is to map these parameter to generic of type INTEGER.

Section [vlog]

Syntax

ForceUnsignedIntegerToVHDLInteger = {0 | 1}

- 0 — Off
- 1 — On (default)

FsmImplicitTrans

This variable controls recognition of FSM Implicit Transitions.

Sections [vcom], [vlog]

Syntax

FsmImplicitTrans = {0 | 1}

- 0 — Off (default)
- 1 — On Enables recognition of implied same state transitions.

Related Topics

[vcom -fsmimplicittrans](#) |
[-nofsmimplicittrans](#)
[vlog -fsmimplicittrans](#) |
[-nofsmimplicittrans](#)

FsmResetTrans

This variable controls the recognition of asynchronous reset transitions in FSMs.

Sections [vcom], [vlog]

Syntax

FsmResetTrans = {0 | 1}

- 0 — Off
- 1 — On (default)

Related Topics

[vcom -fsmresettrans](#) | [-nofsmresettrans](#)
[vlog -fsmresettrans](#) | [-nofsmresettrans](#)

FsmSingle

This variable controls the recognition of FSMs with a single-bit current state variable.

Section [vcom], [vlog]

Syntax

FsmSingle = { 0 | 1 }

0 — Off

1 — On (default)

Related Topics

[vcom -fmsingle](#) | [-nofmsingle](#)

[vlog -fmsingle](#) | [-nofmsingle](#)

FsmXAssign

This variable controls the recognition of FSMs where a current-state or next-state variable has been assigned “X” in a case statement.

Section [vlog]

Syntax

FsmXAssign = { 0 | 1 }

0 — Off

1 — On (default)

Related Topics

[vlog -fsmxassign](#) | [-nofsmxassign](#)

GenerateFormat

This variable controls the format of the old-style VHDL for ... generate statement region name for each iteration.

Section [vsim]

Syntax

GenerateFormat = <non-quoted string>

<non-quoted string> — The default is %s__%d. The format of the argument must be unquoted, and must contain the conversion codes %s and %d, in that order. This string should not contain any uppercase or backslash (\) characters.

The %s represents the generate statement label and the %d represents the generate parameter value at a particular iteration (this is the position number if the generate parameter is of an enumeration type). Embedded whitespace is allowed (but discouraged) while leading and trailing whitespace is ignored. Application of the format must result in a unique region name over all loop iterations for a particular immediately enclosing scope so that name lookup can function properly.

Related Topics

[OldVhdlForGenNames](#) modelsim.ini variable [Naming Behavior of VHDL For Generate Blocks](#)

GenerateLoopIterationMax

This variable specifies the maximum number of iterations permitted for a generate loop; restricting this permits the implementation to recognize infinite generate loops.

Section [vopt]

Syntax

GenerateLoopIterationMax = <n>

<n> — Any natural integer greater than or equal to 0, where the default is 100000.

GenerateRecursionDepthMax

This variable specifies the maximum depth permitted for a recursive generate instantiation; restricting this permits the implementation to recognize infinite recursions.

Section [vopt]

Syntax

GenerateRecursionDepthMax = <n>

<n> — Any natural integer greater than or equal to 0, where the default is 200.

GenerousIdentifierParsing

Controls parsing of identifiers input to the simulator. If this variable is on (value = 1), either VHDL extended identifiers or Verilog escaped identifier syntax may be used for objects of either language kind. This provides backward compatibility with older *.do* files, which often contain pure VHDL extended identifier syntax, even for escaped identifiers in Verilog design regions.

Section [vsim]

Syntax

GenerousIdentifierParsing = {0 | 1}

0 — Off

1 — On (default)

GlobalSharedObjectsList

This variable instructs ModelSim to load the specified PLI/FLI shared objects with global symbol visibility.

Section [vsim]

Syntax

GlobalSharedObjectsList = <filename>

<filename> — A comma separated list of filenames.

semicolon (;) — (default) Prevents initiation of the variable by commenting the variable line.

You can override this variable by specifying **vsim -gblso**.

Hazard

This variable turns on Verilog hazard checking (order-dependent accessing of global variables).

Section [vlog]

Syntax

Hazard = {0 | 1}

0 — Off (default)

1 — On

ieee

This variable sets the path to the library containing IEEE and Synopsys arithmetic packages.

Section [library]

Syntax

ieee = <path>

< path > — Any valid path, including environment variables where the default is \$MODEL_Tech/./ieee.

IgnoreError

This variable instructs ModelSim to disable runtime error messages.

Section [vsim]

Syntax

IgnoreError = {0 | 1}

0 — Off (default)

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the Tcl [set Command Syntax](#).

IgnoreFailure

This variable instructs ModelSim to disable runtime failure messages.

Section [vsim]

Syntax

IgnoreFailure = {0 | 1}

0 — Off (default)

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the Tcl [set Command Syntax](#).

IgnoreNote

This variable instructs ModelSim to disable runtime note messages.

Section [vsim]

Syntax

IgnoreNote = {0 | 1}

0 — Off (default)

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the Tcl [set Command Syntax](#).

IgnorePragmaPrefix

This variable instructs the compiler to ignore synthesis and coverage pragmas with the specified prefix name. The affected pragmas will be treated as regular comments.

Section [vcom, vlog]

Syntax

IgnorePragmaPrefix — <prefix> | ""

<prefix> — Specifies a user defined string.

"" — (default) No string.

You can override this variable by specifying **vcom -ignorepragmaprefix** or **vlog -ignorepragmaprefix**.

ignoreStandardRealVector

This variable instructs ModelSim to ignore the REAL_VECTOR declaration in package STANDARD when compiling with **vcom -2008**. For more information refer to the REAL_VECTOR section in **Help > Technotes > vhdl2008migration** technote.

Section [vcom]

Syntax

IgnoreStandardRealVector = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying **vcom -ignoreStandardRealVector**.

IgnoreSVAError

This variable instructs ModelSim to disable SystemVerilog assertion messages for Error severity.

Section [vsim]

Syntax

IgnoreSVAError = {0 | 1}

0 — Off (default)

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

IgnoreSVAFatal

This variable instructs ModelSim to disable SystemVerilog assertion messages for Fatal severity.

Section [vsim]

Syntax

IgnoreSVAFatal = 0 | 1 }

0 — Off. (default) SystemVerilog assertion messages enabled.

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

IgnoreSVAInfo

This variable instructs ModelSim to disable SystemVerilog assertion messages for Info severity.

Section [vsim]

Syntax

IgnoreSVAInfo = {0 | 1 }

0 — Off, (default) Info severity messages enabled.

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

IgnoreSVAWarning

This variable instructs ModelSim to disable SystemVerilog assertion messages for Warning severity.

Section [vsim]

Syntax

IgnoreSVWarning = {0 | 1}

0 — Off, (default) SystemVerilog assertion messages for warning severity enabled.

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

IgnoreVitalErrors

This variable instructs ModelSim to ignore VITAL compliance checking errors.

Section [vcom]

Syntax

IgnoreVitalErrors = {0 | 1}

0 — Off, (default) Allow VITAL compliance checking errors.

1 — On

You can override this variable by specifying [vcom -ignorevitalerrors](#).

IgnoreWarning

This variable instructs ModelSim to disable runtime warning messages.

Section [vsim]

Syntax

IgnoreWarning = {0 | 1}

0 — Off, (*default*) Enable runtime warning messages.

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

ImmediateContinuousAssign

This variable instructs ModelSim to run continuous assignments before other normal priority processes that are scheduled in the same iteration. This event ordering minimizes race differences between optimized and non-optimized designs and is the default behavior.

Section [vsim]

Syntax

ImmediateContinuousAssign = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vsim -noimmedca](#).

IncludeRecursionDepthMax

This variable limits the number of times an include file can be called during compilation. This prevents cases where an include file could be called repeatedly.

Section [vlog]

Syntax

IncludeRecursionDepthMax = <n>

<n> — an integer that limits the number of loops. A setting of 0 would allow one pass through before issuing an error, 1 would allow two passes, and so on.

InitOutCompositeParam

This variable controls how subprogram output parameters of array and record types are treated.

Section [vcom]

Syntax

InitOutCompositeParam = {0 | 1 | 2}

0 — Use the default for the language version being compiled.

1 — (default) Always initialize the output parameter to its default or “left” value immediately upon entry into the subprogram.

2 — Do not initialize the output parameter.

Variables

You can override this variable by specifying `vcom -initoutcompositeparam` or `vopt -initoutcompositeparam`.

IterationLimit

This variable specifies a limit on simulation kernel iterations allowed without advancing time.

Section [vlog], [vsim]

Syntax

IterationLimit= <n>

n — Any positive integer where the default is 5000.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

LargeObjectSilent

This variable controls whether “large object” warning messages are issued or not. Warning messages are issued when the limit specified in the variable `LargeObjectSize` is reached.

Section [vsim]

Syntax

LargeObjectSilent = {0 | 1}

0 — On (default).

1 — Off.

LargeObjectSize

This variable specifies the relative size of log, wave, or list objects in bytes that will trigger “large object” messages. This size value is an approximation of the number of bytes needed to store the value of the object before compression and optimization.

Section [vsim]

Syntax

LargeObjectSize= < n >

n — Any positive integer where the default is 500000 bytes.

LibrarySearchPath

This variable specifies the location of one or more resource libraries containing a precompiled package. The behavior of this variable is identical to specifying `vlog -L <libname>`.

Section [vlog]

Syntax

LibrarySearchPath= <variable | <path/lib>...>

variable — Any library variable where the default is:

```
LibrarySearchPath = mtiAvm mtiOvm mtiUvm mtiUPF
```

path/lib — Any valid library path. May include environment variables.

Multiple library paths and variables are specified as a space separated list.

Related Topics

[Specifying Resource Libraries.](#)

License

This variable controls the license file search.

Section [vsim]

Syntax

License = <license_option>

<license_option> — One or more license options separated by spaces where the default is to search all licenses.

Table A-5. License Variable: License Options

license_option	Description
lnonly	only use msimhdlsim
mixedonly	exclude single language licenses
nolnl	exclude language neutral licenses
nomix	exclude msimhdlmix
noqueue	do not wait in license queue if no licenses are available
noslvhdl	exclude qhsimvh
noslvlog	exclude qhsimvl
plus	only use PLUS license
vlog	only use VLOG license

Table A-5. License Variable: License Options

license_option	Description
vhdl	only use VHDL license

You can override this variable by specifying `vsim <license_option>`.

MaxReportRhsCrossProducts

This variable specifies a maximum limit for the number of Cross (bin) products reported against a Cross when a XML or UCDB report is generated. The warning is issued if the limit is crossed.

Section [vsim]

Syntax

MaxReportRhsCrossProducts = <n>

<n> — Any positive integer where the default is 1000.

MaxReportRhsSVCrossProducts

This variable limits the number of "bin_rhs" values associated with cross bins in the XML version of the coverage report for a SystemVerilog design. It also limits the values saved to a UCDB.

Section [vsim]

Syntax

MaxReportRhsSVCrossProducts = <n>

<n> — Any positive integer where the default is 1000.

MaxSVCoverpointBinsDesign

This variable limits the maximum number of Coverpoint bins allowed in the whole design.

Section [vsim]

Syntax

MaxSVCoverpointBinsDesign = <n>

<n> — Any positive integer where the default is 2147483648.

MaxSVCoverpointBinsInst

This variable limits the maximum number of Coverpoint bins allowed in any instance of a Covergroup.

Section [vsim]

Syntax

MaxSVCoverpointBinsInst = <n>

<n> — Any positive integer where the default is 2147483648.

MaxSVCrossBinsDesign

This variable issues a warning when the number of Coverpoint bins in the design exceeds the value specified by <n>.

Section [vsim]

Syntax

MaxSVCrossBinsDesign = <n>

<n> — Any positive integer where the default is 2147483648.

MaxSVCrossBinsInst

This variable issues a warning when the number of Coverpoint bins in any instance of a Covergroup exceeds the value specified by <n>.

Section [vsim]

Syntax

MaxSVCrossBinsInst = <n>

<n> — Any positive integer where the default is 2147483648.

MessageFormat

This variable defines the format of VHDL/PSL/SVA assertion messages as well as normal error messages.

Section [vsim]

Syntax

MessageFormat = <% value>

<% value> — One or more of the variables from [Table A-6](#) where the default is:

** %S: %R\n Time: %T Iteration: %D%I\n.

Table A-6. MessageFormat Variable: Accepted Values

Variable	Description
%S	severity level
%R	report message

Table A-6. MessageFormat Variable: Accepted Values

Variable	Description
%T	time of assertion
%D	delta
%I	instance or region pathname (if available)
%i	instance pathname with process
%O	process name
%K	kind of object path points to; returns Instance, Signal, Process, or Unknown
%P	instance or region path without leaf process
%F	file
%L	line number of assertion, or if from subprogram, line from which call is made
%u	Design unit name in form: library.primary. Returns <protected> if the design unit is protected.
%U	Design unit name in form: library.primary(secondary). Returns <protected> if the design unit is protected.
%%	print '%' character

MessageFormatBreak

This variable defines the format of messages for VHDL/PSL/SVA assertions that trigger a breakpoint.

Section [vsim]

Syntax

MessageFormatBreak = <% value>

<% value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n
```

MessageFormatBreakLine

This variable defines the format of messages for VHDL/PSL/SVA assertions that trigger a breakpoint. %L specifies the line number of the assertion or, if the breakpoint is from a subprogram, the line from which the call is made.

Section [vsim]

Syntax

MessageFormatBreakLine = <% value>

<% value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T Iteration: %D %K: %i File: %F Line: %L\n
```

MessageFormatError

This variable defines the format of all error messages.

If undefined, MessageFormat is used unless the error causes a breakpoint in which case [MessageFormatBreak](#) is used.

Section [vsim]

Syntax

MessageFormatError = <% value>

<% value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n
```

MessageFormatFail

This variable defines the format of messages for VHDL/PSL/SVA Fail assertions.

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case [MessageFormatBreak](#) is used.

Section [vsim]

Syntax

MessageFormatFail = <% value>

<% value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n
```

MessageFormatFatal

This variable defines the format of messages for VHDL/PSL/SVA Fatal assertions.

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case [MessageFormatBreak](#) is used.

Section [vsim]

Syntax

MessageFormatFatal = <% value>

<% value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n   Time: %T   Iteration: %D   %K: %i File: %F\n
```

MessageFormatNote

This variable defines the format of messages for VHDL/PSL/SVA Note assertions.

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case [MessageFormatBreak](#) is used.

Section [vsim]

Syntax

MessageFormatNote = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T   Iteration: %D%i\n
```

MessageFormatWarning

This variable defines the format of messages for VHDL/PSL/SVA Warning assertions.

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case [MessageFormatBreak](#) is used.

Section [vsim]

Syntax

MessageFormatWarning = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T   Iteration: %D%i\n
```

MixedAnsiPorts

This variable permits partial port re-declarations for cases where the port is partially declared in ANSI style and partially non-ANSI.

Section [vlog]

Syntax

MixedAnsiPorts = {0 | 1}

0 — Off, (*default*)

1 — On

You can override this variable by specifying `vlog -mixedansiports`.

modelsim_lib

This variable sets the path to the library containing Mentor Graphics VHDL utilities such as Signal Spy.

Section [library]

Syntax

modelsim_lib = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./modelsim_lib. May include environment variables.

msgmode

This variable controls where the simulator outputs elaboration and runtime messages.

Section [msg_system]

Syntax

msgmode = {tran | wlf | both}

tran — (default) Messages appear only in the transcript.

wlf — Messages are sent to the wlf file and can be viewed in the MsgViewer.

both — Transcript and wlf files.

You can override this variable by specifying **vsim -msgmode**.

Related Topics

[Message Viewer Window](#)

mtiAvm

This variable sets the path to the location of the Advanced Verification Methodology libraries.

Section [library]

Syntax

mtiAvm = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./avm

The behavior of this variable is identical to specifying **vlog -L mtiAvm**.

mtiOvm

This variable sets the path to the location of the Open Verification Methodology libraries.

Section [library]

Syntax

mtiOvm = <path>

<path> — \$MODEL_TECH/./ovm-2.1.2

The behavior of this variable is identical to specifying **vlog -L mtiOvm**.

mtiPA

This variable sets the path to the location of Power Aware libraries.

Section [library]

Syntax

mtiPA = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./pa_lib. May include environment variables.

The behavior of this variable is identical to specifying **vlog -L mtiPA**.

mtiUPF

This variable sets the path to the location of Unified Power Format (UPF) libraries.

Section [library]

Syntax

mtiUPF = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./upf_lib. May include environment variables.

The behavior of this variable is identical to specifying **vlog -L mtiUPF**.

Related Topics

[Specifying Resource Libraries](#)

[Library Usage](#)

mtiUvm

This variable sets the path to the location of the Open Verification Methodology libraries.

Section [library]

Syntax

mtiUvm = <path>

<path> — \$MODEL_TECH/./uvm-1.1

The behavior of this variable is identical to specifying **vlog -L mtiUvm**.

MultiFileCompilationUnit

This variable controls whether Verilog files are compiled separately or concatenated into a single compilation unit.

Section [vlog]

Syntax

MultiFileCompilationUnit = {0 | 1}

0 — (default) Single File Compilation Unit (SFCU) mode.

1 — Multi File Compilation Unit (MFCU) mode.

You can override this variable by specifying `vlog {-mfcu | -sfcu}`.

Related Topics

[SystemVerilog Multi-File Compilation](#)

MvcHome

This variable specifies the location of the installation of Questa Verification IPs (which previously were known as Multi-View Verification Components (MVC)).

Section [vsim]

Syntax

MvcHome = <path>

<path> — Any valid path. May include environment variables.

You can override this variable by specifying `vsim -mvchome`.

NoCaseStaticError

This variable changes case statement static errors to warnings.

Section [vcom]

Syntax

NoCaseStaticError = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vcom -nocasestaticerror`.

Related Topics

vcom **-pedanticerrors**

[PedanticErrors](#)

NoDebug

This variable controls inclusion of debugging info within design units.

Sections [vcom], [vlog]

Syntax

NoDebug = {0 | 1}

0 — Off (default)

1 — On

NoDeferSubpgmCheck

This variable controls the reporting of range and length violations detected within subprograms as errors (instead of as warnings).

Section [vcom]

Syntax

NoDeferSubpgmCheck = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vcom -deferSubpgmCheck](#).

NoIndexCheck

This variable controls run time index checks.

Section [vcom]

Syntax

NoIndexCheck = {0 | 1}

0 — Off (default)

1 — On

You can override NoIndexCheck = 0 by specifying [vcom -noindexcheck](#).

Related Topics

[Range and Index Checking](#)

NoOthersStaticError

This variable disables errors caused by aggregates that are not locally static.

Section [vcom]

Syntax

NoOthersStaticError = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying **vcom -nootersstaticerror**.

Related Topics

[Changing Message Severity Level](#)

[PedanticErrors](#)

NoRangeCheck

This variable disables run time range checking. In some designs this results in a 2x speed increase.

Section [vcom]

Syntax

NoRangeCheck = {0 | 1}

0 — Off (default)

1 — On

You can override this NoRangeCheck = 1 by specifying **vcom -rangecheck**.

Related Topics

[Range and Index Checking](#)

note

This variable changes the severity of the listed message numbers to "note".

Section [msg_system]

Syntax

note = <msg_number>...

Variables

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable setting by specifying the [scom](#), [vcom](#), [vlog](#), [vopt](#), or [vsim](#) command with the **-note** argument.

Related Topics

[verror](#) <msg number> prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [fatal](#), [suppress](#), [warning](#)

NoVital

This variable disables acceleration of the VITAL packages.

Section [vcom]

Syntax

NoVital = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vcom](#) **-novital**.

NoVitalCheck

This variable disables VITAL level 0 and Vital level 1 compliance checking.

Section [vcom]

Syntax

NoVitalCheck = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vcom](#) **-novitalcheck**.

Related Topics

Section 4 of the IEEE Std 1076.4-2004

NumericStdNoWarnings

This variable disables warnings generated within the accelerated `numeric_std` and `numeric_bit` packages.

Section [vsim]

Syntax

NumericStdNoWarnings = {0 | 1}

0 — Off (default)

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

OldVHDLConfigurationVisibility

Controls visibility of VHDL component configurations during compile.

Sections [vcom]

Syntax

OldVHDLConfigurationVisibility = {0 | 1}

0 - Use Language Reference Manual compliant visibility rules when processing VHDL configurations.

1 - (default) Force vcom to process visibility of VHDL component configurations consistent with prior releases.

Related Topics

[vcom -oldconfigvis](#)

[vcom -lrmVHDLConfigVis](#)

OldVhdlForGenNames

This variable instructs the simulator to use a previous style of naming (pre-6.6) for VHDL for ... generate statement iteration names in the design hierarchy. The previous style is controlled by the value of the [GenerateFormat](#) value.

The default behavior is to use the current style names, which is described in the section [“Naming Behavior of VHDL For Generate Blocks”](#).

Section [vsim]

Syntax

OldVhdlForGenNames = {0 | 1}

0 — Off (default)

1 — On

Related Topics

[GenerateFormat](#) modelsim.ini variable

[Naming Behavior of VHDL For Generate Blocks](#)

OnFinish

This variable controls the behavior of ModelSim when it encounters either an assertion failure, a \$finish, or an sc_stop() in the design code.

Section [vsim]

Syntax

OnFinish = {[ask](#) | exit | final | stop}

[ask](#) — (default) In batch mode, the simulation exits. In GUI mode, a dialog box pops up and asks for user confirmation on whether to quit the simulation.

[stop](#) — Causes the simulation to stay loaded in memory. This can make some post-simulation tasks easier.

[exit](#) — The simulation exits without asking for any confirmation.

[final](#) — The simulation executes all final blocks then exits the simulation.

You can override this variable by specifying [vsim -onfinish](#).

OnFinishPendingAssert

This variable prints pending deferred assertion messages. Deferred assertion messages may be scheduled after the \$finish in the same time step. Deferred assertions scheduled to print after the \$finish are printed to the Transcript before exiting. They are printed with severity level NOTE because it is not known whether the assertion is still valid due to being printed in the active region instead of the reactive region where they are normally printed.

Section [vsim]

Syntax

OnFinishPendingAssert = {0 | 1}

0 — Off (default)

1 — On

Optimize_1164

This variable disables optimization for the IEEE std_logic_1164 package.

Section [vcom]

Syntax

Optimize_1164 = {0 | 1}

0 — Off

1 — On (default)

ParallelJobs

This variable may be set to zero (0) to disable parallel processing during vopt code generation phase. Normally a heuristic is used to set this value.

Section [vopt]

Syntax

ParallelJobs = <n>

0 — Off

<n> — Any natural integer greater than or equal to 0.

PathSeparator

This variable specifies the character used for hierarchical boundaries of HDL modules. This variable does not affect file system paths. The argument to PathSeparator must not be the same character as [DatasetSeparator](#). This variable setting is also the default for the [SignalSpyPathSeparator](#) variable.

This variable is used by the [vsim](#) and [vopt](#) commands.

Note



When creating a virtual bus, the PathSeparator variable must be set to either a period (.) or a forward slash (/). For more information on creating virtual buses, refer to the section “[Combining Objects into Buses](#)”.

Section [vsim]

Syntax

PathSeparator = <n>

<n> — Any character except special characters, such as backslash (\), brackets ({}), and so forth, where the default is a forward slash (/).

Related Topics

[Using Escaped Identifiers](#)

[Preserving Design Visibility with the Learn Flow](#)

You can set this variable interactively with the Tcl [set Command Syntax](#).

PedanticErrors

This variable forces display of an error message (rather than a warning) on a variety of conditions. It overrides the [NoCaseStaticError](#) and [NoOthersStaticError](#) variables.

Section [vcom]

Syntax

PedanticErrors = {0 | 1}

0 — Off (default)

1 — On

Related Topics

[vcom -nocasestaticerror](#)

[vcom -noothersstaticerror](#)

[Enforcing Strict 1076 Compliance](#)

PliCompatDefault

This variable specifies the VPI object model behavior within vsim.

Section [vsim]

Syntax

PliCompatDefault = {1995 | 2001 | 2005 | 2009 | latest}

1995 — Instructs vsim to use the object models as defined in IEEE Std 1364-1995.

When you specify this argument, SystemVerilog objects will not be accessible.

Aliases include:

95

1364v1995

1364V1995

VL1995

VPI_COMPATIBILITY_VERSION_1364v1995

1 — On

2001 — Instructs vsim to use the object models as defined in IEEE Std 1364-2001.

When you specify this argument, SystemVerilog objects will not be accessible.

Aliases include:

01
1364v2001
1364V2001
VL2001
VPI_COMPATIBILITY_VERSION_1364v2001

Note

There are a few cases where the 2005 VPI object model is incompatible with the 2001 model, which is inherent in the specifications.

2005 — Instructs vsim to use the object models as defined in IEEE Std 1800-2005 and IEEE Std 1364-2005. Aliases include:

05
1800v2005
1800V2005
SV2005
VPI_COMPATIBILITY_VERSION_1800v2005

2009 — Instructs vsim to use the object models as defined in IEEE Std 1800-2009. Aliases include:

09
1800v2009
1800V2009
SV2009
VPI_COMPATIBILITY_VERSION_1800v2009

latest — (default) This is equivalent to the "**2009**" argument. This is the default behavior if you do not specify this argument or if you specify the argument without an argument.

You can override this variable by specifying `vsim -plicompatdefault`.

Related Topics

[Verilog Interfaces to C](#)

PreserveCase

This variable instructs the VHDL compiler either to preserve the case of letters in basic VHDL identifiers or to convert uppercase letters to lowercase.

Section [vcom]

Syntax

PreserveCase = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vcom -lower` or `vcom -preserve`.

PrintSimStats

This variable instructs the simulator to print out simulation statistics at the end of the simulation before it exits.

Section [vsim]

Syntax

PrintSimStats = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vsim -printsimstats`.

Related Topics

[simstats](#)

PrintSVPackageLoadingAttribute

This variable prints the attribute placed upon SV packages during package import when true (1). The attribute will be ignored when this variable entry is false (0). The attribute name is "package_load_message." The value of this attribute is a string literal.

Section [vlog]

Syntax

PrintSVPackageLoadingAttribute = {0 | 1}

0 — False

1 — True (default)

Protect

This variable enables protect directive processing.

Section [vlog]

Syntax

Protect = {0 | 1}

0 — Off (default)

1 — On

Related Topics

[Compiler Directives](#)

PslOneAttempt

This variable affects PSL directives with top level "always/never" properties. As per strict IEEE Std 1850-2005, an always/never property can either pass or fail. However, by default, ModelSim reports multiple passes and/or failures, which corresponds to multiple attempts made while executing a top level "always/never" property. With this variable, you can force a single attempt to start at the beginning of simulation. The directive will either match (pass), fail, or vacuously-match (provided it is not disabled/aborted). If the "always/never" property fails, the directive is immediately considered a failure and the simulation will not go further. If there is no failure (or disable/abort) until end of simulation then a match (pass) is reported. By default, this feature is off and can only be explicitly turned on using this variable or **`vsim -psloneattempt`**.

Sections [vsim]

Syntax

PslOneAttempt = {0 | 1}

0 — Off (default)

1 — On

PslInfinityThreshold

This variable allows you to specify the number of clock ticks that will represent infinite clock ticks. It only affects PSL strong operators, namely eventually!, until! and until_!. If at End of Simulation an active strong-property has not clocked this number of clock ticks, neither pass nor fail (that is, vacuous match) is returned; else, respective fail/pass is returned. The default value is '0' (zero) which effectively does not check for clock tick condition. This feature can only be explicitly turned on using this variable or **`vsim -pslinfinitethreshold`**.

Sections [vsim]

Syntax

PslOneAttempt = [0 | <n>]

0 — Off (default)

<n> — Any positive integer

Quiet

This variable turns off "loading..." messages.

Sections [vcom], [vlog]

Syntax

Quiet = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying **vlog -quiet** or **vcom -quiet**.

RequireConfigForAllDefaultBinding

This variable instructs the compiler not to generate a default binding during compilation.

Section [vcom]

Syntax

RequireConfigForAllDefaultBinding = {0 | 1}

0 — Off (default)

1 — On

You can override `RequireConfigForAllDefaultBinding = 1` by specifying **vcom -performdefaultbinding**.

Related Topics

[Default Binding](#)

[BindAtCompile](#)

vcom -ignoredefaultbinding

Resolution

This variable specifies the simulator resolution. The argument must be less than or equal to the [UserTimeUnit](#) and must not contain a space between value and units.

Section [vsim]

Syntax

Resolution = {[n]<time_unit>}

[n] — Optional prefix specifying number of time units as 1, 10, or 100.

<time_unit> — fs, ps, ns, us, ms, or sec where the default is ns.

The argument must be less than or equal to the [UserTimeUnit](#) and must not contain a space between value and units, for example:

```
Resolution = 10fs
```

You can override this variable by specifying **vsim -t**. You should set a smaller resolution if your delays get truncated.

Related Topics

[Time](#) command

RunLength

This variable specifies the default simulation length in units specified by the [UserTimeUnit](#) variable.

Section [vsim]

Syntax

RunLength = <n>

<n> — Any positive integer where the default is 100.

You can override this variable by specifying the [run](#) command.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

ScalarOpts

This variable activates optimizations on expressions that do not involve signals, waits, or function/procedure/task invocations.

Sections [vcom], [vlog]

Syntax

ScalarOpts = {0 | 1}

0 — Off (default)

1 — On

SccomLogfile

This variable creates a log file for sccom.

Section [sccom]

Syntax

SccomLogfile = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [sccom -log](#).

SccomVerbose

This variable prints the name of each sc_module encountered during compilation.

Section [sccom]

Syntax

SccomVerbose = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `sccom -verbose`.

ScEnableScSignalWriteCheck

This variable enables a check for multiple writers on a SystemC signal.

Section [vsim]

Syntax

ScEnableScSignalWriteCheck = {0 | 1}

0 — Off (default)

1 — On

ScMainFinishOnQuit

This variable determines when the sc_main thread exits. This variable is used to turn off the execution of remainder of sc_main upon quitting the current simulation session. Disabling this variable (0) has the following effect: If the cumulative length of sc_main() in simulation time units is less than the length of the current simulation run upon quit or restart, sc_main() is aborted in the middle of execution. This can cause the simulator to crash if the code in sc_main is dependent on a particular simulation state.

On the other hand, one drawback of not running sc_main till the end is potential memory leaks for objects created by sc_main. By default, the remainder of sc_main is executed regardless of delays.

Section [vsim]

Syntax

ScMainFinishOnQuit = {0 | 1}

0 — Off

1 — On (default)

ScMainStackSize

This variable sets the stack size for the sc_main() thread process.

Section [vsim]

Syntax

ScMainStackSize = <n>{Kb | Mb | Gb}

<n> — An integer followed by Kb, Mb, Gb where the default is 10Mb.

ScShowIeeeDeprecationWarnings

This variable displays warning messages for many of the deprecated features in Annex C of the IEEE Std 1666-2005, *IEEE Standard SystemC Language Reference Manual*.

Section [vsim]

Syntax

ScShowIeeeDeprecationWarnings = {0 | 1}

0 — Off (default)

1 — On

ScTimeUnit

This variable sets the default time unit for SystemC simulations.

Section [vsim]

Syntax

ScTimeUnit = {[n]<time_unit>}

[n] — Optional prefix specifying number of time units as 1, 10, or 100.

<time_unit> — fs, ps, ns, us, ms, or sec where the default is 1 ns.

ScvPhaseRelationName

This variable changes the precise name used by SCV to specify “phase” transactions in the WLF file.

Section [vsim]

Syntax

ScvPhaseRelationName = <string>

<string> — Any legal string where the default is mti_phase. Legal C-language identifiers are recommended.

Related Topics

[Parallel Transactions](#)

[Recording Phase Transactions](#)

SeparateConfigLibrary

This variable allows the declaration of a VHDL configuration to occur in a different library than the entity being configured. Strict conformance to the VHDL standard (LRM) requires that they be in the same library.

Section [vcom]

Syntax

SeparateConfigLibrary = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vcom -separateConfigLibrary`.

Show_BadOptionWarning

This variable instructs ModelSim to generate a warning whenever an unknown plus argument is encountered.

Section [vlog]

Syntax

Show_BadOptionWarning = {0 | 1}

0 — Off (default)

1 — On

Show_Lint

This variable instructs ModelSim to display lint warning messages.

Sections [vcom], [vlog]

Syntax

Show_Lint = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vlog -lint` or `vcom -lint`.

Show_PslChecksWarnings

This variable instructs ModelSim to display PSL warning messages.

Section [vcom], [vlog]

Syntax

Show_PslChecksWarnings = {0 | 1}

0 — Off

1 — On (default)

Show_source

This variable shows source line containing error.

Sections [vcom], [vlog]

Syntax

Show_source = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying the [vlog -source](#) or [vcom -source](#).

Show_VitalChecksWarnings

This variable enables VITAL compliance-check warnings.

Section [vcom]

Syntax

Show_VitalChecksWarnings = {0 | 1}

0 — Off

1 — On (default)

Show_Warning1

This variable enables unbound-component warnings.

Section [vcom]

Syntax

Show_Warning1 = {0 | 1}

0 — Off

1 — On (default)

Show_Warning2

This variable enables process-without-a-wait-statement warnings.

Section [vcom]

Syntax

Show_Warning2 = {0 | 1}

0 — Off

1 — On (default)

Show_Warning3

This variable enables null-range warnings.

Section [vcom]

Syntax

Show_Warning3 = {0 | 1}

0 — Off

1 — On (default)

Show_Warning4

This variable enables no-space-in-time-literal warnings.

Section [vcom]

Syntax

Show_Warning4 = {0 | 1}

0 — Off

1 — On (default)

Show_Warning5

This variable enables multiple-drivers-on-unresolved-signal warnings.

Section [vcom]

Syntax

Show_Warning5 = {0 | 1}

- 0 — Off
- 1 — On (default)

ShowConstantImmediateAsserts

This variable controls the display of immediate assertions with constant expressions. By default, immediate assertions with constant expressions are displayed in the GUI, in reports, and in the UCDB.

Section [vcom], [vlog], [vopt]

Syntax

ShowConstantImmediateAsserts = {0 | 1}

- 0 — Off
- 1 — On (default)

ShowFunctions

This variable sets the format for Breakpoint and Fatal error messages. When set to 1 (the default value), messages will display the name of the function, task, subprogram, module, or architecture where the condition occurred, in addition to the file and line number. Set to 0 to revert messages to the previous format.

Section [vsim]

Syntax

ShowFunctions = {0 | 1}

- 0 — Off
- 1 — On (default)

ShowUnassociatedScNameWarning

This variable instructs ModelSim to display unassociated SystemC name warnings.

Section [vsim]

Syntax

ShowUnassociatedScNameWarning = {0 | 1}

- 0 — Off (default)
- 1 — On

ShowUndebuggableScTypeWarning

This variable instructs ModelSim to display undebuggable SystemC type warnings.

Section [vsim]

Syntax

ShowUndebuggableScTypeWarning = {0 | 1}

0 — Off

1 — On (default)

ShutdownFile

This variable calls the [write format restart](#) command upon exit and executes the *.do* file created by that command. This variable should be set to the name of the file to be written, or the value "--disable-auto-save" to disable this feature. If the filename contains the pound sign character (#), then the filename will be sequenced with a number replacing the #. For example, if the file is "restart#.do", then the first time it will create the file "restart1.do" and the second time it will create "restart2.do", and so forth.

Section [vsim]

Syntax

ShutdownFile = <filename>.do | <filename>#.do | --disable-auto-save }

<filename>.do — A user defined filename where the default is *restart.do*.

<filename>#.do — A user defined filename with a sequencing character.

--disable-auto-save — Disables auto save.

SignalSpyPathSeparator

This variable specifies a unique path separator for the Signal Spy functions. The argument to SignalSpyPathSeparator must not be the same character as the [DatasetSeparator](#) variable.

Section [vsim]

Syntax

SignalSpyPathSeparator = <character>

<character> — Any character except special characters, such as backslash (\), brackets ({ }), and so forth, where the default is to use the [PathSeparator](#) variable or a forward slash (/).

Related Topics

[Signal Spy](#)

SimulateAssumeDirectives

This variable instructs ModelSim to assume directives are simulated as if they were assert directives.

Section [vsim]

Syntax

SimulateAssumeDirectives = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vsim {-assume | -noassume}`.

Related Topics

[Processing Assume Directives](#)

SimulateImmedAsserts

This variable controls whether or not SVA and VHDL immediate assertion directives will be simulated.

Section [vsim]

Syntax

SimulateImmedAsserts = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vsim {-immedassert | -noimmedassert}`.

SimulatePSL

This variable controls whether or not PSL assertion directives will be elaborated.

Section [vsim]

Syntax

SimulatePSL = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vsim {-psl | -nopsl}`.

SimulateSVA

This variable controls whether or not SVA concurrent assertion directives will be elaborated.

Section [vsim]

Syntax

SimulateSVA = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vsim {-sva | -nosva}`.

SolveACTbeforeSpeculate

This variable specifies the use of the ACT solver next in randomization scenarios where the non-speculative BDD solver is used first (i.e. `SolveEngine = auto` or `bdd`, and `SolveSpeculateFirst = 0`) and fails due to the `SolveGraphMaxSize` or `SolveGraphMaxEval` limits.

Section [vsim]

Syntax

SolveACTbeforeSpeculate = 0 | 1

0 — (default) Do not use the ACT solver immediately following a failed attempt to solve the randomization scenario with the BDD.

1 — Use the ACT solver immediately following a failed attempt to solve the randomization scenario with the BDD.

SolveACTMaxOps

This variable specifies the maximum number of operations that the ACT solver may perform before abandoning an attempt to solve a particular constraint scenario. The value is specified in 1,000,000s of operations.

Section [vsim]

Syntax

SolveACTMaxOps = <n>

n — Any positive integer where the default is 10000 and 0 indicates no limit.

SolveACTMaxTests

This variable specifies the maximum number of tests that the ACT solver may evaluate before abandoning an attempt to solve a particular randomize scenario.

Section [vsim]

Syntax

SolveACTMaxTests = <n>

<n> — Any positive integer where 0 indicates no limit and the default is 2000000.

SolveACTRetryCount

This variable specifies the number of times to retry the ACT solver on a randomization scenario that fails due to the value of the SolveACTMaxTests threshold. The default is 0, meaning that if the first attempt fails after SolveACTMaxTests tests, no subsequent attempts are made, and the solver moves on to the next engine (e.g. the BDD engine). This can be useful in scenarios where the BDD engine is known to fail, and the ACT solver succeeds most of the time. A small nonzero value of SolveACTRetryCount can decrease the percentage of the time that a randomize call might not ultimately succeed.

Section [vsim]

Syntax

SolveACTRetryCount = <n>

<n> — Any positive integer where the default is 0.

SolveArrayResizeMax

This variable specifies the maximum size randomize() will allow a dynamic array to be resized. If randomize() attempts to resize a dynamic array to a value greater than SolveArrayResizeMax, an error will be displayed and randomize() will fail.

Section [vsim]

Syntax

SolveArrayResizeMax = <n>

<n> — Any positive integer (a value of 0 indicates no limit). Default value is 2000.

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

SolveEngine

This variable specifies which solver engine to use when evaluating randomize calls.

Section [vsim]

Syntax

SolveEngine = auto | act | bdd

auto — (default) automatically select the best engine for the current randomize scenario

act — evaluate all randomize scenarios using the ACT solver engine

bdd — evaluate all randomize scenarios using the BDD solver engine

You can override this variable by specifying [vsim -solveengine](#) at the command line.

SolveFailDebug

This variable enables the feature to debug SystemVerilog randomize() failures. Whenever a randomize() failure is detected during simulation, ModelSim displays the minimum set of constraints that caused the randomize() call to fail.

Section [vsim]

Syntax

SolveFailDebug = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vsim -solvefaildebug](#).

Related Topics

[Debugging randomize\(\) Failures](#)

SolveFailDebugMaxSet

When SolveFailDebug is enabled, this variable specifies the maximum size of constraint subsets (in number of constraints) that will be tested for conflicts.

Section [vsim]

Syntax

SolveFailDebugMaxSet = {0 | 1}

0 — Off (default)

1 — On

SolveFailSeverity

This variable allows you to specify the severity of messages that result when a SystemVerilog call to `randomize()` fails.

Section [vsim]

Syntax

SolveFailSeverity = {0 | 1 | 2 | 3 | 4}

0 — No error (default)

1 — Warning

2 — Error

3 — Failure

4 — Fatal

SolveFlags

This variable allows you to modify the behavior of the constraint solver (SystemVerilog `randomize()`) to improve the evaluation performance of some types of constraints.

Section [vsim]

Syntax

SolveFlags = {i | n | r | ""}

i — Disable bit interleaving for `>`, `>=`, `<`, `<=` constraints.

n — Disable bit interleaving for all constraints.

r — Reverse bit interleaving.

"" — (default) No options.

You can override this variable by specifying `vsim -solveflags`.

SolveGraphMaxEval

This variable specifies the maximum number of evaluations that may be performed on the solution graph generated during `randomize()`. This value can be used to force `randomize()` to

Variables

abort if the complexity of the constraint scenario (in time) exceeds the specified limit. The value is specified in 10000s of evaluations.

Section [vsim]

Syntax

SolveGraphMaxEval = <n>

<n>— A non-negative integer where 0 indicates no limit and the default is 10000.

SolveGraphMaxSize

This variable specifies the maximum size of the solution graph that may be generated during a SystemVerilog call to randomize(). You can use this value to force randomize() to abort if the complexity of the constraint scenario exceeds the specified limit. The limit is specified in 1000s of nodes.

Section [vsim]

Syntax

SolveGraphMaxSize = <n>

<n> — A non-negative integer (with the unit of 1000 nodes) where 0 indicates no limit and 10000 is the default.

SolveIgnoreOverflow

This variable instructs the solver to ignore calculation overflow or underflow while evaluating constraints.

Section [vsim]

Syntax

SolveIgnoreOverflow = 0 | 1

0 — (default) Do not ignore overflow or underflow.

1 — Ignore overflow or underflow.

SolveRev

This variable allows you to specify random sequence generator compatibility with a prior letter release for the SystemVerilog solver. (It does not apply to the SystemC/SCV solver.) This option is used to get the same random sequences during simulation as a prior letter release.

Section [vsim]

Syntax

SolveRev = <string> | ""

<string> — A string of a ModelSim release number and letter, such as 6.4a
(SolveRev = 6.4a).

" " — Off (default).

Note

Only prior letter releases (within the same number release) are allowed. For example, in 6.4b you can set "SolveRev= 6.4" or "SolveRev = 6.4a", but cannot set "SolveRev = 6.4g".

You can override this variable by specifying `vsim -solverev`.

SolveSpeculateDistFirst

This variable specifies whether to attempt speculation on solve-before constraints or dist constraints first. Note that while solve-before constraints and dist constraints may be used successfully to yield an LRM-compliant (IEEE Std 180-2009) distribution of solved random variable values, conditional variables do not guarantee an LRM-compliant distribution.

Section [vsim]

Syntax

SolveSpeculateDistFirst = 0 | 1

- 0 — (default) Attempt solve-before constraints first as the basis for speculative randomization. If this fails, attempt dist constraints next. If this fails, and the value of SolveSpeculateLevel is greater than 1, attempt conditional variables as the basis for speculation (subject to the value of SolveSpeculateMaxCondWidth).
- 1 — Attempt dist constraints first, followed by solve-before constraints, followed by conditional variables (if SolveSpeculateLevel is greater than 1).

SolveSpeculateFirst

This variable specifies whether to use a BDD solution or “speculation” for a randomization scenario. Speculation is an attempt to partition complex randomize scenarios by choosing a speculation subset of the variables and constraints.

Section [vsim]

Syntax

SolveSpeculateFirst = 0 | 1

- 0 — (default) Use a single BDD solution instead of speculation as an initial attempt to solve a constraint problem. If the solver succeeds in generating a BDD, that BDD is cached and re-used each time the randomization is repeated. This is the most efficient approach when a randomization is repeated many times, since the initial cost of building the BDD is amortized over many subsequent randomize calls. By contrast, caching is less effective for speculation. If the solver is unable to generate a single

BDD (due to SolveGraphMaxSize or SolveGraphMaxEval limits), it attempts to use speculation (if SolveSpeculateLevel = [1|2]) to partition the constraint problem into smaller, more manageable segments.

- 1 — Use speculation as an initial attempt to solve a constraint problem. This can be useful if randomizations are not repeated.

SolveSpeculateLevel

This variable specifies a level of “speculation” for randomization scenario, which is an attempt to partition complex randomize scenarios by choosing a speculation subset of the variables and constraints. This speculation set is solved independently of the remaining constraints. The solver then attempts to solve the remaining variables and constraints (the dependent set). If this attempt fails, the solver backs up and re-solves the ‘speculation’ set, then retries the dependent set.

Section [vsim]

Syntax

SolveSpeculateLevel = 0 | 1 | 2

- 0 — (default) Do not perform speculation.
- 1 — Perform speculation that yields a distribution that conforms to the IEEE Std 1800-2009 standard.
- 2 — Perform speculation that yields a distribution that may not conform to the IEEE Std 1800-2009 standard.

Currently, distribution constraints and solve-before constraints are used in selecting the ‘speculation’ sets for speculation level 1. Speculation that does not conform to LRM standard includes random variables in condition expressions.

SolveSpeculateMaxCondWidth

This variable specifies the maximum bit width of a variable in a conditional expression that may be considered as the basis for “conditional” speculation. This can be useful for some randomization scenarios where you have a relatively narrow control variable as the basis of a conditional expression and where each of the branches of the expression contains a complex set of constraints.

Section [vsim]

Syntax

SolveSpeculateMaxCondWidth = <n>

- <n> — A non-negative integer; default value is 6.

SolveSpeculateMaxIterations

This variable specifies the maximum number of attempts to make in solving a speculative set of random variables and constraints. Exceeding this limit abandons the current speculative set. If the set is dependent on (solved after) another speculative set, the solver will back up and retry the parent set. If there is no parent speculative set, the overall randomization will fail.

Section [vsim]

Syntax

SolveSpeculateMaxIterations = <n>

<n> — A non-negative integer; default value is 100.

SparseMemThreshold

This variable specifies the size at which memories will automatically be marked as sparse memory. A memory with depth equal to or more than the sparse memory threshold gets marked as sparse automatically, unless specified otherwise in source code or by **vlog +nonsparse**, or **vopt +nonsparse**.

Section [vlog]

Syntax

SpaseMemThreshhold = <n>

<n> — Any non-negative integer where the default is 1048576.

Related Topics

[Sparse Memory Modeling](#)

Startup

This variable specifies a simulation startup macro.

Section [vsim]

Syntax

Startup = {do <DO filename>}

<DO filename> — Any valid macro (do) file where the default is to comment out the line (;).

Related Topics

[do command](#)

[Using a Startup File](#)

std

This variable sets the path to the VHDL STD library.

Section [library]

Syntax

std = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./std. May include environment variables.

std_developerskit

This variable sets the path to the libraries for Mentor Graphics standard developer's kit.

Section [library]

Syntax

std_developerskit = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./std_developerskit. May include environment variables.

StdArithNoWarnings

This variable suppresses warnings generated within the accelerated Synopsys std_arith packages.

Section [vsim]

Syntax

StdArithNoWarnings = {0 | 1}

0 — Off (default)

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

You can set this variable interactively with the [Tcl set Command Syntax](#).

suppress

This variable suppresses the listed message numbers and/or message code strings (displayed in square brackets).

Section [msg_system]

Syntax

suppress = <msg_number>...

<msg_number>...— An unlimited list of message numbers, comma separated.

You can override this variable setting by specifying the [sccom](#), [vcom](#), [vlog](#), [vopt](#), or [vsim](#) command with the **-suppress** argument.

Related Topics

[verror <msg number>](#) prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [fatal](#), [note](#), [warning](#)

SuppressFileTypeReg

This variable suppresses a prompt from the GUI asking if ModelSim file types should be applied to the current version.

Section [vsim]

Syntax

SuppressFileTypeReg = {0 | 1}

0 — Off (default)

1 — On

You can suppress the GUI prompt for ModelSim type registration by setting the SuppressFileTypeReg variable value to 1 in the modelsim.ini file on each server in a server farm. This variable only applies to Microsoft Windows platforms.

Sv_Seed

This variable sets the initial seed for the Random Number Generator (RNG) of the root thread in SystemVerilog.

Section [vsim]

Syntax

`Sv_Seed = <n>`

`<n>` — Any 32-bit signed integer (from -2147483648 to +2147483647) where the default is 0. Integers outside the value range will be assigned either the minimum or maximum value.

You can override this variable by specifying `vsim -sv_seed`.

Related Topics

[Seeding the Random Number Generator \(RNG\)](#)

sv_std

This variable sets the path to the SystemVerilog STD library.

Section [library]

Syntax

`sv_std = <path>`

`<path>` — Any valid path where the default is `$MODEL_TECH/./sv_std`. May include environment variables.

SVAPrintOnlyUserMessage

This variable controls the printing of user-defined assertion error messages along with severity information.

Section [vsim]

Syntax

`SVAPrintOnlyUserMessage = {0 | 1}`

0 — (default) Prints additional information from LRM-defined (IEEE Std 1800-2009) Severity System tasks.

1 — Prints only the severity information and the user message.

SVCovergroupGetInstCoverageDefault

This variable allows you to specify an override for the default value of the "get_inst_coverage" option for Covergroup variables. This is a compile time option which forces "get_inst_coverage" to a user specified default value and supersedes the SystemVerilog specified default value of '0' (zero).

Section [vlog], [vsim]

Syntax

SVCovergroupGetInstCoverageDefault = {0 | 1}

0 — Off (default)

1 — On

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupGoal

This variable is used to override both the default value of option.goal (100, unless otherwise set with the [SVCovergroupGoalDefault](#) compiler control variable), as well as any explicit assignments to covergroup, coverpoint, and cross option.goal placed in SystemVerilog.

Section [vsim]

Syntax

SVCovergroupGoal = <n>

<n> — Any non-negative integer where the default is 100.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupGoalDefault

This variable is used in conjunction with the [SVCovergroupGoal](#) simulator control variable, and overrides the default value of the SystemVerilog covergroup, coverpoint, and cross option.goal (defined to be 100 in the IEEE Std 1800-2009). This variable does not override specific assignments in SystemVerilog source code.

Note



You must re-compile the design after changing the setting of this variable.

Section [vlog]

Syntax

SVCovergroupGoalDefault = <n>

<n> — Any integer where the default is 100.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupMergeInstancesDefault

This variable exists in the [vlog](#) and [vsim](#) sections of the modelsim.ini file.

Section [vlog]

For compilation, this variable overrides for the default value of the *merge_instances* option for the covergroup type. It forces *merge_instances* to default value you specify and supersedes the SystemVerilog specified default value of zero.

Section [vsim]

For simulation, this variable enforces the version 6.5 default behavior of covergroup *get_coverage()* built-in functions, GUI operations, and reports. It changes the default value of *type_option.merge_instances* to ensure the 6.5 default behavior if explicit assignments are not made on *type_option.merge_instances*. Two [vsim](#) command line options, —
-cvgmergeinstances and -nocvgmergeinstances — override this variable setting.

Coverpoint and Cross scopes are now created under covergroup type scopes when *type_option.merge_instances* is 0 (zero). Those coverpoints and crosses are displayed in reports and in the GUI and do not contain any bins. The coverage of such coverpoint or cross items is the weighted average of the corresponding coverpoint or cross instance items from all the instances of the covergroup.

Syntax

SVCovergroupMergeInstancesDefault = {0 | 1}

0 — Off (default)

1 — On

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupPerInstanceDefault

This variable is used to set the default value for SystemVerilog option.per_instance (defined to be 0 in the IEEE Std 1800-2009). It does not override explicit assignments to option.per_instance.

Note



You must re-compile the design after changing the setting of this variable.

Section [vlog]

Syntax

SVCovergroupPerInstanceDefault = {0 | 1}

0 — Off (default)

1 — On

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupSampleInfo

This variable is used to enable generation of more detailed information about the sampling of covergroup, cross, and coverpoints. It provides details about the number of times the covergroup instance and type were sampled, as well as details about why covergroup, cross, and coverpoint were not covered.

Section [vsim]

Syntax

SVCovergroupSampleInfo = <n>

<n> — Any integer where the default is 0.

SVCovergroupStrobe

This variable is used to override both the default value of `type_option.strobe` (0, unless otherwise set with the [SVCovergroupStrobeDefault](#) variable), as well as any user assignments for `covergroup`, `coverpoint`, and `cross` `type_option.strobe`, placed in SystemVerilog.

Section [vsim]

Syntax

`SVCovergroupStrobe = <n>`

<n> — Any integer where the default is 0.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupStrobeDefault

This variable is used in conjunction with the [SVCovergroupStrobe](#) variable, and overrides the SystemVerilog `covergroup` `type_option.strobe` (defined to be 0 in the IEEE Std 1800-2009). It does not override explicit assignments to `type_option.strobe`.

Note

You must re-compile the design after changing the setting of this variable.

Section [vlog]

Syntax

`SVCovergroupStrobeDefault = <n>`

<n> — Any integer where the default is 0.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupTypeGoal

This variable is used to override both the default value of `type_option.goal` (100, unless otherwise set with the [SVCovergroupTypeGoalDefault](#) variable), as well as any user assignments for `covergroup`, `coverpoint`, and `cross` `type_option.goal`, placed in SystemVerilog.

Section [vsim]

Syntax

SVCovergroupTypeGoal = <n>

<n> — Any integer where the default is 100.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupTypeGoalDefault

This variable is used to override the default value of the SystemVerilog covergroup, coverpoint, and cross type_option.goal (defined to be 100 in the IEEE Std 1800-2009). It does not override specific assignments in SystemVerilog source code.

Note



You must re-compile the design after changing the setting of this variable.

Section [vlog]

Syntax

SVCovergroupTypeGoal Default = <n>

<n> — Any integer where the default is 100.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCovergroupZWNNoCollect

This variable is used to disable coverage collection for any coverage item (coverpoint or cross or the entire covergroup), when 0 is assigned as its *option.weight*. Item will not be displayed in any coverage report, nor will it contribute to any coverage score computation.

Section [vsim]

Syntax

SVCovergroupZWNNoCollect = {0 | 1}

0 — Off (default)

1 — On - disables coverage collection for coverage item

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCoverpointAutoBinMax

This variable is used to override both the default value of option.auto_bin_max (64), as well as any explicit assignments in source code to SystemVerilog covergroup option.auto_bin_max.

Section [vsim]

Syntax

SVCoverpointAutoBinMax = <n>

<n> — Any non-negative integer where the default is 64.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCoverpointExprVariablePrefix

When [GenerateLoopIterationMax](#) = 1, this variable sets the prefix in the name of the user-visible variable generated for the coverpoint expression sampled value.

The current settings for both this variable and the [EnableSVCoverpointExprVariable](#) are displayed in the Objects window.

Note

You must re-compile the design after changing the setting of either this variable or the [EnableSVCoverpointExprVariable](#).

Section [vlog]

Syntax

SVCoverpointExprVariablePrefix = [<value><coverpoint name> | expr]

<value> — A user defined string.

<coverpoint name> — The name of the coverpoint.

expr — (default)

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCoverpointWildcardBinValueSizeWarn

This variable sets the threshold value range beyond which a warning for SV Coverpoint wildcard bin size is issued. The default threshold is 4096 (12 wildcard bits).

Section [vsim]

Syntax

SVCoverpointWildcardBinValueSizeWarn = [<value>]

SVCrossNumPrintMissing

This variable is used to override all other settings for the number of missing values that will be printed to the coverage report. It overrides both the default value (0, unless otherwise set with the [SVCrossNumPrintMissingDefault](#) variable), as well as any user assignments placed in SystemVerilog.

Section [vsim]

Syntax

SVCrossNumPrintMissing = <n>

<n> — Any non-negative integer where the default is 0.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVCrossNumPrintMissingDefault

This variable is used in conjunction with [SVCrossNumPrintMissing](#) variable, and overrides the default value of the SystemVerilog covergroup option.cross_num_print_missing (defined to be 0 in the IEEE Std 1800-2009).

Note



You must recompile the design after changing the setting of this variable.

Section [vlog]

Syntax

SVCrossNumPrintMissingDefault = <n>

<n> — Any non-negative integer where the default is 0.

Related Topics

Covergroup options in section 19.7, entitled "Specifying Coverage Options," of the IEEE Std 1800-2009 for SystemVerilog.

SVFileExtensions

This variable defines one or more filename suffixes that identify a file as a SystemVerilog file. To insert white space in an extension, use a backslash (\) as a delimiter. To insert a backslash in an extension, use two consecutive back-slashes (\\).

Section [vlog]

Syntax

SVFileExtensions = sv svp svh

On — Uncomment the variable.

Off — Comment the variable (;).

Svlog

This variable instructs the vlog compiler to compile in SystemVerilog mode. This variable does not exist in the default *modelsim.ini* file, but is added when you select Use SystemVerilog in the Compile Options dialog box > Verilog and SystemVerilog tab.

Section [vlog]

Syntax

Svlog = {0 | 1}

0 — Off (default)

1 — On

synopsys

This variable sets the path to the accelerated arithmetic packages.

Section [vsim]

Syntax

synopsys = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./synopsys. May include environment variables.

SyncCompilerFiles

This variable causes compilers to force data to be written to disk when files are closed.

Section [vcom]

Syntax

SyncCompilerFiles = {0 | 1}

0 — Off (default)

1 — On

ToggleCountLimit

This variable limits the toggle coverage count for a toggle node. After the limit is reached, further activity on the node will be ignored for toggle coverage. All possible transition edges must reach this count for the limit to take effect. For example, if you are collecting toggle data on 0->1 and 1->0 transitions, both transition counts must reach the limit. If you are collecting full data on 6 edge transitions, all 6 must reach the limit. If the limit is set to zero, then it is treated as unlimited.

Section [vsim]

Syntax

ToggleCountLimit = <n>

<n> — Any non-negative integer with a maximum positive value of a 32-bit signed integer and a default of 1.

You can override this variable by specifying **vsim -togglecountlimit** or **toggle add -countlimit**.

ToggleFixedSizeArray

This variable is used to control whether Verilog fixed-size unpacked arrays, VHDL multi-dimensional arrays, and VHDL arrays-of-arrays are included for toggle coverage.

Section [vsim]

Syntax

ToggleFixedSizeArray = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying **vsim {-togglefixedsizearray | -notogglefixedsizearray}**.

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

ToggleMaxFixedSizeArray

This variable is used to control the limit on the size of Verilog fixed-size unpacked arrays, VHDL multi-dimensional arrays, and VHDL arrays-of-arrays that are included for toggle coverage. Increasing the size of the limit has the effect of increasing the size of the array that can be included for toggle coverage.

Section [vsim]

Syntax

ToggleMaxFixedSizeArray = <n>

<n> — Any positive integer where the default is 1024.

You can override this variable by specifying [vsim -togglemaxfixedsizearray](#).

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

ToggleMaxIntValues

This variable sets the maximum number of unique VHDL integer values to record with toggle coverage.

Section [vsim]

Syntax

ToggleMaxIntValues = <n>

<n> — Any positive integer where the default is 100.

You can override this variable by specifying [vsim -togglemaxintvalues](#).

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

ToggleMaxRealValues

This variable sets the maximum number of unique SystemVerilog real values to record with toggle coverage.

Section [vsim]

Syntax

ToggleMaxIntValues = <n>

<n> — Any positive integer where the default is 100.

You can override this variable by specifying [vsim -togglemaxrealvalues](#).

Related Topics

You can set this variable interactively with the Tcl [set Command Syntax](#).

ToggleNoIntegers

This variable controls the automatic inclusion of VHDL integer types in toggle coverage.

Section [vsim]

Syntax

ToggleNoIntegers = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vsim -notoggleints](#).

TogglePackedAsVec

This variable treats Verilog multi-dimensional packed vectors and packed structures as equivalently sized one_dimensional packed vectors for toggle coverage.

Section [vsim]

Syntax

TogglePackedAsVec = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vsim -togglepackedasvec`.

TogglePortsOnly

This variable controls the inclusion into toggle coverage numbers of ports only; when enabled, all internal nodes are not included in the coverage numbers. When disabled, both ports and internal nodes are included. In order for this variable to function properly, you must also use “`vopt +acc=p`”.

Section [vsim]

Syntax

TogglePortsOnly = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vsim -toggleportsonly`.

ToggleVlogEnumBits

This variable treats Verilog enumerated types as equivalently sized one-dimensional packed vectors for toggle coverage.

Section [vsim]

Syntax

ToggleVlogEnumBits = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vsim -togglevlogenumbits`.

ToggleVlogIntegers

This variable controls toggle coverage for SystemVerilog integer types (that is, byte, shortint, int, longint, but not enumeration types).

Section [vsim]

Syntax

ToggleVlogIntegers = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying `vsim [-togglevlogints | -notogglevlogints]`.

ToggleVlogReal

This variable controls toggle coverage for SystemVerilog real value types.

Section [vsim]

Syntax

ToggleVlogReal = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying `vsim {-togglevlogreal | -notogglevlogreal}`.

ToggleWidthLimit

This variable limits the width of signals that are automatically added to toggle coverage with the `+cover=t` argument for `vcom` or `vlog`. The limit applies to Verilog registers and VHDL arrays. A value of 0 is taken as unlimited.

Section [vsim]

Syntax

ToggleWidthLimit = <n>

<n> — Any non-negative integer with a maximum positive value of a 32-bit signed integer and a default of 128.

You can override this variable by specifying `vsim -togglewidthlimit`.

Related Topics

`vcom -togglewidthlimit`

`vlog -togglewidthlimit`

TranscriptFile

This variable specifies a file for saving a command transcript. You can specify environment variables in the pathname.

Note



Once you load a modelsim.ini file with TranscriptFile set to a file location, this location will be used for all output until you override the location with the `transcript file` command. This includes the scenario where you load a new design with a new TranscriptFile variable set to a different file location.

You can determine the current path of the transcript file by executing the `transcript path` command with no arguments.

Section [vsim]

Syntax

TranscriptFile = {<filename> | transcript}

<filename> — Any valid filename where transcript is the default.

Related Topics

[transcript file](#) command

[AssertFile](#)

UCDBFilename

This variable specifies the default unified coverage database file name that is written at the end of the simulation. If this variable is set, the UCDB is saved automatically at the end of simulation. All coverage statistics are saved to the specified *.ucdb* file.

Section [vsim]

Syntax

UCDBFilename = {<filename> | *vsim.ucdb*}

<filename> — Any valid filename where *vsim.ucdb* is the default.

UCDBTestStatusMessageFilter

This variable specifies a regular expression which, if matched when compared against all messages, prevents the status of that message from being propagated to the UCDB TESTSTATUS. If this variable is set, the matching regular expression is ignored for all messages which contain that match.

Section [vsim]

Syntax

UCDBTestStatusMessageFilter =“<subtext>” [“<additional subtext>”]

<subtext> — Subtext of message you wish to be exempt from altering UCDB TESTSTATUS.

UnattemptedImmediateAssertions

This variable controls the inclusion or exclusion of unattempted (un-executed) immediate assertions from the coverage calculations shown in the UCDB and coverage reports.

Section [vsim]

Syntax

UnattemptedImmediateAssertions = {0 | 1}

- 0 — Off, Excluded (default)
- 1 — On, Included

UnbufferedOutput

This variable controls VHDL and Verilog files open for write.

Section [vsim]

Syntax

UnbufferedOutput = {0 | 1}

- 0 — Off, Buffered (default)
- 1 — On, Unbuffered

UpCase

This variable instructs ModelSim to activate the conversion of regular Verilog identifiers to uppercase and allows case insensitivity for module names.

Section [vlog]

Syntax

UpCase = {0 | 1}

- 0 — Off (default)
- 1 — On

Related Topics

[Verilog-XL Compatible Compiler Arguments](#)

UserTimeUnit

This variable specifies the multiplier for simulation time units and the default time units for commands such as [force](#) and [run](#). Generally, you should set this variable to default, in which case it takes the value of the [Resolution](#) variable.

Note



The value you specify for UserTimeUnit does not affect the display in the Wave window. To change the time units for the X-axis in the Wave window, choose Wave > Wave Preferences > Grid & Timeline from the main menu and specify a value for Grid Period.

Section [vsim]

Syntax

UserTimeUnit = {<time_unit> | default}

<time_unit> — fs, ps, ns, us, ms, sec, or default.

Related Topics

[RunLength](#) variable.

You can set this variable interactively with the [Tcl set Command Syntax](#).

UseScv

This variable enables the use of SCV include files and verification library.

Section [sccom]

Syntax

UseScv = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [sccom -scv](#).

UseSVCrossNumPrintMissing

Specify whether to display and report the value of the "cross_num_print_missing" option for the Cross in Covergroups. If not specified then "cross_num_print_missing" is ignored for creating reports and displaying covergroups in GUI. Default is 0, which means ignore "cross_num_print_missing."

Section [vsim]

Syntax

UseSVCrossNumPrintMissing = {0 | 1}

0 — Off (default)

1 — On

UVMControl

This variable controls UVM-Aware debug features. These features work with either a standard Accelera-released open source toolkit or the pre-compiled UVM library package in ModelSim.

Section [vsim]

Syntax

UVMControl={all | certe | disable | msglog | none | struct | trlog | verbose}

You must specify at least one argument. You can enable or disable some arguments by prefixing the argument with a dash (-). Arguments may be specified as multiple instances of -uvmcontrol. Multiple arguments are specified as a comma separated list without spaces. Refer to the argument descriptions for more information.

- all — Enables all UVM-Aware functionality and debug options except disable and verbose. You must specify verbose separately.
- certe — Enables the integration of the elaborated design in the Certe tool. Disables Certe features when specified as -certe.
- disable — Prevents the UVM-Aware debug package from being loaded. Changes the results of randomized values in the simulator.
- msglog — Enables messages logged in UVM to be integrated into the Message Viewer. You must also enable wlf message logging by specifying tran or wlf with vsim -msgmode. Disables message logging when specified as -msglog
- none — Turns off all UVM-Aware debug features. Useful when multiple -uvmcontrol options are specified in a separate script, makefile or alias and you want to be sure all UVM debug features are turned off.
- struct — (default) Enables UVM component instances to appear in the Structure window. UVM instances appear under “uvm_root” in the Structure window. Disables Structure window support when specified as -struct.
- trlog — Enables or disables UVM transaction logging. Logs UVM transactions for viewing in the Wave window. Disables transaction logging when specified as -trlog.
- verbose — Sends UVM debug package information to the transcript. Does not affect functionality. Must be specified separately.

You can also control UVM-Aware debugging with the -uvmcontrol argument to the [vsim](#) command.

verilog

This variable sets the path to the library containing VHDL/Verilog type mappings.

Section [library]

Syntax

verilog = <path>

<path> — Any valid path where the default is \$MODEL_Tech/./verilog. May include environment variables.

Veriuser

This variable specifies a list of dynamically loadable objects for Verilog PLI/VPI applications.

Section [vsim]

Syntax

Veriuser = <name>

<name> — One or more valid shared object names where the default is to comment out the variable.

Related Topics

[vsim -pli](#)

[restart](#) command.

[Registering PLI Applications](#)

VHDL93

This variable enables support for VHDL language version.

Section [vcom]

Syntax

VHDL93 = {0 | 1 | 2 | 3 | 87 | 93 | 02 | 08 | 1987 | 1993 | 2002 | 2008}

0 — Support for VHDL-1987. You can also specify 87 or 1987.

1 — Support for VHDL-1993. You can also specify 93 or 1993.

2 — Support for VHDL-2002 (default). You can also specify 02 or 2002.

3 — Support for VHDL-2008. You can also specify 08 or 2008.

You can override this variable by specifying [vcom](#) {-87 | -93 | -2002 | -2008}.

VhdlVariableLogging

This switch makes it possible for process variables to be recursively logged or added to the Wave and List windows (process variables can still be logged or added to the Wave and List windows explicitly with or without this switch). For example with this vsim switch, `log -r /*` will log process variables as long as `vopt` is specified with `+acc=v` and the variables are not filtered out by the WildcardFilter (via the "Variable" entry).

Note



Logging process variables is inherently expensive on simulation performance because of their nature. It is recommended that they not be logged, or added to the Wave and List windows. However, if your debugging needs require them to be logged, then use of this switch will lessen the performance hit in doing so.

Section [vsim]

Syntax

VhdlVariableLogging = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vsim -novhdlvariablelogging](#).

Related Topics

[vsim -vhdlvariablelogging](#)

vital2000

This variable sets the path to the VITAL 2000 library.

Section [library]

Syntax

vital2000 = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./vital2000. May include environment variables.

vlog95compat

This variable instructs ModelSim to disable SystemVerilog and Verilog 2001 support, making the compiler revert to IEEE Std 1364-1995 syntax.

Section [vlog]

Syntax

vlog95compat = {0 | 1}

0 — Off (default)

1 — On

You can override this variable by specifying [vlog -vlog95compat](#).

VoptFlow

This variable controls whether ModelSim operates in optimized mode or full visibility mode.

Section [vsim]

Syntax

VoptFlow = {0 | 1}

0 — Off, Design is compiled and simulated without optimizations, maintaining full visibility.

1 — On (default), vopt is invoked automatically on the design and the design is fully optimized.

You can override `VoptFlow = 0` by specifying `vsim -vopt`.

Related Topics

[Optimizing Designs with vopt](#)

[Using SV Bind With or Without vopt](#)

WarnConstantChange

This variable controls whether a warning is issued when the change command changes the value of a VHDL constant or generic.

Section [vsim]

Syntax

WarnConstantChange = {0 | 1}

0 — Off

1 — On (default)

Related Topics

[change](#) command

warning

This variable changes the severity of the listed message numbers to "warning".

Section [msg_system]

Syntax

warning = <msg_number>...

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable setting by specifying the `sccom`, `vcom`, `vlog`, `vopt`, or `vsim` command with the `-warning` argument.

Related Topics

[error](#) <msg number> prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [fatal](#), [note](#), [suppress](#)

WaveSignalNameWidth

This variable controls the number of visible hierarchical regions of a signal name shown in the [Wave Window](#).

Section [vsim]

Syntax

WaveSignalNameWidth = <n>

<n> — Any non-negative integer where the default is 0 (display full path). 1 displays only the leaf path element, 2 displays the last two path elements, and so on.

You can override this variable by specifying [configure -signalnamewidth](#).

WLFCacheSize

This variable sets the number of megabytes for the WLF reader cache. WLF reader caching caches blocks of the WLF file to reduce redundant file I/O.

Section [vsim]

Syntax

WLFCacheSize = <n>

<n> — Any non-negative integer where the default is 2000M.

You can override this variable by specifying [vsim -wlf cachesize](#).

Related Topics

[WLF File Parameter Overview](#)

WLFCollapseMode

This variable controls when the WLF file records values.

Section [vsim]

Syntax

WLFCollapseMode = {0 | 1 | 2}

0 — Preserve all events and event order. Same as [vsim -nowlfcollapse](#).

1 — (default) Only record values of logged objects at the end of a simulator iteration. Same as [vsim -wlfcollapsedelta](#).

2 — Only record values of logged objects at the end of a simulator time step. Same as [vsim -wlfcollapsetime](#).

You can override this variable by specifying [vsim](#) {-nowlfcollapse | -wlfcollapsedelta | -wlfcollapsetime}.

Related Topics

[WLF File Parameter Overview](#)

WLFCompress

This variable enables WLF file compression.

Section [vsim]

Syntax

WLFCompress = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vsim -nowlfcompress](#).

Related Topics

[WLF File Parameter Overview](#)

vsim -wlfcompress

You can set this variable in the [The Runtime Options Dialog](#).

vsim -nowlfcompress

WLFDeleteOnQuit

This variable specifies whether a WLF file should be deleted when the simulation ends.

Section [vsim]

Syntax

WLFDeleteOnQuit = {0 | 1}

0 — Off (default), Do not delete.

1 — On

You can override this variable by specifying [vsim -nowlfdeleteonquit](#).

Related Topics

[WLF File Parameter Overview](#)

`vsim -wlfdeleteonquit`

You can set this variable in the [The Runtime Options Dialog](#).

`vsim -nowlfdeleteonquit`

WLFFileLock

This variable controls overwrite permission for the WLF file.

Section [vsim]

Syntax

WLFFileLock = {0 | *I*}

0 — Allow overwriting of the WLF file.

I — (default) Prevent overwriting of the WLF file.

You can override this variable by specifying `vsim -wlflock` or `vsim -nowlflock`.

Related Topics

[WLF File Parameter Overview](#)

`vsim -wlflock`

WLFFilename

This variable specifies the default WLF file name.

Section [vsim]

Syntax

WLFFilename = {<filename> | *vsim.wlf*}

<filename> — User defined WLF file to create.

vsim.wlf — (default) filename

You can override this variable by specifying `vsim -wlf`.

Related Topics

[WLF File Parameter Overview](#)

You can set this variable interactively with the [Tcl set Command Syntax](#).

WLFOptimize

This variable specifies whether the viewing of waveforms is optimized.

Section [vsim]

Syntax

WLFOptimize = {0 | 1}

0 — Off

1 — On (default)

You can override this variable by specifying [vsim -nowlfopt](#).

Related Topics

[WLF File Parameter Overview](#)

[vsim -wlfopt](#).

WLFSaveAllRegions

This variable specifies the regions to save in the WLF file.

Section [vsim]

Syntax

WLSaveAllRegions= {0 | 1}

0 — (default), Only save regions containing logged signals.

1 — Save all design hierarchy.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

WLFSimCacheSize

This variable sets the number of megabytes for the WLF reader cache for the current simulation dataset only. WLF reader caching caches blocks of the WLF file to reduce redundant file I/O. This makes it easier to set different sizes for the WLF reader cache used during simulation, and those used during post-simulation debug. If the WLFSimCacheSize variable is not specified, the [WLFCacheSize](#) variable is used.

Section [vsim]

Syntax

WLFSimCacheSize = <n>

<n> — Any non-negative integer where the default is 500.

You can override this variable by specifying [vsim -wlfsimcachesize](#).

Related Topics

[WLF File Parameter Overview](#)

WLFSizeLimit

This variable limits the WLF file by size (as closely as possible) to the specified number of megabytes; if both size (`WLFSizeLimit`) and time (`WLFTimeLimit`) limits are specified the most restrictive is used.

Section [vsim]

Syntax

`WLFSizeLimit = <n>`

`<n>` — Any non-negative integer in units of MB where the default is 0 (unlimited).

You can override this variable by specifying `vsim -wlfslim`.

Related Topics

[WLF File Parameter Overview](#)

[Limiting the WLF File Size](#)

You can set this variable interactively with the Tcl [set Command Syntax](#).

WLFTimeLimit

This variable limits the 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.

Section [vsim]

Syntax

`WLFTimeLimit = <n>`

`<n>` — Any non-negative integer in units of MB where the default is 0 (unlimited).

You can override this variable by specifying `vsim -wlfylim`.

Related Topics

[WLF File Parameter Overview](#)

You can set this variable in the [The Runtime Options Dialog](#).

[Limiting the WLF File Size](#)

You can set this variable interactively with the [Tcl set Command Syntax](#).

WLFUseThreads

This variable specifies whether the logging of information to the WLF file is performed using multithreading.

Section [vsim]

Syntax

WLFUseThreads = {0 | 1}

0 — Off, (default) Windows systems only, or when one processor is available.

1 — On Linux systems only, with more than one processor on the system. When this behavior is enabled, the logging of information is performed by the secondary processor while the simulation and other tasks are performed by the primary processor.

You can override this variable by specifying [vsim -nowlfopt](#).

Related Topics

[Multithreading on Linux Platforms](#)

Commonly Used modelsim.ini Variables

Several of the more commonly used *modelsim.ini* variables are further explained below.



Tip: When a design is loaded, you can use the [where](#) command to display which *modelsim.ini* or ModelSim Project File (*.mpf*) file is in use.

Common Environment Variables

You can use environment variables in an initialization file. Insert a dollar sign (\$) before the name of the environment variable so that its defined value is used. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

Note



The MODEL_TECH environment variable is a special variable that is set by ModelSim (it is not user-definable). ModelSim sets this value to the name of the directory from which the VCOM or VLOG compilers or the VSIM simulator was invoked. This directory is used by other ModelSim commands and operations to find the libraries.

Hierarchical Library Mapping

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

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

Creating a Transcript File

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, and so forth. To do this, set the value for the [TranscriptFile](#) line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnsrpt
```

You can prevent overwriting older transcript files by including a pound sign (#) in **the name of the file**. The simulator replaces the '#' character with the next available sequence number when saving a new transcript file.

When you invoke **vsim** using the default *modelsim.ini* file, a transcript file is opened in the current working directory. If you then change (cd) to another directory that contains a different *modelsim.ini* file with a **TranscriptFile** variable setting, the simulator continues to save to the

original transcript file in the former location. You can change the location of the transcript file to the current working directory by:

- changing the preference setting (**Tools > Edit Preferences > By Name > Main > file**).
- using the [transcript file](#) command.

To limit the amount of disk space used by the transcript file, you can set the maximum size of the transcript file with the [transcript sizelimit](#) command.

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

Using a Startup File

The system initialization file allows you to specify a command or a *.do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the [do](#) command for additional information on creating do files.

Turning Off Assertion Messages

You can turn off assertion messages from your VHDL code by setting a variable in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Turning off Warnings from Arithmetic Packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

These variables can also be set interactively using the Tcl [set Command Syntax](#). This capability provides an answer to a common question about disabling warnings at time 0. You might enter commands like the following in a DO file or at the ModelSim prompt:

```
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

Force Command Defaults

The [force](#) command has **-freeze**, **-drive**, and **-deposit** arguments. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. 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**, **-nofcovers**, **-nolist**, **-nolog**, and **-nowave** arguments. 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**, **-nofcovers**, **-nolist**, **-nolog**, and **-nowave**.

Example:

```
DefaultRestartOptions = -nolog -force
```

VHDL Standard

You can specify which version of the 1076 Std ModelSim follows by default using the [VHDL93](#) variable:

Commonly Used modelsim.ini Variables

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

Opening VHDL Files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the [DelayFileOpen](#) option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

Appendix B

Location Mapping

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.

Referencing Source Files with Location Maps

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

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

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

Using Location Mapping

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

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

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

Use these two steps to map your files:

1. Set the environment variable `MGC_LOCATION_MAP` to the path of your location map file.
2. Specify the mappings from physical pathnames to logical pathnames:

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

$IEEE
/usr/modeltech/ieee
```

Pathname Syntax

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

How Location Mapping Works

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

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

Mapping with TCL Variables

Two Tcl variables may also be used to specify alternative source-file paths; `SourceDir` and `SourceMap`. You would define these variables in a `modelsim.tcl` file. See [The modelsim.tcl File](#) for details.

Appendix C

Error and Warning Messages

Message System

The ModelSim message system helps you identify and troubleshoot problems while using the application. The messages display in a standard format in the Transcript pane. Accordingly, you can also access them from a saved transcript file (see [Saving the Transcript File](#) for more details).

Message Format

The format for the messages is:

```
** <SEVERITY LEVEL>: ([<Tool>[-<Group>]]-<MsgNum>) <Message>
```

- **SEVERITY LEVEL** — may be one of the following:

Table C-1. Severity Level Types

severity level	meaning
Note	This is an informational message.
Warning	There may be a problem that will affect the accuracy of your results.
Error	The tool cannot complete the operation.
Fatal	The tool cannot complete execution.
INTERNAL ERROR	This is an unexpected error that should be reported to your support representative.

- **Tool** — indicates which ModelSim tool was being executed when the message was generated. For example, tool could be vcom, vdel, vsim, and so forth.
- **Group** — indicates the topic to which the problem is related. For example group could be PLI, VCD, and so forth.

Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few arguments.
```

Getting More Information

Each message is identified by a unique MsgNum id. You can access additional information about a message using the unique id and the [verror](#) command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

Changing Message Severity Level

You can suppress or change the severity of notes, warnings, and errors that come from **vcom**, **vlog**, and **vsim**. You cannot change the severity of or suppress Fatal or Internal messages.

There are three ways to modify the severity of or suppress notes, warnings, and errors:

- Use the `-error`, `-fatal`, `-note`, `-suppress`, and `-warning` arguments to [sccom](#), [vcom](#), [vlog](#), [vopt](#), or [vsim](#). See the command descriptions in the Reference Manual for details on those arguments.
- Use the [suppress](#) command.
- Set a permanent default in the `[msg_system]` section of the *modelsim.ini* file. See [modelsim.ini Variables](#) for more information.

Suppressing Warning Messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

Suppressing VCOM Warning Messages

Use the `-nowarn <category_number>` argument with the [vcom command](#) to suppress a specific warning message. For example:

```
vcom -nowarn 1
```

suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the Main window **Compile > Compile Options** menu selections or the *modelsim.ini* file (see [modelsim.ini Variables](#)).

The warning message category numbers are:

```
1 = unbound component
2 = process without a wait statement
3 = null range
4 = no space in time literal
5 = multiple drivers on unresolved signal
6 = VITAL compliance checks ("VitalChecks" also works)
7 = VITAL optimization messages
8 = lint checks
9 = signal value dependency at elaboration
10 = VHDL-1993 constructs in VHDL-1987 code
11 = PSL warnings
13 = constructs that coverage can't handle
14 = locally static error deferred until simulation run
```

These numbers are unrelated to `vcom` arguments that are specified by numbers, such as `vcom -87` – which disables support for VHDL-1993 and 2002.

Suppressing VLOG Warning Messages

As with the `vcom` command, you can use the `-nowarn <category_number>` argument with the `vlog` command to suppress a specific warning message. The warning message category numbers for `vlog` are:

```
11 = PSL warnings
12 = non-LRM compliance in order to match Cadence behavior
13 = constructs that coverage can't handle
15 = SystemVerilog assertions using local variable
```

Or, you can use the `+nowarn<CODE>` argument with the `vlog` command to suppress a specific warning message. Warnings that can be disabled include the `<CODE>` name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

suppresses decay warning messages.

Suppressing VOPT Warning Messages

Use the `-nowarn <category_number>` argument with the `vopt` command to suppress a specific warning message. For example:

```
vopt -nowarn 1
```

suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the Main window **Compile > Compile Options** menu selections or the `modelsim.ini` file (see [modelsim.ini Variables](#)).

The warning message category numbers are:

- 1 = unbound component
- 2 = process without a wait statement
- 3 = null range
- 4 = no space in time literal
- 5 = multiple drivers on unresolved signal
- 6 = VITAL compliance checks ("VitalChecks" also works)
- 7 = VITAL optimization messages
- 8 = lint checks
- 9 = signal value dependency at elaboration
- 10 = VHDL-1993 constructs in VHDL-1987 code
- 11 = PSL warnings
- 12 = non-LRM compliance in order to match Cadence behavior
- 13 = constructs that coverage can't handle
- 14 = locally static error deferred until simulation run
- 15 = SystemVerilog assertions using local variable

Or, you can use the **+nowarn<CODE>** argument with the **vopt** command to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vopt +nowarnDECAY
```

suppresses decay warning messages.

Suppressing VSIM Warning Messages

Use the **+nowarn<CODE>** argument to **vsim** to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vsim +nowarnTFMPC
```

suppresses warning messages about too few port connections.

You can use **vsim -msglimit <msg_number>[,<msg_number>,...]** to limit the number of times specific warning message(s) are displayed to five. All instances of the specified messages are suppressed after the limit is reached.

Exit Codes

The table below describes exit codes used by ModelSim tools.

Table C-2. Exit Codes

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing

Table C-2. Exit Codes

Exit code	Description
3	Cannot create a system process (execv, fork, spawn, and so forth.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, and so forth.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
22	SystemC link error
23	SystemC DPI internal error
24	SystemC archive error
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
100	GUI Tcl initialization failure
101	GUI Tk initialization failure
102	GUI IncrTk initialization failure

Table C-2. Exit Codes

Exit code	Description
111	X11 display error
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

Miscellaneous Messages

This section describes miscellaneous messages which may be associated with ModelSim.

Compilation of DPI Export TFs Error

```
# ** Fatal: (vsim-3740) Can't locate a C compiler for compilation of  
DPI export tasks/functions.
```

- Description — ModelSim was unable to locate a C compiler to compile the DPI exported tasks or functions in your design.
- Suggested Action — Make sure that a C compiler is visible from where you are running the simulation.

Empty port name warning

```
# ** WARNING: [8] <path/file_name>: empty port name in port list.
```

- Description — ModelSim reports these warnings if you use the **-lint** argument to **vlog**. It reports the warning for any NULL module ports.
- Suggested action — If you wish to ignore this warning, do not use the **-lint** argument.

Lock message

```
waiting for lock by user@user. Lockfile is <library_path>/_lock
```

- Description — The `_lock` file is created in a library when you begin a compilation into that library, and it is removed when the compilation completes. This prevents simultaneous updates to the library. If a previous compile did not terminate properly, ModelSim may fail to remove the `_lock` file.
- Suggested action — Manually remove the `_lock` file after making sure that no one else is actually using that library.

Metavalue detected warning

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

- Description — This warning is an assertion being issued by the IEEE **numeric_std** package. It indicates that there is an 'X' in the comparison.
- Suggested action — The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue arrow in a Source window will be pointing at the line following the line with the comparison.

These messages can be turned off by setting the **NumericStdNoWarnings** variable to 1 from the command line or in the `modelsim.ini` file.

Sensitivity list warning

```
signal is read by the process but is not in the sensitivity list
```

- Description — ModelSim outputs this message when you use the **-check_synthesis** argument to **vcom**. It reports the warning for any signal that is read by the process but is not in the sensitivity list.
- Suggested action — There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, your only option is to not use the **-check_synthesis** argument.

Tcl Initialization error 2

```
Tcl_Init Error 2 : Can't find a usable Init.tcl in the following
  directories :
  ../../tcl/tcl8.3 .
```

- Description — This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

```
modeltech-base.mis
modeltech-docs.mis
install.<platform>
```

If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

This message could also occur if the file or directory was deleted or corrupted.

- Suggested action — Reinstall ModelSim with all three files.

Too few port connections

```
# ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port
  connections. Expected 2, found 1.
# Region: /foo/tb
```

- Description — This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn't necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn't connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

Here are some examples of legal instantiations that will and will not cause the warning message.

Module definition:

```
module foo (a, b, c, d);
```

Instantiation that does not connect all pins but will not produce the warning:

```
foo inst1(e, f, g, ); // positional association
foo inst1(.a(e), .b(f), .c(g), .d()); // named association
```

Instantiation that does not connect all pins but will produce the warning:

```
foo inst1(e, f, g); // positional association
foo inst1(.a(e), .b(f), .c(g)); // named association
```

Any instantiation above will leave pin *d* unconnected but the first example has a placeholder for the connection. Here's another example:

```
foo inst1(e, , g, h);
```

```
foo inst1(.a(e), .b(), .c(g), .d(h));
```

- Suggested actions —
 - Check that there is not an extra comma at the end of the port list. (for example, `model(a,b,)`). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
 - If you are purposefully leaving pins unconnected, you can disable these messages using the `+nowarnTFMPC` argument to `vsim`.

VSIM license lost

```
Console output:  
Signal 0 caught... Closing vsim vlm child.  
vsim is exiting with code 4  
FATAL ERROR in license manager
```

```
transcript/vsim output:  
# ** Error: VSIM license lost; attempting to re-establish.  
#   Time: 5027 ns   Iteration: 2  
# ** Fatal: Unable to kill and restart license process.  
#   Time: 5027 ns   Iteration: 2
```

- Description — ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.
- Suggested action — Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

Failed to find libswift entry

```
** Error: Failed to find LMC Smartmodel libswift entry in project file.  
# Fatal: Foreign module requested halt
```

- Description — ModelSim could not locate the **libswift** entry and therefore could not link to the Logic Modeling library.
- Suggested action — Uncomment the appropriate **libswift** entry in the `[lmc]` section of the `modelsim.ini` or project `.mpf` file. See [VHDL SmartModel Interface](#) for more information.

Detecting Infinite Zero-Delay Loops

If you receive an error similar to:

```
# ** Error: (vsim-3601) Iteration limit reached at time 132025 ps.
```

it is because ModelSim has ended the simulation after 5000 iterations in a zero-delay oscillation. The [IterationLimit](#) `modelsim.ini` variable sets the number of iterations before issuing the error.

Follow these steps to find and debug the zero-delay loop causing the error.

1. Re-run `vopt` with `+acc` to open full visibility to the design.
2. Re-run `vsim` with `+autofindloop`.

This will produce the `vsim-3601` error again, but this time with information about zero-delay loops.

```
# ** Error: (vsim-3601) Iteration limit reached at time 132025 ps.  
# This is a zero-delay loop:  
#   /top/#ALWAYS#5      src/alcomb.sv:5  
#   /top/#ALWAYS#8      src/alcomb.sv:8  
#   /top/#ALWAYS#14     src/alcomb.sv:14
```

3. Use this information to debug any loops in your design. The following are potential coding techniques that lead to zero-delay loops:

- o A missing or incorrectly applied SDF annotation to a netlist.
- o An RTL design with an asynchronous feedback loop with no delays.
- o Processes without wait statements or sensitivity lists, for example:

```
a <= not b;  
b <= not a;
```

- o A long string of assignments in VHDL, such as:

```
a1 <= a0;  
a2 <= a1;  
...  
a500 <= a499;
```

sccom Error Messages

This section describes `sccom` error messages which may be associated with ModelSim.

Failed to load sc lib error: undefined symbol

```
# ** Error: (vsim-3197) Load of  
"/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so"  
failed:ld.so.1:  
/home/icds_nut/modelsim/5.8a/sunos5/vsimk:  
fatal: relocation error: file  
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so:  
symbol_Z28host_respond_to_vhdl_requestPm:  
referenced symbol not found.  
# ** Error: (vsim-3676) Could not load shared library  
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so  
for SystemC module 'host_xtor'.
```

- Description — The causes for such an error could be:

- o missing symbol definition
- o bad link order specified in `scom -link`
- o multiply defined symbols (see [Multiple Symbol Definitions](#))

1. Suggested action —

- o If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an external "C" function:

```
extern "C" void myFunc();
```

- o The order in which you place the **-link** option within the **scom -link** command is critical. Make sure you have used it appropriately. See [scom](#) for syntax and usage information. See [Misplaced -link Option](#) for further explanation of error and correction.

Multiply defined symbols

```
work/sc/gensrc/test_ringbuf.o: In function
    `test_ringbuf::clock_generator(void)':
work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
    `test_ringbuf::clock_generator(void)'
work/sc/test_ringbuf.o(.text+0x4): first defined here
```

- Meaning — The most common type of error found during **scom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one `.o` file. Several causes are likely:
 - o A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (that is, not just referenced or prototyped, but truly defined) in a `.h` file, you can't include that `.h` file in more than one `.cpp` file.
 - o Another cause of errors is due to ModelSim's name association feature. The name association feature automatically generates `.cpp` files in the work library. These files "include" your header files. Thus, while it might appear as though you have included your header file in only one `.cpp` file, from the linker's point of view, it is included in multiple `.cpp` files.
- Suggested action — Make sure you don't have any out-of-line functions. Use the "inline" keyword. See [Multiple Symbol Definitions](#).

Enforcing Strict 1076 Compliance

The optional **-pedanticerrors** argument to `vcom` enforces strict compliance to the IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual (LRM) in the cases listed below. The default behavior for these cases is to issue an insuppressible warning message. If

you compile with **-pedanticerrors**, the warnings change to an error, unless otherwise noted. Descriptions in quotes are actual warning/error messages emitted by **vcom**. As noted, in some cases you can suppress the warning using **-nowarn [level]**.

- Type conversion between array types, where the element subtypes of the arrays do not have identical constraints.
- "Extended identifier terminates at newline character (0xa)."
- "Extended identifier contains non-graphic character 0x%x."
- "Extended identifier \"%s\" contains no graphic characters."
- "Extended identifier \"%s\" did not terminate with backslash character."
- "An abstract literal and an identifier must have a separator between them."

This is for forming physical literals, which comprise an optional numeric literal, followed by a separator, followed by an identifier (the unit name). Warning is level 4, which means "-nowarn 4" will suppress it.

- In VHDL 1993 or 2002, a subprogram parameter was declared using VHDL 1987 syntax (which means that it was a class VARIABLE parameter of a file type, which is the only way to do it in VHDL 1987 and is illegal in later VHDLs). Warning is level 10.
- "Shared variables must be of a protected type." Applies to VHDL 2002 only.
- Expressions evaluated during elaboration cannot depend on signal values. Warning is level 9.
- "Non-standard use of output port '%s' in PSL expression." Warning is level 11.
- "Non-standard use of linkage port '%s' in PSL expression." Warning is level 11.
- Type mark of type conversion expression must be a named type or subtype, it can't have a constraint on it.
- When the actual in a PORT MAP association is an expression, it must be a (globally) static expression. The port must also be of mode IN.
- The expression in the CASE and selected signal assignment statements must follow the rules given in Section 8.8 of the IEEE Std 1076-2002. In certain cases we can relax these rules, but **-pedanticerrors** forces strict compliance.
- A CASE choice expression must be a locally static expression. We allow it to be only globally static, but **-pedanticerrors** will check that it is locally static. Same rule for selected signal assignment statement choices. Warning level is 8.
- When making a default binding for a component instantiation, ModelSim's non-standard search rules found a matching entity. Section 5.2.2 of the IEEE Std 1076-2002 describes the standard search rules. Warning level is 1.

- Both FOR GENERATE and IF GENERATE expressions must be globally static. We allow non-static expressions unless **-pedanticerrors** is present.
- When the actual part of an association element is in the form of a conversion function call [or a type conversion], and the formal is of an unconstrained array type, the return type of the conversion function [type mark of the type conversion] must be of a constrained array subtype. We relax this (with a warning) unless **-pedanticerrors** is present when it becomes an error.
- OTHERS choice in a record aggregate must refer to at least one record element.
- In an array aggregate of an array type whose element subtype is itself an array, all expressions in the array aggregate must have the same index constraint, which is the element's index constraint. No warning is issued; the presence of **-pedanticerrors** will produce an error.
- Non-static choice in an array aggregate must be the only choice in the only element association of the aggregate.
- The range constraint of a scalar subtype indication must have bounds both of the same type as the type mark of the subtype indication.
- The index constraint of an array subtype indication must have index ranges each of whose both bounds must be of the same type as the corresponding index subtype.
- When compiling VHDL 1987, various VHDL 1993 and 2002 syntax is allowed. Use **-pedanticerrors** to force strict compliance. Warnings are all level 10.
- For a FUNCTION having a return type mark that denotes a constrained array subtype, a RETURN statement expression must evaluate to an array value with the same index range(s) and direction(s) as that type mark. This language requirement (Section 8.12 of the IEEE Std 1076-2002) has been relaxed such that ModelSim displays only a compiler warning and then performs an implicit subtype conversion at run time.

To enforce the prior compiler behavior, use `vcom -pedanticerrors`.

Appendix D

Verilog Interfaces to C

This appendix describes the ModelSim implementation of the Verilog interfaces:

- Verilog PLI (Programming Language Interface)
- VPI (Verilog Procedural Interface)
- SystemVerilog DPI (Direct Programming Interface).

These three interfaces provide a mechanism for defining 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](#)). In addition, you may write your own PLI/VPI/DPI applications.

Implementation Information

This chapter describes only the details of using the PLI/VPI/DPI with ModelSim Verilog and SystemVerilog.

- ModelSim SystemVerilog implements DPI as defined in the IEEE Std 1800-2005.
- The PLI implementation (TF and ACC routines) as defined in IEEE Std 1364-2001 is retained for legacy PLI applications. However, this interface was deprecated in IEEE Std 1364-2005 and subsequent IEEE Std 1800-2009 (SystemVerilog) standards. New applications should not rely on this functionality being present and should instead use the VPI.
- VPI Implementation — The VPI is partially implemented as defined in the IEEE Std 1364-2005 and IEEE Std 1800-2005. The list of currently supported functionality can be found in the following file:

```
<install_dir>/docs/technotes/Verilog_VPI.note
```

The simulator allows you to specify whether it runs in a way compatible with the IEEE Std 1364-2001 object model or the combined IEEE Std 1364-2005/IEEE Std 1800-2005 object models. By default, the simulator uses the combined 2005 object models. This control is accessed through the `vsim -plicompatdefault` switch or the `PliCompatDefault` variable in the `modelsim.ini` file.

The following table outlines information you should know about when performing a simulation with VPI and HDL files using the two different object models.

Table D-1. VPI Compatibility Considerations

Simulator Compatibility: -plicompatdefault	VPI Files	HDL Files	Notes
2001	2001	2001	When your VPI and HDL are written based on the 2001 standard, be sure to specify, as an argument to vsim, “-plicompatdefault 2001”.
2005	2005	2005	When your VPI and HDL are written based on the 2005 standard, you do not need to specify any additional information to vsim because this is the default behavior
2001	2001	2005	New SystemVerilog objects in the HDL will be completely invisible to the application. This may be problematic, for example, for a delay calculator, which will not see SystemVerilog objects with delay on a net.
2001	2005	2001	It is possible to write a 2005 VPI that is backwards-compatible with 2001 behavior by using mode-neutral techniques. The simulator will reject 2005 requests if it is running in 2001 mode, so there may be VPI failures.
2001	2005	2005	You should only use this setup if there are other VPI libraries in use for which it is absolutely necessary to run the simulator in 2001-mode. This combination is not recommended when the simulator is capable of supporting the 2005 constructs.
2005	2001	2001	This combination is not recommended. You should change the -plicompatdefault argument to 2001.
2005	2001	2005	This combination is most likely to result in errors generated from the VPI as it encounters objects in the HDL that it does not understand.
2005	2005	2001	This combination should function without issues, as SystemVerilog is a superset of Verilog. All that is happening here is that the HDL design is not using the full subset of objects that both the simulator and VPI ought to be able to handle.

GCC Compiler Support for use with C Interfaces

You must acquire the gcc/g++ compiler for your given platform as defined in the sections [Compiling and Linking C Applications for Interfaces](#) and [Compiling and Linking C++ Applications for Interfaces](#).

Registering PLI Applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, ModelSim Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of `s_tfcell` structures. This structure is declared in the `veriusers.h` include file as follows:

```
typedef int (*p_tffn)();
typedef struct t_tfcell {
    short type; /* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data; /* passed as data argument of callback function */
    p_tffn checktf; /* argument checking callback function */
    p_tffn sizetf; /* function return size callback function */
    p_tffn calltf; /* task or function call callback function */
    p_tffn misctf; /* miscellaneous reason callback function */
    char *tfname; /* name of system task or function */
    /* The following fields are ignored by ModelSim Verilog */
    int forwref;
    char *tfveritool;
    char *tferrmessage;
    int hash;
    struct t_tfcell *left_p;
    struct t_tfcell *right_p;
    char *namecell_p;
    int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (`checktf`, `sizetf`, `calltf`, and `misctf`) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the `calltf` function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the `data` field (many PLI applications don't use this field). The `type` field defines the entry as either a system task (`USERTASK`) or a system function that returns either a register (`USERFUNCTION`) or a real (`USERREALFUNCTION`). The `tfname` field is the system task or function name (it must begin with \$). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an `init_usertfs` function, and then a `veriusertfs` array. If `init_usertfs` is found, the simulator calls that function so that it can call `mti_RegisterUserTF()` for each system task or function defined. The `mti_RegisterUserTF()` function is declared in `veriusers.h` as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
    {0} /* last entry must be 0 */
};
```

Alternatively, you can add an init_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if a PLI shared library does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see [Compiling and Linking C Applications for Interfaces](#)). The PLI applications are specified as follows (note that on a Windows platform the file extension would be *.dll*):

- As a list in the Veriuser entry in the *modelsim.ini* file:
Veriuser = pliapp1.so pliapp2.so pliappn.so
- As a list in the PLIOBJS environment variable:
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
- As a -pli argument to the simulator (multiple arguments are allowed):
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so

The various methods of specifying PLI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

Registering VPI Applications

Each VPI application must register its system tasks and functions and its callbacks with the simulator. To accomplish this, one or more user-created registration routines must be called at simulation startup. Each registration routine should make one or more calls to `vpi_register_systf()` to register user-defined system tasks and functions and `vpi_register_cb()` to register callbacks. The registration routines must be placed in a table named `vlog_startup_routines` so that the simulator can find them. The table must be terminated with a 0 entry.

Example D-1. VPI Application Registration

```
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }
PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }
void RegisterMySystfs( void )
{
    vpiHandle tmpH;
    s_cb_data callback;
    s_vpi_systf_data systf_data;

    systf_data.type          = vpiSysFunc;
    systf_data.sysfunctype  = vpiSizedFunc;
    systf_data.tfname       = "$myfunc";
    systf_data.calltf       = MyFuncCalltf;
    systf_data.compiletf    = MyFuncCompiletf;
    systf_data.sizetf       = MyFuncSizetf;
    systf_data.user_data    = 0;
    tmpH = vpi_register_systf( &systf_data );
    vpi_free_object(tmpH);

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

    callback.reason         = cbStartOfSimulation;
    callback.cb_rtn         = MyStartOfSimCB;
    callback.user_data      = 0;
    tmpH = vpi_register_cb( &callback );
    vpi_free_object(tmpH);
}
```

```
void (*vlog_startup_routines[ ] ) () = {  
    RegisterMySystfs,  
    0 /* last entry must be 0 */  
};
```

Loading VPI applications into the simulator is the same as described in [Registering PLI Applications](#).

Using PLI and VPI Together

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.

Registering DPI Applications

DPI applications do not need to be registered. However, each DPI imported or exported task or function must be identified using SystemVerilog ‘import “DPI-C”’ or ‘export “DPI-C”’ syntax. Examples of the syntax follow:

```
export "DPI-C" task t1;  
task t1(input int i, output int o);  
.  
.  
.  
end task  
import "DPI-C" function void f1(input int i, output int o);
```

Your C code must provide imported functions or tasks. An imported task must return an int value, "1" indicating that it is returning due to a disable, or "0" indicating otherwise.

The default flow is to supply C/C++ files on the `vlog` command line. The `vlog` compiler will automatically compile the specified C/C++ files and prepare them for loading into the simulation. For example,

```
vlog dut.v imports.c  
vsim top -do <do_file>
```

Optionally, DPI C/C++ files can be compiled externally into a shared library. For example, third party IP models may be distributed in this way. The shared library may then be loaded into the simulator with either the command line option `-sv_lib <lib>` or `-sv_liblist <bootstrap_file>`. For example,

```
vlog dut.v
gcc -shared -Bsymbolic -o imports.so imports.c
vsim -sv_lib imports top -do <do_file>
```

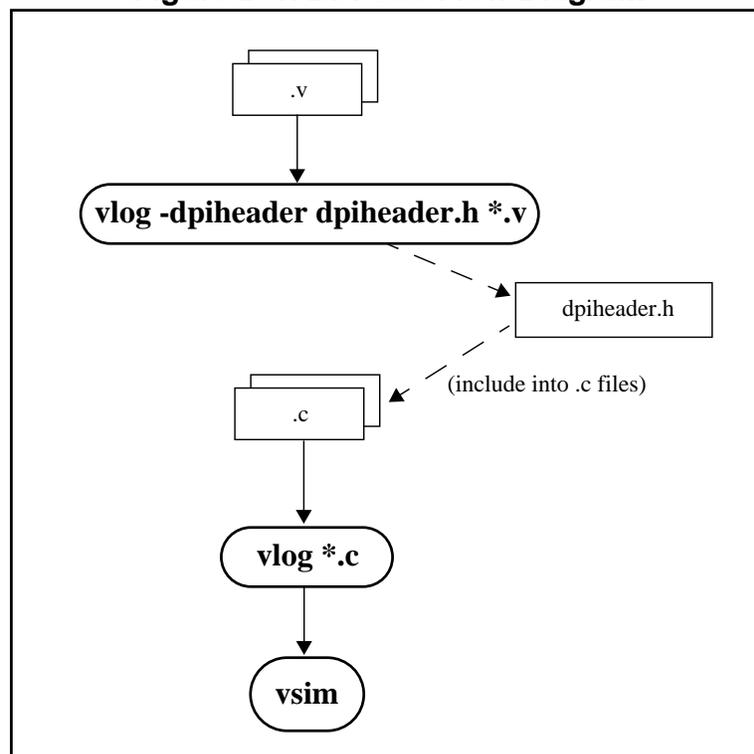
The `-sv_lib` option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see [DPI File Loading](#).

You can also use the command line options `-sv_root` and `-sv_liblist` to control the process for loading imported functions and tasks. These options are defined in the IEEE Std 1800-2005.

DPI Use Flow

Correct use of ModelSim DPI depends on the flow presented in this section.

Figure D-1. DPI Use Flow Diagram



1. Run `vlog` to generate a `dpiheader.h` file.

This file defines the interface between C and ModelSim for exported and imported tasks and functions. Though the `dpiheader.h` is a user convenience file rather than a requirement, including `dpiheader.h` in your C code can immediately solve problems

caused by an improperly defined interface. An example command for creating the header file would be:

```
vlog -dpiheader dpiheader.h files.v
```

2. Include the *dpiheader.h* file in your C code.

ModelSim recommends that any user DPI C code that accesses exported tasks/functions, or defines imported tasks/functions, should include the *dpiheader.h* file. This allows the C compiler to verify the interface between C and ModelSim.

3. Compile the C code using vlog. For example:

```
vlog *.c
```

4. Simulate the design. For example

```
vsim top
```

DPI and the vlog Command

You can specify C/C++ files on the [vlog](#) command line, and the command will invoke the correct C/C++ compiler based on the file type passed. For example, you can enter the following command:

```
vlog verilog1.v verilog2.v mydpcode.c
```

This command compiles all Verilog files and C/C++ files into the work library. The vsim command automatically loads the compiled C code at elaboration time.

It is possible to pass custom C compiler flags to vlog using the **-ccflags** option. vlog does not check the validity of option(s) you specify with -ccflags. The options are directly passed on to the compiler, and if they are not valid, an error message is generated by the C compiler.

You can also specify C/C++ files and options in a **-f** file, and they will be processed the same way as Verilog files and options in a **-f** file.

It is also possible to pass custom C/C++ linker flags to vsim using the **-ldflags** option. For example,

```
vsim top -ldflags '-lcrypt'
```

This command tells vsim to pass -lcrypt to the GCC linker.

The [qverilog command](#) also accepts C/C++ files on the command line. It works similarly to vlog, but automatically invokes vsim at the end of compilation.

Deprecated Legacy DPI Flows

Legacy use flows may be in use for certain designs from previous versions of ModelSim. These customized flows may have involved use of `-dpiexportobj`, `-dpiexportonly`, or `-nodpiexports`, and may have been employed for the following scenarios:

- runtime work library locked
- running parallel vsim simulations on the same design (distributed vsim simulation)
- complex dependency between FLI/PLI/SystemC and DPI

None of the former special handling is required for these scenarios as of version 10.0d and above. The recommended use flow is as documented in “[DPI Use Flow](#)”.

Platform Specific Information

On Windows, complex flows involving DPI combined with PLI or SystemC require special handling. Contact Customer Support for further assistance.

When Your DPI Export Function is Not Getting Called

This issue can arise in your C code due to the way the C linker resolves symbols. It happens if a name you choose for a SystemVerilog export function happens to match a function name in a custom, or even standard C library (for example, “pow”). In this case, your C compiler will bind calls to the function in that C library, rather than to the export function in the SystemVerilog simulator.

The symptoms of such a misbinding can be difficult to detect. Generally, the misbound function silently returns an unexpected or incorrect value.

To determine if you have this type of name aliasing problem, consult the C library documentation (either the online help or man pages) and look for function names that match any of your export function names. You should also review any other shared objects linked into your simulation and look for name aliases there. To get a comprehensive list of your export functions, you can use the vsim `-dpiheader` option and review the generated header file.

If you are using an external compilation flow, make sure to use `-Bsymbolic` on the GCC link line. For more information, see “[Correct Linking of Shared Libraries with -Bsymbolic](#)”.

Troubleshooting a Missing DPI Import Function

DPI uses C function linkage. If your DPI application is written in C++, it is important to remember to use extern “C” declaration syntax appropriately. Otherwise the C++ compiler will produce a mangled C++ name for the function, and the simulator is not able to locate and bind the DPI call to that function.

Also, if you do not use the **-Bsymbolic** argument on the command line for specifying a link, the system may bind to an incorrect function, resulting in unexpected behavior. For more information, see [Correct Linking of Shared Libraries with -Bsymbolic](#).

Simplified Import of Library Functions

In addition to the traditional method of importing FLI, PLI, and C library functions, a simplified method can be used: you can declare VPI and FLI functions as DPI-C imports. When you declare VPI and FLI functions as DPI-C imports, the C implementation of the import is not required.

Also, on most platforms (see [Platform Specific Information](#)), you can declare most standard C library functions as DPI-C imports.

The following example is processed directly, without DPI C code:

```
package cmath;
    import "DPI-C" function real sin(input real x);
    import "DPI-C" function real sqrt(input real x);
endpackage

package fli;
    import "DPI-C" function mti_Cmd(input string cmd);
endpackage

module top;
    import cmath::*;
    import fli::*;
    int status, A;
    initial begin
        $display("sin(0.98) = %f", sin(0.98));
        $display("sqrt(0.98) = %f", sqrt(0.98));
        status = mti_Cmd("change A 123");
        $display("A = %1d, status = %1d", A, status);
    end
endmodule
```

To simulate, you would simply enter a command such as: **vsim top**.

Precompiled packages are available with that contain import declarations for certain commonly used C calls.

```
<installDir>/verilog_src/dpi_cpack/dpi_cpackages.sv
```

You do not need to compile this file, it is automatically available as a built-in part of the SystemVerilog simulator.

Platform Specific Information

On Windows, only FLI and PLI commands may be imported in this fashion. C library functions are not automatically importable. They must be wrapped in user DPI C functions.

Optimizing DPI Import Call Performance

You can optimize the passing of some array data types across SystemVerilog/SystemC language boundary. Most of the overhead associated with argument passing is eliminated if the following conditions are met:

- DPI import is declared as a DPI-C function, not a task.
- DPI function port mode is input or inout.
- DPI calls are not hierarchical. The actual function call argument must not make use of hierarchical identifiers.
- For actual array arguments and return values, do not use literal values or concatenation expressions. Instead, use explicit variables of the same datatype as the formal array arguments or return type.
- DPI formal arguments can be either fixed-size or open array. They can use the element types `int`, `shortint`, `byte`, or `longint`.

Fixed-size array arguments — declaration of the actual array and the formal array must match in both direction and size of the dimension. For example: `int_formal[2:0]` and `int_actual[4:2]` match and are qualified for optimization. `int_formal[2:0]` and `int_actual[2:4]` do not match and will not be optimized.

Open-array arguments — Actual arguments can be either fixed-size arrays or dynamic arrays. The topmost array dimension should be the only dimension considered open. All lower dimensions should be fixed-size subarrays or scalars. High performance actual arguments: `int_arr1[10]`, `int_arr2[]`, `int_arr3[][2]` `int_arr4[][2][2]`. A low performance actual argument would be `slow_arr[2][][2]`.

DPI Arguments of Parameterized Datatypes

DPI import and export TF's can be written with arguments of parameterized data types. For example, assuming T1 and T2 are type parameters:

```
import "DPI-C" function T1 impf(input T2 arg);
```

This feature is only supported when the **vopt** flow is used (see [Optimizing Designs with vopt](#)). On occasion, the tool may not be able to resolve type parameters while building the optimized design, in which case the workaround is to rewrite the function without using parameterized types. The LRM rules for tf signature matching apply to the finally resolved value of type parameters. See the IEEE Std 1800-2005, Section 26.4.4 for further information on matching rules.

Making Verilog Function Calls from non-DPI C Models

Working in certain FLI or PLI C applications, you might want to interact with the simulator by directly calling Verilog DPI export functions. Such applications may include complex 3rd party integrations, or multi-threaded C test benches. Normally calls to export functions from PLI or FLI code are illegal. These calls are referred to as out-of-the-blue calls, since they do not originate in the controlled environment of a DPI import tf.

You can configure the ModelSim tool to allow out-of-the-blue Verilog function calls either for all simulations (`DpiOutOfTheBlue = 1` in `modelsim.ini` file), or for a specific simulation (`vsim -dpioutoftheblue 1`). See `DpiOutOfTheBlue` for information about debugging support for a SystemC method or a SystemC thread.

The following is an example in which PLI code calls a SystemVerilog export function:

```
vlog test.sv
gcc -shared -o pli.so pli.c
vsim -pli pli.so top -dpioutoftheblue 1
```

Here is an example in which SystemC calls a SystemVerilog export function:

```
vlog test.sv
sccom test.cpp
sccom -link
vsim top sc_top
```

No `-dpioutoftheblue` specification is required for SystemC calls.

One restriction applies: only Verilog functions may be called out-of-the-blue. It is illegal to call Verilog tasks in this way. The simulator issues an error if it detects such a call.

Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code

In some instances you may need to share C/C++ code across different shared objects that contain PLI and/or DPI code. There are two ways you can achieve this goal:

- The easiest is to include the shared code in an object containing PLI code, and then make use of the `vsim -gblso` option.
- Another way is to define a standalone shared object that only contains shared function definitions, and load that using `vsim -gblso`. In this case, the process does not require PLI or DPI loading mechanisms, such as `-pli` or `-sv_lib`.

You should also take into consideration what happens when code in one global shared object needs to call code in another global shared object. In this case, place the `-gblso` argument for the calling code on the `vsim` command line *after* you place the `-gblso` argument for the called code. This is because `vsim` loads the files in the specified order and you must load called code before calling code in all cases.

Circular references aren't possible to achieve. If you have that kind of condition, you are better off combining the two shared objects into a single one.

For more information about this topic please refer to the section "[Loading Shared Objects with Global Symbol Visibility](#)."

Compiling and Linking C Applications for Interfaces

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C applications so that they can be loaded by ModelSim. Various native C/C++ compilers are supported on different platforms. The gcc compiler is supported on all platforms.

The following PLI/VPI/DPI routines are declared in the include files located in the ModelSim `<install_dir>/include` directory:

- `acc_user.h` — declares the ACC routines
- `veriusers.h` — declares the TF routines
- `vpi_user.h` — declares the VPI routines
- `svdpi.h` — declares DPI routines

The following instructions assume that the PLI, VPI, or DPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries for PLI/VPI see [PLI and VPI File Loading](#). For DPI loading instructions, see [DPI File Loading](#).

For all UNIX Platforms

The information in this section applies to all UNIX platforms.

app.so

If `app.so` is not in your current directory, you must tell the OS where to search for the shared object. You can do this one of two ways:

- Add a path before `app.so` in the command line option or control variable (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
`LD_LIBRARY_PATH_32= <library path without filename> (for 32-bit)`

or

`LD_LIBRARY_PATH_64= <library path without filename> (for 64-bit)`

Correct Linking of Shared Libraries with **-Bsymbolic**

In the examples shown throughout this appendix, the **-Bsymbolic** linker option is used with the compilation (**gcc** or **g++**) or link (**ld**) commands to correctly resolve symbols. This option instructs the linker to search for the symbol within the local shared library and bind to that symbol if it exists. If the symbol is not found within the library, the linker searches for the symbol within the vsimk executable and binds to that symbol, if it exists.

When using the **-Bsymbolic** option, the linker may warn about symbol references that are not resolved within the local shared library. It is safe to ignore these warnings, provided the symbols are present in other shared libraries or the vsimk executable. (An example of such a warning would be a reference to a common API call such as `vpi_printf()`).

Windows Platforms — C

- Microsoft Visual Studio 2008

Refer to the section “[Creating .dll or .exe Files using Compiled .lib files](#)” in the *Installation and Licensing Guide* for information on using Microsoft Visual Studio 2008.

For 32-bit:

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

For 64-bit:

```
cl -c -I<install_dir>\modeltech\include app.c  
link -dll -export:<init_function> app.obj <install_dir>\win64\mtipli.lib -out:app.dll
```

For the Verilog PLI, the `<init_function>` should be `"init_usertfs"`. Alternatively, if there is no `init_usertfs` function, the `<init_function>` specified on the command line should be `"veriusertfs"`. For the Verilog VPI, the `<init_function>` should be `"vlog_startup_routines"`. These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

If you need to run the profiler (see [Profiling Performance and Memory Use](#)) on a design that contains PLI/VPI code, add these two switches to the link commands shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the `.dll` that the profiler can use in its report.

If you have Cygwin installed, make sure that the Cygwin `link.exe` executable is not in your search path ahead of the Microsoft Visual Studio 2008 `link` executable. If you mistakenly bind your dll's with the Cygwin `link.exe` executable, the `.dll` will not function

properly. It may be best to rename or remove the Cygwin *link.exe* file to permanently avoid this scenario.

- MinGW

For 32-bit:

```
gcc -c -I<install_dir>\include app.c
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win32 -lmtipli
```

For 64-bit:

```
gcc -c -I<install_dir>\include app.c
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win64 -lmtipli
```

The ModelSim tool requires the use of the MinGW gcc compiler rather than the Cygwin gcc compiler. Remember to add the path to your gcc executable in the Windows environment variables. Refer to [SystemC Supported Platforms](#) for more information.

32-bit Linux Platform — C

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify `-lc` to the `ld` command.

- GNU C Compiler version gcc 3.2 or later

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

If you are using ModelSim with RedHat version 7.1 or below, you also need to add the **-noinherit-exec** switch when you specify **-Bsymbolic**.

The compiler switch **-freg-struct-return** must be used when compiling any FLI application code that contains foreign functions that return real or time values.

64-bit Linux Platform — C

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- GNU C Compiler version gcc 3.2 or later

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

To compile for 32-bit operation, specify the `-m32` argument on both the compile and link gcc command lines.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library *libm*, specify `-lm` to the `ld` command:

```
gcc -c -fPIC -I<install_dir>/modeltech/include math_app.c  
gcc -shared -Bsymbolic -o math_app.so math_app.o -lm
```

Compiling and Linking C++ Applications for Interfaces

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

Since ModelSim's PLI/VPI/DPI functions have a standard C prototype, you must prevent the C++ compiler from mangling the PLI/VPI/DPI function names. This can be accomplished by using the following type of extern:

```
extern "C"  
{  
  <PLI/VPI/DPI application function prototypes>  
}
```

The header files *veriususer.h*, *acc_user.h*, and *vpi_user.h*, *svdpi.h*, and *dpiheader.h* already include this type of extern. You must also put the PLI/VPI/DPI shared library entry point (*veriusertfs*, *init_usertfs*, or *vlog_startup_routines*) inside of this type of extern.

You must also place an ‘extern “C”’ declaration immediately before the body of every import function in your C++ source code, for example:

```
extern "C"  
int myimport(int i)  
{  
    vpi_printf("The value of i is %d\n", i);  
}
```

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C++ applications so that they can be loaded by ModelSim.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see [DPI File Loading](#).

For PLI/VPI only

If *app.so* is not in your current directory you must tell Linux where to search for the shared object. You can do this one of two ways:

- Add a path before *app.so* in the foreign attribute specification. (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
LD_LIBRARY_PATH_32= <library path without filename> (32-bit)

or
 LD_LIBRARY_PATH_64= <library path without filename> (64-bit)

Windows Platforms — C++

- Microsoft Visual Studio 2008

Refer to the section “[Creating .dll or .exe Files using Compiled .lib files](#)” in the *Installation and Licensing Guide* for information on using Microsoft Visual Studio 2008.

For 32-bit:

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj
                        <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

For 64-bit:

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj
                        <install_dir>\modeltech\win64\mtipli.lib /out:app.dll
```

The **-GX** argument enables exception handling.

For the Verilog PLI, the **<init_function>** should be "init_usertfs". Alternatively, if there is no init_usertfs function, the **<init_function>** specified on the command line should be "veriusertfs". For the Verilog VPI, the **<init_function>** should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

If you need to run the profiler (see [Profiling Performance and Memory Use](#)) on a design that contains PLI/VPI code, add these two switches to the link command shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

If you have Cygwin installed, make sure that the Cygwin *link.exe* executable is not in your search path ahead of the Microsoft Visual C *link* executable. If you mistakenly bind your dll's with the Cygwin *link.exe* executable, the *.dll* will not function properly. It may be best to rename or remove the Cygwin *link.exe* file to permanently avoid this scenario.

- MinGW

For 32-bit:

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -Bsymbolic -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
```

For 64-bit:

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -Bsymbolic -o app.dll app.o -L<install_dir>\modeltech\win64 -lmtipli
```

ModelSim requires the use of the MinGW gcc compiler rather than the Cygwin gcc compiler.

32-bit Linux Platform — C++

- GNU C++ Version 2.95.3 or Later

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp
g++ -shared -Bsymbolic -fPIC -o app.so app.o
```

64-bit Linux Platform — C++

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp
g++ -shared -Bsymbolic -o app.so app.o
```

To compile for 32-bit operation, specify the `-m32` argument on both the `g++` compiler command line as well as the `g++ -shared` linker command line.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library `libm`, specify `-lm` to the `ld` command:

```
g++ -c -fPIC -I<install_dir>/modeltech/include math_app.cpp
g++ -shared -Bsymbolic -o math_app.so math_app.o -lm
```

Specifying Application Files to Load

PLI and VPI file loading is identical. DPI file loading uses switches to the `vsim` command.

PLI and VPI File Loading

The PLI/VPI applications are specified as follows:

- As a list in the Veriuser entry in the `modelsim.ini` file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```
- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```
- As a `-pli` argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

Note



On Windows platforms, the file names shown above should end with *.dll* rather than *.so*.

The various methods of specifying PLI/VPI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

See also “[modelsim.ini Variables](#)” for more information on the *modelsim.ini* file.

DPI File Loading

This section applies only to external compilation flows. It is not necessary to use any of these options in the default autocompile flow (using *vlog* to compile). DPI applications are specified to *vsim* using the following SystemVerilog arguments:

Table D-2. vsim Arguments for DPI Application Using External Compilation Flows

Argument	Description
<code>-sv_lib <name></code>	specifies a library name to be searched and used. No filename extensions must be specified. (The extensions ModelSim expects are: <i>.dll</i> for Win32/Win64, <i>.so</i> for all other platforms.)
<code>-sv_root <name></code>	specifies a new prefix for shared objects as specified by <code>-sv_lib</code>
<code>-sv_liblist <bootstrap_file></code>	specifies a “bootstrap file” to use. See The format for <code><bootstrap_file></code> is as follows: <pre>#!SV_LIBRARIES <path>/<to>/<shared>/<library> <path>/<to>/<another> ...</pre> No extension is expected on the shared library.

When the simulator finds an imported task or function, it searches for the symbol in the collection of shared objects specified using these arguments.

For example, you can specify the DPI application as follows:

```
vsim -sv_lib dpiapp1 -sv_lib dpiapp2 -sv_lib dpiappn top
```

It is a mistake to specify DPI import tasks and functions (tf) inside PLI/VPI shared objects. However, a DPI import tf can make calls to PLI/VPI C code, providing that **vsim -gblso** was used to mark the PLI/VPI shared object with global symbol visibility. See [Loading Shared Objects with Global Symbol Visibility](#).

Loading Shared Objects with Global Symbol Visibility

On Unix platforms you can load shared objects such that all symbols in the object have global visibility. To do this, use the **-gblso** argument to **vsim** when you load your PLI/VPI application. For example:

```
vsim -pli obj1.so -pli obj2.so -gblso obj1.so top
```

The **-gblso** argument works in conjunction with the `GlobalSharedObjectList` variable in the `modelsim.ini` file. This variable allows user C code in other shared objects to refer to symbols in a shared object that has been marked as global. All shared objects marked as global are loaded by the simulator earlier than any non-global shared objects.

PLI Example

The following example shows a small but complete PLI application for Linux.

```
hello.c:
#include "veriusertfs.h"
static PLI_INT32 hello()
{
    io_printf("Hi there\n");
    return 0;
}
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, hello, 0, "$hello"},
    {0} /* last entry must be 0 */
};
hello.v:
module hello;
    initial $hello;
endmodule
Compile the PLI code for a 32-bit Linux Platform:
% gcc -c -I <install_dir>/questasim/include hello.c
% gcc -shared -Bsymbolic -o hello.so hello.o -lc
Compile the Verilog code:
% vlib work
% vlog hello.v
Simulate the design:
vsim -c -pli hello.so hello
# Loading ./hello.so
```

VPI Example

The following example is a trivial, but complete VPI application. A general VPI example can be found in `<install_dir>/modeltech/examples/verilog/vpi`.

```
hello.c:
```

```

#include "vpi_user.h"
static PLI_INT32 hello(PLI_BYTE8 * param)
{
    vpi_printf( "Hello world!\n" );
    return 0;
}
void RegisterMyTfs( void )
{
    s_vpi_systf_data systf_data;
    vpiHandle systf_handle;
    systf_data.type          = vpiSysTask;
    systf_data.sysfunctype  = vpiSysTask;
    systf_data.tfname       = "$hello";
    systf_data.calltf       = hello;
    systf_data.compiletf    = 0;
    systf_data.sizetf       = 0;
    systf_data.user_data    = 0;
    systf_handle = vpi_register_systf( &systf_data );
    vpi_free_object( systf_handle );
}
void (*vlog_startup_routines[])() = {
    RegisterMyTfs,
    0
};
hello.v:
module hello;
    initial $hello;
endmodule
Compile the 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

```

DPI Example

The following example is a trivial but complete DPI application. For win32 platforms, an additional step is required. For additional examples, see the `<install_dir>/modeltech/examples/systemverilog/dpi` directory.

```

hello_c.c:
#include "svdpi.h"
#include "dpiheader.h"
int c_task(int i, int *o)
{
    printf("Hello from c_task()\n");
    verilog_task(i, o); /* Call back into Verilog */
    *o = i;
    return(0); /* Return success (required by tasks) */
}

```

```
hello.v:
module hello_top;
  int ret;
  export "DPI-C" task verilog_task;
  task verilog_task(input int i, output int o);
    #10;
    $display("Hello from verilog_task()");
  endtask
  import "DPI-C" context task c_task(input int i, output int o);
  initial
  begin
    c_task(1, ret); // Call the c task named 'c_task()'
  end
endmodule
Compile the Verilog code:
% vlib work
% vlog -sv -dpiheader dpiheader.h hello.v hello_c.c
Simulate the design:
% vsim -c hello_top -do "run -all; quit -f"
# Loading work.hello_c
VSIM 1> run -all
# Hello from c_task()
# Hello from verilog_task()
VSIM 2> quit
```

The PLI Callback reason Argument

The second argument to a PLI callback function is the reason argument. The values of the various reason constants are defined in the *veriusers.h* include file. See the IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the *misctf* callback functions under the following circumstances:

`reason_endofcompile`

For the completion of loading the design.

`reason_finish`

For the execution of the `$finish` system task or the **quit** command.

`reason_startofsave`

For the start of execution of the **checkpoint** command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to `tf_write_save()` until it is called with `reason_save`.

`reason_save`

For the execution of the **checkpoint** command. This is when the PLI application must save its state with calls to `tf_write_save()`.

`reason_startofrestart`

For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to `tf_read_restart()` until it is called with

`reason_restart`. The `reason_startofrestart` value is passed only for a restore command, and not in the case that the simulator is invoked with `-restore`.

`reason_restart`

For the execution of the **restore** command. This is when the PLI application must restore its state with calls to `tf_read_restart()`.

`reason_reset`

For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be reloaded. (See the **-keeploaded** and **-keeploadedrestart** arguments to **vsim** for related information.)

`reason_endofreset`

For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`

For the execution of the `$stop` system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`

For the execution of the **environment** command or selecting a scope in the structure window. Also for the call to `acc_set_interactive_scope()` if the `callback_flag` argument is non-zero.

`reason_paramvc`

For the change of value on the system task or function argument.

`reason_synch`

For the end of time step event scheduled by `tf_synchronize()`.

`reason_rosynch`

For the end of time step event scheduled by `tf_rosynchronize()`.

`reason_reactivate`

For the simulation event scheduled by `tf_setdelay()`.

`reason_paramdrc`

Not supported in ModelSim Verilog.

`reason_force`

Not supported in ModelSim Verilog.

`reason_release`

Not supported in ModelSim Verilog.

`reason_disable`

Not supported in ModelSim Verilog.

The `sizetf` Callback Function

A user-defined system function specifies the width of its return value with the `sizetf` callback function, and the simulator calls this function while loading the design. The following details on the `sizetf` callback function are not found in the IEEE Std 1364:

- If you omit the `sizetf` function, then a return width of 32 is assumed.
- The `sizetf` function should return 0 if the system function return value is of Verilog type "real".
- The `sizetf` function should return -32 if the system function return value is of Verilog type "integer".

PLI Object Handles

Many of the object handles returned by the PLI ACC routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the `acc_close()` routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after `acc_close()` is called. The following object types are created on demand in ModelSim Verilog:

`accOperator` (`acc_handle_condition`)
`accWirePath` (`acc_handle_path`)
`accTerminal` (`acc_handle_terminal`, `acc_next_cell_load`, `acc_next_driver`, and `acc_next_load`)
`accPathTerminal` (`acc_next_input` and `acc_next_output`)
`accTchkTerminal` (`acc_handle_tchkarg1` and `acc_handle_tchkarg2`)
`accPartSelect` (`acc_handle_conn`, `acc_handle_pathin`, and `acc_handle_pathout`)

If your PLI application uses these types of objects, then it is important to call `acc_close()` to free the memory allocated for these objects when the application is done using them.

If your PLI application places value change callbacks on `accRegBit` or `accTerminal` objects, *do not* call `acc_close()` while these callbacks are in effect.

Third Party PLI Applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a `veriusser.c` file. The `veriusser.c` file contains the registration information as described above in [Registering PLI Applications](#). To prepare the application for ModelSim Verilog, you must compile the `veriusser.c` file and link it to the object files to create a dynamically loadable object (see [Compiling and Linking C Applications for Interfaces](#)). For example, if you have a `veriusser.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 Linux operating system:

```
% gcc -c -I<install_dir>/modeltech/include veriusers.c
% gcc -shared -Bsymbolic -o app.so veriusers.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the **Veriuser** entry in the *modesim.ini* file, the **-pli** simulator argument, or the PLIOBJS environment variable (see [Registering PLI Applications](#)).

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:

Table D-3. Supported VHDL Objects

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 ModelSim 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:

Table D-4. Supported ACC Routines

Routines		
acc_append_delays	acc_free	acc_next
acc_append_pulsere	acc_handle_by_name	acc_next_bit
acc_close	acc_handle_calling_mod_m	acc_next_cell
acc_collect	acc_handle_condition	acc_next_cell_load
acc_compare_handles	acc_handle_conn	acc_next_child
acc_configure	acc_handle_hiconn	acc_next_driver
acc_count	acc_handle_interactive_scope	acc_next_hiconn
acc_fetch_argc	acc_handle_loconn	acc_next_input
acc_fetch_argv	acc_handle_modpath	acc_next_load
acc_fetch_attribute	acc_handle_notifier	acc_next_loconn
acc_fetch_attribute_int	acc_handle_object	acc_next_modpath
acc_fetch_attribute_str	acc_handle_parent	acc_next_net
acc_fetch_defname	acc_handle_path	acc_next_output
acc_fetch_delay_mode	acc_handle_pathin	acc_next_parameter
acc_fetch_delays	acc_handle_pathout	acc_next_port
acc_fetch_direction	acc_handle_port	acc_next_portout
acc_fetch_edge	acc_handle_scope	acc_next_primitive
acc_fetch_fullname	acc_handle_simulated_net	acc_next_scope
acc_fetch_fulltype	acc_handle_tchk	acc_next_specparam
acc_fetch_index	acc_handle_tchkarg1	acc_next_tchk
acc_fetch_location	acc_handle_tchkarg2	acc_next_terminal
acc_fetch_name	acc_handle_terminal	acc_next_topmod
acc_fetch_paramtype	acc_handle_tfang	acc_object_in_typelist
acc_fetch_paramval	acc_handle_itfang	acc_object_of_type
acc_fetch_polarity	acc_handle_tfinst	acc_product_type
acc_fetch_precision	acc_initialize	acc_product_version
acc_fetch_pulsere		acc_release_object
acc_fetch_range		acc_replace_delays
acc_fetch_size		acc_replace_pulsere
acc_fetch_tfang		acc_reset_buffer
acc_fetch_itfang		acc_set_interactive_scope
acc_fetch_tfang_int		acc_set_pulsere
acc_fetch_itfang_int		acc_set_scope
acc_fetch_tfang_str		acc_set_value
acc_fetch_itfang_str		acc_vcl_add
acc_fetch_timescale_info		acc_vcl_delete
acc_fetch_type		acc_version
acc_fetch_type_str		
acc_fetch_value		

`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 (task and function) routines;

Table D-5. Supported TF Routines

Routines		
io_mcdprintf	tf_getrealtime	tf_scale_longdelay
io_printf	tf_igetrealtime	tf_scale_realdelay
mc_scan_plusargs	tf_gettime	tf_setdelay
tf_add_long	tf_igettime	tf_isetdelay
tf_asynchoff	tf_gettimeprecision	tf_setlongdelay
tf_iasynchoff	tf_igettimeprecision	tf_isetlongdelay
tf_asynchon	tf_gettimeunit	tf_setrealdelay
tf_iasynchon	tf_igettimeunit	tf_isetrealdelay
tf_clearalldelays	tf_getworkarea	tf_setworkarea
tf_iclearalldelays	tf_igetworkarea	tf_isetworkarea
tf_compare_long	tf_long_to_real	tf_sizep
tf_copypvc_flag	tf_longtime_tostr	tf_isizep
tf_icopypvc_flag	tf_message	tf_spname
tf_divide_long	tf_mipname	tf_ispname
tf_dofinish	tf_imipname	tf_strdelputp
tf_dostop	tf_movepvc_flag	tf_istrdelputp
tf_error	tf_imovepvc_flag	tf_strgetp
tf_evaluatep	tf_multiply_long	tf_istrgetp
tf_ievaluatep	tf_nodeinfo	tf_strgettime
tf_exprinfo	tf_inodeinfo	tf_strlongdelputp
tf_iexprinfo	tf_nump	tf_istrlongdelputp
tf_getcstringp	tf_inump	tf_strrealdelputp
tf_igetcstringp	tf_propagatep	tf_istrrealdelputp
tf_getinstance	tf_ipropagatep	tf_subtract_long
tf_getlongp	tf_putlongp	tf_synchronize
tf_igetlongp	tf_iputlongp	tf_isynchronize
tf_getlongtime	tf_putp	tf_testpvc_flag
tf_igetlongtime	tf_iputp	tf_itestpvc_flag
tf_getnextlongtime	tf_putrealp	tf_text
tf_getp	tf_iputrealp	tf_typep
tf_igetp	tf_read_restart	tf_itypep
tf_getpchange	tf_real_to_long	tf_unscale_longdelay
tf_igetpchange	tf_rosynchronize	tf_unscale_realdelay
tf_getrealp	tf_irosynchronize	tf_warning
tf_igetrealp		tf_write_save

SystemVerilog DPI Access Routines

ModelSim SystemVerilog supports all routines defined in the "svdpi.h" file defined in the IEEE Std 1800-2005.

Verilog-XL Compatible Routines

The following PLI routines are not defined in IEEE Std 1364, but ModelSim Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc_decompile_expr** routine. The **condition** argument must be a handle obtained from the **acc_handle_condition** routine. The value returned by **acc_decompile_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **\$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the **aof_hightime** argument.

64-bit Support for 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.

Using 64-bit ModelSim with 32-bit Applications

If you have 32-bit PLI/VPI/DPI applications and wish to use 64-bit ModelSim, you will need to port your code to 64 bits by moving from the ILP32 data model to the LP64 data model. We strongly recommend that you consult the 64-bit porting guides for Sun.

PLI/VPI Tracing

The foreign interface tracing feature is available for tracing PLI and VPI function calls. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files that can be used to replay what the foreign interface code did.

The Purpose of Tracing Files

The purpose of the logfile is to aid you in debugging PLI or VPI code. The primary purpose of the replay facility is to send the replay files to 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` with the `-trace_foreign` argument:

Syntax

```
vsim
  -trace_foreign <action> [-tag <name>]
```

Arguments

<action>

Can be either the value 1, 2, or 3. Specifies one of the following actions:

Table D-6. Values for action Argument

Value	Operation	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	writes all above files

-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 mistf routines), and Verilog VCL callbacks.

Checkpointing and Interface Code

The checkpoint feature in ModelSim captures the state of PLI/VPI/DPI code. See [The PLI Callback reason Argument](#) for reason arguments that apply to checkpoint/restore.

You can use the FLI interface `mti_AddDPISaveRestoreCB()` to save and restore the states of C code. This DPI checkpoint/restore support is limited to linux and linux_x86_64 platforms if there are active DPI threads at the time of creating simulation checkpoint.

You can find an example in the `<install_dir>/examples/systemverilog/dpi/checkpoint` directory.

Checkpointing Code that Works with Heap Memory

If checkpointing code that works with heap memory, use `mti_Malloc()` rather than raw `malloc()` or `new`. Any memory allocated with `mti_Malloc()` is guaranteed to be restored correctly. Any memory allocated with raw `malloc()` will not be restored correctly, and simulator crashes can result.

Debugging Interface Application Code

ModelSim offers the optional C Debug feature. This tool allows you to interactively debug SystemC/C/C++ source code with the open-source **`gdb`** debugger. See [C Debug](#) for details. If you don't have access to C Debug, continue reading for instructions on how to attach to an external C debugger.

In order to debug your PLI/VPI/DPI application code in a debugger, you must first:

1. Compile the application code with debugging information (using the **`-g`** option) and without optimizations (for example, don't use the **`-O`** option).
2. Load **`vsim`** into a debugger.

Even though **`vsim`** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **`vsimk`**, the simulation kernel where your application code is loaded (for example, "`ddd `which vsimk``"), or you can attach the debugger to an already running **`vsim`** process. In the second case, you must attach to the PID for **`vsimk`**, and you must specify the full path to the **`vsimk`** executable (for example, "`gdb <modelsim_install_directory>/sunos5/vsimk 1234`").

On Linux systems you can use either **`gdb`** or **`ddd`**.

3. Set an entry point using breakpoint.

Since initially the debugger recognizes only **vsim**'s PLI/VPI/DPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI/DPI function that is called by your application code. An easy way to set an entry point is to put a call to `acc_product_version()` as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run **vsim** with the usual arguments.

When the breakpoint is reached, the shared library containing your application code has been loaded.

4. In some debuggers, you must use the **share** command to load the application's symbols.

At this point all of the application's symbols should be visible. You can now set breakpoints in and single step through your application code.

Appendix E

Command and Keyboard Shortcuts

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Command Shortcuts

- You may abbreviate command syntax, with the following limitation: the minimum number of characters required to execute a command are those that make it unique. Note that new commands may disable existing shortcuts. For this reason, ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.
- You can enter multiple commands on one line if they are separated by semi-colons (;). For example:

```
vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
```

The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

```
vsim -c -do "run 20 ; simstats ; quit -f" top
```

You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

```
vsim -do "run 20 ; echo [simstats]; quit -f" -c top
```

Command History Shortcuts

You can review the simulator command history, or reuse previously entered commands with the following shortcuts at the ModelSim/VSIM prompt:!
<string>

Table E-1. Command History Shortcuts

Shortcut	Description
!!	repeats the last command
!n	repeats command number n; n is the VSIM prompt number (for example, for this prompt: VSIM 12>, n =12)

Table E-1. Command History Shortcuts (cont.)

Shortcut	Description
! <code><string></code>	shows a list of executed commands that start with <code><string></code> ; Use the up and down arrows to choose from the list
!abc	repeats the most recent command starting with "abc"
^xyz^ab^	replaces "xyz" in the last command with "ab"
up arrow and down arrow keys	scrolls through the command history
Ctrl-N (UNIX only)	scroll to the next command
Ctrl-P (UNIX only)	scroll to the previous command
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

Main and Source Window Mouse and Keyboard Shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all **Notepad** windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Table E-2. Mouse Shortcuts

Mouse - UNIX and Windows	Result
Click the left mouse button	relocate the cursor
Click and drag the left mouse button	select an area
Shift-click the left mouse button	extend selection
Double-click the left mouse button	select a word
Double-click and drag the left mouse button	select a group of words
Ctrl-click the left mouse button	move insertion cursor without changing the selection
Click the left mouse button on a previous ModelSim or VSIM prompt	copy and paste previous command string to current prompt
Click the middle mouse button	paste selection to the clipboard
Click and drag the middle mouse button	scroll the window

Table E-3. Keyboard Shortcuts

Keystrokes - UNIX and Windows	Result
Left Arrow Right Arrow	move cursor left or right one character
Ctrl + Left Arrow Ctrl + Right Arrow	move cursor left or right one word
Shift + Any Arrow	extend text selection
Ctrl + Shift + Left Arrow Ctrl + Shift + Right Arrow	extend text selection by one word
Up Arrow Down Arrow	Transcript window: scroll through command history Source window: move cursor one line up or down
Ctrl + Up Arrow Ctrl + Down Arrow	Transcript window: moves cursor to first or last line Source window: moves cursor up or down one paragraph
Alt + /	Open a pop-up command prompt for entering commands.
Ctrl + Home	move cursor to the beginning of the text
Ctrl + End	move cursor to the end of the text
Backspace Ctrl + h (UNIX only)	delete character to the left
Delete Ctrl + d (UNIX only)	delete character to the right
Esc (Windows only)	cancel
Alt	activate or inactivate menu bar mode
Alt-F4	close active window
Home Ctrl + a	move cursor to the beginning of the line
Ctrl + Shift + a	select all contents of active window
Ctrl + b	move cursor left
Ctrl + d	delete character to the right
End Ctrl + e	move cursor to the end of the line
Ctrl + f (UNIX) Right Arrow (Windows)	move cursor right one character
Ctrl + k	delete to the end of line

Table E-3. Keyboard Shortcuts (cont.)

Keystrokes - UNIX and Windows	Result
Ctrl + n	move cursor one line down (Source window only under Windows)
Ctrl + o (UNIX only)	insert a new line character at the cursor
Ctrl + p	move cursor one line up (Source window only under Windows)
Ctrl + s (UNIX) Ctrl + f (Windows)	find
Ctrl + t	reverse the order of the two characters on either side of the cursor
Ctrl + u	delete line
Page Down Ctrl + v (UNIX only)	move cursor down one screen
Ctrl + x	cut the selection
Ctrl + s Ctrl + x (UNIX Only)	save
Ctrl + v	paste the selection
Ctrl + a (Windows Only)	select the entire contents of the widget
Ctrl + \	clear any selection in the widget
Ctrl + - (UNIX) Ctrl + / (UNIX) Ctrl + z (Windows)	undoes previous edits in the Source window
Meta + < (UNIX only)	move cursor to the beginning of the file
Meta + > (UNIX only)	move cursor to the end of the file
Page Up Meta + v (UNIX only)	move cursor up one screen
Ctrl + c	copy selection
F3	Performs a Find Next action in the Source window.
F4 Shift+F4	Change focus to next pane in main window Change focus to previous pane in main window
F5 Shift+F5	Toggle between expanding and restoring size of pane to fit the entire main window Toggle on/off the pane headers.
F8	search for the most recent command that matches the characters typed (Main window only)

Table E-3. Keyboard Shortcuts (cont.)

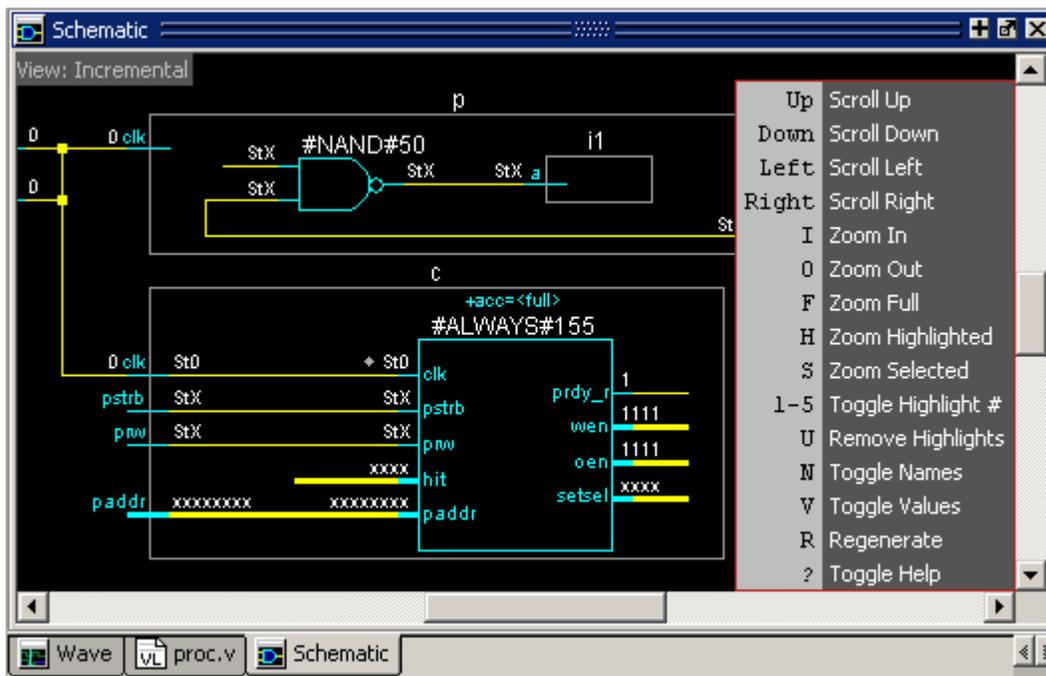
Keystrokes - UNIX and Windows	Result
F9	run simulation
F10	continue simulation
F11 (Windows only)	single-step
F12 (Windows only)	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

List of Keyboard Shortcuts in GUI Windows

You can open a dynamic list of all keyboard shortcuts, predetermined and user defined, for most windows by entering Ctrl-Shift-?

Figure E-1. Schematic Window Keyboard Shortcuts



You can create user defined keyboard shortcuts and change predetermined shortcuts. Refer to [User Defined Keyboard Shortcuts](#) for more information.

List Window Keyboard Shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Table E-4. List Window Keyboard Shortcuts

Key - UNIX and Windows	Action
Left Arrow	scroll listing left (selects and highlights the item to the left of the currently selected item)
Right Arrow	scroll listing right (selects and highlights the item to the right of the currently selected item)
Up Arrow	scroll listing up
Down Arrow	scroll listing down
Page Up Ctrl + Up Arrow	scroll listing up by page
Page Down Ctrl + Down Arrow	scroll listing down by page
Tab	searches forward (down) to the next transition on the selected signal
Shift + Tab	searches backward (up) to the previous transition on the selected signal
Shift + Left Arrow Shift + Right Arrow	extends selection left/right
Ctrl + f (Windows) Ctrl + s (UNIX)	opens the Find dialog box to find the specified item label within the list display

Wave Window Mouse and Keyboard Shortcuts

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

Table E-5. Wave Window Mouse Shortcuts

Mouse action ¹	Result
Ctrl + Click left mouse button and drag 	zoom area (in)
Ctrl + Click left mouse button and drag 	zoom out

Table E-5. Wave Window Mouse Shortcuts

Mouse action ¹	Result
Ctrl + Click left mouse button and drag 	zoom fit
Click left mouse button and drag	moves closest cursor
Ctrl + Click left mouse button on a scroll bar arrow	scrolls window to very top or bottom (vertical scroll) or far left or right (horizontal scroll)
Click middle mouse button in scroll bar (UNIX only)	scrolls window to position of click
Shift + scroll with middle mouse button	scrolls window

1. If you choose **Wave > Mouse Mode > Zoom Mode**, you do not need to press the Ctrl key.

Table E-6. Wave Window Keyboard Shortcuts

Keystroke	Action
s	bring into view and center the currently active cursor
i Shift + i +	zoom in (mouse pointer must be over the cursor or waveform panes)
o Shift + o -	zoom out (mouse pointer must be over the cursor or waveform panes)
f Shift + f	zoom full (mouse pointer must be over the cursor or waveform panes)
l Shift + l	zoom last (mouse pointer must be over the cursor or waveform panes)
r Shift + r	zoom range (mouse pointer must be over the cursor or waveform panes)
m	zooms all open Wave windows to the zoom range of the active window.
Up Arrow Down Arrow	scrolls entire window up or down one line, when mouse pointer is over waveform pane scrolls highlight up or down one line, when mouse pointer is over pathname or values pane
Left Arrow	scroll pathname, values, or waveform pane left

Table E-6. Wave Window Keyboard Shortcuts

Keystroke	Action
Right Arrow	scroll pathname, values, or waveform pane right
Page Up	scroll waveform pane up by a page
Page Down	scroll waveform pane down by a page
Tab	search forward (right) to the next transition on the selected signal - finds the next edge
Shift + Tab	search backward (left) to the previous transition on the selected signal - finds the previous edge
Ctrl+G	automatically create a group for the selected signals by region with the name Group<n>. If you use this shortcut on signals for which there is already a "Group<n>" they will be placed in that region's group rather than creating a new one.
Ctrl + F (Windows) Ctrl + S (UNIX)	open the find dialog box; searches within the specified field in the pathname pane for text strings
Ctrl + Left Arrow Ctrl + Right Arrow	scroll pathname, values, or waveform pane left or right by a page

Appendix F

Setting GUI Preferences

The ModelSim GUI is programmed using Tcl/Tk. It is highly customizable. You can control everything from window size, position, and color to the text of window prompts, default output filenames, and so forth. You can even add buttons and menus that run user-programmable Tcl code.

Most user GUI preferences are stored as Tcl variables in the *.modelsim* file on Unix/Linux platforms or the Registry on Windows platforms. The variable values save automatically when you exit ModelSim. Some of the variables are modified by actions you take with menus or windows (for example, resizing a window changes its geometry variable). Or, you can edit the variables directly either from the prompt in the Transcript window or the **Tools > Edit Preferences** menu item.

Customizing the Simulator GUI Layout

There are five predefined layout modes that the GUI will load dependent upon which part of the simulation flow you are currently in. They include:

- **NoDesign** — This layout is the default view when you first open the GUI or quit out of an active simulation.
- **Simulate** — This layout appears after you have begun a simulation with *vsim*.
- **Coverage** — This layout appears after you have begun a simulation with the *-coverage* switch or loaded a UCDB dataset.
- **VMgmt** — This layout appears after you have loaded a dataset containing test plan information.

These layout modes are fully customizable and the GUI stores your manipulations in the *.modelsim* file (UNIX and Linux) or the registry (Windows) when you exit the simulation or change to another layout mode. The types of manipulations that are stored include: showing, hiding, moving, and resizing windows.

Layout Mode Loading Priority

The GUI stores your manipulations on a directory by directory basis and attempts to load a layout mode in the following order:

1. **Directory** — The GUI attempts to load any manipulations for the current layout mode based on your current working directory.

2. **Last Used** — If there is no layout related to your current working directory, the GUI attempts to load your last manipulations for that layout mode, regardless of your directory.
3. **Default** — If you have never manipulated a layout mode, or have deleted the *.modelsim* file or the registry, the GUI will load the default appearance of the layout mode.

Configure Window Layouts Dialog Box

The Configure Window Layouts dialog box allows you to alter the default behavior of the GUI layouts. You can display this dialog box by selecting the **Layout > Configure** menu item. The elements of this dialog box include:

- **Specify a Layout to Use** — This pane allows you to map which layout is used for the four actions. Refer to the section [Changing Layout Mode Behavior](#) for additional information.
- **Save window layout automatically** — This option (on by default) instructs the GUI to save any manipulations to the layout mode upon exit or changing the layout mode.
- **Save Window Layout by Current Directory** — This option (on by default) instructs the tool to save the final state of the GUI layout on a directory by directory basis. This means that the next time you open the GUI from a given directory, the tool will load your previous GUI settings.
- **Window Restore Properties Button**— Opens the Window Restore Properties Dialog Box. Refer to [Configuring Default Windows for Restored Layouts](#) for more information.

Creating a Custom Layout Mode

To create a custom layout, follow these steps:

1. Rearrange the GUI as you see fit.
2. Select **Layout > Save Layout As**.

This displays the Save Current Window Layout dialog box.

3. In the Save Layout As field, type in a new name for the layout mode.
4. Click OK.

The layout is saved to the *.modelsim* file or registry. You can then access this layout mode from the Layout menu or the Layout toolbar.

Changing Layout Mode Behavior

To assign which predefined or custom layout appears in each mode, follow these steps:

1. Create your custom layouts as described in [Creating a Custom Layout Mode](#).
2. Select **Layout > Configure**.
This displays the [Configure Window Layouts Dialog Box](#).
3. Select which layout you want the GUI to load for each scenario. This behavior affects the [Layout Mode Loading Priority](#).
4. Click OK.

The layout assignment is saved to the *.modelsim* file or registry.

Resetting a Layout Mode to its Default

To get a layout back to the default arrangement, follow these steps:

1. Load the layout mode you want to reset via the Layout menu or the Layout toolbar.
2. Select **Layout > Reset**.

Deleting a Custom Layout Mode

To delete a custom layout, follow these steps:

1. Load a custom layout mode from the Layout menu or the Layout toolbar.
2. Select **Layout > Delete**.
Displays the Delete Custom Layout dialog box.
3. Select the custom layout you wish to delete.
4. **Delete**.

Configuring Default Windows for Restored Layouts

The Window Restore Properties Dialog Box allows you to specify which windows will be restored when a layout is reloaded.

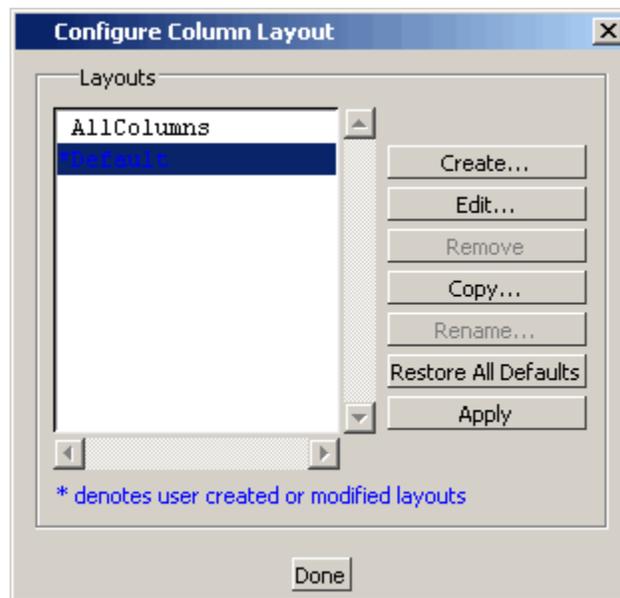
1. Select **Layout > Configure** to open the **Configure Window Layouts** dialog box.
2. Click the **Window Restore Properties** button to open the **Window Restore Properties** dialog box
3. Select the windows you want to have opened when a new layout is loaded. Windows that are not selected will not load until specified with the [view](#) command or by selecting **View > <window>**.

You can also work with window layouts by specifying **layout suppressstype <window>**, **layout restoretype**, or **layout showsuppresstypes**. Refer to the [layout](#) command for more information.

Configuring the Column Layout

Some windows allow you to configure the column layout using the Configure Column Layout dialog ([Configure Column Layout Dialog](#)).

Figure F-1. Configure Column Layout Dialog

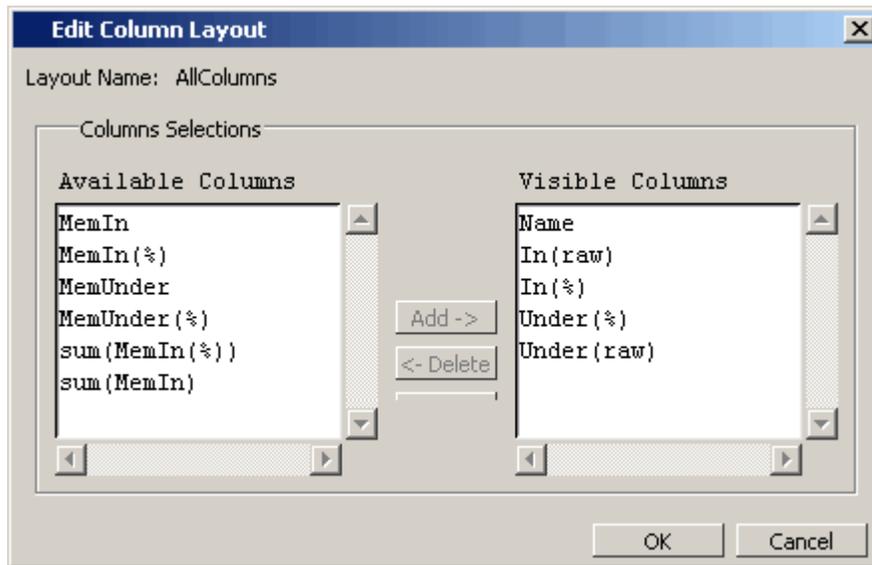


This dialog can also be opened by right-clicking a column heading and selecting Configure Column Layout from the popup menu; or by selecting “Configure ColumnLayout” from the drop-down list in the [Column Layout Toolbar](#).

An asterisk (*) prefix and blue font indicate column layouts are in their default state and which have been added or modified.

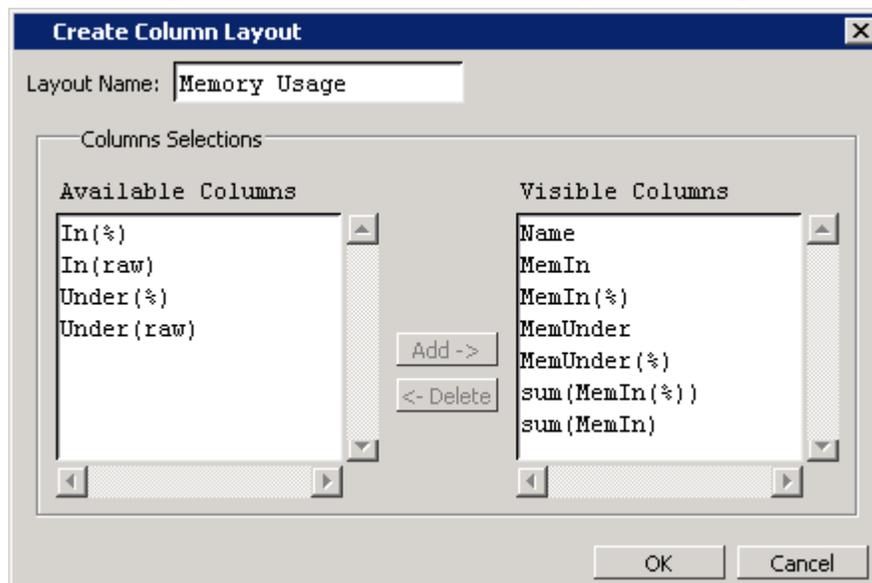
Click the Edit button to open the Edit Column Layout dialog, where you can add and remove columns from the display and change their order ([Figure F-2](#)).

Figure F-2. Edit Column Layout Dialog



Or, click the Create button to create a customized column layout for your application. The Create Column Layout window allows you to select the columns that you want to appear in the customized layout. For example, in [Figure F-3](#), we have created a Memory Usage layout for the Ranked Profile window that includes only those columns related to memory usage.

Figure F-3. Create Column Layout Dialog



Simulator GUI Preferences

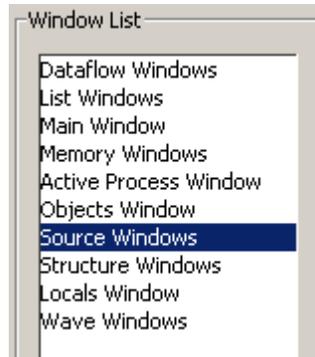
Simulator GUI preferences are stored by default either in the *.modelsim* file in your HOME directory on UNIX/Linux platforms or the Registry on Windows platforms.

Setting Preference Variables from the GUI

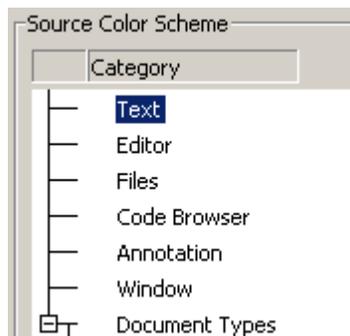
To edit a variable value from the GUI, select **Tools > Edit Preferences**.

The dialog organizes preferences into two tab groups: By Window and By Name. The By Window tab primarily allows you to change colors and fonts for various GUI objects. For example, if you want to change the color of the text in the Source window, do the following:

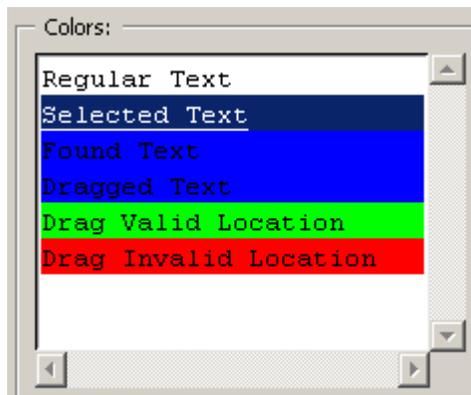
1. Select "Source Windows" from the Window List column.



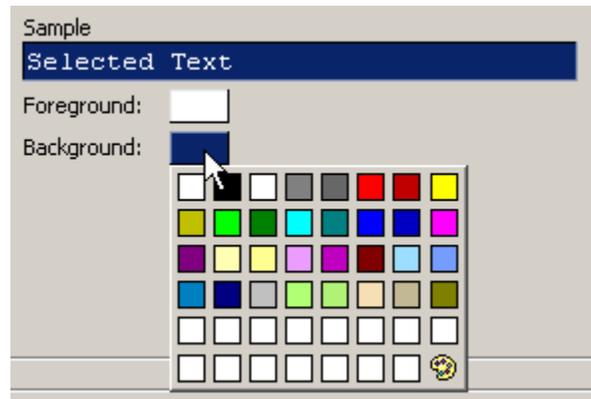
2. Select "Text" from the Source Color Scheme column.



3. Click the type of text you want to change (Regular Text, Selected Text, Found Text, and so forth) from the Colors area.



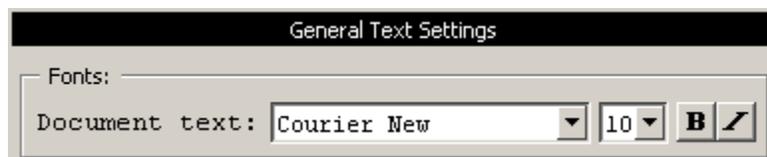
- Click the “Foreground” or “Background” color block.



- Select a color from the palette.

To change the font type and/or size of the window selected in the Windows List column, use the Fonts section of the By Window tab that appears under “General Text Settings” (Figure F-4).

Figure F-4. Change Text Fonts for Selected Windows



You can also make global font changes to all GUI windows with the Fonts section of the By Window tab (Figure F-5).

Figure F-5. Making Global Font Changes

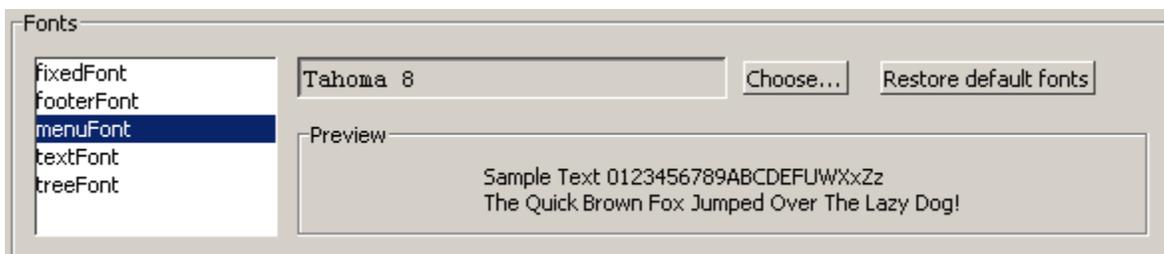


Table F-1. Global Fonts

Global Font Name	Description
fixedFont	for all text in Source window and Notepad display, and in all text entry fields or boxes
footerFont	for all footer text that appears in footer of Main window and all undocked windows

Table F-1. Global Fonts

Global Font Name	Description
menuFont	for all menu text
textFont	for Transcript window text and text in list boxes
treeFont	for all text that appears in any window that displays a hierarchical tree

The By Name tab lists every Tcl variable in a tree structure. The procedure for changing a Tcl variable is:

1. Expand the tree.
2. Highlight a variable.
3. Click **Change Value** to edit the current value.

Clicking **OK** or **Apply** at the bottom of the Preferences dialog changes the variable, and the change is saved when you exit ModelSim.

You can search for information in the By Name tab by using the the **Find** button. However, the **Find** button will only search expanded preference items, therefore it is suggested that you click the **Expand All** button before searching within this tab.

Setting Preference Variables from the Command Line

Use the Tcl [set Command Syntax](#) to customize preference variables from the Main window command line. For example:

```
set <variable name> <variable value>
```

Saving GUI Preferences

GUI preferences are saved automatically when you exit the tool.

If you prefer to store GUI preferences elsewhere, set the [MODELSIM_PREFERENCES](#) environment variable to designate where these preferences are stored. Setting this variable causes ModelSim to use a specified path and file instead of the default location. Here are some additional points to keep in mind about this variable setting:

- The file does not need to exist before setting the variable as ModelSim will initialize it.
- If the file is read-only, ModelSim will not update or otherwise modify the file.
- This variable may contain a relative pathname, in which case the file is relative to the working directory at the time the tool is started.

The modelsim.tcl File

Previous versions saved user GUI preferences into a *modelsim.tcl* file. Current versions will still read in a *modelsim.tcl* file if it exists. ModelSim searches for the file as follows:

- use `MODELSIM_TCL` environment variable if it exists (if `MODELSIM_TCL` is a list of files, each file is loaded in the order that it appears in the list); else
- use `./modelsim.tcl`; else
- use `$(HOME)/modelsim.tcl` if it exists

Note that in versions 6.1 and later, ModelSim will save to the *.modelsim* file any variables it reads in from a *modelsim.tcl* file (except for `user_hook` variables). The values from the *modelsim.tcl* file will override like variables in the *.modelsim* file.

User_hook Variables

`User_hook` variables allow you to add buttons and menus to the GUI. `User_hook` variables can only be stored in a *modelsim.tcl* file. They are not stored in *.modelsim*. If you need to use `user_hook` variables, you must create a *modelsim.tcl* file to store them.

Appendix G

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.

Table G-1. Files Accessed During Startup

File	Purpose
<i>modelsim.ini</i>	contains initial tool settings; see modelsim.ini Variables for specific details on the <i>modelsim.ini</i> file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is <code>mgc_location_map</code>
<i>pref.tcl</i>	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics
.modelsim (UNIX) or Windows registry	contains last working directory, project file, printer defaults, and other user-customized GUI characteristics
<i>modelsim.tcl</i>	contains user-customized settings for fonts, colors, prompts, other GUI characteristics; maintained for backwards compatibility with older versions (see The modelsim.tcl File)
<i><project_name>.mpf</i>	if available, loads last project file which is specified in the registry (Windows) or <code>\$(HOME)/.modelsim</code> (UNIX); see What are Projects? for details on project settings

Initialization Sequence

The following list describes in detail ModelSim's initialization sequence. The sequence includes a number of conditional structures, the results of which are determined by the existence of certain files and the current settings of environment variables.

In the steps below, names in uppercase denote environment variables (except `MTI_LIB_DIR` which is a Tcl variable). Instances of `$(NAME)` denote paths that are determined by an environment variable (except `$(MTI_LIB_DIR)` which is determined by a Tcl variable).

1. Determines the path to the executable directory (`./modeltech/<platform>`). Sets `MODEL_TECH` to this path, *unless* `MODEL_TECH_OVERRIDE` exists, in which case `MODEL_TECH` is set to the same value as `MODEL_TECH_OVERRIDE`.

Environment Variables used: [MODEL_TECH](#), [MODEL_TECH_OVERRIDE](#)

2. Finds the `modelsim.ini` file by evaluating the following conditions:
 - use `$(MODELSIM)` (which specifies the directory location and name of a `modelsim.ini` file) if it exists; else
 - use `$(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

Environment Variables used: [MODELSIM](#), [MGC_WD](#), [MGC_HOME](#)

You can determine which `modelsim.ini` file was used by executing the [where](#) command.

3. Finds the location map file by evaluating the following conditions:
 - use `MGC_LOCATION_MAP` if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else
 - use `mgc_location_map` if it exists; else
 - use `$(HOME)/mgc/mgc_location_map`; else
 - use `$(HOME)/mgc_location_map`; else
 - use `$(MGC_HOME)/etc/mgc_location_map`; else
 - use `$(MGC_HOME)/shared/etc/mgc_location_map`; else
 - use `$(MODEL_TECH)/mgc_location_map`; else
 - use `$(MODEL_TECH)/../mgc_location_map`; else
 - use no map

Environment Variables used: [MGC_LOCATION_MAP](#), [HOME](#), [MGC_HOME](#), [MODEL_TECH](#)

4. Reads various variables from the [vsim] section of the *modelsim.ini* file. See [modelsim.ini Variables](#) for more details.
5. Parses any command line arguments that were included when you started ModelSim and reports any problems.
6. Defines the following environment variables:
 - use MODEL_TECH_TCL if it exists; else
 - set MODEL_TECH_TCL=\$(MODEL_TECH)/../tcl
 - set TCL_LIBRARY=\$(MODEL_TECH_TCL)/tcl8.4
 - set TK_LIBRARY=\$(MODEL_TECH_TCL)/tk8.4
 - set ITCL_LIBRARY=\$(MODEL_TECH_TCL)/itcl3.0
 - set ITK_LIBRARY=\$(MODEL_TECH_TCL)/itk3.0
 - set VSIM_LIBRARY=\$(MODEL_TECH_TCL)/vsim

Environment Variables used: [MODEL_TECH_TCL](#), [TCL_LIBRARY](#), [TK_LIBRARY](#), [MODEL_TECH](#), [ITCL_LIBRARY](#), [ITK_LIBRARY](#), [VSIM_LIBRARY](#)

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).
9. The next four steps relate to initializing the graphical user interface.
10. Sets Tcl variable MTI_LIB_DIR=\$(MODEL_TECH_TCL)
Environment Variables used: [MTI_LIB_DIR](#), [MODEL_TECH_TCL](#)
11. Loads \$(MTI_LIB_DIR)/vsim/pref.tcl.
Environment Variables used: [MTI_LIB_DIR](#)
12. Loads GUI preferences, project file, and so forth, from the registry (Windows) or \$(HOME)/.modelsim (UNIX).
Environment Variables used: [HOME](#)
13. Searches for the *modelsim.tcl* file by evaluating the following conditions:
 - use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
 - use ./modelsim.tcl; else
 - use \$(HOME)/modelsim.tcl if it exists

Environment Variables used: [HOME](#), [MODEL_TECH_TCL](#)

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

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler or simulator 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.

Environment Variables

Environment Variable Expansion

The shell commands **vcom**, **vlog**, **vsim**, and **vmap**, no longer expand environment variables in filename arguments and options. Instead, variables should be expanded by the shell beforehand, in the usual manner. The -f switch that most of these commands support now performs environment variable expansion throughout the file.

Environment variable expansion is still performed in the following places:

- Pathname and other values in the *modelsim.ini* file
- Strings used as file pathnames in VHDL and Verilog
- VHDL Foreign attributes
- The **PLIOBJS** environment variable may contain a path that has an environment variable.
- Verilog ``uselib` file and dir directives
- Anywhere in the contents of a -f file

The recommended method for using flexible pathnames is to make use of the MGC Location Map system (see [Using Location Mapping](#)). When this is used, then pathnames stored in libraries and project files (.mpf) will be converted to logical pathnames.

If a file or path name contains the dollar sign character (\$), and must be used in one of the places listed above that accepts environment variables, then the explicit dollar sign must be escaped by using a double dollar sign (\$\$).

Setting Environment Variables

Before compiling or simulating, several environment variables may be set to provide the functions described below. You set the variables as follows:

- Windows — through the System control panel, refer to “[Creating Environment Variables in Windows](#)” for more information.
- Linux/UNIX — typically through the *.login* script.

The LM_LICENSE_FILE variable is required; all others are optional.

DOPATH

The toolset uses the DOPATH environment variable to search for DO files (macros). DOPATH consists of a colon-separated (semi-colon for Windows) list of paths to directories. You can override this environment variable with the DOPATH Tcl preference variable.

The DOPATH environment variable isn't accessible when you invoke vsim from a UNIX shell or from a Windows command prompt. It is accessible once ModelSim or vsim is invoked. If you need to invoke from a shell or command line and use the DOPATH environment variable, use the following syntax:

```
vsim -do "do <dofile_name>" <design_unit>
```

DP_INIFILE

The DP_INIFILE environment variable points to a file that contains preference settings for the Source window. By default, this file is created in your \$HOME directory. You should only set this variable to a different location if your \$HOME directory does not exist or is not writable.

EDITOR

The EDITOR environment variable specifies the editor to invoke with the [edit](#) command

From the Windows platform, you could set this variable from within the Transcript window with the following command:

```
set PrefMain(Editor) {c:/Program Files/Windows NT/Accessories/wordpad.exe}
```

Where you would replace the path with that of your desired text editor. The braces ({ }) are required because of the spaces in the pathname

HOME

The toolset uses the HOME environment variable to look for an optional graphical preference file and optional location map file. Refer to [modelsim.ini Variables](#) for additional information.

ITCL_LIBRARY

Identifies the pathname of the [incr]Tcl library; set by ModelSim to the same path as MODEL_Tech_TCL; must point to libraries supplied by Mentor Graphics.

ITK_LIBRARY

Identifies the pathname of the [incr]Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Mentor Graphics.

LD_LIBRARY_PATH

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used for both 32-bit and 64-bit shared libraries on Linux systems.

LD_LIBRARY_PATH_32

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used only for 32-bit shared libraries on Linux systems.

LD_LIBRARY_PATH_64

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used only for 64-bit shared libraries on Linux systems.

LM_LICENSE_FILE

The toolset's file manager uses the LM_LICENSE_FILE environment variable to find the location of the license file. The argument may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files. The environment variable is required.

MGC_AMS_HOME

Specifies whether vcom adds the declaration of REAL_VECTOR to the STANDARD package. This is useful for designers using VHDL-AMS to test digital parts of their model.

MGC_HOME

Identifies the pathname of the MGC product suite.

MGC_LOCATION_MAP

The toolset uses the MGC_LOCATION_MAP environment variable to find source files based on easily reallocated "soft" paths.

MGC_WD

Identifies the Mentor Graphics working directory. This variable is used in the initialization sequence.

MODEL_TECH

Do not set this variable. The toolset automatically sets the MODEL_TECH environment variable to the directory in which the binary executable resides.

MODEL_TECH_OVERRIDE

Provides an alternative directory path for the binary executables. Upon initialization, the product sets MODEL_TECH to this path, if set.

MODEL_TECH_TCL

Specifies the directory location of Tcl libraries for Tcl/Tk and vsim, and may also be used to specify a startup DO file. This variable defaults to *<installDIR>/tcl*, however you may set it to an alternate path.

MODELSIM

The toolset uses the MODELSIM environment variable to find the *modelsim.ini* file. The argument consists of a path including the file name.

An alternative use of this variable is to set it to the path of a project file (*<Project_Root_Dir>/<Project_Name>.mpf*). This allows you to use project settings with command line tools. However, if you do this, the *.mpf* file will replace *modelsim.ini* as the initialization file for all tools.

MODELSIM_PREFERENCES

The MODELSIM_PREFERENCES environment variable specifies the location to store user interface preferences. Setting this variable with the path of a file instructs the toolset to use this file instead of the default location (your HOME directory in UNIX or in the registry in Windows). The file does not need to exist beforehand, the toolset will initialize it. Also, if this file is read-only, the toolset will not update or otherwise modify the file. This variable may contain a relative pathname – in which case the file will be relative to the working directory at the time ModelSim is started.

MODELSIM_TCL

identifies the pathname to a user preference file (for example, C:\questasim\modelsim.tcl); can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX); note that user preferences are now stored in the *.modelsim* file (Unix) or registry (Windows); QuestaSim will still read this environment variable but it will then save all the settings to the *.modelsim* file when you exit ModelSim.

MTI_COSIM_TRACE

The MTI_COSIM_TRACE environment variable creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls. You should set this variable to any value before invoking the simulator.

MTI_LIB_DIR

Identifies the path to all Tcl libraries installed with ModelSim.

MTI_LIBERTY_PATH

Identifies the pathname of the Liberty library containing Liberty logic cell definitions. Refer to Liberty Library Models for more information about the Liberty library modeling standard.

MTI_TF_LIMIT

The MTI_TF_LIMIT environment variable limits the size of the VSOUT temp file (generated by the toolset's kernel). Set the argument of this variable to the size of k-bytes

The environment variable TMPDIR controls the location of this file, while STDOUT controls the name. The default setting is 10, and a value of 0 specifies that there is no limit. This variable does *not* control the size of the transcript file.

MTI_RELEASE_ON_SUSPEND

The MTI_RELEASE_ON_SUSPEND environment variable allows you to turn off or modify the delay for the functionality of releasing all licenses when operation is suspended. The default setting is 10 (in seconds), which means that if you do not set this variable your licenses will be released 10 seconds after your run is suspended. If you set this environment variable with an argument of 0 (zero) ModelSim will not release the licenses after being suspended. You can change the default length of time (number of seconds) by setting this environment variable to an integer greater than 0 (zero).

MTI_USELIB_DIR

The MTI_USELIB_DIR environment variable specifies the directory into which object libraries are compiled when using the **-compile_uselibs** argument to the **vlog** command

MTI_VCO_MODE

The MTI_VCO_MODE environment variable specifies which version of the toolset to use on platforms that support both 32- and 64-bit versions when the executables are invoked from the *modeltech/bin* directory by a Unix shell command (using full path specification or PATH search). Acceptable values are either "32" or "64" (do not include quotes). If you do not set this variable, the default is to use 32-bit mode, even on 64-bit machines.

MTI_VOPT_FLOW

The environment variable `MTI_VOPT_FLOW` determines whether `vopt` is used as part of the `vsim` command. This variable is overridden by `vsim` switches `-vopt` and `-novopt`, but it overrides the `VoptFlow` setting in `modelsim.ini`.

Setting `MTI_VOPT_FLOW` to 0 means do not use `vopt` (`-novopt`). Setting it to any other value means use `vopt` (`-vopt`).

NOMMAP

When set to 1, the `NOMMAP` environment variable disables memory mapping in the toolset. You should only use this variable when running on Linux 7.1 because it will decrease the speed with which ModelSim reads files.

PLIOBJS

The toolset uses the `PLIOBJS` environment variable to search for PLI object files for loading. The argument consists of a space-separated list of file or path names

STDOUT

The argument to the `STDOUT` environment variable specifies a filename to which the simulator saves the `VSOUT` temp file information. Typically this information is deleted when the simulator exits. The location for this file is set with the `TMPDIR` variable, which allows you to find and delete the file in the event of a crash, because an unnamed `VSOUT` file is not deleted after a crash.

TCL_LIBRARY

Identifies the pathname of the Tcl library; set by ModelSim to the same pathname as `MODEL_TECH_TCL`; must point to libraries supplied by Mentor Graphics.

TK_LIBRARY

Identifies the pathname of the Tk library; set by ModelSim to the same pathname as `MODEL_TECH_TCL`; must point to libraries supplied by Mentor Graphics.

TMP

(Windows environments) The `TMP` environment variable specifies the path to a `tempnam()` generated file (`VSOUT`) containing all stdout from the simulation kernel.

TMPDIR

(UNIX environments) The `TMPDIR` environment variable specifies the path to a `tempnam()` generated file (`VSOUT`) containing all stdout from the simulation kernel.

VSIM_LIBRARY

Identifies the pathname of the Tcl files that are used by ModelSim; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Mentor Graphics.

Creating Environment Variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

1. From your desktop, right-click your **My Computer** icon and select **Properties**
2. In the System Properties dialog box, select the Advanced tab
3. Click Environment Variables
4. In the Environment Variables dialog box and User variables for <user> pane, select New:
5. In the New User Variable dialog box, add the new variable with this data

```
Variable name: MY_PATH  
Variable value: \temp\work
```

6. OK (New User Variable, Environment Variable, and System Properties dialog boxes)

Library Mapping with Environment Variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command to add library mappings to the current *modelsim.ini* file.

Table G-2. Add Library Mappings to modelsim.ini File

Prompt Type	Command	Result added to <i>modelsim.ini</i>
DOS prompt	vmap MY_VITAL %MY_PATH%	MY_VITAL = c:\temp\work
ModelSim or vsim prompt	vmap MY_VITAL \"\$MY_PATH ¹ or vmap MY_VITAL {\$MY_PATH}	MY_VITAL = \$MY_PATH

1. The dollar sign (\$) character is Tcl syntax that indicates a variable. The backslash (\) character is an escape character that prevents the variable from being evaluated during the execution of **vmap**.

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
```

Use braces ({ }) for cases where the path contains multiple items that need to be escaped, such as spaces in the pathname or backslash characters. For example:

```
vmap celllib {$LIB_INSTALL_PATH/Documents And Settings/All/celllib}
```

Referencing Environment Variables

There are two ways to reference environment variables within ModelSim. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
use std.textio.all;
entity test is end;
architecture only of test is
begin
    process
        FILE in_file : text is in "$ENV_VAR_NAME";
    begin
        wait;
    end process;
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

Note



Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

Removing Temp Files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the Graphical User Interface. In normal circumstances the file is deleted when the simulator exits. If ModelSim crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

Appendix H

Third-Party Model Support

This appendix provides information about using ModelSim with the Synopsys SmartModels and Synopsys hardware modeling.

Synopsys SmartModels

You can use the Synopsys SWIFT-based SmartModel library with ModelSim. The SmartModel library is a collection of behavioral models supplied in binary form with a procedural interface that is accessed by the simulator.

This section only describes the specifics of using SmartModels with ModelSim.

Note

A 32-bit SmartModel will not run with a 64-bit version of the simulator. When trying to load the operating system specific 32-bit library into the 64-bit executable, the pointer sizes will be incorrect.

VHDL SmartModel Interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

To enable the SmartModel interface you must do the following:

1. Set the **LMC_HOME** environment variable to the root of the SmartModel library installation directory. Consult SmartModel documentation for details.
2. Uncomment the appropriate **libswift** entry in the *modelsim.ini* file for your operating system.
 - **libswift** — This variable points to the dynamic link library software that accesses the SmartModels.
3. If you are running the Windows operating system, you must also comment out the default **libsm** entry (precede the line with a semicolon (;)) and uncomment the **libsm** entry for the Windows operating system.
 - **libsm** — This variable points to the dynamic link library that interfaces the foreign architecture to the SmartModel software.

By default, the **libsm** entry points to the *libsm.sl* supplied in the ModelSim installation directory indicated by the **MODEL_TECH** environment variable. ModelSim automatically sets the **MODEL_TECH** environment variable to the appropriate directory containing the executables and binaries for the current operating system.

The **libswift** and **libsm** entries are found under the [lmc] section of the *modelsim.ini* file located in the ModelSim installation directory.

Creating Foreign Architectures with `sm_entity`

The **sm_entity** tool automatically creates entities and foreign architectures for SmartModels.

1. Create the entity and foreign architecture using the `sm_entity` tool.

By default, the **sm_entity** tool writes an entity and foreign architecture to stdout, but you can redirect it to a file with the following syntax

```
sm_entity -all > sml.vhd
```

2. Compile the entity and foreign architecture into a library named *lmc*.

For example, the following commands compile the entity and foreign architecture:

```
vlib lmc
```

```
vcom -work lmc sml.vhd
```

3. Generate a component declaration

You will need to generate a component declaration for the SmartModels so that you can instantiate the them in your VHDL design. 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
```

4. Create a package of SmartModel component declarations

Edit the resulting *smlcomp.vhd* file to turn it into a package as follows:

```
library ieee;  
use ieee.std_logic_1164.all;  
package sml is  
    <component declarations go here>  
end sml;
```

5. Compile the package into the **lmc** library:

```
vcom -work lmc smlcomp.vhd
```

6. Reference the SmartModels in your design.

Add the following **library** and **use** clauses to your code:

```
library lmc;
```

```
use lmc.sml.all;
```

sm_entity Syntax

```
sm_entity [-] [-xe] [-xa] [-c] [-all] [-v] [-93] [-modelsimini <ini_filepath>]  
[<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.
- -modelsimini <ini_filepath> — Load an alternate initialization file that replaces the current initialization file. Overrides the file path specified in the MODELSIM environment variable. Specify either an absolute or relative path the initialization file. On Windows systems the path separator should be a forward slash (/).
- <SmartModelName> — Name of a SmartModel.

Example Output

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;
```

Based on the above example, the following are details about the entity:

- The entity name is the SmartModel name.
- 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 *MemoryFile* generics represent the SmartModel attributes of the same name. **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**.

Based on the above example, the following are details about the architecture:

- 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.

SmartModel Vector Ports

The entities generated by **sm_entity** only contain single-bit ports, never vectored ports. This is necessary because ModelSim correlates entity ports with the SmartModel SWIFT interface by name. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```
component cy7c285
  generic ( TimingVersion : STRING := "CY7C285-65";
           DelayRange : STRING := "Max";
           MemoryFile : STRING := "memory" );
  port ( A : in std_logic_vector (15 downto 0);
        CS : in std_logic;
        O : out std_logic_vector (7 downto 0);
        WAIT_PORT : inout std_logic );
end component;

for all: cy7c285
  use entity work.cy7c285
  port map (A0 => A(0),
           A1 => A(1),
           A2 => A(2),
           A3 => A(3),
           A4 => A(4),
           A5 => A(5),
           A6 => A(6),
           A7 => A(7),
           A8 => A(8),
           A9 => A(9),
           A10 => A(10),
           A11 => A(11),
           A12 => A(12),
           A13 => A(13),
           A14 => A(14),
           A15 => A(15),
           CS => CS,
           O0 => O(0),
           O1 => O(1),
           O2 => O(2),
           O3 => O(3),
           O4 => O(4),
           O5 => O(5),
           O6 => O(6),
           O7 => O(7),
           WAIT_PORT => WAIT_PORT );
```

Command Channel

The command channel lets you invoke SmartModel specific commands. ModelSim provides access to the Command Channel from the command line.

The form of a SmartModel command is:

```
lmc {<instance_name> | -all} "<SmartModel command>"
```

- **instance_name** — 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). For example, to turn timing checks off for SmartModel */top/u1*:

```
lmc /top/u1 "SetConstraints Off"
```

- **-all** — applies the command to all SmartModel instances. For example, to turn timing checks off for all SmartModel instances:

```
lmc -all "SetConstraints Off"
```

There are also some SmartModel commands that apply globally to the current simulation session rather than to models. The form of a SmartModel session command is:

```
lmcsession "<SmartModel session command>"
```

SmartModel Windows

Some models in the SmartModel library provide access to internal registers with a feature called SmartModel Windows. The simulator interface to this feature is described below.

Window names that are not valid VHDL or Verilog identifiers are converted to VHDL extended identifiers. For example, with a window named *z1I10.GSR.OR*, the tool treats the name as `\z1I10.GSR.OR\` (for all commands including `lmcwin`, `add wave`, and `examine`). You must then use that name in all commands. For example,

```
add wave /top/swift_model/\z1I10.GSR.OR\
```

Extended identifiers are case sensitive.

ReportStatus

The **ReportStatus** command displays model information, including the names of window registers. For example,

```
lmc /top/u1 ReportStatus
```

SmartModel Windows description:

```
WA "Read-Only (Read Only)"  
WB "1-bit"  
WC "64-bit"
```

This model contains window registers named *wa*, *wb*, and *wc*. These names can be used in subsequent window (**lmcwin**) commands.

SmartModel lmcwin Commands

Each of the following commands 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** — displays the current value of a window.

lmcwin read <window_instance> [-<radix>]

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** — writes a value into a window.

lmcwin write <window_instance> <value>

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** — enables continuous monitoring of a window.

lmcwin enable <window_instance>

The specified window is added to the model instance as a signal (with the same name as the window) of type **std_logic** or **std_logic_vector**. This signal's values can then be referenced in simulator commands that read signal values, such as the [add list](#) command 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** — disables continuous monitoring of a window.

lmcwin disable <window_instance>

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** — disables the effect of a previous **lmcwin write** command on a window net.

lmcwin release <window_instance>

Some windows are actually nets, and the **lmcwin write** command behaves more like a continuous force on the 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 SmartModel library provides an optional library of Verilog modules and a PLI application that communicates between a simulator's PLI and the SWIFT simulator interface.

Synopsys Hardware Models

A hardware model allows simulation of a device using the actual silicon installed as a hardware model. The hardware modeling system is a network resource with a procedural interface that is accessed by the simulator.

This section only describes the specifics of using hardware models with ModelSim.

VHDL Hardware Model Interface

The simulator 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 simulator locates the hardware modeler interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **hm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly.

These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory.

The simulator automatically loads both the **libhm** and **libsfi** libraries when it elaborates a hardware model foreign architecture.

- **libhm** — This variable points to the dynamic link library that interfaces the foreign architecture to the hardware modeler software.

By default, **libhm** points to the *libhm.sl* supplied in the installation directory indicated by the **MODEL_Tech** environment variable. The tool 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.

- **libsfi** — This variable points to the dynamic link library software that accesses the hardware modeler.

Uncomment the appropriate **libsfi** setting 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 Synopsys hardware modeling documentation.

Creating Foreign Architectures with `hm_entity`

The **hm_entity** tool automatically creates entities and foreign architectures for hardware models.

1. Create the entity and foreign architecture using the `hm_entity` tool.

By default, **hm_entity** writes the entity and foreign architecture to stdout, but you can redirect it to a file with the following syntax:

```
hm_entity LMTEST.MDL > lntest.vhd
```

2. Compile the entity and foreign architecture into a library named *lmc*.

For example, the following commands compile the entity and foreign architecture:

```
vlib lmc  
vcom -work lmc lntest.vhd
```

3. Generate a component declaration.

You will need to generate a component declaration so that you can instantiate the hardware model in your VHDL design. 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
```

4. Place the component declaration into your design.

Paste the resulting component declaration into the appropriate place in your design or into a package.

hm_entity Syntax

hm_entity [-xe] [-xa] [-c] [-93] [-modelsimini <ini_filepath>] <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.
- -modelsimini <ini_filepath> — Load an alternate initialization file that replaces the current initialization file. Overrides the file path specified in the MODELSIM environment variable. Specify either an absolute or relative path the initialization file. On Windows systems the path separator should be a forward slash (/).
- <shell software filename> — Hardware model shell software filename.

Example Output

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;
```

Based on the above example, the following are details about the entity:

- 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.
- 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".

Based on the above example, the following are details about the architecture:

- 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.

Hardware Model 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 not 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.

- Enable/disable test vector logging for the specified hardware model.
`lm_vectors on|off <instance_name> [<filename>]`
- Enable/disable timing measurement for the specified hardware model.
`lm_measure_timing on|off <instance_name> [<filename>]`
- Enable/disable timing checks for the specified hardware model.
`lm_timing_checks on|off <instance_name>`
- Enable/disable pattern looping for the specified hardware model.
`lm_loop_patterns on|off <instance_name>`

- Enable/disable unknown propagation for the specified hardware model.

```
lm_unknowns on|off <instance_name>
```


— Symbols —

- , 947, 952
- .ini control variables
 - AssertFile, 1285
 - AssertionActiveThreadMonitor, 1285
 - AssertionActiveThreadMonitorLimit, 1285
 - AssertionCover, 1286
 - AssertionDebug, 1286, 1290
 - AssertionEnable, 1286
 - AssertionEnableVacuousPassActionBlock, 1287
 - AssertionFailAction, 1287
 - AssertionFailLocalVarLog, 1287
 - AssertionFailLog, 1288
 - AssertionLimit, 1288
 - AssertionPassLog, 1288
 - BreakOnAssertion, 1290
 - CheckPlusargs, 1291
 - CheckpointCompressMode, 1291
 - CommandHistory, 1293
 - ConcurrentFileLimit, 1293
 - CoverAtLeast, 1294
 - CoverCountAll, 1295
 - CoverEnable, 1295
 - CoverExcludeDefault, 1295
 - CoverLimit, 1296
 - CoverLog, 1296
 - CoverWeight, 1300
 - DefaultForceKind, 1302
 - DefaultRadix, 1302
 - DefaultRestartOptions, 1303
 - DelayFileOpen, 1304
 - DumpportsCollapse, 1306
 - ErrorFile, 1308
 - GenerateFormat, 1312
 - GenerousIdentifierParsing, 1313
 - GlobalSharedObjectList, 1314
 - IgnoreError, 1314
 - IgnoreFailure, 1315
 - IgnoreNote, 1315
 - ignoreStandardRealVector, 1316
 - IgnoreSVAError, 1316
 - IgnoreSVAFatal, 1317
 - IgnoreSVAInfo, 1317
 - IgnoreSVAWarning, 1317
 - IgnoreWarning, 1318
 - IterationLimit, 1320
 - License, 1321
 - MessageFormat, 1323
 - MessageFormatBreak, 1324
 - MessageFormatBreakLine, 1324
 - MessageFormatError, 1325
 - MessageFormatFail, 1325
 - MessageFormatFatal, 1325
 - MessageFormatNote, 1326
 - MessageFormatWarning, 1326
 - NumericStdNoWarnings, 1332
 - OldVhdlForGenNames, 1333
 - ParallelJobs, 1335
 - PathSeparator, 1335
 - PslInfinityThreshold, 1339
 - PslOneAttempt, 1339
 - Resolution, 1340
 - RunLength, 1341
 - ScTimeUnit, 1343
 - ShowUnassociatedScNameWarning, 1347
 - ShowUndebuggableScTypeWarning, 1348
 - SimulateAssumeDirectives, 1349
 - SimulateImmedAsserts, 1349
 - SimulatePSL, 1349
 - SimulateSVA, 1350
 - SolveACTbeforeSpeculate, 1350
 - SolveACTMaxOps, 1350
 - SolveACTMaxTests, 1351
 - SolveACTRetryCount, 1351
 - SolveArrayResizeMax, 1351
 - SolveEngine, 1352
 - SolveFailDebug, 1352

- SolveFailDebugMaxSet, [1352](#)
 - SolveFailSeverity, [1353](#)
 - SolveFlags, [1353](#)
 - SolveGraphMaxEval, [1353](#)
 - SolveGraphMaxSize, [1354](#)
 - SolveIgnoreOverflow, [1354](#)
 - SolveRev, [1354](#)
 - SolveSpeculateDistFirst, [1355](#)
 - SolveSpeculateFirst, [1355](#)
 - SolveSpeculateLevel, [1356](#)
 - SolveSpeculateMaxCondWidth, [1356](#)
 - SolveSpeculateMaxIterations, [1357](#)
 - Startup, [1357](#)
 - StdArithNoWarnings, [1358](#)
 - Sv_Seed, [1359](#)
 - SVCovergroupGoal, [1361](#)
 - SVCovergroupStrobe, [1364](#)
 - SVCovergroupTypeGoal, [1364](#)
 - SVCovergroupZWNNoCollect, [1365](#)
 - SVCoverpointAutoBinMax, [1366](#)
 - SVCrossNumPrintMissing, [1367](#)
 - ToggleFixedSizeArray, [1369](#)
 - ToggleMaxFixedSizeArray, [1370](#)
 - ToggleMaxIntValues, [1370](#)
 - ToggleMaxRealValues, [1371](#)
 - ToggleNoIntegers, [1371](#)
 - TogglePackedAsVec, [1371](#)
 - TogglePortsOnly, [1372](#)
 - ToggleVlogEnumBits, [1372](#)
 - ToggleVlogIntegers, [1372](#)
 - ToggleVlogReal, [1373](#)
 - ToggleWidthLimit, [1373](#)
 - TranscriptFile, [1373](#)
 - UCDBFilename, [1374](#)
 - UnattemptedImmediateAssertions, [1374](#)
 - UnbufferedOutput, [1375](#)
 - UserTimeUnit, [1375](#)
 - Veriuser, [1377](#)
 - VoptFlow, [1379](#)
 - WarnConstantChange, [1380](#)
 - WaveSignalNameWidth, [1380](#)
 - WLFCacheSize, [1381](#)
 - WLFCollapseMode, [1381](#)
 - WLFCompress, [1382](#)
 - WLFDeleteOnQuit, [1382](#)
 - WLFFilename, [1383](#)
 - WLFOptimize, [1383](#)
 - WLFSaveAllRegions, [1384](#)
 - WLFsimCacheSize, [1384](#)
 - WLFSizeLimit, [1385](#)
 - WLFTimeLimit, [1385](#)
 - .ini VHDL compiler control variables
 - ShowConstantImmediateAsserts, [1347](#)
 - .modelsim file
 - in initialization sequence, [1461](#)
 - purpose, [1459](#)
 - .so, shared object file
 - loading PLI/VPI/DPI C applications, [1419](#)
 - loading PLI/VPI/DPI C++ applications, [1422](#)
 - #, comment character, [1254](#)
 - +acc option, design object visibility, [349](#)
 - +protect
 - compile for encryption
 - Compile
 - with +protect, [307](#)
 - \$coverage_save, [465](#)
 - \$disable_signal_spy, [1177](#)
 - \$enable_signal_spy, [1179](#)
 - \$finish
 - behavior, customizing, [1334](#)
 - \$load_coverage_db(), using, [1070](#)
 - \$sdf_annotate system task, [1221](#)
 - \$unit scope, visibility in SV declarations, [430](#)
- Numerics —**
- 1076, IEEE Std, [67](#)
 - differences between versions, [390](#)
 - 1364, IEEE Std, [67](#), [420](#), [481](#), [1079](#)
 - 1364-2005
 - IEEE std, [299](#), [1227](#)
 - 2-state toggle coverage, [924](#)
 - 3-state toggle coverage, [924](#)
 - 64-bit libraries, [379](#)
 - 64-bit platforms
 - choosing over 32-bit, [1466](#)
 - 64-bit time
 - now variable, [1260](#)
 - Tcl time commands, [1262](#)
 - 64-bit vsim, using with 32-bit FLI apps, [1436](#)

— A —

+acc option, design object visibility, 349
 ACC routines, 1433
 accelerated packages, 378
 AcceptLowerCasePragmasOnly .ini file variable, 1284
 access
 hierarchical objects, 1175
 limitations in mixed designs, 548
 ACT, 1076, 1081
 Action
 set for cover directives, 995
 Actions
 assertions, 993
 actions
 atv window, 1043
 Active driver
 path details, 881
 Active Processes pane, 223
 see also windows, Active Processes pane
 active thread monitor, 1035
 Active time indicator
 schematic
 Schematic
 active time indicator, 810
 active window, selecting, 102
 Add bookmark
 source window, 263
 Add cursor
 to Wave window, 726
 AddPragmaPrefix .ini file variable, 1284
 aggregates, SystemC, 522
 Algorithm
 negative timing constraint, 451
 alias name, definition, 929
 AmsStandard .ini file variable, 1284
 analog sidebar, 291
 Analysis
 root thread
 assertions, 1046
 annotate
 local variables, 1048
 annotating differences, wave compare, 792
 API, 420
 api_version in error message, 542

Application programming interface (API), 420
 architecture simulator state variable, 1259
 archives
 described, 372
 argc simulator state variable, 1259
 arguments
 passing to a DO file, 1267
 arguments, accessing command-line, 537
 Arithmetic Constraint Technology (ACT), 1076, 1081
 arithmetic package warnings, disabling, 1388
 array
 random dynamic
 size constraints, 1080, 1351
 array of sc_signal<T>, 522
 Ascending expressions
 ATV window, 116
 assert debug, 1034
 replicator parameters, 1035
 Assert directives
 names, 986
 -assertdebug, 1029
 AssertFile .ini file variable, 1285
 Assertion
 counts, 997
 assertion atv command, 1030
 Assertion debugging
 using -assertdebug, 1029
 assertion directives
 report on active, 142
 Assertion expressions
 ascending/descending, 116
 assertion fail messages, 1030
 assertion profile command, 1005
 Assertion thread viewing
 enable, 1030
 assertion thread viewing
 actions, 1043
 annotating local variables, 1048
 enable, 1030
 expression hierarchy, 1045
 expression pane, 1040
 multiple clocks, 1045
 navigating atv window, 1039
 open

- from Assertions pane, 1037
 - from menu bar, 1037
 - from Message Viewer, 1038
 - from Wave window, 1037
- thread state, 1047
- thread viewer pane, 1041
- time in thread viewer pane, 1041
- AssertionActiveThreadMonitor .ini file variable, 1285
- AssertionActiveThreadMonitorLimit .ini file variable, 1285
- AssertionCover .ini file variable, 1286
- AssertionDebug .ini file variable, 1286, 1290
- AssertionEnable .ini file variable, 1286
- AssertionEnableVacuousPassActionBlock .ini file variable, 1287
- AssertionFailAction .ini file variable, 1287
- AssertionFailLocalVarLog .ini file variable, 1287
- AssertionFailLog .ini file variable, 1288
- AssertionLimit .ini file variable, 1288
- AssertionPassLog .ini file variable, 1288
- Assertions
 - break severity, 991, 1280, 1281
 - clockdeclarations, 1022
 - coding guidelines, 982
 - compare, 1010
 - compiling and simulating, 1025
 - cumulative threads, 1005
 - deferred, 1027
 - enable, 988
 - filter thread start times, 1037
 - filtering, 1064
 - immediate
 - deferred, 1027
 - memory profiling
 - enabling, 989
 - performance profiling
 - enabling, 989
 - save, 1032, 1067
 - set actions, 993
- assertions
 - active thread monitor, 1035
 - analyzing failures, 1034
 - analyzing in the GUI, 1002
 - binding, 548, 986
 - concurrent, 982
 - configuring, 987
 - during elab/optimize, 1026
 - embedded assertions, 1015
 - enable pass/fail logging, 991
 - external file, contained in, 1019
 - file and line number, 1323
 - immediate, 982
 - inline, 1015
 - library and use clauses, 1021
 - limitations, 1026
 - message display, 1281
 - message logging, 990
 - messages
 - alternate output file, 1013
 - turning off, 1388
 - multiclocked properties, 1022
 - recompile after changes, 1026
 - reporting, 1012
 - set fail limits, 992
 - setting format of messages, 1323
 - simulating, 997
 - SystemVerilog, 982
 - thread state, 1047
 - unlocked properties, 1022
 - view in Analysis window, 1002
 - viewing in Wave window, 1005
 - warnings, locating, 1323
- Assertions browser
 - display options, 1003
- Assertions window, 140
 - column descriptions, 141
- AssertionThreadLimit .ini file variable, 1289
- AssertionThreadLimitAction .ini file variable, 1289
- Assume directives
 - processing, 986
- assume directives
 - SimulateAssumeDirectives .ini variable, 1349
- Asymmetric encryption, 325
- AtLeast counts, PSL
 - configuring cover directives
 - AtLeast counts, 996

- attribute
 - definition of, [642](#)
 - ATV
 - ascending expressions, [116](#)
 - graphic symbols, [147](#)
 - highlighting
 - root thread analysis, [1046](#)
 - sub-expression failures, [1043](#)
 - toolbar, [115](#), [116](#)
 - view grid icon, [116](#)
 - atv actions
 - expand/contract hierarchy, [1045](#)
 - hover mouse
 - thread state, [1047](#)
 - view multiple clocks, [1045](#)
 - ATV log command, [1030](#)
 - atv window
 - actions, [1043](#)
 - annotating local variables, [1048](#)
 - expression hierarchy, [1045](#)
 - expression pane, [1040](#)
 - multiple clocks, [1045](#)
 - navigating, [1039](#)
 - thread state, [1047](#)
 - thread viewer pane, [1041](#)
 - thread viewer pane pane, [1041](#)
 - time in thread viewer pane pane, [1041](#)
 - ATVStartTimeKeepCount .ini file variable, [1289](#)
 - auto exclusion
 - fsm transitions, [934](#)
 - auto find bp command, [1141](#)
 - auto step mode, C Debug, [1141](#)
 - Autofill text entry
 - find, [78](#), [1163](#)
 - automatic command help, [282](#)
 - automatic saving of UCDB, [1097](#)
 - automating testbench, [1075](#)
 - program blocks, [1085](#)
- B —
- bad magic number error message, [705](#)
 - base (radix)
 - List window, [761](#)
 - Wave window, [752](#)
 - batch-mode simulations, [66](#)
 - BDD, [1076](#)
 - Binary Decision Diagram (BDD), [1076](#)
 - bind
 - hierarchical references, [551](#)
 - restrictions, [549](#)
 - port mapping, [551](#)
 - to VHDL enumerated types, [552](#)
 - what can be bound, [548](#), [549](#)
 - bind construct
 - limitations for SystemC, [496](#), [558](#)
 - SystemC binding
 - to Verilog/SV design unit, [496](#)
 - SystemVerilog, [548](#), [986](#)
 - bind statement
 - in compilation unit scope, [556](#)
 - syntax, [549](#)
 - BindAtCompile .ini file variable, [1290](#)
 - binding, VHDL, default, [394](#)
 - bins
 - names and unions, [1058](#)
 - bitwise format, [792](#)
 - black-boxing
 - blocking assignments, [443](#)
 - Bookmarks
 - clear all in Source window, [263](#)
 - Source window, [263](#)
 - bookmarks
 - Wave window, [742](#)
 - Boolean
 - failed expression, [1040](#)
 - bound block
 - hierarchical references, [551](#)
 - Break
 - on assertion, [1281](#)
 - break
 - stop simulation run, [120](#), [129](#)
 - Break severity
 - assertions, [1280](#)
 - for assertions, [991](#)
 - BreakOnAssertion .ini file variable, [1290](#)
 - Breakpoints
 - C code, [1137](#)
 - command execution, [260](#), [871](#)
 - conditional, [260](#), [870](#)

- use of SystemVerilog keyword *this*, 260
 - deleting, 257, 781
 - edit, 258, 778, 781
 - in SystemVerilog class methods, 260
 - load, 260
 - Run Until Here, 874
 - save, 260
 - saving/restoring, 782
 - set with GUI, 256
 - setting automatically in C code, 1141
 - Source window, viewing in, 242
 - SystemC constructor/destructor, 519
 - unavailable with vopt, 334
- .bsm file, 817, 843
- bubble diagram
 - using the mouse, 182, 183
- buffered/unbuffered output, 1375
- Buffers
 - in schematic, 805
- busses
 - RTL-level, reconstructing, 715
 - user-defined, 770
- buswise format, 792
- C —
- C applications
 - compiling and linking, 1419
 - debugging, 1135
- C Debug, 1135
 - auto find bp, 1141
 - auto step mode, 1141
 - debugging functions during elaboration, 1144
 - debugging functions when exiting, 1148
 - function entry points, finding, 1141
 - initialization mode, 1144
 - registered function calls, identifying, 1141
 - running from a DO file, 1137
 - Stop on quit mode, 1148
- C++ applications
 - compiling and linking, 1422
- Call Stack pane, 148
- cancelling scheduled events, performance, 418
- canonical name, definition, 929
- capacity analysis, 1165
- Case sensitivity
 - for VHDL and Verilog, 385, 423, 546
- case sensitivity
 - named port associations, 594
- Case statements
 - exclude allfalse branch, 936
- Causality
 - in schematic, 809
- Causality trace
 - path times bar, 893
- Causality traceback, 877
 - active driver path details, 881
 - from command line, 894
 - from GUI, 879
 - from Objects or Schematic window, 885
 - from Source window, 883
 - from specific time, 890
 - from Wave window, 880
 - multiple drivers, 891
 - post-sim debug, 879
 - set report destination, 895
 - setting preferences, 896
 - to all possible drivers, 889
 - to driving process, 885
 - to root cause, 887
 - trace cursor, 882
 - trace to seq process, 880
 - usage flow, 877
 - viewing path details
 - in Schematic window, 891
 - in Wave window, 893
- causality, tracing in Dataflow window, 839
- Cause
 - show, 877, 879, 880
- cdbg_wait_for_starting command, 1137
- cell libraries, 459
- change command
 - modifying local variables, 462
- chasing X, 814, 839
- check_synthesis argument
 - warning message, 1399
- CheckPlusargs .ini file variable, 1291
- checkpoint/restore
 - checkpointing a running simulation, 635
 - foreign C code and heap memory, 635

- CheckpointCompressMode .ini file variable, [1291](#)
- CheckSynthesis .ini file variable, [1292](#)
- cin support, SystemC, [536](#)
- Class objects
 - view in Wave window, [768](#)
- class of sc_signal<T>, [522](#)
- cleanup
 - SystemC state-based code, [514](#)
- clean-up of SystemC state-based code, [514](#)
- Clear bookmarks
 - source window, [263](#)
- CLI commands for debugging transactions, [676](#)
- Click and sprout
 - schematic window
 - incremental view, [232](#), [797](#)
- clock change, sampling signals at, [776](#)
- clock cycles
 - display in timeline, [749](#)
- Clock declarations, [1022](#)
- clock declarations
 - restrictions, [1023](#)
- clocked comparison, [789](#)
- Clocking block inout display, [297](#)
- clocks
 - assertions, [1045](#)
- Code Coverage, [274](#)
 - \$coverage_save system function, [466](#)
 - condition coverage, [900](#)
 - enabling with vsim, [903](#)
 - excluding lines/files, [934](#)
 - exclusion filter files
 - used in multiple simulation runs, [953](#)
 - expression coverage, [900](#), [911](#)
 - FSM coverage, [900](#)
 - Instance Coverage pane, [188](#)
 - Main window coverage data, [906](#)
 - missed coverage, [160](#)
 - pragma exclusions, [938](#), [950](#)
 - reports, [954](#)
 - Source window data, [247](#)
 - toggle coverage, [900](#)
 - toggle coverage in Signals window, [924](#)
 - toggle details, [165](#)
- code coverage
 - and optimization, [959](#)
 - branch coverage, [900](#)
 - condition coverage, [911](#)
 - design is not fully covered, [959](#)
 - see also* Coverage
 - statement coverage, [900](#)
- code coverage, elaboration time, [901](#)
- Code preview
 - in schematic window, [803](#)
- code profiling, [1151](#)
- Coding
 - assertion guidelines, [982](#)
- collapsing ports, and coverage reporting, [958](#)
- collapsing time and delta steps, [713](#)
- Color
 - for traces, [835](#)
 - radix
 - example, [83](#)
- colorization, in Source window, [264](#), [875](#)
- Colorize
 - Transcript window, [280](#)
- Column layout
 - configure, [119](#), [1452](#)
 - create, [1453](#)
 - edit, [1452](#)
- Combine Selected Signals dialog box, [198](#), [759](#)
- combining signals, busses, [770](#)
- command completion, [282](#)
- Command line
 - initiating causality traceback, [894](#)
- CommandHistory .ini file variable, [1293](#)
- command-line arguments, accessing, [537](#)
- command-line mode, [64](#)
- commands
 - event watching in DO file, [1267](#)
 - system, [1258](#)
 - vcd2wlf, [1245](#)
 - VSIM Tcl commands, [1262](#)
- comment character
 - Tcl and DO files, [1254](#)
- Commonly Used modelsim.ini Variables, [1386](#)
- Compare
 - assertions, [1010](#)
- compare

- add signals, [786](#)
- adding regions, [787](#)
- by signal, [786](#)
- clocked, [789](#)
- difference markers, [791](#)
- displayed in list window, [792](#)
- icons, [792](#)
- options, [789](#)
- pathnames, [790](#)
- reference dataset, [785](#)
- reference region, [787](#)
- tab, [786](#)
- test dataset, [786](#)
- timing differences, [791](#)
- tolerance, [789](#)
- values, [791](#)
- wave window display, [790](#)
- compare by region, [787](#)
- compare commands, [784](#)
- compare signal, virtual
 - restrictions, [770](#)
- compare simulations, [703](#)
- compilation
 - multi-file issues (SystemVerilog), [430](#)
 - SDF files, [1219](#)
- compilation unit scope, [430](#)
- Compile
 - encryption
 - 'include, [304](#)
 - VHDL, [384](#)
- compile
 - SystemC
 - reducing non-debug compile time, [498](#)
- Compile directive
 - encryption
 - 'include, [304](#)
- compile order
 - auto generate, [359](#)
 - changing, [358](#)
 - SystemVerilog packages, [426](#)
- Compiler Control Variable
 - SystemC
 - CppPath, [1300](#)
 - SccomLogfile, [1341](#)
 - SccomVerbose, [1342](#)
 - ScvPhaseRelationName, [1343](#)
 - UseScv, [1376](#)
- Compiler Control Variables
- Verilog
 - AcceptLowerCasePragmasOnly, [1284](#)
 - CoverCells, [1294](#)
 - EnableSVCoverpointExprVariable, [1306](#)
 - ExtendedToggleMode, [1308](#)
 - GenerateLoopIterationMax, [1313](#)
 - GenerateRecursionDepthMax, [1313](#)
 - Hazard, [1314](#)
 - LibrarySearchPath, [1321](#)
 - MultiFileCompilationUnit, [1329](#)
 - Protect, [1338](#)
 - Quiet, [1339](#)
 - Show_BadOptionWarning, [1344](#)
 - Show_Lint, [1344](#)
 - Show_PslChecksWarnings, [1345](#)
 - SparseMemThreshold, [1357](#)
 - SVCovergroupGoalDefault, [1361](#)
 - SVCovergroupPerInstanceDefault, [1363](#)
 - SVCovergroupSampleInfo, [1363](#)
 - SVCovergroupStrobeDefault, [1364](#)
 - SVCovergroupTypeGoalDefault, [1365](#)
 - SVCoverpointExprVariablePrefix, [1366](#)
 - SVCrossNumPrintMissingDefault, [1367](#)
 - UpCase, [1375](#)
 - vlog95compat, [1379](#)
- VHDL, [1308](#)
 - AmsStandard, [1284](#)
 - BindAtCompile, [1290](#)
 - CheckSynthesis, [1292](#)
 - CoverageFEC, [1296](#), [1299](#)
 - CoverageShortCircuit, [1298](#)
 - CoverSub, [1298](#)
 - EmbeddedPsl, [1306](#)
 - Explicit, [1308](#)
 - IgnoreVitalErrors, [1318](#)
 - NoCaseStaticError, [1329](#)
 - NoDebug, [1330](#)
 - NoIndexCheck, [1330](#)

- NoOthersStaticError, [1331](#)
- NoRangeCheck, [1331](#)
- NoVital, [1332](#)
- NoVitalCheck, [1332](#)
- Optimize_1164, [1334](#)
- PedanticErrors, [1336](#)
- RequireConfigForAllDefaultBinding, [1340](#)
- ScalarOpts, [1341](#)
- Show_source, [1345](#)
- Show_VitalChecksWarning, [1345](#)
- Show_Warning1, [1345](#)
- Show_Warning2, [1346](#)
- Show_Warning3, [1346](#)
- Show_Warning4, [1346](#)
- Show_Warning5, [1346](#)
- VHDL93, [1378](#)
- compiler directives, [474](#)
 - IEEE Std 1364-2000, [475](#)
 - XL compatible compiler directives, [477](#)
- CompilerTempDir .ini file variable, [1293](#)
- compiling
 - overview, [62](#)
 - changing order in the GUI, [358](#)
 - gensrc errors during, [542](#)
 - grouping files, [359](#)
 - order, changing in projects, [358](#)
 - properties, in projects, [367](#)
 - range checking in VHDL, [387](#)
 - SystemC, [497](#)
 - converting sc_main(), [531](#)
 - exporting top level module, [498](#)
 - for source level debug, [498](#)
 - invoking sccom, [498](#)
 - linking the compiled source, [511](#)
 - modifying source code, [531](#)
 - replacing sc_start(), [531](#)
 - using sccom vs. raw C++ compiler, [500](#)
 - Verilog, [422](#)
 - incremental compilation, [426](#)
 - XL 'uselib compiler directive, [432](#)
 - XL compatible options, [431](#)
 - VHDL, [383](#)
 - VITAL packages, [404](#)
- compiling C code, gcc, [1421](#)
- component declaration
 - generating SystemC from Verilog or VHDL, [618](#)
 - generating VHDL from Verilog, [590](#)
 - vgencomp for SystemC, [618](#)
 - vgencomp for VHDL, [590](#)
- component, default binding rules, [394](#)
- Compressing files
 - VCD tasks, [1242](#)
- concurrent assertions, [982](#)
- ConcurrentFileLimit .ini file variable, [1293](#)
- Conditional breakpoints, [260](#)
 - use of SystemVerilog keyword *this*, [260](#)
- conditional breakpoints
 - use of keyword *this*, [260](#)
- configuration
 - Verilog, support, [601](#)
- configuration simulator state variable, [1259](#)
- configurations
 - instantiation in mixed designs, [588](#)
 - Verilog, [434](#)
- Configure
 - column layout, [119](#), [1452](#)
 - encryption envelope, [300](#)
- confusing toggle numbers, [930](#)
- connectivity, exploring, [800](#), [833](#)
- Constants
 - VHDL, [595](#)
- Constrained random
 - initial seed value, [466](#)
- Constrained random stimulus, [1075](#)
- constrained random stimulus
 - constraints, [1076](#)
- Constrained random tests
 - enabling/disabling, [1083](#)
 - inheriting constraints, [1082](#)
 - rand and randc modifiers, [1077](#)
- Constraint algorithm
 - negative timing checks, [451](#)
- Constraint solver
 - set to previous release, [1084](#)
- constraint solver, [1075](#)
- constraints
 - random dynamic array size, [1080](#), [1351](#)
- construction parameters, SystemC, [538](#)

- Constructor
 - breakpoint, [519](#)
- Contains, [78, 80, 81](#)
- context menus
 - Library tab, [374](#)
- control function, SystemC, [559](#)
- control_foreign_signal() function, [548](#)
- Convergence
 - delay solution, [451](#)
- convert real to time, [407](#)
- convert time to real, [406](#)
- converting to a module, [531](#)
- Counts
 - assertion, [997](#)
- Cover directives
 - cumulative threads, [1005](#)
 - display options, [168](#)
 - filtering, [1064](#)
 - recursive display mode, [168](#)
 - save, [1032, 1067](#)
 - set action, [995](#)
 - viewing in Wave window, [1005](#)
- cover directives, [1012](#)
 - active thread monitor, [1035](#)
 - analyzing in the GUI, [1002](#)
 - AtLeast counts, [996](#)
 - change default configuration, [994](#)
 - count mode, [1009](#)
 - coverage percentage, [166](#)
 - limiting, [996](#)
 - SystemVerilog
 - cover directives, [1027](#)
 - viewing in Wave window, [1005](#)
 - weighting, [995](#)
- Cover directives browser
 - display options, [1004](#)
- Cover Directives tab
 - column descriptions, [167](#)
- Cover Directives window, [166](#)
- Coverage
 - during optimization, [343](#)
- coverage
 - auto exclusions
 - fsm transitions, [934](#)
 - data, automatic saving of, [1097](#)
 - directive, Analysis pane, [166](#)
 - editing UCDB, [1111](#)
 - enable FSMs, [968](#)
 - fsm exclusions, [949](#)
 - fsm states, [967](#)
 - fsm transitions, [967](#)
 - missing expression, [912](#)
 - ranking most effective tests, [1111](#)
 - setting default mode, [958](#)
 - UCDB, [1089](#)
- Coverage .ini file variable, [1293](#)
- coverage calculation for toggles, [929](#)
- Coverage Details window, [159, 162](#)
- Coverage exclude
 - allfalse branch in case stmt, [936](#)
- Coverage exclusion pragmas
 - fsm, [950](#)
- coverage numbers, mismatching, [249, 866](#)
- coverage reports, [954](#)
 - default mode, [958](#)
 - ensuring all signals appear, [958](#)
 - HML format, [959](#)
 - xml format, [958](#)
- coverage toggle_ignore pragma, [947](#)
- \$coverage_save system function, [466](#)
- CoverageFEC .ini file variable, [1296, 1299](#)
- CoverageShortCircuit .ini file variable, [1298](#)
- CoverAtLeast .ini file variable, [1294](#)
- CoverCells .ini file variable, [1294](#)
- CoverCountAll .ini file variable, [1295](#)
- CoverEnable .ini file variable, [1295](#)
- CoverExcludeDefault .ini file variable, [1295](#)
- covergroup types
 - reporting, [1062](#)
- Covergroups
 - add/modify filter, [1065](#)
 - create filter, [1064](#)
 - filter data, [1064](#)
 - filtering, [1064](#)
- Covergroups window, [169, 1060](#)
 - column descriptions, [170](#)
- CoverLimit .ini file variable, [1296](#)
- CoverLog .ini file variable, [1296](#)
- CoverOpt .ini file variable, [1297](#)

- CoverRespectHandL .ini file variable, [1297](#), [1298](#)
 - CoverSub .ini file variable, [1298](#)
 - CoverThreadLimit .ini file variable, [1299](#)
 - CoverThreadLimitAction .ini file variable, [1299](#)
 - CoverWeight .ini file variable, [1300](#)
 - covreport.xml, [959](#)
 - CppOptions .ini file variable, [1300](#)
 - CppPath .ini file variable, [1300](#)
 - Create column layout, [1453](#)
 - create debug, [796](#)
 - create debug database, [831](#)
 - CreateDirForFileAccess .ini file variable, [1301](#)
 - Creating do file, [86](#), [764](#), [782](#)
 - Cumulative Threads
 - assertions, [1005](#)
 - cover directives, [1005](#)
 - current exclusions
 - pragmas, [938](#)
 - Cursor
 - add, [726](#)
 - cursor linking, [293](#)
 - Cursors
 - linking, [728](#)
 - sync all active, [727](#)
 - cursors
 - adding, deleting, locking, naming, [724](#)
 - link to Dataflow window, [820](#), [847](#)
 - measuring time with, [727](#)
 - saving waveforms between, [765](#)
 - trace events with, [839](#)
 - Wave window, [727](#), [765](#)
 - Custom color
 - for trace, [835](#)
 - Custom column layout, [1453](#)
 - Customize
 - columns, [119](#), [1452](#)
 - Customize GUI
 - fonts, [1455](#)
 - customizing
 - via preference variables, [1453](#)
- D —
- deltas
 - explained, [395](#)
 - daemon
 - JobSpy, [1196](#)
 - dashed signal lines, [291](#)
 - database
 - post-sim debug, [796](#), [831](#)
 - Dataflow
 - post-sim debug database
 - create, [831](#)
 - post-sim debug flow, [830](#)
 - sprout limit readers, [834](#)
 - Dataflow window, [172](#), [829](#)
 - extended mode, [829](#)
 - pan, [824](#)
 - zoom, [824](#)
 - see also* windows, Dataflow window
 - dataflow.bsm file, [817](#), [843](#)
 - Dataset Browser, [710](#)
 - Dataset Snapshot, [712](#)
 - datasets, [703](#)
 - managing, [710](#)
 - opening, [708](#)
 - prevent dataset prefix display, [712](#)
 - reference, [785](#)
 - test, [786](#)
 - view structure, [708](#)
 - visibility, [709](#)
 - DatasetSeparator .ini file variable, [1301](#)
 - debug database
 - create, [796](#), [831](#)
 - debug flow
 - post-simulation, [797](#), [830](#)
 - debuggable SystemC objects, [516](#)
 - Debugging
 - randomize() failures, [1080](#)
 - debugging
 - assertion failures, [1034](#)
 - C code, [1135](#)
 - null value, [446](#)
 - SIGSEGV, [446](#)
 - SystemC channels and variables, [526](#)
 - debugging the design, overview, [63](#)
 - default binding
 - BindAtCompile .ini file variable, [1290](#)
 - disabling, [395](#)
 - default binding rules, [394](#)

- default clock
 - in assertions, [1022](#)
- default coverage mode, setting, [958](#)
- Default editor, changing, [1463](#)
- default SystemC parameter values, overriding, [538](#)
- DefaultForceKind .ini file variable, [1302](#)
- DefaultRadix .ini file variable, [1302](#)
- DefaultRestartOptions .ini file variable, [1303](#)
- DefaultRestartOptions variable, [1389](#)
- Deferred assertions, [1027](#)
- delay
 - delta delays, [395](#)
 - modes for Verilog models, [459](#)
- Delay solution convergence, [451](#)
- DelayFileOpen .ini file variable, [1304](#)
- deleting library contents, [373](#)
- delta collapsing, [713](#)
- delta simulator state variable, [1259](#)
- Delta time
 - recording
 - for expanded time viewing, [732](#)
- deltas
 - in List window, [774](#)
 - referencing simulator iteration
 - as a simulator state variable, [1259](#)
- dependent design units, [384](#)
- descriptions of HDL items, [262](#)
- design library
 - creating, [373](#)
 - logical name, assigning, [375](#)
 - mapping search rules, [376](#)
 - resource type, [371](#)
 - VHDL design units, [383](#)
 - working type, [371](#)
- design object icons, described, [76](#)
- design object visibility, +acc, [349](#)
- design optimization with vopt, [331](#)
- design portability and SystemC, [499](#)
- design units, [371](#)
- Destructor
 - breakpoint, [519](#)
- DEVICE
 - matching to specify path delays, [1225](#)
- dialogs
 - Runtime Options, [1278](#)
- Direct Programming Interface, [1407](#)
- Direct programming interface (DPI), [420](#)
- directive coverage, [166](#)
- directives
 - PSL
 - in procedural blocks, [1015](#)
- directories
 - moving libraries, [376](#)
- disable_signal_spy, [1177](#)
- Display mode
 - expanded time, [736](#)
 - recursive
 - Display mode
 - show all contexts, [1003](#)
- display mode
 - recursive, [168](#)
 - show all contextsCover directives
 - show all contexts display mode, [168](#)
- Display options
 - assertions browser, [1003](#)
 - cover directives, [168](#)
 - cover directives browser, [1004](#)
- display preferences
 - Wave window, [747](#)
- displaymsgmode .ini file variable, [1304](#)
- distributed delay mode, [460](#)
- dividers
 - Wave window, [753](#)
- DLL files, loading, [1419](#), [1422](#)
- DO files (macros)
 - creating from a saved transcript, [280](#)
 - error handling, [1270](#)
 - executing at startup, [1357](#), [1465](#)
 - parameters, passing to, [1267](#)
 - Tcl source command, [1270](#)
- DOPATH environment variable, [1463](#)
- DPI, [420](#)
 - and qverilog command, [1414](#)
 - export TFs, [1398](#)
 - missing DPI import function, [1415](#)
 - optimizing import call performance, [1417](#)
 - registering applications, [1412](#)
 - use flow, [1413](#)
- DPI access routines, [1435](#)

DPI export TFs, [1398](#)
 DPI/VPI/PLI, [1407](#)
 DpiCppPath .ini file variable, [1305](#)
 DpiOutOfTheBlue .ini file variable, [1305](#)
 Driver
 path details, [881](#)
 Drivers
 multiple, [891](#)
 drivers
 Dataflow Window, [833](#)
 show in Dataflow window, [777](#)
 Wave window, [777](#)
 dumpports tasks, VCD files, [1240](#)
 DumpportsCollapse .ini file variable, [1306](#)
 dynamic array, [1080](#), [1351](#)
 dynamic module array
 SystemC, [523](#)

— E —

Edit
 column layout, [1452](#)
 edit
 breakpoints, [258](#), [778](#), [781](#)
 Editing
 in notepad windows, [1442](#)
 in the Main window, [1442](#)
 in the Source window, [1442](#)
 Editing the modelsim.ini file, [1278](#)
 editing UCDBs, [1111](#)
 EDITOR environment variable, [1463](#)
 editor, default, changing, [1463](#)
 elab_defer_fli argument, [640](#)
 elaboration file
 creating, [638](#)
 loading, [638](#)
 modifying stimulus, [638](#)
 simulating with PLI or FLI models, [640](#)
 elaboration, and code coverage, [901](#)
 embedded wave viewer, [808](#), [837](#)
 EmbeddedPsl .ini file variable, [1306](#)
 empty port name warning, [1398](#)
 Enable
 assertion thread viewing, [1030](#)
 assertions, [988](#)
 enable_signal_spy, [1179](#)

EnableSVCoverpointExprVariable .ini file
 variable, [1306](#)
 EnableTypeOf .ini file variable, [1307](#)
 Encoding
 methods, [325](#)
 encrypt
 IP code
 public keys, [327](#)
 undefined macros, [309](#)
 vendor-defined macros, [311](#)
 IP source code, [299](#)
 usage models, [309](#)
 protect pragmas, [309](#)
 vencrypt utility, [309](#)
 vencrypt command
 header file, [310](#), [315](#)
 vlog +protect, [323](#)
 encrypting IP code
 vencrypt utility, [309](#)
 Encryption
 asymmetric, [325](#)
 compile with +protect, [307](#)
 configuring envelope, [300](#)
 creating envelope, [299](#)
 default asymmetric method for Questa, [325](#)
 default symmetric method for Questa, [325](#)
 envelopes
 how they work, [326](#)
 for multiple simulators
 Encryption
 portable, [305](#)
 language-specific usage, [309](#)
 methods, [325](#)
 proprietary compile directives, [321](#)
 protection expressions, [302](#)
 unsupported, [303](#)
 raw, [326](#)
 runtime model, [308](#)
 symmetric, [325](#)
 usage models for VHDL, [314](#)
 using 'include, [304](#)
 using Mentor Graphics public key, [327](#)
 vlog +protect, [312](#)
 encryption
 'protect compiler directive, [322](#), [475](#)

- securing pre-compiled libraries, 323
- Encryption and Encoding methods, 325
- end_of_construction() function, 537
- end_of_simulation() function, 537
- ENDFILE function, 401
- ENDLINE function, 400
- endpoints
 - in HDL code, 1023
- 'endprotect compiler directive, 322, 475
- entities
 - default binding rules, 394
- entity simulator state variable, 1259
- EnumBaseInit .ini file variable, 1307
- environment variables, 1462
 - expansion, 1462
 - referencing from command line, 1469
 - referencing with VHDL FILE variable, 1469
 - setting, 1462
 - setting in Windows, 1468
 - TranscriptFile, specifying location of, 1373
 - used in Solaris linking for FLI, 1419, 1422
 - used in Solaris linking for
 - PLI/VPI/DPI/FLI, 1464
 - using with location mapping, 1391
 - variable substitution using Tcl, 1258
- EOS Note column
 - assertion directives, 142
- error
 - can't locate C compiler, 1398
- error .ini file variable, 1307
- ErrorFile .ini file variable, 1308
- errors
 - "api_version" in, 542
 - bad magic number, 705
 - DPI missing import function, 1415
 - getting more information, 1394
 - libswift entry not found, 1401
 - multiple definition, 543
 - out-of-line function, 543
 - severity level, changing, 1394
 - SystemC loading, 540
 - SystemVerilog, missing declaration, 1329
 - Tcl_init error, 1400
 - void function, 543
 - VSIM license lost, 1401
 - zero-delay iteration loop, 1401
- escaped identifiers, 457, 594
 - Tcl considerations, 458
- EVCD files
 - exporting, 1213
 - importing, 1214
- event order
 - in optimized designs, 351
 - in Verilog simulation, 441
- event queues, 441
- Event time
 - recording
 - for expanded time viewing, 732
- Event traceback
 - toolbar button, 879
 - using schematic window, 809
- event watching commands, placement of, 1267
- events, tracing, 839
- Exclude
 - toggle nodes, 946
- Exclude allfalse
 - in case statements, 936
- exclusion filter files
 - excluding udp truth table rows, 935
 - used in multiple simulation runs, 953
- Exclusion pragmas
 - for FSM
 - simultaneous behavior, 952
 - simultaneous behavior - toggles, 947
- Exclusions
 - pragmas
 - fsm, 950
- exclusions
 - lines and files, 934
 - Z values, 920
- exit codes, 1396
- exiting tool on sc_stop or \$finish, 1334
- exiting tool, customizing, 1334
- Exlude
 - toggle node transitions, 946
- expand
 - environment variables, 1462
- expand net, 800, 833
- Expanded Time

- customizing Wave window, [735](#)
- expanding/collapsing sim time, [737](#)
 - with commands, [738](#)
 - with menu selections, [737](#)
 - with toolbar buttons, [737](#)
- in Wave and List, [730](#)
- recording, [731](#)
 - delta time, [732](#)
 - even time, [732](#)
- selecting display mode, [736](#)
 - with command, [737](#)
 - with menus, [736](#)
 - with toolbar buttons, [736](#)
- switching time mode, [737](#)
- terminology, [731](#)
- viewing in List window, [738](#)
- viewing in Wave window, [732](#)
- Explicit .ini file variable, [1308](#)
- export TFs, in DPI, [1398](#)
- Exporting SystemC modules
 - to Verilog, [607](#)
- exporting SystemC modules
 - to VHDL, [618](#)
- exporting top SystemC module, [498](#)
- Expression Builder, [745](#)
 - configuring a List trigger with, [775](#)
 - saving expressions to Tcl variable, [746](#)
- expression hierarchy
 - atv window, [1045](#)
- expression pane, [1040](#)
 - atv window, [1040](#)
- Expressions
 - ascending/descending, [116](#)
- expressions
 - missing coverage, cause, [912](#)
- extended identifiers
 - in mixed designs, [589](#), [617](#)
- Extended system tasks
 - Verilog, [473](#)
- extended toggles, conversion of, [928](#)
- ExtendedToggleMode .ini file variable, [1308](#)

— F —

- F8 function key, [1444](#)
- Failed boolean
 - ATV window, [1040](#)

- Fatal .ini file variable, [1309](#)
- Fatal error
 - SIGSEGV, [447](#)
- FecEffort .ini file variable, [1309](#)
- FIFOs, viewing SystemC, [523](#)
- File compression
 - VCD tasks, [1242](#)
- file compression
 - SDF files, [1217](#)
- file I/O
 - TextIO package, [397](#)
- file-line breakpoints, [256](#)
 - edit, [258](#), [781](#)
- files
 - .modelsim, [1459](#)
- Files window, [176](#), [274](#)
- files, grouping for compile, [359](#)
- Filter, [80](#)
 - add/modify
 - for covergroups, [1065](#)
 - assertion thread start times, [1037](#)
 - assertions/cover directives, [1064](#)
 - covergroups, [1064](#)
 - create
 - for cover directives, [1064](#)
 - setup
 - for assertions/cover directives, [1064](#)
 - covergroups, [1064](#)
- Filtering
 - Contains field, [78](#), [81](#)
- filtering
 - signals in Objects window, [219](#)
- filtering data in UCDB, [1129](#)
- filters
 - for Code Coverage
 - used in multiple simulation runs, [953](#)
- Find, [78](#)
 - in Structure window, [268](#)
 - prefill text entry field, [78](#), [1163](#)
 - stop, [744](#)
- find
 - inline search bar, [263](#), [281](#)
- Finite State Machines, [963](#)
 - coverage exclusions, [949](#)
 - enable coverage, [968](#)

enable recognition, [968](#)
 recognition reporting, [976](#)
 state coverage, [967](#)
 transition coverage, [967](#)
 unsupported design styles, [964](#)
 Fixed point radix, [85](#)
 fixed-point types, in SystemC
 compiling for support, [535](#)
 construction parameters for, [538](#)
 FLI, [393](#)
 debugging, [1135](#)
 floatfixlib .ini file variable, [1310](#)
 Flow
 causality tracing, [877](#)
 FocusFollowsMouse, [102](#)
 Folded instance
 schematic, [807](#)
 folders, in projects, [365](#)
 fonts
 scaling, [78](#)
 setting preferences, [1455](#)
 forall keyword, [1035](#)
 force command
 defaults, [1389](#)
 ForceUnsignedIntegerToVhdlInteger .ini file
 variable, [1310](#)
 foreign language interface, [393](#)
 foreign model loading
 SmartModels, [1471](#)
 foreign module declaration
 Verilog example, [602](#)
 VHDL example, [612](#)
 foreign module declaration, SystemC, [601](#)
 Format
 saving/restoring, [86](#), [764](#)
 signal
 Wave window, [291](#)
 format file, [763](#)
 Wave window, [763](#)
 FPGA libraries, importing, [380](#)
 FSM
 coverage exclusions, [950](#)
 view current state, [971](#)
 fsm transitions
 auto exclusions, [934](#)

FsmResetTrans .ini file variable, [1311](#)
 FsmSingle .ini file variable, [1311](#)
 FsmXAssign .ini file variable, [1312](#)
 Full view
 schematic
 display redundant buffers, [805](#)
 display redundant inverters, [805](#)
 Function call, debugging, [148](#)
 function calls, identifying with C Debug, [1141](#)
 Functional Coverage
 reports
 with timestamps, [1063](#)
 Functional coverage
 html report, [1063](#)
 reporting, [1061](#)
 text report, [1061](#)
 white box testing, [1049](#)
 functional coverage
 aggregation in Structure view, [1093](#)
 calculating metrics, [1051](#)
 reporting
 covergroup type objects, [1062](#)
 viewing statistics in the GUI, [1059](#)
 functions
 SystemC
 control, [559](#)
 observe, [559](#)
 unsupported, [534](#)
 virtual, [716](#)

— G —

-g C++ compiler option, [517](#)
 g++, alternate installations, [499](#)
 Gate-level
 metastability, [460](#)
 gdb debugger, [1135](#)
 generate statements, Veilog, [435](#)
 GenerateFormat .ini file variable, [1312](#)
 GenerateLoopIterationMax .ini file variable,
 [1313](#)
 GenerateRecursionDepthMax .ini variable,
 [1313](#)
 generic support
 SC instantiating VHDL, [612](#)
 generics

- passing to `sc_foreign_module` (VHDL), 612
 - SystemC instantiating VHDL, 612
 - VHDL, 567
 - generics, integer
 - passing as template arguments, 614
 - generics, integer and non-integer
 - passing as constructor arguments, 613
 - GenerousIdentifierParsing .ini file variable, 1313
 - get_resolution() VHDL function, 405
 - Global GUI changes
 - fonts, 1455
 - Global signal radix, 84, 218, 290, 753
 - global visibility
 - PLI/FLI shared objects, 1426
 - GLOBALPATHPULSE
 - matching to specify path delays, 1225
 - GlobalSharedObjectsList .ini file variable, 1314
 - Glob-style, 80
 - graphic interface, 719, 795, 829, 851
 - UNIX support, 67
 - Graphic symbols
 - ATV window, 147
 - Grid
 - ATV window, 116
 - Grid Engine
 - JobSpy integration, 1203
 - grouping files for compile, 359
 - grouping objects, Monitor window, 286
 - groups
 - in wave window, 756
 - GUI preferences
 - fonts, 1455
 - GUI_expression_format
 - GUI expression builder, 745
 - Guidelines
 - for coding assertions, 982
- H —
- hardware model interface, 1478
 - Hazard .ini file variable, 1314
 - hazards
 - limitations on detection, 445
 - HDL sub-expressions
 - failures in ATV window, 1043
 - help
 - command help, 282
 - Hierarchical access
 - mixed-language, 548
 - Hierarchical references, 546
 - bind, 551
 - supported objects, 550
 - SystemC/mixed-HDL designs, 559
 - SystemVerilog, 559
 - hierarchy
 - driving signals in, 1181
 - forcing signals in, 406, 1189
 - referencing signals in, 406, 1185
 - releasing signals in, 406, 1193
 - Highlight trace, 835
 - Highlighting
 - assertion thread
 - root thread analysis, 1046
 - highlighting, in Source window, 264, 875
 - Highlights
 - in Source window, 246, 864
 - history
 - of commands
 - shortcuts for reuse, 1441
 - hm_entity, 1479
 - HOLD
 - matching to Verilog, 1225
 - HOME environment variable, 1463
 - HTML format
 - coverage reports, 959
 - HTML report
 - functional coverage, 1063
 - Hypertext link, 243
- | —
- I/O
 - TextIO package, 397
 - icons
 - shapes and meanings, 76
 - identifiers
 - escaped, 457, 594
 - ieee .ini file variable, 1314
 - IEEE libraries, 378
 - IEEE Std 1076, 67
 - differences between versions, 390

- IEEE Std 1364, [67](#), [420](#), [481](#), [1079](#)
 - IEEE Std 1364-2005, [299](#), [1227](#)
 - IgnoreError .ini file variable, [1314](#)
 - IgnoreFailure .ini file variable, [1315](#)
 - IgnoreNote .ini file variable, [1315](#)
 - IgnorePragmaPrefix .ini file variable, [1315](#)
 - ignoreStandardRealVector .ini file variable
 - .ini compiler control variables
 - ignoreStandardRealVector, [1316](#)
 - IgnoreSVAError .ini file variable, [1316](#)
 - IgnoreSVAFatal .ini file variable, [1317](#)
 - IgnoreSVAInfo .ini file variable, [1317](#)
 - IgnoreSVAWarning .ini file variable, [1317](#)
 - IgnoreVitalErrors .ini file variable, [1318](#)
 - IgnoreWarning .ini file variable, [1318](#)
 - Immediate assertions
 - deferred, [1027](#)
 - immediate assertions, [982](#), [1374](#)
 - break severity, [1281](#)
 - configuring in GUI, [990](#)
 - message logging, [1316](#)
 - importing EVCD files, waveform editor, [1214](#)
 - importing FPGA libraries, [380](#)
 - IncludeRecursionDepthMax .ini file variable, [1319](#)
 - incremental compilation
 - automatic, [427](#)
 - manual, [427](#)
 - with Verilog, [426](#)
 - Incremental view
 - click and sprout, [232](#), [797](#)
 - schematic
 - display redundant buffers, [805](#)
 - display redundant inverters, [805](#)
 - schematic window
 - adding objects, [234](#), [798](#)
 - index checking, [387](#)
 - Inheriting constraints, [1082](#)
 - \$init_signal_driver, [1181](#)
 - init_signal_driver, [1181](#)
 - \$init_signal_spy, [1185](#)
 - init_signal_spy, [406](#), [1185](#)
 - init_usertfs function, [1147](#), [1409](#)
 - Initial random seed value, [466](#)
 - initialization of SystemC state-based code, [514](#)
 - initialization sequence, [1459](#)
 - inline assertions
 - PSL, [1015](#)
 - inline search bar, [263](#), [281](#)
 - inlining
 - VHDL subprograms, [387](#)
 - input ports
 - matching to INTERCONNECT, [1224](#)
 - matching to PORT, [1224](#)
 - Instance
 - folded/unfolded, [807](#)
 - instantiation in mixed-language design
 - Verilog from VHDL, [588](#)
 - VHDL from Verilog, [592](#)
 - instantiation in SystemC-Verilog design
 - SystemC from Verilog, [607](#)
 - Verilog from SystemC, [600](#)
 - instantiation in SystemC-VHDL design
 - VHDL from SystemC, [610](#)
 - instantiation in VHDL-SystemC design
 - SystemC from VHDL, [617](#)
 - INTERCONNECT
 - matching to input ports, [1224](#)
 - interconnect delays, [1231](#)
 - Inverters
 - in schematic, [805](#)
 - IOPATH
 - matching to specify path delays, [1224](#)
 - IP code
 - encrypt, [299](#)
 - public keys, [327](#)
 - undefined macros, [309](#)
 - vendor-defined macros, [311](#)
 - encryption usage models, [309](#), [314](#)
 - using protect pragmas, [309](#)
 - vencrypt usage models, [309](#)
 - iteration_limit, infinite zero-delay loops, [397](#)
 - IterationLimit .ini file variable, [1320](#)
- J —
- JobSpy
 - daemon, [1196](#)
- K —
- keyboard shortcuts
 - List window, [1446](#)

- Main window, [1442](#)
- Source window, [1442](#)
- user defined, [88](#)
- Wave window, [1446](#)
- keywords
 - SystemVerilog, [423](#)
- L —
- L work, [429](#)
- Language Reference Manual (LRM), [67](#), [420](#)
- language templates, [253](#), [853](#)
- language versions, VHDL, [390](#)
- Layout
 - columns, [119](#), [1452](#)
- Liberty Cell Libraries, optimizing, [345](#)
- libraries
 - 64-bit and 32-bit in same library, [379](#)
 - creating, [373](#)
 - design libraries, creating, [373](#)
 - design library types, [371](#)
 - design units, [371](#)
 - group use, setting up, [377](#)
 - IEEE, [378](#)
 - importing FPGA libraries, [380](#)
 - mapping
 - from the command line, [375](#)
 - from the GUI, [375](#)
 - hierarchically, [1387](#)
 - search rules, [376](#)
 - modelsim_lib, [405](#)
 - moving, [376](#)
 - multiple libraries with common modules, [429](#)
 - naming, [375](#)
 - others clause, [377](#)
 - predefined, [378](#)
 - refreshing library images, [379](#)
 - resource libraries, [371](#)
 - std library, [378](#)
 - Synopsys, [378](#)
 - Verilog, [428](#), [565](#), [1248](#)
 - VHDL library clause, [377](#)
 - working libraries, [371](#)
 - working vs resource, [61](#)
 - working with contents of, [373](#)
- library map file, Verilog configurations, [434](#)
- library mapping, overview, [61](#)
- library maps, Verilog 2001, [434](#)
- library simulator state variable, [1259](#)
- library, definition, [61](#)
- LibrarySearchPath .ini file variable, [1321](#)
- libsm, [1471](#)
- libswift, [1471](#)
 - entry not found error, [1401](#)
- License .ini file variable, [1321](#)
- licensing
 - License variable in .ini file, [1321](#)
- limit toggle coverage, [932](#)
- Limitations
 - PSL, [1026](#)
 - PSL in 0-in, [1015](#)
- Limiting WLF file, [706](#)
- Link cursors, [728](#)
- link cursors, [293](#)
- linking SystemC source, [511](#)
- List pane
 - see also* pane, List pane
- List window, [196](#), [722](#)
 - expanded time viewing, [738](#)
 - setting triggers, [775](#)
 - waveform comparison, [792](#)
 - see also* windows, List window
- LM_LICENSE_FILE environment variable, [1464](#)
- Load
 - breakpoints, [260](#)
- loading the design, overview, [63](#)
- Local variables
 - print to Transcript, [1030](#)
- local variables
 - annotating, [1048](#)
 - modifying with change command, [462](#)
- Locals window, [200](#)
 - see also* windows, Locals window
- location maps, referencing source files, [1391](#)
- locations maps
 - specifying source files with, [1391](#)
- lock message, [1399](#)
- locking cursors, [724](#)
- log file
 - overview, [703](#)

- see also* WLF files
- Log local variables, [1030](#)
- Logic Modeling
 - SmartModel
 - command channel, [1476](#)
 - SmartModel Windows
 - lmcwin commands, [1477](#)
 - memory arrays, [1478](#)
- long simulations
 - saving at intervals, [712](#)
- LRM, [67](#), [420](#)
- LSF
 - JobSpy integration, [1203](#)
- M —
- MacroNestingLevel simulator state variable, [1259](#)
- macros (DO files), [1266](#)
 - creating from a saved transcript, [280](#)
 - depth of nesting, simulator state variable, [1259](#)
 - error handling, [1270](#)
 - parameters
 - as a simulator state variable (n), [1259](#)
 - passing, [1267](#)
 - total number passed, [1259](#)
 - startup macros, [1388](#)
- Main window, [95](#)
 - see also* windows, Main window
- malloc()
 - checkpointing foreign C code, [635](#)
- mapping
 - data types, [562](#)
 - libraries
 - from the command line, [375](#)
 - hierarchically, [1387](#)
 - symbols
 - Dataflow window, [817](#), [843](#)
 - SystemC in mixed designs, [587](#)
 - SystemC to Verilog, [581](#)
 - SystemC to VHDL, [588](#)
 - Verilog states in mixed designs, [565](#)
 - Verilog states in SystemC designs, [580](#)
 - Verilog to SytemC, port and data types, [580](#)
 - Verilog to VHDL data types, [562](#)
 - VHDL to SystemC, [574](#)
 - VHDL to Verilog data types, [567](#)
- mapping libraries, library mapping, [375](#)
- mapping signals, waveform editor, [1214](#)
- math_complex package, [378](#)
- math_real package, [378](#)
- MaxReportRhsCrossProducts .ini file variable, [1322](#)
- MaxReportRhsSVCrossProducts .ini file variable, [1322](#)
- MaxSVCoverpointBinsDesign .ini file variable, [1322](#)
- MaxSVCoverpointBinsInst .ini file variable, [1322](#)
- MaxSVCrossBinsDesign .ini file variable, [1323](#)
- MaxSVCrossBinsInst .ini file variable, [1323](#)
- mc_scan_plusargs()
 - using with an elaboration file, [639](#)
- memories
 - navigation, [208](#)
 - saving formats, [206](#)
 - selecting memory instances, [206](#)
 - sparse memory modeling, [478](#)
 - viewing contents, [206](#)
 - viewing multiple instances, [206](#)
- memory
 - capacity analysis, [1165](#)
 - modeling in VHDL, [408](#)
- memory allocation
 - checkpointing foreign C code, [635](#)
- memory allocation profiler, [1152](#)
- memory leak, cancelling scheduled events, [418](#)
- memory leaks and transaction handles, [665](#)
- Memory profile data
 - viewing PSL, [1005](#)
- Memory profiling
 - assertions/cover directives, [989](#)
- memory tab
 - memories you can view, [204](#)
- merge
 - test-associated, [1114](#)
- merging
 - results from multiple simulation runs, [953](#)
- message logging
 - assertions, [990](#)
- message system, [1393](#)

- Message Viewer Display Options dialog box, [215](#)
- Message Viewer tab, [210](#)
- MessageFormat .ini file variable, [1323](#)
- MessageFormatBreak .ini file variable, [1324](#)
- MessageFormatBreakLine .ini file variable, [1324](#)
- MessageFormatError .ini file variable, [1325](#)
- MessageFormatFail .ini file variable, [1325](#)
- MessageFormatFatal .ini file variable, [1325](#)
- MessageFormatNote .ini file variable, [1326](#)
- MessageFormatWarning .ini file variable, [1326](#)
- Messages, [210](#)
- messages, [1393](#)
 - assertion failures, [1030](#)
 - bad magic number, [705](#)
 - empty port name warning, [1398](#)
 - exit codes, [1396](#)
 - getting more information, [1394](#)
 - lock message, [1399](#)
 - long description, [1394](#)
 - metavalue detected, [1399](#)
 - redirecting, [1373](#)
 - sensitivity list warning, [1399](#)
 - suppressing warnings from arithmetic packages, [1388](#)
 - Tcl_init error, [1400](#)
 - too few port connections, [1400](#)
 - turning off assertion messages, [1388](#)
 - VSIM license lost, [1401](#)
 - warning, suppressing, [1394](#)
- Metastability
 - approximating for Verilog, [460](#)
- metavalue detected warning, [1399](#)
- MFCU, [430](#)
- MGC_LOCATION_MAP env variable, [1391](#)
- MGC_LOCATION_MAP variable, [1464](#)
- MinGW gcc, [1421](#), [1423](#)
- mismatching coverage numbers, [249](#), [866](#)
- Missed Coverage pane, [160](#)
- missing DPI import function, [1415](#)
- missing expression coverage, [912](#)
- MixedAnsiPorts .ini file variable, [1326](#)
- Mixed-language
 - optimizing DPI import call performance, [1417](#)
- Mixed-language
 - hierarchical references, [546](#)
- mixed-language simulation, [545](#)
 - access limitations, [548](#)
- mode, setting default coverage, [958](#)
- MODEL_Tech environment variable, [1465](#)
- MODEL_Tech_Tcl environment variable, [1465](#)
- modeling memory in VHDL, [408](#)
- MODELSIM environment variable, [1465](#)
- modelsim_lib, [405](#)
- modelsim_lib .ini file variable, [1327](#)
- MODELSIM_PREFERENCES variable, [1456](#), [1465](#)
- modelsim.ini
 - found by the tool, [1460](#)
 - default to VHDL93, [1389](#)
 - delay file opening with, [1390](#)
 - editing,, [1278](#)
 - environment variables in, [1386](#)
 - force command default, setting, [1389](#)
 - hierarchical library mapping, [1387](#)
 - opening VHDL files, [1390](#)
 - restart command defaults, setting, [1389](#)
 - transcript file created from, [1387](#)
 - turning off arithmetic package warnings, [1388](#)
 - turning off assertion messages, [1388](#)
- modelsim.tcl, [1457](#)
- modes of operation, [64](#)
- Modified field, Project tab, [363](#)
- modify
 - breakpoints, [781](#)
- modifying local variables, [462](#)
- modifying UCDB data, [1111](#)
- modules
 - handling multiple, common names, [429](#)
 - with unnamed ports, [591](#)
- Monitor window
 - grouping/ungrouping objects, [286](#)
- mouse shortcuts
 - Main window, [1442](#)
 - Source window, [1442](#)

Wave window, 1446
 .mpf file, 353
 loading from the command line, 370
 order of access during startup, 1459
 msgmode .ini file variable, 1327
 msgmode variable, 210
 mti_cosim_trace environment variable, 1466
 mti_inhibit_inline attribute, 387
 MTI_SYSTEMC macro, 499
 MTI_TF_LIMIT environment variable, 1466
 MTI_VCO_MODE environment variable, 1466
 MTI_VOPT_FLOW, 1467
 mtiAvm .ini file variable, 1327
 mtiOvm .ini file variable, 1327
 mtiPA .ini file variable, 1328
 mtiUPF .ini file variable, 1328
 mtiUvm .ini file variable, 1328
 multi file compilation unit (MFCU), 430
 multiclocked assertions, 1022
 multi-file compilation issues, SystemVerilog, 430
 MultiFileCompilationUnit .ini file variable, 1329
 Multiple simulations, 703
 multiple test data records, troubleshooting, 1108
 MvcHome .ini file variable, 1329

— N —

n simulator state variable, 1259
 Name field
 Project tab, 362
 name visibility in Verilog generates, 435
 names, modules with the same, 429
 naming optimized designs, 332
 navigating
 atv window, 1039
 Negative timing
 algorithm for calculating delays, 448
 check limits, 448
 constraint algorithm, 451
 constraints, 449
 delay solution convergence, 451
 syntax for \$recrem, 450
 syntax for \$setuphold, 448

 using delayed inputs for checks, 456
 Negative timing checks, 448
 nets
 Dataflow window, displaying in, 172, 829
 values of
 displaying in Objects window, 217
 saving as binary log file, 703
 waveforms, viewing, 287
 new function
 initialize SV object handle, 446
 Nlview widget Symlib format, 818, 844
 NoCaseStaticError .ini file variable, 1329
 NOCHANGE
 matching to Verilog, 1227
 NoDebug .ini file variable, 1330
 NoDeferSubpgmCheck .ini file variable, 1330
 NoIndexCheck .ini file variable, 1330
 NOMMAP environment variable, 1467
 non-blocking assignments, 443
 Non-synthesizable, 795
 NoOthersStaticError .ini file variable, 1331
 NoRangeCheck .ini file variable, 1331
 note .ini file variable, 1331
 Notepad windows, text editing, 1442
 -notrigger argument, 776
 NoVital .ini file variable, 1332
 NoVitalCheck .ini file variable, 1332
 Now simulator state variable, 1259
 now simulator state variable, 1260
 null value
 debugging, 446
 numeric_bit package, 378
 numeric_std package, 378
 disabling warning messages, 1388
 NumericStdNoWarnings .ini file variable, 1332

— O —

object
 defined, 66
 Object handle
 initialize with new function, 446
 objects
 virtual, 714
 Objects window, 217
 objects, viewable SystemC, 516

- observe function, SystemC, [559](#)
 - observe_foreign_signal() function, [548](#)
 - OldVhdlForGenNames .ini file variable, [1333](#)
 - OnFinish .ini file variable, [1334](#)
 - OnFinishPendingAssert .ini file variable, [1334](#)
 - operating systems supported, *See Installation Guide*
 - Optimization
 - specifying coverage, [343](#)
 - Optimization Configurations, [365](#)
 - optimizations
 - design object visibility, [349](#)
 - event order issues, [351](#)
 - timing checks, [351](#)
 - Verilog designs, [331](#), [685](#)
 - VHDL subprogram inlining, [387](#)
 - Optimize_1164 .ini file variable, [1334](#)
 - optimized designs
 - naming, [332](#)
 - optimized designs, unnamed, [334](#)
 - order of events
 - in optimized designs, [351](#)
 - ordering files for compile, [358](#)
 - organizing projects with folders, [365](#)
 - OSCI simulator, differences with vsim, [534](#)
 - Others clause
 - libraries, [377](#)
 - Override
 - with toggle add, [932](#)
 - overview, simulation tasks, [58](#)
- P —
- packages
 - standard, [378](#)
 - textio, [378](#)
 - util, [405](#)
 - VITAL 1995, [402](#)
 - VITAL 2000, [402](#)
 - page setup
 - Dataflow window, [822](#), [848](#)
 - pan, Dataflow window, [824](#)
 - parallel transaction
 - definition of, [643](#)
 - ParallelJobs .ini file variable, [1335](#)
 - parameter support
 - SC instantiating Verilog, [602](#)
 - SystemC instantiating Verilog, [602](#)
 - Verilog instantiating SC, [608](#)
 - Verilog instantiating SystemC, [608](#)
 - parameters
 - making optional, [1268](#)
 - passing from Verilog to SC, [608](#)
 - passing to sc_foreign_module (Verilog), [603](#)
 - using with macros, [1267](#)
 - parameters (Verilog to SC)
 - passing as constructor arguments, [603](#)
 - passing integer as template arguments, [604](#)
 - path delay mode, [460](#)
 - path delays,matching to DEVICE statements, [1225](#)
 - path delays,matching to GLOBALPATHPULSE statements, [1225](#)
 - path delays,matching to IOPATH statements, [1224](#)
 - path delays,matching to PATHPULSE statements, [1225](#)
 - Path times bar
 - causality trace, [893](#)
 - pathnames
 - comparisons, [790](#)
 - hiding in Wave window, [748](#)
 - PATHPULSE
 - matching to specify path delays, [1225](#)
 - PathSeparator .ini file variable, [1335](#)
 - PedanticErrors .ini file variable, [1336](#)
 - performance
 - cancelling scheduled events, [418](#)
 - improving for Verilog simulations, [331](#), [685](#)
 - Performance profiling
 - assertions/cover directives, [989](#)
 - PERIOD
 - matching to Verilog, [1227](#)
 - phase transaction
 - definition of, [643](#)
 - platforms
 - choosing 32- versus 64-bit, [1466](#)
 - platforms supported, *See Installation Guide*
 - PLI

- design object visibility and auto +acc, 349
- loading shared objects with global symbol visibility, 1426
- specifying which apps to load, 1410
- Veriuser entry, 1410
- PLI/VPI, 481
 - debugging, 1135
 - tracing, 1436
- PLI/VPI/DPI, 1407
 - registering DPIapplications, 1412
 - specifying the DPI file to load, 1425
- PliCompatDefault .ini file variable, 1336
- PLIOBJS environment variable, 1410, 1467
- plusargs
 - changing behavior of, 1291
- PORT
 - matching to input ports, 1224
- port collapsing, toggle coverage, 958
- Port driver data, capturing, 1245
- ports, unnamed, in mixed designs, 591
- ports, VHDL and Verilog, 565
- Postscript
 - saving a waveform in, 764
 - saving the Dataflow display in, 821, 847
- Post-sim debug
 - causality tracing, 879
- post-sim debug flow, 797, 830
- Pragma
 - FSM exclusion
 - simultaneous behavior, 952
 - toggle exclusion
 - simultaneous behavior, 947
- pragmas, 938
 - protecting IP code, 309
 - synthesis pragmas, 1284
- precision
 - in timescale directive, 439
 - simulator resolution, 439
- precision, simulator resolution, 558
- PrefCoverage(DefaultCoverageMode), 958
- PrefCoverage(pref_InitFilterFrom), 937
- preference variables
 - .ini files, located in, 1283
 - editing, 1453
 - saving, 1453
- Preferences
 - schematic, 236
- preferences
 - saving, 1453
 - Wave window display, 747
- Prefill text entry
 - find, 78, 1163
- PrefMemory(ExpandPackedMem) variable, 205
- Preoptimized Design Unit, 331, 337
- PreserveCase .ini file variable, 1337
- preserving case of VHDL identifiers, 1337
- primitives, symbols in Dataflow window, 817, 843
- printing
 - Dataflow window display, 821, 847
 - waveforms in the Wave window, 764
- printing simulation stats, 1338
- PrintSimStats .ini file variable, 1338
- PrintSVPackageLoadingAttribute .ini file variable, 1338
- Procedural blocks
 - PSL directives, 1015
- Profile data
 - memory usage
 - assertions, 1005
 - cover directives, 1005
- profile report command, 1163
- Profiler, 1151
 - %parent fields, 1158
 - clear profile data, 1154
 - enabling memory profiling, 1152
 - enabling statistical sampling, 1154
 - getting started, 1152
 - handling large files, 1153
 - Hierarchical View, 1157
 - interpreting data, 1155
 - memory allocation, 1152
 - memory allocation profiling, 1154
 - profile report command, 1163
 - Profile Report dialog, 1164
 - Ranked View, 1156
 - report option, 1163
 - results, viewing, 1155
 - statistical sampling, 1152

- Structural View, 1159
 - viewing profile details, 1160
 - Profiling
 - assertions/cover directives, 989
 - program blocks, 1085
 - Programming Language Interface, 481, 1407
 - project tab
 - sorting, 363
 - project window
 - information in, 362
 - projects, 353
 - accessing from the command line, 370
 - adding files to, 356
 - benefits, 353
 - close, 362
 - compile order, 358
 - changing, 358
 - compiler properties in, 367
 - compiling files, 357
 - creating, 355
 - creating simulation configurations, 363
 - folders in, 365
 - grouping files in, 359
 - loading a design, 360
 - MODELSIM environment variable, 1465
 - open and existing, 362
 - overview, 353
 - Proprietary compile directives
 - encryption, 321
 - protect
 - source code, 299
 - Protect .ini file variable, 1338
 - 'protect compiler directive, 322, 475
 - protect pragmas
 - encrypting IP code, 309
 - protected types, 410
 - Protection expressions, 302
 - PSL
 - assertion debug, 1029
 - assertion messages
 - alternate output file, 1013
 - assertions
 - embedded, 1015
 - in external file, 1019
 - inline, 1015
 - library and use clauses, 1021
 - reporting, 1012
 - view in Analysis window, 1002
 - viewing in Wave window, 1005
 - clock declarations
 - default clock, 1022
 - multi-clocked properties, 1022
 - partially clocked, 1022
 - restrictions, 1023
 - clockdeclarations
 - unlocked properties, 1022
 - compiling
 - elaboration/optimization, 1026
 - embedded assertions, 1025
 - external assertions, 1026
 - recompile after changes, 1026
 - configuring assertions, 987
 - enable pass/fail logging, 991
 - set actions, 993
 - set fail limits, 992
 - configuring cover directives
 - change default configuration, 994
 - limiting, 996
 - weighting, 995
 - cover directives
 - count mode, 1009
 - endpoints
 - in HDL code, 1023
 - limitations, 1026
 - limitations in 0-in, 1015
 - replicator parameters, 1035
 - simulating assertions, 997
- PSL assertions
 - clock declarations, 1022
 - compiling and simulating, 1025
- PSL directives
 - in procedural blocks, 1015
- PsInfinityThreshold .ini file variable, 1339
- PsOneAttempt .ini file variable, 1339
- Public encryption key, 327
- Public encryption keys, 327
- Q —
- quick reference
 - table of simulation tasks, 58
 - Quiet .ini file variable, 1339

- qverilog
 - one-step flow, 421
- qverilog command
 - DPI support, 1414
- R —
- race condition, problems with event order, 441
- Radix
 - change in Watch pane, 283
 - color
 - example, 83
 - set globally, 84, 218, 290, 753
 - setting fixed point, 85
 - setting for Objects window, 84, 218
- radix
 - List window, 761
 - SystemVerilog types, 289, 752
 - user-defined, 81
 - definition body, 82
 - Wave window, 752
- Radix define command
 - setting radix color, 83
- rand
 - SystemVerilog class modifier, 1077
- randc
 - supported types, 1077
- Random constraints
 - stability across versions, 1084
- random dynamic array, 1080, 1351
- random number generator
 - seeding, 1084
- Random stimulus
 - constrained, 1075
- random stimulus
 - constraints, 1076
- randomize, 1077
- randomize()
 - solveflags attribute, 1079
- randomize() failures
 - debugging, 1080
- range checking, 387
- rank most effective tests, 1111
- ranking
 - most effective tests, 1111
- Raw encryption, 326
- Readers
 - sprout limit in dataflow, 834
- readers and drivers, 800, 833
- real type, converting to time, 407
- Recall breakpoints, 782
- reconstruct RTL-level design busses, 715
- Recording
 - expanded time, 731
- RECOVERY
 - matching to Verilog, 1226
- RECREM
 - matching to Verilog, 1226
- Recursive display mode, 168, 1003
- redirecting messages, TranscriptFile, 1373
- Redundant buffers
 - in schematic, 805
- Redundant inverters
 - in schematic, 805
- reference region, 787
- refreshing library images, 379
- regions
 - virtual, 717
- registered function calls, 1141
- registers
 - values of
 - displaying in Objects window, 217
 - saving as binary log file, 703
 - waveforms, viewing, 287
- regular-expression, 81
- REMOVAL
 - matching to Verilog, 1226
- renaming items in UCDB, 1111
- replicator parameters, 1035
- Reporting
 - causality trace, 895
- reporting, 1012
 - code coverage, 954
- Reports
 - functional coverage, 1061
 - html, 1063
 - text, 1061
- reports
 - fsm recognition, 976
- RequireConfigForAllDefaultBinding .ini file
 - variable, 1340
- resolution

- in SystemC simulation, 512
 - mixed designs, 558
 - returning as a real, 405
 - truncated value, 513
 - truncated values, 441, 513, 1262
 - verilog simulation, 439
 - VHDL simulation, 393
 - Resolution .ini file variable, 1340
 - resolution simulator state variable, 1260
 - Resolving VCD values, 1247
 - when force cmd used, 1247
 - resource libraries
 - specifying, 377
 - restart command
 - defaults, 1389
 - in GUI, 109
 - Restore
 - breakpoints, 782
 - Restoring
 - window format, 86, 764
 - results, saving simulations, 703
 - return to VSIM prompt on sc_stop or \$finish, 1334
 - Root cause, 887
 - Root thread analysis
 - assertions, 1046
 - RTL-level design busses
 - reconstructing, 715
 - RunLength .ini file variable, 1341
 - Runtime
 - encryption, 308
 - Runtime Options dialog, 1278
- S —
- Save
 - assertions, 1032, 1067
 - breakpoints, 260
 - cover directives, 1032, 1067
 - saveLines preference variable, 280
 - Saving
 - window format, 86, 764
 - saving
 - simulation options in a project, 363
 - waveforms, 703
 - sc_argc() function, 537
 - sc_argv() function, 537
 - sc_clock() functions, moving, 531
 - sc_cycle() function, 534
 - sc_fifo, 523
 - sc_fix and sc_ufix, 538
 - sc_fixed and sc_ufixed, 538
 - sc_foreign_module, 611
 - sc_foreign_module (Verilog)
 - and parameters, 603
 - sc_foreign_module (VHDL)
 - and generics, 612
 - sc_main(), 531
 - sc_main() function, 491
 - sc_main() function, converting, 531
 - SC_MODULE_EXPORT macro, 498
 - sc_signal() functions, moving, 531
 - sc_signed and sc_unsigned, 538
 - sc_start() function, 534
 - sc_start() function, replacing in SystemC, 534
 - sc_start(), replacing, 531
 - sc_stop()
 - behavior of, 538
 - behavior, customizing, 1334
 - ScalarOpts .ini file variable, 1341
 - scaling fonts, 78
 - sccom
 - using sccom vs. raw C++ compiler, 500
 - sccom -link command, 511, 618
 - SccomLogfile .ini file variable, 1341
 - SccomVerbose .ini file variable, 1342
 - scgenmod, using, 601, 611
 - Schematic
 - click and sprout, 232, 797
 - code preview, 803
 - display preferences, 236
 - folding/unfolding instances, 807
 - post-sim debug database
 - create, 796
 - post-sim debug flow, 797
 - show drivers/readers, 800
 - tooltip, 803
 - views, 232, 797
 - Schematic window, 795
 - add objects to incr view, 234, 798
 - display redundant buffers, 805
 - display redundant inverters, 805

- path times bar, [893](#)
- show causality path, [891](#)
- synthesizable parts of design, [795](#)
- ScMainFinishOnQuit .ini file variable, [1342](#)
- ScMainStackSize .ini file variable, [1343](#)
- ScTimeUnit .ini file variable, [1343](#)
- ScvPhaseRelationName .ini variable, [1342](#), [1343](#)
- SDF, [63](#)
 - compiled SDF, [1219](#)
 - disabling timing checks, [1231](#)
 - errors and warnings, [1219](#)
 - instance specification, [1218](#)
 - interconnect delays, [1231](#)
 - mixed VHDL and Verilog designs, [1231](#)
 - specification with the GUI, [1218](#)
 - troubleshooting, [1232](#)
 - Verilog
 - \$sdf_annotate system task, [1223](#)
 - optional conditions, [1230](#)
 - optional edge specifications, [1229](#)
 - rounded timing values, [1230](#)
 - SDF to Verilog construct matching, [1224](#)
 - VHDL
 - resolving errors, [1221](#)
 - SDF to VHDL generic matching, [1220](#)
- SDF annotate
 - \$sdf_annotate system task, [1221](#)
- SDF annotation
 - matching single timing check, [1233](#)
- SDF DEVICE
 - matching to Verilog constructs, [1225](#)
- SDF GLOBALPATHPULSE
 - matching to Verilog constructs, [1225](#)
- SDF HOLD
 - matching to Verilog constructs, [1225](#)
- SDF INTERCONNECT
 - matching to Verilog constructs, [1224](#)
- SDF IOPATH
 - matching to Verilog constructs, [1224](#)
- SDF NOCHANGE
 - matching to Verilog constructs, [1227](#)
- SDF PATHPULSE
 - matching to Verilog constructs, [1225](#)
- SDF PERIOD
 - matching to Verilog constructs, [1227](#)
- SDF PORT
 - matching to Verilog constructs, [1224](#)
- SDF RECOVERY
 - matching to Verilog constructs, [1226](#)
- SDF RECREM
 - matching to Verilog constructs, [1226](#)
- SDF REMOVAL
 - matching to Verilog constructs, [1226](#)
- SDF SETUPHOLD
 - matching to Verilog constructs, [1226](#)
- SDF SKEW
 - matching to Verilog constructs, [1226](#)
- SDF WIDTH
 - matching to Verilog constructs, [1227](#)
- \$sdf_done, [470](#)
- Search
 - in Structure window, [268](#)
 - prefill text entry field, [78](#), [1163](#)
 - stop, [744](#)
- search
 - inline search bar
 - Source window, [263](#)
 - Transcript, [281](#)
- Search bar, [78](#)
- searching
 - Expression Builder, [745](#)
 - Verilog libraries, [428](#), [594](#)
- seeding random number generator, [1084](#)
- sensitivity list warning, [1399](#)
- SeparateConfigLibrary .ini file variable, [1344](#)
- Setting radix
 - fixed point, [85](#)
- SETUP
 - matching to Verilog, [1225](#)
- SETUPHOLD
 - matching to Verilog, [1226](#)
- Severity
 - break on assertion, [991](#)
 - break on assertions, [1280](#)
- severity, changing level for errors, [1394](#)
- SFCU, [430](#)
- Share
 - user-defined types, [595](#)

- shared library
 - building in SystemC, [108](#), [511](#)
- shared objects
 - loading FLI applications
 - see FLI Reference manual
 - loading PLI/VPI/DPI C applications, [1419](#)
 - loading PLI/VPI/DPI C++ applications, [1422](#)
 - loading with global symbol visibility, [1426](#)
- Shortcuts
 - text editing, [1442](#)
- shortcuts
 - command history, [1441](#)
 - command line caveat, [1441](#)
 - List window, [1446](#)
 - Main window, [1442](#)
 - Source window, [1442](#)
 - Wave window, [1446](#)
- Show All Contexts display mode, [168](#)
- Show all possible drivers
 - Drivers
 - show all possible, [889](#)
- Show cause, [877](#), [879](#), [880](#)
 - active driver path details, [881](#)
 - from command line, [894](#)
 - from GUI, [879](#)
 - from Objects or Schematic window, [885](#)
 - from Source window, [883](#)
 - from specific time, [890](#)
 - from Wave window, [880](#)
 - in Schematic window, [891](#)
 - in Wave window, [893](#)
 - multiple drivers, [891](#)
 - set report destination, [895](#)
 - setting preferences
 - Preferences
 - causality traceback, [896](#)
 - trace cursor, [882](#)
- Show driver, [885](#)
- show drivers
 - Dataflow window, [833](#)
 - Wave window, [777](#)
- Show drivers/readers
 - schematic, [800](#)
- Show root cause, [887](#)
- Show_BadOptionWarning .ini file variable, [1344](#)
- Show_Lint .ini file variable, [1344](#)
- Show_PslChecksWarnings .ini file variable, [1345](#)
- Show_source .ini file variable, [1345](#)
- Show_VitalChecksWarning .ini file variable, [1345](#)
- Show_Warning1 .ini file variable, [1345](#)
- Show_Warning2 .ini file variable, [1346](#)
- Show_Warning3 .ini file variable, [1346](#)
- Show_Warning4 .ini file variable, [1346](#)
- Show_Warning5 .ini file variable, [1346](#)
- ShowConstantImmediateAsserts .ini file variable, [1347](#)
- ShowFunctions .ini file variable, [1347](#)
- ShowUnassociatedScNameWarning .ini file variable, [1347](#)
- ShowUndebuggableScTypeWarning .ini file variable, [1348](#)
- ShutdownFile .ini file variable, [1348](#)
- Signal
 - create virtual, [771](#)
- signal breakpoints
 - edit, [778](#)
- Signal format
 - Wave window, [291](#)
- signal groups
 - in wave window, [756](#)
- signal interaction
 - Verilog and SystemC, [574](#)
- Signal radix
 - for Objects window
 - Objects window
 - setting signal radix, [84](#), [218](#)
 - set globally, [84](#), [218](#), [290](#), [753](#)
- Signal Segmentation Violations
 - debugging, [446](#)
- Signal Spy, [406](#), [1185](#)
 - disable, [1177](#)
 - enable, [1179](#)
 - using in PSL assertions, [1021](#)
- \$signal_force, [1189](#)
- signal_force, [406](#), [1189](#)
- \$signal_release, [1193](#)

- signal_release, [406](#), [1193](#)
- signals
 - combining into a user-defined bus, [770](#)
 - dashed, [291](#)
 - Dataflow window, displaying in, [172](#), [829](#)
 - driving in the hierarchy, [1181](#)
 - filtering in the Objects window, [219](#)
 - hierarchy
 - driving in, [1181](#)
 - referencing in, [406](#), [1185](#)
 - releasing anywhere in, [1193](#)
 - releasing in, [406](#), [1193](#)
 - sampling at a clock change, [776](#)
 - transitions, searching for, [740](#)
 - types, selecting which to view, [219](#)
 - values of
 - displaying in Objects window, [217](#)
 - forcing anywhere in the hierarchy, [406](#), [1189](#)
 - saving as binary log file, [703](#)
 - virtual, [715](#)
 - waveforms, viewing, [287](#)
- SignalSpyPathSeparator .ini file variable, [1348](#)
- SIGSEGV
 - fatal error message, [447](#)
- SIGSEGV error, [446](#)
- SimulateAssumeDirectives .ini file variable, [1349](#)
- SimulateImmedAsserts .ini file variable, [1349](#)
- SimulatePSL .ini file variable, [1349](#)
- SimulateSVA .ini file variable, [1350](#)
- simulating
 - batch mode, [64](#)
 - command-line mode, [64](#)
 - comparing simulations, [703](#)
 - default run length, [1280](#)
 - iteration limit, [1280](#)
 - mixed language designs
 - compilers, [546](#)
 - libraries, [546](#)
 - resolution limit in, [558](#)
 - mixed Verilog and SystemC designs
 - channel and port type mapping, [574](#)
 - Verilog port direction, [580](#)
 - Verilog state mapping, [580](#)
 - mixed Verilog and VHDL designs
 - Verilog parameters, [564](#)
 - Verilog state mapping, [565](#)
 - VHDL and Verilog ports, [565](#)
 - VHDL generics, [567](#)
 - mixed VHDL and SystemC designs
 - SystemC state mapping, [587](#)
 - VHDL port direction, [587](#)
 - VHDL port type mapping, [582](#)
 - VHDL sc_signal data type mapping, [583](#)
 - saving dataflow display as a Postscript file, [821](#), [847](#)
 - saving options in a project, [363](#)
 - saving simulations, [703](#)
 - saving waveform as a Postscript file, [764](#)
 - SystemC, [489](#), [511](#)
 - usage flow for SystemC only, [491](#)
 - Verilog, [439](#)
 - delay modes, [459](#)
 - hazard detection, [445](#)
 - optimizing performance, [331](#), [685](#)
 - resolution limit, [439](#)
 - XL compatible simulator options, [457](#)
 - VHDL, [388](#)
 - viewing results in List pane, [196](#)
 - viewing results in List window, [722](#)
 - VITAL packages, [404](#)
- simulating the design, overview, [63](#)
- simulation
 - basic steps for, [60](#)
 - time, current, [1259](#), [1260](#)
- Simulation Configuration
 - creating, [363](#)
- simulation task overview, [58](#)
- simulations
 - event order in, [441](#)
 - saving results, [703](#)
 - saving results at intervals, [712](#)
- Simulator Control Variables
 - SystemVerilog
 - SVAPrintOnlyUserMessage, [1360](#)
- simulator resolution
 - mixed designs, [558](#)
 - returning as a real, [405](#)

- SystemC, [512](#)
- Verilog, [439](#)
- VHDL, [393](#)
- simulator state variables, [1259](#)
- simulator, difference from OSCI, [534](#)
- single file compilation unit (SFCU), [430](#)
- sizeof callback function, [1430](#)
- SKEW
 - matching to Verilog, [1226](#)
- sm_entity, [1472](#)
- SmartModels
 - creating foreign architectures with sm_entity, [1472](#)
 - invoking SmartModel specific commands, [1476](#)
 - linking to, [1471](#)
 - lmcwin commands, [1477](#)
 - memory arrays, [1478](#)
 - Verilog interface, [1478](#)
 - VHDL interface, [1471](#)
- so, shared object file
 - loading PLI/VPI/DPI C applications, [1419](#)
 - loading PLI/VPI/DPI C++ applications, [1422](#)
- SolveACTbeforeSpeculate .ini file variable, [1350](#)
- SolveACTMaxOps .ini file variable, [1350](#)
- SolveACTMaxTests .ini file variable, [1351](#)
- SolveACTRetryCount .ini file variable, [1351](#)
- SolveArrayResizeMax .ini file variable, [1351](#)
- SolveEngine .ini file variable, [1352](#)
- SolveFailDebug .ini file variable, [1352](#)
- SolveFailDebugMaxSet .ini file variable, [1352](#)
- SolveFailSeverity .ini file variable, [1353](#)
- Solveflags
 - attribute, [1079](#)
- SolveFlags .ini file variable, [1353](#)
- SolveGraphMaxEval .ini file variable, [1353](#)
- SolveGraphMaxSize .ini file variable, [1354](#)
- SolveIgnoreOverflow .ini file variable, [1354](#)
- solver failures, [1083](#)
- SolveRev .ini file variable, [1354](#)
- SolveSpeculateDistFirst .ini file variable, [1355](#)
- SolveSpeculateFirst .ini file variable, [1355](#)
- SolveSpeculateLevel .ini file variable, [1356](#)
- SolveSpeculateMaxCondWidth .ini file variable, [1356](#)
- SolveSpeculateMaxIterations .ini file variable, [1357](#)
- Source
 - textual connectivity, [252](#)
- Source annotation
 - Annotation, [250](#), [859](#)
- source annotation, [250](#), [859](#)
- source code mismatches
 - bypassing warning, [1109](#)
- Source code pragmas, [950](#)
- source code pragmas, [938](#)
- source code, security, [322](#), [323](#), [475](#)
- source files
 - Debug, [250](#), [859](#)
- source files, referencing with location maps, [1391](#)
- source files, specifying with location maps, [1391](#)
- source highlighting, customizing, [264](#), [875](#)
- source libraries
 - arguments supporting, [431](#)
- Source window, [242](#), [851](#)
 - clear highlights, [246](#), [864](#)
 - code coverage data, [247](#)
 - colorization, [264](#), [875](#)
 - inline search bar, [263](#)
 - Run Until Here, [874](#)
 - tab stops in, [264](#), [875](#)
 - see also* windows, Source window
- source-level debug
 - SystemC, enabling, [517](#)
- sparse memory modeling, [478](#)
- SparseMemThreshold .ini file variable, [1357](#)
- specify path delays
 - matching to DEVICE construct, [1225](#)
 - matching to GLOBALPATHPULSE construct, [1225](#)
 - matching to IOPATH statements, [1224](#)
 - matching to PATHPULSE construct, [1225](#)
- Sprout limit
 - readers in dataflow, [834](#)
- Stability
 - across versions, [1084](#)

- Standard Delay Format (SDF), [63](#)
- standards supported, [67](#)
- start_of_simulation() function, [537](#)
- Startup
 - macro in the modelsim.ini file, [1357](#)
- startup
 - files accessed during, [1459](#)
 - macros, [1388](#)
 - startup macro in command-line mode, [64](#)
 - using a startup file, [1388](#)
- Startup .ini file variable, [1357](#)
- State machines
 - view current state, [971](#)
- state variables, [1259](#)
- statement coverage
 - count for "for" loops, [909](#)
- statistical sampling profiler, [1152](#)
- statistics, toggle
 - confusing, [930](#)
- status bar
 - Main window, [101](#)
- Status field
 - Project tab, [362](#)
- std .ini file variable, [1358](#)
- std_arith package
 - disabling warning messages, [1388](#)
- std_developerskit .ini file variable, [1358](#)
- std_logic_arith package, [378](#)
- std_logic_signed package, [378](#)
- std_logic_textio, [378](#)
- std_logic_unsigned package, [378](#)
- StdArithNoWarnings .ini file variable, [1358](#)
- STDOUT environment variable, [1467](#)
- steps for simulation, overview, [60](#)
- Stimulus
 - constrained random, [1075](#)
- stimulus
 - modifying for elaboration file, [638](#)
- Stop wave drawing, [744](#)
- stripping levels in UCDB, [1111](#)
- struct of sc_signal<T>, [522](#)
- structure, [628](#)
- Structure window
 - find item, [268](#)
- subprogram inlining, [387](#)
- subprogram write is ambiguous error, fixing, [399](#)
- substreams
 - definition of, [643](#)
- suppress .ini file variable, [1359](#)
- SuppressFileTypeReg .ini file variable, [1359](#)
- suspended thread, [1140](#)
- sv_seed, [1084](#)
- Sv_Seed .ini variable, [1359](#)
- sv_std .ini file variable, [1360](#)
- SVA
 - viewing in Wave window, [1005](#)
- SVAPrintOnlyUserMessage .ini file variable, [1360](#)
- SVCovergroupGetInstCoverageDefault .ini file variable, [1360](#)
- SVCovergroupGoal .ini file variable, [1361](#)
- SVCovergroupGoalDefault .ini file variable, [1361](#)
- SVCovergroupMergeInstancesDefault .ini file variable, [1362](#)
- SVCovergroupPerInstanceDefault .ini file variable, [1363](#)
- SVCovergroupSampleInfo .ini file variable, [1363](#)
- SVCovergroupStrobe .ini file variable, [1364](#)
- SVCovergroupStrobeDefault .ini file variable, [1364](#)
- SVCovergroupTypeGoal .ini file variable, [1364](#)
- SVCovergroupTypeGoalDefault .ini file variable, [1365](#)
- SVCovergroupZWNNoCollectl .ini file variable, [1365](#)
- SVCoverpointAutoBinMax .ini file variable, [1366](#)
- SVCoverpointExprVariablePrefix .ini file variable, [1366](#)
- SVCrossNumPrintMissing .ini file variable, [1367](#)
- SVCrossNumPrintMissingDefault .ini file variable, [1367](#)
- SVFileExtensions .ini file variable, [1368](#)
- Svlog .ini file variable, [1368](#)
- symbol mapping

- Dataflow window, [817](#), [843](#)
- symbolic link to design libraries (UNIX), [376](#)
- symbols
 - ATV window, [147](#)
- Symmetric encryption, [325](#)
- Sync active cursors, [727](#)
- SyncCompilerFiles .ini file variable, [1369](#)
- Synopsis hardware modeler, [1478](#)
- synopsys .ini file variable, [1368](#)
- Synopsys libraries, [378](#)
- syntax highlighting, [264](#), [875](#)
- synthesis
 - pragmas, [1284](#)
 - rule compliance checking, [1292](#), [1298](#), [1306](#)
- Synthesizable, [795](#)
- System calls
 - VCD, [1240](#)
- system calls
 - Verilog, [461](#)
- system commands, [1258](#)
- System functions
 - \$coverage_save, [465](#)
- System tasks
 - VCD, [1240](#)
- system tasks
 - proprietary, [466](#)
 - Verilog, [461](#)
 - Verilog-XL compatible, [471](#)
- SystemC
 - aggregates of signals/ports, [522](#)
 - calling member import functions in SC scope, [625](#)
 - cin support, [536](#)
 - compiling for source level debug, [498](#)
 - compiling optimized code, [498](#)
 - component declaration for instantiation, [618](#)
 - construction parameters, [538](#)
 - constructor/destructor breakpoints, [519](#)
 - control function, [559](#)
 - converting sc_main(), [531](#)
 - converting sc_main() to a module, [531](#)
 - debugging of channels and variables, [526](#)
 - declaring/calling member import functions in SV, [624](#)
 - dynamic module array, [523](#)
 - exporting sc_main, example, [532](#)
 - exporting top level module, [498](#)
 - fixed-point types, [538](#)
 - foreign module declaration, [601](#)
 - generic support, instantiating VHDL, [612](#)
 - hierarchical references in mixed designs, [559](#)
 - instantiation criteria in Verilog design, [607](#)
 - instantiation criteria in VHDL design, [617](#)
 - linking the compiled source, [511](#)
 - maintaining design portability, [499](#)
 - mapping states in mixed designs, [587](#)
 - VHDL, [588](#)
 - memories, viewing, [524](#)
 - mixed designs with Verilog, [545](#)
 - mixed designs with VHDL, [545](#)
 - observe function, [559](#)
 - parameter support, Verilog instances, [602](#)
 - prim channel aggregates, [522](#)
 - reducing non-debug compile time, [498](#)
 - replacing sc_start(), [531](#)
 - sc_clock(), moving to SC_CTOR, [531](#)
 - sc_fifo, [523](#)
 - sc_signal(), moving to SC_CTOR, [531](#)
 - signals, viewing global and static, [522](#)
 - simulating, [511](#)
 - source code, modifying for vsim, [531](#)
 - stack space for threads, [540](#)
 - state-based code, initializing and cleanup, [514](#)
 - troubleshooting, [540](#)
 - unsupported functions, [534](#)
 - user defined signals and ports, viewable, [516](#)
 - viewable objects, [516](#)
 - viewable types, [515](#)
 - viewable/debuggable objects, [516](#)
 - viewing FIFOs, [523](#)
 - virtual functions, [514](#)
- SystemC binding
 - to Verilog/SV design unit, [496](#), [548](#), [986](#)
- SystemC modules

- exporting for use in Verilog, 607
 - exporting for use in VHDL, 618
 - SystemVerilog, 260
 - assertions, 982
 - concurrent, 982
 - immediate, 982
 - reporting, 1012
 - class methods, conditional breakpoints, 260
 - configuring assertions, 987
 - set actions, 993
 - set fail limits, 992
 - configuring cover directives
 - AtLeast counts, 996
 - limiting, 996
 - weighting, 995
 - cover directives
 - count mode, 1009
 - hierarchical references, 559
 - keyword considerations, 423
 - multi-file compilation, 430
 - object handle
 - initialize with new function, 446
 - random number generator, 1084
 - randomize, 1077
 - system functions
 - \$coverage save, 465
 - SystemVerilog bind
 - limitations for SystemC, 496, 558
 - SystemVerilog classes
 - constrained random tests, 1076
 - view in Wave window, 768
 - SystemVerilog DPI
 - specifying the DPI file to load, 1425
 - SystemVerilog types
 - radix, 289, 752
- T —
- tab stops
 - Source window, 264, 875
 - Tcl, ?? to 1264
 - command separator, 1257
 - command substitution, 1256
 - command syntax, 1252
 - evaluation order, 1257
 - history shortcuts, 1441
 - preference variables, 1453
 - relational expression evaluation, 1257
 - time commands, 1262
 - variable
 - substitution, 1258
 - VSIM Tcl commands, 1262
 - with escaped identifiers, 458
 - Tcl_init error message, 1400
 - temp files, VSOUT, 1469
 - template arguments
 - passing integer generics as, 614
 - passing integer parameters as, 604
 - terminology
 - for expanded time, 731
 - test data records, unique, 1108
 - test-associated merge, 1114
 - testbench
 - automating, 1075
 - testbench, accessing internal objectsfrom, 1175
 - Text
 - filtering, 80
 - text and command syntax, 69
 - Text editing, 1442
 - Text report
 - functional coverage, 1061
 - TEXTIO
 - buffer, flushing, 401
 - TextIO package
 - alternative I/O files, 401
 - containing hexadecimal numbers, 400
 - dangling pointers, 400
 - ENDFILE function, 401
 - ENDLINE function, 400
 - file declaration, 398
 - implementation issues, 399
 - providing stimulus, 401
 - standard input, 399
 - standard output, 399
 - WRITE procedure, 399
 - WRITE_STRING procedure, 400
 - Textual connectivity
 - in the Source window, 252
 - TF routines, 1435
 - TFMPC
 - explanation, 1400
 - thread

- debugging, [1140](#)
- thread viewer pane, [1041](#)
 - atv window, [1041](#)
 - understanding time, [1041](#)
- threads
 - assertions, [1035](#)
 - cover directives, [1035](#)
- time
 - current simulation time as a simulator statevariable, [1259](#), [1260](#)
 - measuring in Wave window, [727](#)
 - resolution in SystemC, [512](#)
 - time resolution as a simulator state variable, [1260](#)
 - truncated values, [441](#), [513](#), [1262](#)
- time collapsing, [713](#)
- Time mode switching
 - expanded time, [737](#)
- time resolution
 - in mixed designs, [558](#)
 - in Verilog, [439](#)
 - in VHDL, [393](#)
- Time stamps
 - func coverage, [1063](#)
- time type
 - converting to real, [406](#)
 - in mixed designs, [568](#)
- time unit
 - in SystemC, [512](#)
- timeline
 - display clock cycles, [749](#)
- timescale directive warning
 - investigating, [440](#)
- Timestamp
 - func coverage reports, [1063](#)
- timing
 - differences shown by comparison, [791](#)
 - disabling checks, [1231](#)
 - in optimized designs, [351](#)
- Timing checks
 - delay solution convergence, [451](#)
 - negative
 - constraint algorithm, [451](#)
 - constraints, [449](#)
 - syntax for \$crem, [450](#)
 - syntax for \$setuphold, [448](#)
 - using delayed inputs for checks, [456](#)
 - negative check limits, [448](#)
- TMPDIR environment variable, [1467](#)
- to_real VHDL function, [406](#)
- to_time VHDL function, [407](#)
- Toggle add
 - override +cover=tx, [932](#)
- toggle counts, understanding, [928](#)
- Toggle coverage
 - excluding node transitions, [946](#)
 - excluding nodes, [946](#)
- toggle coverage, [924](#)
 - 2-state, [924](#)
 - 3-state, [924](#)
 - count limit, [1369](#)
 - excluding bus bits, [947](#)
 - excluding enum signals, [948](#)
 - extended and regular, coverage computation, [929](#)
 - limiting, [932](#)
 - max VHDL integer values, [1369](#), [1370](#)
 - port collapsing, [958](#)
 - reporting, duplication of elements, [958](#)
 - reporting, ordering of nodes, [958](#)
 - Verilog/SV supported data types, [925](#)
 - viewing in Signals window, [924](#)
- toggle coverage, and performance, [924](#)
- Toggle node transitions
 - exlcude from coverage, [946](#)
- toggle numbers, confusing, [930](#)
- toggle, extended
 - conversion of, [928](#)
- ToggleCountLimit .ini file variable, [1369](#)
- ToggleFixedSizeArray .ini file variable, [1369](#)
- ToggleMaxFixedSizeArray .ini file variable, [1370](#)
- ToggleMaxIntValues .ini file variable, [1370](#)
- ToggleMaxRealValues .ini file variable, [1371](#)
- ToggleNoIntegers .ini file variable, [1371](#)
- TogglePackedAsVec .ini file variable, [1371](#)
- TogglePortsOnly .ini file variable, [1372](#)
- ToggleVlogEnumBits .ini file variable, [1372](#)
- ToggleVlogIntegers .ini file variable, [1372](#)
- ToggleVlogReal .ini file variable, [1373](#)

- ToggleWidthLimit .ini file variable, [1373](#)
 - tolerance
 - leading edge, [789](#)
 - trailing edge, [789](#)
 - too few port connections, explanation, [1400](#)
 - tool structure, [57](#)
 - Toolbar
 - filter, [80](#)
 - toolbar
 - ATV, [115](#), [116](#)
 - Main window, [114](#)
 - Toolbar buttons
 - event traceback, [879](#)
 - Tooltip
 - schematic, [803](#)
 - trace cursor, [882](#)
 - Tracing
 - causality, [877](#), [882](#)
 - active driver path details, [881](#)
 - from command line, [894](#)
 - from GU, [879](#)
 - from Objects or Schematic window, [885](#)
 - from Source window, [883](#)
 - from specific time, [890](#)
 - from Wave window, [880](#)
 - multiple drivers, [891](#)
 - post-sim debug, [879](#)
 - set report destination, [895](#)
 - setting preferences, [896](#)
 - to all possible drivers, [889](#)
 - to driving process, [885](#)
 - to first seq process, [880](#)
 - to root cause, [887](#)
 - usage flow, [877](#)
 - viewing path details
 - in Schematic window, [891](#)
 - in Wave window, [893](#)
 - events
 - schematic, [809](#)
 - tracing
 - events, [839](#)
 - source of unknown, [814](#), [839](#)
 - transaction
 - definition of, [642](#)
 - transaction handles and memory leaks, [665](#)
 - transactions
 - CLI commands for debugging, [676](#)
 - parallel, [643](#)
 - phase, [643](#)
 - Transcript
 - colorize, [280](#)
 - inline search bar, [281](#)
 - transcript
 - command help, [282](#)
 - disable file creation, [281](#), [1388](#)
 - file name, specified in modelsim.ini, [1387](#)
 - saving, [280](#)
 - saving as a DO file, [280](#)
 - Transcript window
 - changing buffer size, [280](#)
 - changing line count, [280](#)
 - TranscriptFile .ini file variable, [1373](#)
 - triggers, in the List window, [775](#)
 - triggers, in the List window, setting, [772](#)
 - troubleshooting
 - DPI, missing import funtion, [1415](#)
 - multiple test data records, [1108](#)
 - SystemC, [540](#)
 - unexplained behaviors, SystemC, [540](#)
 - TSSI
 - in VCD files, [1245](#)
 - type
 - converting real to time, [407](#)
 - converting time to real, [406](#)
 - Type field, Project tab, [363](#)
 - type-based coverage
 - constructor parameters, [1056](#)
 - Types
 - sharing user-defined, [595](#)
 - types
 - virtual, [717](#)
 - types, fixed-point in SystemC, [535](#)
 - types, viewable SystemC, [515](#)
- U —
- UCDB, [1089](#)
 - automatic saving of, [1097](#)
 - editing, [1111](#)
 - loading into current simulation
 - \$load_coverage_db() function, [1070](#)

- modifying contents, [1111](#)
 - stripping and adding levels, [1111](#)
- UCDB filtering, [1129](#)
- UCDBFilename .ini file variable, [1374](#)
- UDP, [422](#), [426](#), [428](#), [429](#), [437](#), [439](#), [459](#)
- UnattemptedImmediateAssertions .ini file variable, [1374](#)
- UnbufferedOutput .ini file variable, [1375](#)
- unlocked
 - assertion properties, [1022](#)
- undefined symbol, error, [541](#)
- unexplained behavior during simulation, [540](#)
- unexplained simulation behavior, [540](#)
- unfolded instance
 - schematic, [807](#)
- ungrouping
 - in wave window, [758](#)
- ungrouping objects, Monitor window, [286](#)
- unit delay mode, [460](#)
- unknowns, tracing, [814](#), [839](#)
- unnamed designs, [334](#)
- unnamed ports, in mixed designs, [591](#)
- unsupported functions in SystemC, [534](#)
- UpCase .ini file variable, [1375](#)
- Usage flow
 - causality tracing, [877](#)
- usage models
 - encrypting IP code, [309](#)
 - vencrypt utility, [309](#)
- use clause, specifying a library, [378](#)
- use flow
 - Code Coverage, [901](#)
 - DPI, [1413](#)
 - SystemC-only designs, [491](#)
- user-defined bus, [714](#), [770](#)
- user-defined primitive (UDP), [422](#), [426](#), [428](#), [429](#), [437](#), [439](#), [459](#)
- user-defined radix, [81](#)
 - definition body, [82](#)
- User-defined types
 - sharing, [595](#)
- UserTimeUnit .ini file variable, [1375](#)
- UseScv .ini file variable, [1376](#)
- UseSVCrossNumPrintMissing .ini file variable, [1376](#)

- util package, [405](#)
- UVM-Aware debug
 - UVMControl .ini file variable, [1376](#)
- UVMControl .ini file variable, [1376](#)

— V —

- values
 - of HDL items, [262](#)
- variables
 - editing,, [1278](#)
 - environment, [1462](#)
 - expanding environment variables, [1462](#)
 - LM_LICENSE_FILE, [1464](#)
 - modelsim.ini, [1283](#)
 - reading from the .ini file, [1261](#)
 - setting environment variables, [1462](#)
 - simulator state variables
 - iteration number, [1259](#)
 - name of entity or module as a variable, [1259](#)
 - resolution, [1259](#)
 - simulation time, [1259](#)
 - values of
 - displaying in Objects window, [217](#)
 - saving as binary log file, [703](#)
- VCD files
 - capturing port driver data, [1245](#)
 - case sensitivity, [1236](#)
 - creating, [1235](#)
 - dumpports tasks, [1240](#)
 - exporting created waveforms, [1213](#)
 - from VHDL source to VCD output, [1242](#)
 - stimulus, using as, [1237](#)
 - supported TSSI states, [1245](#)
 - translate into WLF, [1245](#)
 - VCD system tasks, [1240](#)
- VCD values
 - resolving, [1247](#)
 - when force cmd used, [1247](#)
- vcd2wlf command, [1245](#)
- vencrypt command
 - header file, [310](#), [315](#)
- Verification
 - with constrained random stimulus, [1075](#)
- Verilog
 - ACC routines, [1433](#)

- approximating metastability, [460](#)
- capturing port driver data with -dumpports, [1245](#)
- case sensitivity, [423](#), [546](#)
- cell libraries, [459](#)
- compiler directives, [474](#)
- compiling and linking PLI C applications, [1419](#)
- compiling and linking PLI C++ applications, [1422](#)
- compiling design units, [422](#)
- compiling with XL 'uselib compiler directive, [432](#)
- component declaration, [590](#)
- configuration support, [601](#)
- configurations, [434](#)
- DPI access routines, [1435](#)
- event order in simulation, [441](#)
- extended system tasks, [473](#)
- force and release, [457](#)
- generate statements, [435](#)
- instantiation criteria in mixed-language design, [588](#)
- instantiation criteria in SystemC design, [600](#)
- instantiation of VHDL design units, [592](#)
- language templates, [253](#)
- library usage, [428](#)
- mapping states in mixed designs, [565](#)
- mapping states in SystemC designs, [580](#)
- mixed designs with SystemC, [545](#)
- mixed designs with VHDL, [545](#)
- parameter support, instantiating SystemC, [608](#)
- parameters, [564](#)
- port direction, [580](#)
- resource libraries, [377](#)
- sdf_annotate system task, [1221](#)
- simulating, [439](#)
 - delay modes, [459](#)
 - XL compatible options, [457](#)
- simulation hazard detection, [445](#)
- simulation resolution limit, [439](#)
- SmartModel interface, [1478](#)
- source code viewing, [242](#)
- standards, [67](#)
- system tasks, [461](#)
- TF routines, [1435](#)
- to SystemC, channel and port type mapping, [574](#)
- XL compatible compiler options, [431](#)
- XL compatible routines, [1436](#)
- XL compatible system tasks, [471](#)
- verilog .ini file variable, [1377](#)
- Verilog 2001
 - disabling support, [1379](#)
- Verilog PLI/VP/DPII
 - registering VPI applications, [1411](#)
- Verilog PLI/VPI
 - 64-bit support in the PLI, [1436](#)
 - debugging PLI/VPI code, [1437](#)
- Verilog PLI/VPI/DPI
 - compiling and linking PLI/VPI C++ applications, [1422](#)
 - compiling and linking PLI/VPI/CPI C applications, [1419](#)
 - PLI callback reason argument, [1428](#)
 - PLI support for VHDL objects, [1431](#)
 - registering PLI applications, [1409](#)
 - specifying the PLI/VPI file to load, [1424](#)
- Verilog/SV supported data types
 - for toggle coverage, [925](#)
- Verilog-XL
 - compatibility with, [419](#), [1049](#)
- Veriuser .ini file variable, [1377](#), [1410](#)
- Veriuser, specifying PLI applications, [1410](#)
- veriuser.c file, [1430](#)
- Version compaitbility
 - constraint solver, [1084](#)
- Version compatibility
 - constraint solver, [1084](#)
- VHDL
 - .ini compiler control variables
 - ShowConstantImmediateAsserts, [1347](#)
 - case sensitivity, [385](#), [546](#)
 - compile, [384](#)
 - compiling design units, [383](#)
 - conditions and expressions, automatic conversion of H and L., [1297](#)
 - constants, [595](#)

- creating a design library, 383
- delay file opening, 1390
- dependency checking, 384
- encryption, 314
- file opening delay, 1390
- foreign language interface, 393
- hardware model interface, 1478
- instantiation criteria in SystemC design, 610
- instantiation from Verilog, 592
- instantiation of Verilog, 562
- language templates, 253
- language versions, 390
- library clause, 377
- mixed designs with SystemC, 545
- mixed designs with Verilog, 545
- object support in PLI, 1431
- optimizations
 - inlining, 387
- port direction, 587
- port type mapping, 582
- resource libraries, 377
- sc_signal data type mapping, 583
- simulating, 388
- SmartModel interface, 1471
- source code viewing, 242, 852
- standards, 67
- timing check disabling, 388
- variables
 - logging, 1378
 - viewing, 1378
- VITAL package, 378
- VHDL binding
 - to Verilog/SV design unit, 548, 986
- VHDL utilities, 405, 406, 1185
 - get_resolution(), 405
 - to_real(), 406
 - to_time(), 407
- VHDL-1987, compilation problems, 390
- VHDL-1993
 - enabling support for, 1378
- VHDL-2002
 - enabling support for, 1378
- VHDL-2008
 - package STANDARD
 - REAL_VECTOR, 1316
- VHDL93 .ini file variable, 1378
- VhdlVariableLogging .ini file variable, 1378
- view assertion failures, 1034
- View grid
 - ATV window, 116
- viewing, 210
 - library contents, 373
 - waveforms, 703
- viewing FIFOs, 523
- Viewing files for the simulation, 176
- Views
 - schematic, 232, 797
- virtual compare signal, restrictions, 770
- virtual functions in SystemC, 514
- virtual hide command, 715
- virtual objects, 714
 - virtual functions, 716
 - virtual regions, 717
 - virtual signals, 715
 - virtual types, 717
- virtual region command, 717
- virtual regions
 - reconstruct RTL hierarchy, 717
- virtual save command, 716
- Virtual signal
 - create, 771
- virtual signal command, 715
- virtual signals
 - reconstruct RTL-level design busses, 715
 - reconstruct the original RTL hierarchy, 715
 - virtual hide command, 715
- visibility
 - column in structure tab, 709
 - design object and +acc, 349
 - design object and vopt, 331
 - of declarations in \$unit, 430
- VITAL
 - compiling and simulating with accelerated VITAL packages, 404
 - compliance warnings, 403
 - disabling optimizations for debugging, 404
 - specification and source code, 402
 - VITAL packages, 403
- vital2000 .ini file variable, 1379

vl_logic, 1248
 vlog, 920
 vlog command
 +protect argument, 312, 323
 vlog95compat .ini file variable, 1379
 Vopt
 suppress warning messages, 1395
 vopt
 and breakpoints, 334
 vopt command, 331, 685
 Vopt Control Variables
 ParallelJobs, 1335
 VoptFlow, 1379
 VoptFlow .ini file variable, 1379
 VPI, registering applications, 1411
 VPI/PLI, 481
 VPI/PLI/DPI, 1407
 compiling and linking C applications, 1419
 compiling and linking C++ applications, 1422
 VSIM license lost, 1401
 VSIM prompt, returning to, 1334
 vsim, differences with OSCI simulator, 534
 VSOUT temp file, 1469

— W —

WarnConstantChange .ini file variable, 1380
 warning .ini file variable, 1380
 warning #6820, 1109
 warnings
 disabling at time 0, 1389
 empty port name, 1398
 exit codes, 1396
 getting more information, 1394
 messages, long description, 1394
 metavalue detected, 1399
 multiple test data records, 1108
 severity level, changing, 1394
 suppressing VCOM warning messages, 1394
 suppressing VLOG warning messages, 1395
 suppressing VOPT warning messages, 1395
 suppressing VSIM warning messages, 1396

Tcl initialization error 2, 1400
 too few port connections, 1400
 turning off warnings from arithmetic packages, 1388
 waiting for lock, 1399
 Wave drawing
 stop, 744
 wave groups, 756
 add items to existing, 758
 creating, 756
 deleting, 758
 drag from Wave to List, 759
 drag from Wave to Transcript, 759
 removing items from existing, 758
 ungrouping, 758
 Wave Log Format (WLF) file, 703
 wave log format (WLF) file
 see also WLF files
 wave viewer, Dataflow window, 808, 837
 Wave window, 287, 720
 compare waveforms, 790
 cursor linking, 728
 customizing for expanded time, 735
 expanded time viewing, 730, 732
 format signal, 291
 in the Dataflow window, 808, 837
 saving layout, 763
 show causality, 893
 sync active cursors, 727
 timeline
 display clock cycles, 749
 values column, 791
 view SV class objects, 768
 see also windows, Wave window
 wave window
 dashed signal lines, 291
 Waveform Compare
 adding clocks, 788
 adding regions, 787
 adding signals, 786
 annotating differences, 792
 clocked comparison, 789
 compare by region, 787
 compare by signal, 786
 compare options, 789

- compare tab, [786](#)
- comparison commands, [784](#)
- comparison method, [788](#)
- differences in text format, [793](#)
- flattened designs, [794](#)
- hierarchical designs, [794](#)
- icons, [792](#)
- initiating with GUI, [785](#)
- introduction, [782](#)
- leading edge tolerance, [789](#)
- list window display, [792](#)
- mixed-language support, [783](#)
- pathnames, [790](#)
- reference dataset, [785](#)
- reference region, [787](#)
- saving and reloading, [793](#)
- setup options, [783](#)
- signals with different names, [784](#)
- test dataset, [786](#)
- timing differences, [791](#)
- trailing edge tolerance, [789](#)
- using comparison wizard, [783](#)
- using the GUI, [785](#)
- values column, [791](#)
- wave window display, [790](#)
- Waveform Comparison
 - created waveforms, using with, [1214](#)
 - difference markers, [791](#)
- waveform editor
 - creating waveforms, [1207](#)
 - editing waveforms, [1209](#)
 - mapping signals, [1214](#)
 - saving stimulus files, [1212](#)
 - simulating, [1212](#)
 - Waveform Compare, using with, [1214](#)
- waveform logfile
 - overview, [703](#)
 - see also* WLF files
- waveforms, [703](#)
 - optimize viewing of, [1383](#)
 - saving between cursors, [765](#)
 - viewing, [287](#)
- WaveSignalNameWidth .ini file variable, [1380](#)
- White box testing
 - functional coverage, [1049](#)
- WIDTH
 - matching to Verilog, [1227](#)
- Window format
 - saving/restoring, [86](#), [764](#)
- Windows
 - Assertions, [140](#)
- windows
 - Active Processes pane, [223](#)
 - Dataflow window, [172](#), [829](#)
 - zooming, [824](#)
 - List window, [196](#), [722](#)
 - display properties of, [760](#)
 - formatting HDL items, [761](#)
 - saving data to a file, [767](#)
 - setting triggers, [772](#), [775](#)
 - Locals window, [200](#)
 - Main window, [95](#)
 - status bar, [101](#)
 - text editing, [1442](#)
 - time and delta display, [101](#)
 - toolbar, [114](#)
 - Objects window, [217](#)
 - Process window
 - specifying next process to be executed, [166](#)
 - viewing processing in the region, [166](#)
 - Signals window
 - VHDL and Verilog items viewed in, [217](#)
 - Source window, [242](#), [851](#)
 - Run Until Here, [874](#)
 - text editing, [1442](#)
 - viewing HDL source code, [242](#)
 - Variables window
 - VHDL and Verilog items viewed in, [200](#)
 - Wave window, [287](#), [720](#)
 - adding HDL items to, [722](#)
 - cursor measurements, [727](#)
 - display preferences, [747](#)
 - display range (zoom), changing, [740](#)
 - format file, saving, [763](#)
 - path elements, changing, [1380](#)
 - time cursors, [727](#)
 - zooming, [740](#)

- WLF file
 - limiting, [706](#)
 - WLF file parameters
 - cache size, [705](#), [706](#)
 - collapse mode, [706](#)
 - compression, [706](#)
 - delete on quit, [706](#)
 - filename, [706](#)
 - indexing, [706](#)
 - multithreading, [707](#)
 - optimization, [706](#)
 - overview, [705](#)
 - size limit, [706](#)
 - time limit, [706](#)
 - WLF files
 - collapsing events, [713](#)
 - optimizing waveform viewing, [1383](#)
 - saving, [704](#)
 - saving at intervals, [712](#)
 - WLFCacheSize .ini file variable, [1381](#)
 - WLFCollapseMode .ini file variable, [1381](#)
 - WLFCompress .ini variable, [1382](#)
 - WLFDeleteOnQuit .ini variable, [1382](#)
 - WLFFilename .ini file variable, [1383](#)
 - WLFOptimize .ini file variable, [1383](#)
 - WLFSaveAllRegions .ini file variable, [1384](#)
 - WLFSimCacheSize .ini variable, [1384](#)
 - WLFSizeLimit .ini variable, [1385](#)
 - WLFTimeLimit .ini variable, [1385](#)
 - WLFUseThreads .ini file variable, [1386](#)
 - work library, [372](#)
 - creating, [373](#)
 - write format restart, [86](#), [764](#), [782](#)
 - WRITE procedure, problems with, [399](#)
- X —
- X
 - tracing unknowns, [814](#), [839](#)
 - xml format
 - coverage reports, [958](#)
- Z —
- Z values, automatically excluded, [920](#)
 - zero delay elements, [395](#)
 - zero delay mode, [460](#)
 - zero-delay error, [1401](#)
 - zero-delay loop, infinite, [397](#)
 - zero-delay oscillation, [397](#)
 - zero-delay race condition, [441](#)
 - zoom
 - Dataflow window, [824](#)
 - saving range with bookmarks, [742](#)

Third-Party Information

This section provides information on third-party software that may be included in the ModelSim SE product, including any additional license terms.

- [*Third-Party Software for Questa and Modelsim Products*](#)

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement ("Order(s)"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2000), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer requests any change or enhancement to Software, whether in the course of receiving support or consulting services, evaluating Software, performing beta testing or otherwise, any inventions, product

improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.
4. **BETA CODE.**
 - 4.1. Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's 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.
 - 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
 - 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.
5. **RESTRICTIONS ON USE.**
 - 5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use it except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark.
 - 5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
 - 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at <http://supportnet.mentor.com/about/legal/>.
7. **AUTOMATIC CHECK FOR UPDATES; PRIVACY.** Technological measures in Software may communicate with servers of Mentor Graphics or its contractors for the purpose of checking for and notifying the user of updates and to ensure that the Software in use is licensed in compliance with this Agreement. Mentor Graphics will not collect any personally identifiable data in this process and will not disclose any data collected to any third party without the prior written consent of Customer, except to Mentor Graphics' outside attorneys or as may be required by a court of competent jurisdiction.
8. **LIMITED WARRANTY.**
 - 8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
 - 8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
10. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
11. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10. THE PROVISIONS OF THIS SECTION 11 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
12. **INFRINGEMENT.**
 - 12.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

- 12.2. If a claim is made under Subsection 12.1 Mentor Graphics may, at its option and expense, (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 12.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 12.4. THIS SECTION 12 IS SUBJECT TO SECTION 9 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS FOR DEFENSE, SETTLEMENT AND DAMAGES, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.
13. **TERMINATION AND EFFECT OF TERMINATION.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term.
 - 13.1. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
 - 13.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
14. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products and information about the products to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.
15. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.
16. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
17. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 17 shall survive the termination of this Agreement.
18. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the United States. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, USA, if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. This section shall not

restrict Mentor Graphics' right to bring an action against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

19. **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.
20. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 100615, Part No. 246066