



A Siemens Business

ModelSim® User's Manual

Software Version 2020.3

Unpublished work. © Siemens 2020

This document contains information that is confidential and proprietary to Mentor Graphics Corporation, Siemens Industry Software Inc., or their affiliates (collectively, "Siemens"). 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 confidential and proprietary information.

This document is for information and instruction purposes. Siemens 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 Siemens to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Siemens products are set forth in written agreements between Siemens and its customers. **End User License Agreement** — You can print a copy of the End User License Agreement from: mentor.com/eula.

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 Siemens whatsoever.

SIEMENS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

SIEMENS SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL OR PUNITIVE DAMAGES, LOST DATA OR PROFITS, EVEN IF SUCH DAMAGES WERE FORESEEABLE, ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF SIEMENS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

LICENSE RIGHTS APPLICABLE TO THE U.S. GOVERNMENT: This document explains the capabilities of commercial products that were developed exclusively at private expense. If the products are acquired directly or indirectly for use by the U.S. Government, then the parties agree that the products and this document are considered "Commercial Items" and "Commercial Computer Software" or "Computer Software Documentation," as defined in 48 C.F.R. §2.101 and 48 C.F.R. §252.227-7014(a)(1) and (a)(5), as applicable. Software and this document may only be used under the terms and conditions of the End User License Agreement referenced above as required by 48 C.F.R. §12.212 and 48 C.F.R. §227.7202. The U.S. Government will only have the rights set forth in the End User License Agreement, which supersedes any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Siemens or other parties. No one is permitted to use these Marks without the prior written consent of Siemens or the owner of the Marks, as applicable. The use herein of third party Marks is not an attempt to indicate Siemens as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A list of Siemens' trademarks may be viewed at: www.plm.automation.siemens.com/global/en/legal/trademarks.html and mentor.com/trademarks.

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Support Center: support.sw.siemens.com

Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Table of Contents

Chapter 1	
Introduction.....	29
Operational Structure and Flow.....	29
Basic Steps for Simulation.....	31
Files and Map Libraries	32
What is a Library?	32
Resource Libraries	32
Mapping the Logical Work to the Physical Work Directory	33
Step 1 — Create Work and Resource Libraries	33
Step 2 — Compile the Design	35
Step 3 — Load the Design for Simulation	36
Step 4 — Simulate the Design	37
Step 5 — Debug the Design	38
General Modes of Operation	39
Command Line Mode	40
Startup Variable Flow.....	40
Here-Document Flow	41
I/O Redirection Flow	41
Supported Commands for Command Line Mode	42
Batch Mode Simulation	43
Saving Batch Mode Simulation Data.....	43
Simulator Control Variables.....	44
Default stdout Messages	46
Tool Statistics Messages.....	46
Definition of an Object	47
Standards Supported	47
Text Conventions	48
Chapter 2	
Projects	49
What are Projects?.....	50
What are the Benefits of Projects?	50
Project Conversion Between Simulator Versions.....	51
Getting Started with Projects.....	52
Open a New Project	52
Add Source Files to the Project	54
Compile the Files	55
Change Compile Order	56
Auto-Generate the Compile Order	57
Grouping Files	57
Simulate a Design.....	58
The Project Window	60

Creating a Simulation Configuration	61
Organizing Projects with Folders	64
Adding a Project Folder	64
Set File Properties and Project Settings	66
File Compilation Properties	66
Project Settings	68
Convert Pathnames to Softnames for Location Mapping	68
Setting Custom Double-click Behavior	69
Access Projects from the Command Line	69
Chapter 3	
Design Libraries	71
Design Library Overview	72
Design Unit Information	72
Working Library Versus Resource Libraries	72
Working with Design Libraries	74
Creating a Library	74
Library Size	75
Library Window Contents	76
Map a Logical Name to a Design Library	77
Mapping a Library with the GUI	77
Mapping a Library from the Command Line	78
Manual Mapping of the modelsim.ini File	78
Move a Library	79
Setting Up Libraries for Group Use	79
Verilog Resource Libraries	81
Library Search Rules	81
Handling Sub-Modules with the Same Name	82
The LibrarySearchPath Variable	83
VHDL Resource Libraries	84
Predefined Libraries	84
Alternate IEEE Libraries Supplied	85
Regenerating Your Design Libraries	85
Importing FPGA Libraries	86
Protect Source Code	87
Chapter 4	
VHDL Simulation	89
Mixed-Language Support	89
Basic VHDL Usage	89
Compilation and Simulation of VHDL	91
Creating a Design Library for VHDL	91
Compilation of a VHDL Design—the vcom Command	91
Simulation of a VHDL Design—the vsim Command	96
Usage Characteristics and Requirements	97
Differences Between Supported Versions of the VHDL Standard	97
Naming Behavior of VHDL for Generate Blocks	100
Simulator Resolution Limit for VHDL	101

Table of Contents

Default Binding	102
Delta Delays	103
The TextIO Package	107
Syntax for File Declaration	107
STD_INPUT and STD_OUTPUT Within ModelSim	108
TextIO Implementation Issues	108
Alternative Input/Output Files	111
The TEXTIO Buffer	111
Input Stimulus to a Design	111
VITAL Usage and Compliance	112
VITAL Source Code	112
VITAL 1995 and 2000 Packages	112
VITAL Compliance	113
Compiling and Simulating with Accelerated VITAL Packages	113
VHDL Utilities Package (util)	114
Modeling Memory	118
Examples of Different Memory Models	118
Effects on Performance by Canceling Scheduled Events	128
VHDL Access Object Debugging	129
Terminology and Naming Conventions	129
VHDL Access Type	130
Limitations	131
Default Behavior—Logging and Debugging Disabled	132
Logging and Debugging Enabled	132
The examine and describe Commands	134

Chapter 5	
Verilog and SystemVerilog Simulation	137
Mixed-Language Support - Verilog Top	137
Standards, Nomenclature, and Conventions	138
Supported Variations in Source Code	139
for Loops	139
Naming Macros with Integers	139
Basic Verilog Usage	141
Verilog Compilation	142
Creating a Working Library	142
Invoking the Verilog Compiler	142
Verilog Case Sensitivity	143
Parsing SystemVerilog Keywords	143
Recognizing SystemVerilog Files by File Name Extension	144
Initializing enum Variables	145
Incremental Compilation	145
Library Usage	148
SystemVerilog Multi-File Compilation	150
Declarations in Compilation Unit Scope	150
Macro Definitions and Compiler Directives in Compilation Unit Scope	150
Verilog-XL Compatible Compiler Arguments	152
Arguments Supporting Source Libraries	152

Verilog-XL uselib Compiler Directive.	153
Verilog Configurations	156
Configurations and the Library Named work	156
Verilog Generate Statements	158
Name Visibility in Generate Statements	158
Verilog Simulation	159
Simulator Resolution Limit (Verilog)	159
Modules Without Timescale Directives	160
Multiple Timescale Directives	161
Choosing the Resolution for Verilog	162
Event Ordering in Verilog Designs	163
Event Queues	163
Controlling Event Queues with Blocking or Non-Blocking Assignments	165
Debugging Event Order Issues	167
Hazard Detection	167
Hazard Detection and Optimization Levels	167
Signal Segmentation Violations	168
Negative Timing Checks	171
vsim Arguments Related to Timing Checks	171
Commands Supporting Negative Timing Check Limits	172
Negative Timing Constraint Algorithm	177
Using Delayed Inputs for Timing Checks	181
Re-evaluation of Zero-Delay Output Schedules	182
Force and Release Statements in Verilog	183
Verilog-XL Compatible Simulator Arguments	183
Using Escaped Identifiers	185
Tcl and Escaped Identifiers	185
Cell Libraries	186
SDF Timing Annotation	186
Delay Modes	187
Delay Modes and the Verilog Standard	187
Distributed Delay Mode	190
Path Delay Mode	190
Unit Delay Mode	190
Zero Delay Mode	190
SystemVerilog System Tasks and Functions	191
IEEE Std 1800-2012 System Tasks and Functions	191
Using the \$typename Data Query Function	195
Task and Function Names Without Round Braces ‘()’	196
Verilog-XL Compatible System Tasks and Functions	198
Supported Tasks and Functions Mentioned in IEEE Std 1364	198
Supported Tasks and Functions Not Described in IEEE Std 1364	198
Extensions to Supported System Tasks	201
Unsupported Verilog-XL System Tasks	201
String Class Methods for Matching Patterns	202
Compiler Directives	205
IEEE Std 1364 Compiler Directives	206
Verilog-XL Compatible Compiler Directives	206
Unmatched Virtual Interface Declarations	207

Table of Contents

Verilog PLI and SystemVerilog DPI	209
Extensions to SystemVerilog DPI	209
SystemVerilog Class Debugging	210
Enabling Class Debug	210
The Class Instance Identifier	211
Obtaining the CIID with the examine Command	211
Obtaining the CIID With a System Function	211
Logging Class Types and Class Instances	212
Working with Class Types	213
Authoritative and Descriptive Class Type Names	214
Finding the Class Type Syntax	214
Viewing Class Types in the GUI	216
Working with Class Instances	219
The Class Instances Window	219
Viewing Class Instances in the Wave Window	221
The Locals Window	223
The Watch Window	223
The Call Stack Window	224
Working with Class Path Expressions	225
Class Path Expression Syntax	225
Adding a Class Path Expression to the Wave Window	226
Class Path Expression Values	226
Casting a Class Variable to a Specific Type	226
Class Objects vs Class Path Expressions	228
Disabling Class Path Expressions	228
Conditional Breakpoints in Dynamic Code	228
Stepping Through Your Design	229
The Run Until Here Feature	230
Command Line Interface	231
Class Instance Values	231
Class Instance Properties	231
Calling Functions	232
The classinfo Commands	234
Class Instance Garbage Collection	241
Default Garbage Collector Settings	241
Changing the Garbage Collector Configuration	242
Running the Garbage Collector	243
Autofindloop and the Autofindloop Report	243
Chapter 6	
Mixed-Language Simulation	249
Basic Mixed-Language Flow	249
Different Compilers with Common Design Libraries	250
Case Sensitivity	250
Hierarchical References	251
The SystemVerilog bind Construct in Mixed-Language Designs	253
Syntax of bind Statement	253
Allowed Bindings	253

Hierarchical References to a VHDL Object from a Verilog/SystemVerilog Scope	254
Mapping of Types	256
Port Mapping with VHDL and Verilog Enumerated Types	256
VHDL Instance Mapping	258
Simulator Resolution Limit	262
Runtime Modeling Semantics	263
Hierarchical References to SystemVerilog	263
Mapping Data Types	264
Verilog and SystemVerilog to VHDL Mappings	264
VHDL To Verilog and SystemVerilog Mappings	268
VHDL Instantiating Verilog or SystemVerilog	276
Verilog/SystemVerilog Instantiation Criteria Within VHDL	276
Component Declaration for VHDL Instantiating Verilog	276
vgencomp Component Declaration when VHDL Instantiates Verilog	277
Modules with Bidirectional Pass Switches	278
Modules with Unnamed Ports	279
Verilog or SystemVerilog Instantiating VHDL	281
VHDL Instantiation Criteria Within Verilog	281
Entity and Architecture Names and Escaped Identifiers	282
Named Port Associations	283
Generic Associations	283
Sharing User-Defined Types	284
Using a Common VHDL Package	284
Using a Common SystemVerilog Package	287
Chapter 7	
Recording Simulation Results With Datasets	291
Saving a Simulation to a WLF File	293
Saving at Intervals with Dataset Snapshot	293
Saving Memories to the WLF	295
WLF File Parameter Overview	295
Limiting the WLF File Size	297
Opening Datasets	298
Dataset Structure	299
Structure Window Columns	299
Managing Multiple Datasets	301
Managing Multiple Datasets in the GUI	301
Managing Multiple Datasets from the Command Line	301
Restricting the Dataset Prefix Display	303
Collapsing Time and Delta Steps	303
Virtual Objects	305
Virtual Signals	305
Virtual Functions	306
Virtual Regions	307
Virtual Types	307

Table of Contents

Chapter 8	
Waveform Analysis.....	309
Wave Window Overview.....	310
Objects You Can View	311
Adding Objects to the Wave Window.....	312
Inserting Signals in a Specific Location.....	313
Working with Cursors	315
Adding Cursors.....	317
Editing Cursor Properties	317
Jump to a Signal Transition	318
Measuring Time with Cursors in the Wave Window.....	318
Syncing All Active Cursors	319
Linking Cursors	319
Understanding Cursor Behavior	320
Shortcuts for Working with Cursors.....	321
Two Cursor Mode.....	322
Enable Two Cursor Mode	322
Additional Mouse Actions	322
Expanded Time in the Wave Window.....	323
Expanded Time Terminology	323
Recording Expanded Time Information	324
Viewing Expanded Time Information in the Wave Window.....	325
Customizing the Expanded Time Wave Window Display.....	327
Expanded Time Display Modes	329
Menu Selections for Expanded Time Display Modes	329
Toolbar Selections for Expanded Time Modes	329
Command Selection of Expanded Time Mode	330
Switching Between Time Modes	330
Expanding and Collapsing Simulation Time	330
Expanded Time with examine and Other Commands	331
Zooming the Wave Window Display	332
Zooming with the Menu, Toolbar and Mouse	332
Saving Zoom Range and Scroll Position with Bookmarks.....	333
Editing Bookmarks	334
Searching in the Wave Window	335
Searching for Values or Transitions	335
Search with the Expression Builder	337
Using the Expression Builder for Expression Searches	337
Saving an Expression to a Tcl Variable	339
Searching for a Particular Value.....	339
Evaluating Only on Clock Edges	339
Filtering the Wave Window Display	340
Formatting the Wave Window.....	341
Setting Wave Window Display Preferences.....	342
Hiding/Showing Path Hierarchy.....	342
Double-Click Behavior in the Wave Window	343
Setting the Timeline to Count Clock Cycles	343
Formatting Objects in the Wave Window	345

Changing Radix (base) for the Wave Window.....	346
Dividing the Wave Window.....	348
Splitting Wave Window Panes.....	349
Wave Groups	350
Creating a Wave Group	351
Grouping Signals through Menu Selection	351
Adding a Group of Contributing Signals	352
Grouping Signals with the add wave Command	353
Grouping Signals with a Keyboard Shortcut	353
Deleting or Ungrouping a Wave Group	354
Adding Items to an Existing Wave Group	354
Removing Items from an Existing Wave Group	354
Miscellaneous Wave Group Features	355
Composite Signals or Buses	356
Creating Composite Signals through Menu Selection	356
Saving the Window Format.....	357
Exporting Waveforms from the Wave window.....	359
Exporting the Wave Window as a Bitmap Image.....	359
Printing the Wave Window to a Postscript File	359
Printing the Wave Window on the Windows Platform	360
Saving Waveform Sections for Later Viewing.....	361
Saving Waveforms Between Two Cursors.....	361
Viewing Saved Waveforms	362
Working With Multiple Cursors	363
Viewing System Verilog Interfaces.....	364
Working with Virtual Interfaces	365
Adding Virtual Interface References to the Wave Window.....	365
Combining Objects into Buses	367
Extracting a Bus Slice.....	367
Wave Extract/Pad Bus Dialog Box.....	368
Splitting a Bus into Several Smaller Buses	369
Using the Virtual Signal Builder	370
Creating a Virtual Signal	371
Miscellaneous Tasks	374
Examining Waveform Values.....	374
Displaying Drivers of the Selected Waveform.....	374
Sorting a Group of Objects in the Wave Window	375
Creating and Managing Breakpoints	376
Signal Breakpoints	377
Setting Signal Breakpoints with the when Command	377
Setting Signal Breakpoints with the GUI.....	377
Modifying Signal Breakpoints	378
File-Line Breakpoints	380
Setting File-Line Breakpoints Using the bp Command	380
Setting File-Line Breakpoints Using the GUI	380
Modifying a File-Line Breakpoint	381
Saving and Restoring Breakpoints	382

Table of Contents

Chapter 9	
Debugging with the Dataflow Window	383
Dataflow Window Overview	383
Dataflow Usage Flow	385
Live Simulation Debug Flow	385
Post-Simulation Debug Flow Details	387
Create the Post-Sim Debug Database.....	387
Use the Post-Simulation Debug Database	388
Common Tasks for Dataflow Debugging	389
Add Objects to the Dataflow Window	389
Exploring the Connectivity of the Design	392
Analyzing a Scalar Connected to a Wide Bus	392
Control the Display of Readers and Nets	394
Controlling the Display of Redundant Buffers and Inverters.....	395
Track Your Path Through the Design	395
Explore Designs with the Embedded Wave Viewer.....	396
Tracing Events	398
Tracing the Source of an Unknown State (StX)	399
Finding Objects by Name in the Dataflow Window.....	400
Automatically Tracing All Paths Between Two Nets.....	401
Dataflow Concepts.....	403
Symbol Mapping.....	404
User-Defined Symbols	406
Current vs. Post-Simulation Command Output	407
Dataflow Window Graphic Interface Reference	408
What Can I View in the Dataflow Window?	408
How is the Dataflow Window Linked to Other Windows?	408
How Can I Print and Save the Display?	409
Save a .eps File and Printing the Dataflow Display from UNIX	409
Print from the Dataflow Display on Windows Platforms	409
Configure Page Setup.....	410
How Do I Configure Window Options?.....	411
Chapter 10	
Source Window	413
Opening Source Files.....	414
Changing File Permissions	414
Updates to Externally Edited Source Files	415
Navigating Through Your Design	415
Data and Objects in the Source Window	417
Object Values and Descriptions	417
Setting Simulation Time in the Source Window	418
Search for Source Code Objects	419
Searching for One Instance of a String.....	419
Searching for All Instances of a String.....	419
Searching for the Original Declaration of an Object	420
Debugging and Textual Connectivity	421
Hyperlinked Text	421

Highlighted Text in the Source Window	421
Drag Objects Into Other Windows	422
Breakpoints	423
Setting Individual Breakpoints in a Source File	423
Setting Breakpoints with the bp Command	424
Editing Breakpoints	425
Using the Modify Breakpoints Dialog Box	425
Deleting Individual Breakpoints	427
Deleting Groups of Breakpoints	427
Saving and Restoring Source Breakpoints	427
Setting Conditional Breakpoints	429
Setting a Breakpoint For a Specific Instance	430
Setting a Breakpoint For a Specified Value of Any Instance	431
Run Until Here	431
Source Window Bookmarks	433
Setting and Removing Bookmarks	433
Source Window Preferences	433
Chapter 11	
Signal Spy	435
Signal Spy Concepts	436
Signal Spy Formatting Syntax	436
Signal Spy Supported Types	437
Signal Spy Reference	439
disable_signal_spy	440
enable_signal_spy	442
init_signal_driver	444
init_signal_spy	448
signal_force	452
signal_release	456
Chapter 12	
Generating Stimulus with Waveform Editor	459
Getting Started with the Waveform Editor	461
Using Waveform Editor Prior to Loading a Design	461
Using Waveform Editor After Loading a Design	462
Accessing the Create Pattern Wizard	463
Creating Waveforms with Wave Create Command	464
Editing Waveforms	464
Selecting Parts of the Waveform	466
Selection and Zoom Percentage	467
Auto Snapping of the Cursor	468
Stretching and Moving Edges	468
Simulating Directly from Waveform Editor	468
Exporting Waveforms to a Stimulus File	469
Driving Simulation with the Saved Stimulus File	470
Signal Mapping and Importing EVCD Files	470
Saving the Waveform Editor Commands	471

Table of Contents

Chapter 13	
Standard Delay Format (SDF) Timing Annotation	473
Specifying SDF Files for Simulation	474
Instance Specification	474
SDF Specification with the GUI	474
Errors and Warnings	475
VHDL VITAL SDF	476
SDF to VHDL Generic Matching	477
Resolving Errors	477
Verilog SDF	479
\$sd _f _annotate	480
SDF to Verilog Construct Matching	482
Retain Delay Behavior	485
Optional Edge Specifications	487
Optional Conditions	488
Rounded Timing Values	488
SDF for Mixed VHDL and Verilog Designs	489
Interconnect Delays	489
Disabling Timing Checks	489
Troubleshooting	491
Specifying the Wrong Instance	491
Matching a Single Timing Check	492
Mistaking a Component or Module Name for an Instance Label	492
Forgetting to Specify the Instance	492
Reporting Unannotated Specify Path Objects	493
Failing to Find Matching Specify Module Path	494
Chapter 14	
Value Change Dump (VCD) Files	497
Creating a VCD File	498
Four-State VCD File	498
Extended VCD File	498
VCD Case Sensitivity	499
Using Extended VCD as Stimulus	500
Simulating with Input Values from a VCD File	500
Replacing Instances with Output Values from a VCD File	502
Port Order Issues	503
VCD Commands and VCD Tasks	504
Compressing Files with VCD Tasks	505
VCD File from Source to Output	506
VHDL Source Code	506
VCD Simulator Commands	506
VCD to WLF	509
Capturing Port Driver Data	509
Resolving Values	511
Default Behavior	511
When force Command is Used	511
Extended Data Type for VHDL (vl_logic)	512

Ignoring Strength Ranges	513
Chapter 15	
Tcl and DO Files	515
Tcl Features	516
Tcl References	516
Tcl Command Syntax	517
If Command Syntax	520
Command Substitution	520
Command Separator	521
Multiple-Line Commands	521
Evaluation Order	521
Tcl Relational Expression Evaluation	521
Variable Substitution	522
System Commands	522
ModelSim Replacements for Tcl Commands	522
Simulator State Variables	524
Referencing Simulator State Variables	526
Special Considerations for the now Variable	527
List Processing	527
Simulator Tcl Commands	529
Simulator Tcl Time Commands	530
Time Conversion Tcl Commands	530
Time Relations Tcl Commands	530
Tcl Time Arithmetic Commands	531
Tcl Examples	531
DO Files	534
Creating DO Files	534
Using Parameters with DO Files	535
Deleting a File from a .do Script	535
Making Script Parameters Optional	536
Breakpoint Flow Control in Nested DO files	537
Useful Commands for Handling Breakpoints and Errors	539
Error Action in DO File Scripts	539
Using the Tcl Source Command with DO Files	540
Appendix A	
modelsim.ini Variables	541
Organization of the modelsim.ini File	546
Making Changes to the modelsim.ini File	546
Editing modelsim.ini Variables	547
Overriding the Default Initialization File	547
The Runtime Options Dialog	548
Variables	552
AccessObjDebug	558
AddPragmaPrefix	559
AllowCheckpointCpp	560
AmsStandard	561

Table of Contents

AppendClose	562
AssertFile	563
BatchMode	564
BatchTranscriptFile	565
BindAtCompile	566
BreakOnAssertion	567
BreakOnMessage	568
CheckPlusargs	569
CheckpointCompressMode	570
CheckSynthesis	571
ClassDebug	572
CommandHistory	573
common	574
CompilerTempDir	575
ConcurrentFileLimit	576
CreateDirFor FileAccess	577
CreateLib	578
data_method	579
DatasetSeparator	580
DefaultForceKind	581
DefaultLibType	582
DefaultRadix	583
DefaultRadixFlags	584
DefaultRestartOptions	585
DelayFileOpen	586
displaymsgmode	587
DpiOutOfTheBlue	588
DumpportsCollapse	589
EnumBaseInit	590
error	591
ErrorFile	592
Explicit	593
fatal	594
FlatLibPageSize	595
FlatLibPageDeletePercentage	596
FlatLibPageDeleteThreshold	597
floatfixlib	598
ForceSigNextIter	599
ForceUnsignedIntegerToVHDLInteger	600
FsmImplicitTrans	601
FsmResetTrans	602
FsmSingle	603
FsmXAssign	604
GCThreshold	605
GCThresholdClassDebug	606
GenerateFormat	607
GenerousIdentifierParsing	608
GlobalSharedObjectList	609
Hazard	610

ieee	611
IgnoreError	612
IgnoreFailure	613
IgnoreNote	614
IgnorePragmaPrefix	615
ignoreStandardRealVector	616
IgnoreVitalErrors	617
IgnoreWarning	618
ImmediateContinuousAssign	619
IncludeRecursionDepthMax	620
InitOutCompositeParam	621
IterationLimit	622
keyring	623
LargeObjectSilent	624
LargeObjectSize	625
LibrarySearchPath	626
MessageFormat	627
MessageFormatBreak	628
MessageFormatBreakLine	629
MessageFormatError	630
MessageFormatFail	631
MessageFormatFatal	632
MessageFormatNote	633
MessageFormatWarning	634
MixedAnsiPorts	635
modelsim_lib	636
MsgLimitCount	637
msgmode	638
mtiAvm	639
mtiOvm	640
MultiFileCompilationUnit	641
NoCaseStaticError	642
NoDebug	643
NoDeferSubpgmCheck	644
NoIndexCheck	645
NoOthersStaticError	646
NoRangeCheck	647
note	648
NoVitalCheck	649
NumericStdNoWarnings	650
OldVHDLConfigurationVisibility	651
OldVhdlForGenNames	652
OnFinish	653
Optimize_1164	654
osvvm	655
PathSeparator	656
PedanticErrors	657
PreserveCase	658
PrintSimStats	659

Table of Contents

Quiet	660
RequireConfigForAllDefaultBinding	661
Resolution	662
RunLength	663
SeparateConfigLibrary	664
Show_BadOptionWarning	665
Show_Lint	666
Show_source	667
Show_VitalChecksWarnings	668
Show_Warning1	669
Show_Warning2	670
Show_Warning3	671
Show_Warning4	672
Show_Warning5	673
ShowFunctions	674
ShutdownFile	675
SignalForceFunctionUseDefaultRadix	676
SignalSpyPathSeparator	677
SmartDbgSym	678
Startup	679
Stats	680
std	682
std_developerskit	683
StdArithNoWarnings	684
suppress	685
SuppressFileTypeReg	686
sv_std	687
SvExtensions	688
SVFileSuffixes	692
Svlog	693
SVPrettyPrintFlags	694
synopsys	696
SyncCompilerFiles	697
toolblock	698
TranscriptFile	699
UnbufferedOutput	700
UndefSyms	701
UserTimeUnit	702
UVMControl	703
verilog	704
Veriuser	705
VHDL93	706
VhdlSeparatePduPackage	707
VhdlVariableLogging	708
vital2000	709
vlog95compat	710
WarnConstantChange	711
warning	712
WaveSignalNameWidth	713

wholefile	714
WildcardFilter	715
WildcardSizeThreshold	716
WildcardSizeThresholdVerbose	717
WLFCacheSize	718
WLFCollapseMode	719
WLFCompress	720
WLFDeleteOnQuit	721
WLFFileLock	722
WLFFilename	723
WLFOptimize	724
WLFSaveAllRegions	725
WLFSimCacheSize	726
WLFSizeLimit	727
WLFTimeLimit	728
WLFUpdateInterval	729
WLFUseThreads	730
WrapColumn	731
WrapMode	732
WrapWSColumn	733
Commonly Used modelsim.ini Variables	734
Common Environment Variables	734
Hierarchical Library Mapping	735
Creating a Transcript File	735
Using a Startup File	736
Turn Off Assertion Messages	736
Turn Off Warnings from Arithmetic Packages	736
Force Command Defaults	737
Restart Command Defaults	737
VHDL Standard	737
Delay Opening VHDL Files	738
Appendix B Location Mapping	739
Referencing Source Files with Location Maps	740
Using Location Mapping	740
Pathname Syntax	741
How Location Mapping Works	741
Appendix C Error and Warning Messages	743
Message System	744
Message Format	744
Getting More Information	745
Message Severity Level	745
Syntax Error Debug Flow	745
Suppression of Warning Messages	746
Exit Codes	747

Table of Contents

Miscellaneous Messages	749
Enforcing Strict 1076 Compliance	752
Appendix D	
Verilog Interfaces to C	755
Implementation Information	755
GCC Compiler Support for use with C Interfaces.....	756
Registering PLI Applications.....	756
Registering DPI Applications	758
DPI Use Flow.....	760
DPI and the vlog Command	761
Deprecated Legacy DPI Flows	762
When Your DPI Export Function is Not Getting Called	762
Troubleshooting a Missing DPI Import Function.....	762
Simplified Import of Library Functions	763
Optimizing DPI Import Call Performance	763
Making Verilog Function Calls from non-DPI C Models	764
Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code	765
Compiling and Linking C Applications for Interfaces	766
Windows Platforms — C	766
Compiling and Linking C++ Applications for Interfaces	768
For PLI only	768
Windows Platforms — C++	769
Specifying Application Files to Load	770
PLI and VPI File Loading.....	770
DPI File Loading	770
DPI Example	771
The PLI Callback reason Argument	772
The sizetf Callback Function.....	774
PLI Object Handles	774
Support for VHDL Objects	775
IEEE Std 1364 ACC Routines.....	776
IEEE Std 1364 TF Routines.....	777
SystemVerilog DPI Access Routines	780
Verilog-XL Compatible Routines	780
PLI/VPI Tracing	781
The Purpose of Tracing Files	781
Invoking a Trace	781
Debugging Interface Application Code	782
Appendix E	
System Initialization	785
Files Accessed During Startup.....	785
Initialization Sequence.....	786
Environment Variables	789
Expansion of Environment Variables	789
Setting Environment Variables.....	790
Creating Environment Variables in Windows	795

Library Mapping with Environment Variables.....	795
Node-Locked License File	796
Referencing Environment Variables.....	796
Removal of Temporary Files (VSOUT).....	797

Third-Party Information

End-User License Agreement with EDA Software Supplemental Terms

List of Figures

Figure 1-1. Operational Structure and Flow	30
Figure 1-2. Work Library.....	35
Figure 1-3. Compiled Design.....	36
Figure 2-1. Create Project Dialog Box	52
Figure 2-2. Project Window Detail	53
Figure 2-3. Add items to the Project Dialog	53
Figure 2-4. Create Project File Dialog.....	54
Figure 2-5. Add file to Project Dialog.....	54
Figure 2-6. Click Plus Sign to Show Design Hierarchy	56
Figure 2-7. Setting Compile Order	57
Figure 2-8. Grouping Files.....	58
Figure 2-9. Add Simulation Configuration Dialog Box — Design Tab	59
Figure 2-10. Structure Window with Projects	59
Figure 2-11. Project Window Overview	60
Figure 2-12. Add Simulation Configuration Dialog Box	62
Figure 2-13. Simulation Configuration in the Project Window.....	63
Figure 2-14. Add Folder Dialog Box.....	64
Figure 2-15. Specifying a Project Folder.....	65
Figure 2-16. Project Compiler Settings Dialog Box	65
Figure 2-17. Specifying File Properties.....	67
Figure 2-18. Project Settings Dialog Box	68
Figure 3-1. Creating a New Library.....	75
Figure 3-2. The Library Window—Design Unit Information in the Workspace	76
Figure 3-3. Edit Library Mapping Dialog Box	78
Figure 3-4. Sub-Modules with the Same Name.....	82
Figure 3-5. Import Library Wizard	87
Figure 4-1. VHDL Delta Delay Process	104
Figure 5-1. Fatal Signal Segmentation Violation (SIGSEGV)	169
Figure 5-2. Current Process Where Error Occurred	169
Figure 5-3. Blue Arrow Indicating Where Code Stopped Executing	169
Figure 5-4. Null Values in the Locals Window	170
Figure 5-5. Classes in the Class Tree Window	216
Figure 5-6. Class in the Class Graph Window.....	217
Figure 5-7. Classes in the Structure Window	218
Figure 5-8. The Class Instances Window	220
Figure 5-9. Placing Class Instances in the Wave Window	222
Figure 5-10. Class Information Popup in the Wave Window	222
Figure 5-11. Class Viewing in the Watch Window	223
Figure 5-12. Class Path Expressions in the Wave Window	226
Figure 5-13. /top/a Cast as c1 and c1prime	227

Figure 5-14. Casting c1 to c1prime	228
Figure 5-15. Extensions for a Class Type	239
Figure 5-16. Garbage Collector Configuration	242
Figure 7-1. Displaying Two Datasets in the Wave Window	292
Figure 7-2. Dataset Snapshot Dialog Box	294
Figure 7-3. Open Dataset Dialog Box	298
Figure 7-4. Structure Tabs	299
Figure 7-5. The Dataset Browser	301
Figure 7-6. Virtual Objects Indicated by Orange Diamond	305
Figure 8-1. The Wave Window	311
Figure 8-2. Insertion Point Bar	313
Figure 8-3. Grid and Timeline Properties	316
Figure 8-4. Original Names of Wave Window Cursors	318
Figure 8-5. Sync All Active Cursors	319
Figure 8-6. Cursor Linking Menu	320
Figure 8-7. Configure Cursor Links Dialog	320
Figure 8-8. Waveform Pane with Collapsed Event and Delta Time	325
Figure 8-9. Waveform Pane with Expanded Time at a Specific Time	326
Figure 8-10. Waveform Pane with Event Not Logged	326
Figure 8-11. Waveform Pane with Expanded Time Over a Time Range	327
Figure 8-12. New Bookmark Dialog	334
Figure 8-13. Wave Signal Search Dialog Box	336
Figure 8-14. Expression Builder Dialog Box	337
Figure 8-15. Selecting Signals for Expression Builder	338
Figure 8-16. Display Tab of the Wave Window Preferences Dialog Box	342
Figure 8-17. Grid and Timeline Tab of Wave Window Preferences Dialog Box	344
Figure 8-18. Clock Cycles in Timeline of Wave Window	344
Figure 8-19. Wave Format Menu Selections	345
Figure 8-20. Format Tab of Wave Properties Dialog	345
Figure 8-21. Changing Signal Radix	346
Figure 8-22. Global Signal Radix Dialog in Wave Window	347
Figure 8-23. Separate Signals with Wave Window Dividers	348
Figure 8-24. Splitting Wave Window Panes	349
Figure 8-25. Wave Groups Denoted by Red Diamond	352
Figure 8-26. Contributing Signals Group	353
Figure 8-27. Save Format Dialog	357
Figure 8-28. Waveform Save Between Cursors	361
Figure 8-29. Wave Filter Dialog	362
Figure 8-30. Wave Filter Dataset	363
Figure 8-31. Virtual Interface Objects Added to Wave Window	366
Figure 8-32. Signals Combined to Create Virtual Bus	367
Figure 8-33. Wave Extract/Pad Bus Dialog Box	368
Figure 8-34. Virtual Signal Builder	370
Figure 8-35. Virtual Signal Builder Help	371
Figure 8-36. Creating a Virtual Signal	372

List of Figures

Figure 8-37. Virtual Signal in the Wave Window	373
Figure 8-38. Modifying the Breakpoints Dialog	378
Figure 8-39. Signal Breakpoint Dialog	379
Figure 8-40. Breakpoints in the Source Window	381
Figure 8-41. File Breakpoint Dialog Box	382
Figure 9-1. The Dataflow Window (undocked) - ModelSim	384
Figure 9-2. Dataflow Debugging Usage Flow	386
Figure 9-3. Dot Indicates Input in Process Sensitivity List	390
Figure 9-4. CurrentTime Label in Dataflow Window	391
Figure 9-5. Controlling Display of Redundant Buffers and Inverters	395
Figure 9-6. Green Highlighting Shows Your Path Through the Design	395
Figure 9-7. Highlight Selected Trace with Custom Color	396
Figure 9-8. Wave Viewer Displays Inputs and Outputs of Selected Process	397
Figure 9-9. Unknown States Shown as Red Lines in Wave Window	399
Figure 9-10. Dataflow: Point-to-Point Tracing	402
Figure 9-11. The Print Postscript Dialog	409
Figure 9-12. The Print Dialog	410
Figure 9-13. The Page Setup Dialog	410
Figure 9-14. Dataflow Options Dialog	411
Figure 10-1. Setting Context from Source Files	416
Figure 10-2. Examine Pop Up	417
Figure 10-3. Current Time Label in Source Window	418
Figure 10-4. Enter an Event Time Value	418
Figure 10-5. Bookmark All Instances of a Search	420
Figure 10-6. Breakpoint in the Source Window	423
Figure 10-7. Editing Existing Breakpoints	426
Figure 10-8. Source Code for <i>source.sv</i>	429
Figure 12-1. Waveform Editor: Library Window	461
Figure 12-2. Results of Create Wave Operation	462
Figure 12-3. Opening Waveform Editor from Objects Windows	462
Figure 12-4. Create Pattern Wizard	463
Figure 12-5. Wave Edit Toolbar	464
Figure 12-6. Manipulating Waveforms with the Wave Edit Toolbar and Cursors	467
Figure 12-7. Export Waveform Dialog	469
Figure 12-8. Evcd Import Dialog	470
Figure 13-1. SDF Tab in Start Simulation Dialog	475
Figure 15-1. Breakpoint Flow Control in Nested DO Files	538
Figure A-1. Runtime Options Dialog: Defaults Tab	549
Figure A-2. Runtime Options Dialog Box: Message Severity Tab	550
Figure A-3. Runtime Options Dialog Box: WLF Files Tab	551
Figure D-1. DPI Use Flow Diagram	760

List of Tables

Table 1-1. General Modes for ModelSim	39
Table 1-2. Possible Definitions of an Object, by Language	47
Table 1-3. Text Conventions	48
Table 4-1. Using the examine Command to Obtain VHDL Integer Data	134
Table 4-2. Using the examine Command to Obtain VHDL String Data	135
Table 4-3. Using the examine Command to Obtain VHDL Record Data	135
Table 5-1. Evaluation 1 of always Statements	164
Table 5-2. Evaluation 2 of always Statement	164
Table 5-3. Utility System Tasks and Functions	191
Table 5-4. Utility System Functions	192
Table 5-5. Utility System Math Functions	192
Table 5-6. Utility System Analysis Tasks and Functions	192
Table 5-7. Input/Output System Tasks and Functions	193
Table 5-8. Input/Output System Memory and Argument Tasks	194
Table 5-9. Input/Output System File I/O Tasks	194
Table 5-10. Other System Tasks and Functions	194
Table 5-11. Stepping Within the Current Context.	230
Table 5-12. Garbage Collector Modes	241
Table 5-13. CLI Garbage Collector Commands and INI Variables	242
Table 6-1. VHDL Types Mapped To SystemVerilog Port Vectors	256
Table 6-2. SystemVerilog-to-VHDL Data Type Mapping	264
Table 6-3. Verilog Parameter to VHDL Mapping	266
Table 6-4. Verilog States Mapped to std_logic and bit	267
Table 6-5. VHDL to SystemVerilog Data Type Mapping	268
Table 6-6. VHDL Generics to Verilog Mapping	269
Table 6-7. Mapping VHDL bit to Verilog States	269
Table 6-8. Mapping VHDL std_logic Type to Verilog States	269
Table 6-9. Mapping Table for Verilog-style Declarations	271
Table 6-10. Mapping Table for SystemVerilog-style Declarations	272
Table 6-11. Mapping Literals from VHDL to SystemVerilog	285
Table 6-12. Supported Types Inside VHDL Records	286
Table 6-13. Supported Types Inside SystemVerilog Structure	287
Table 7-1. WLF File Parameters	295
Table 7-2. Structure Tab Columns	299
Table 7-3. vsim Arguments for Collapsing Time and Delta Steps	303
Table 8-1. Add Objects to the Wave Window	312
Table 8-2. Actions for Cursors	315
Table 8-3. Find Previous and Next Transition Icons	318
Table 8-4. Two Cursor Zoom	322
Table 8-5. Recording Delta and Event Time Information	324

Table 8-6. Menu Selections for Expanded Time Display Modes	329
Table 8-7. Actions for Bookmarks	334
Table 9-1. Icon and Menu Selections for Exploring Design Connectivity	392
Table 9-2. Dataflow Window Links to Other Windows and Panes	408
Table 10-1. Open a Source File	414
Table 11-1. Signal Spy Reference Comparison	436
Table 12-1. Signal Attributes in Create Pattern Wizard	463
Table 12-2. Waveform Editing Commands	465
Table 12-3. Selecting Parts of the Waveform	466
Table 12-4. Wave Editor Mouse/Keyboard Shortcuts	468
Table 12-5. Formats for Saving Waveforms	469
Table 12-6. Examples for Loading a Stimulus File	470
Table 13-1. Matching SDF to VHDL Generics	477
Table 13-2. Matching SDF IOPATH to Verilog	482
Table 13-3. Matching SDF INTERCONNECT and PORT to Verilog	482
Table 13-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog	482
Table 13-5. Matching SDF DEVICE to Verilog	483
Table 13-6. Matching SDF SETUP to Verilog	483
Table 13-7. Matching SDF HOLD to Verilog	483
Table 13-8. Matching SDF SETUPHOLD to Verilog	483
Table 13-9. Matching SDF RECOVERY to Verilog	484
Table 13-10. Matching SDF REMOVAL to Verilog	484
Table 13-11. Matching SDF RECREM to Verilog	484
Table 13-12. Matching SDF SKEW to Verilog	484
Table 13-13. Matching SDF WIDTH to Verilog	484
Table 13-14. Matching SDF PERIOD to Verilog	485
Table 13-15. Matching SDF NOCHANGE to Verilog	485
Table 13-16. RETAIN Delay Usage (default)	486
Table 13-17. RETAIN Delay Usage (with +vlog_retain_same2same_on)	486
Table 13-18. Matching Verilog Timing Checks to SDF SETUP	487
Table 13-19. SDF Data May Be More Accurate Than Model	487
Table 13-20. Matching Explicit Verilog Edge Transitions to Verilog	487
Table 13-21. SDF Timing Check Conditions	488
Table 13-22. SDF Path Delay Conditions	488
Table 13-23. Disabling Timing Checks	490
Table 14-1. VCD Commands and SystemTasks	504
Table 14-2. VCD Dumpport Commands and System Tasks	504
Table 14-3. VCD Commands and System Tasks for Multiple VCD Files	504
Table 14-4. Driver States	509
Table 14-5. State When Direction is Unknown	509
Table 14-6. Driver Strength	510
Table 14-7. VCD Values When Force Command is Used	511
Table 14-8. Values for file_format Argument	513
Table 14-9. Sample Driver Data	514
Table 15-1. Tcl Backslash Sequences	518

List of Tables

Table 15-2. Changes to ModelSim Commands	523
Table 15-3. Simulator State Variables	524
Table 15-4. Tcl List Commands	527
Table 15-5. Simulator-Specific Tcl Commands	529
Table 15-6. Tcl Time Conversion Commands	530
Table 15-7. Tcl Time Relation Commands	530
Table 15-8. Tcl Time Arithmetic Commands	531
Table 15-9. Commands for Handling Breakpoints and Errors in DO scripts	539
Table A-1. Commands for Overriding the Default Initialization File	548
Table A-2. Runtime Option Dialog: Defaults Tab Contents	549
Table A-3. Runtime Option Dialog: Message Severity Tab Contents	550
Table A-4. Runtime Option Dialog: WLF Files Tab Contents	551
Table A-5. MessageFormat Variable: Accepted Values	627
Table C-1. Severity Level Types	744
Table C-2. Exit Codes	747
Table D-1. vsim Arguments for DPI Application Using External Compilation Flows	770
Table D-2. Supported VHDL Objects	775
Table D-3. Supported ACC Routines	776
Table D-4. Supported TF Routines	779
Table D-5. Values for action Argument	781
Table E-1. Files That ModelSim Accesses During Startup	785
Table E-2. Add Library Mappings to modelsim.ini File	795

Chapter 1 Introduction

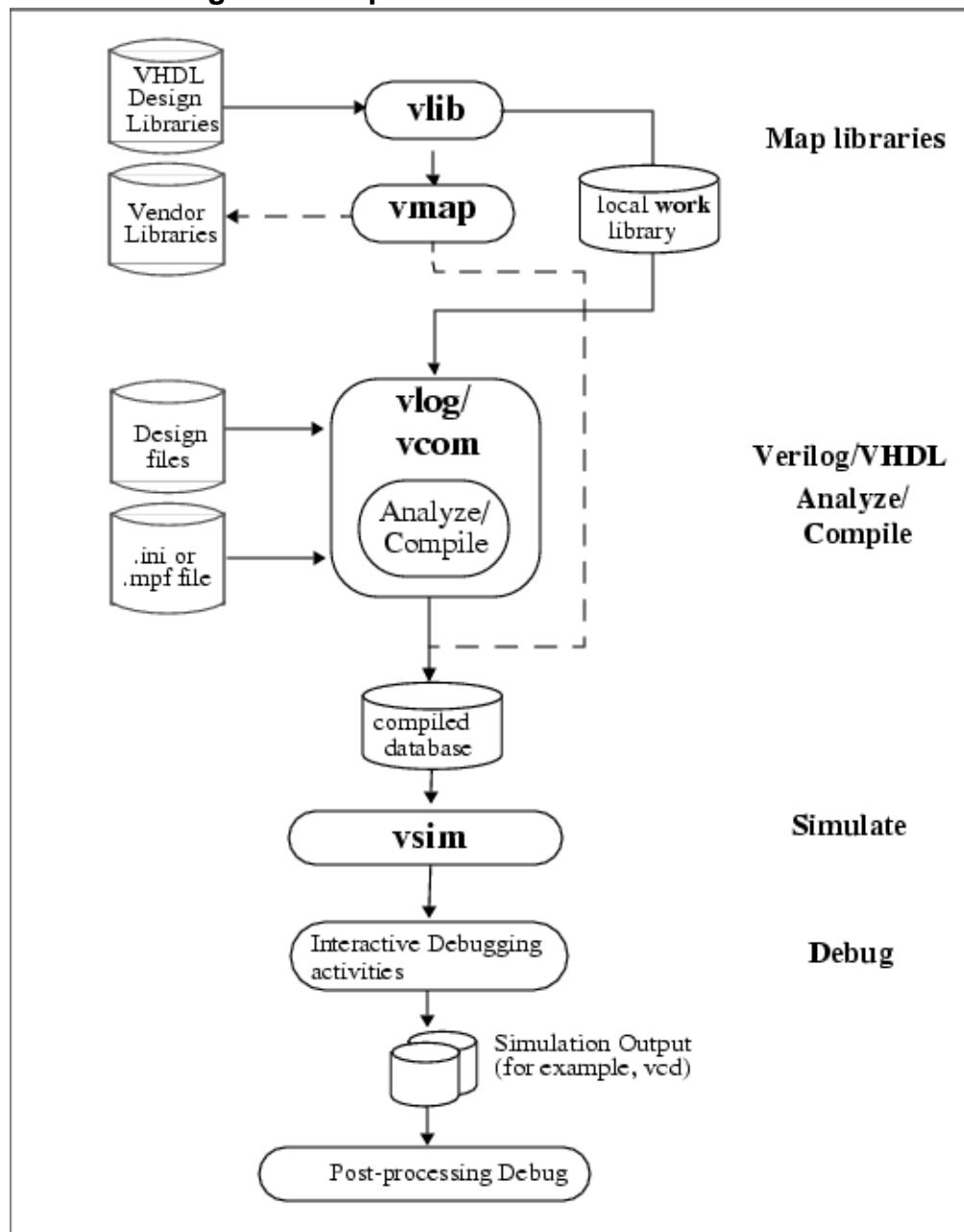
ModelSim provides you with a simulation, debug, and verification platform for validating FPGA and SoC designs.

For more complete information on current support for ModelSim, refer to the Installation and Licensing Guide.

Operational Structure and Flow	29
Basic Steps for Simulation	31
General Modes of Operation	39
Default stdout Messages	46
Definition of an Object	47
Standards Supported.....	47
Text Conventions	48

Operational Structure and Flow

A series of commands provides the structure and flow for verifying a design with ModelSim.

Figure 1-1. Operational Structure and Flow

Basic Steps for Simulation

You must have the proper files and library setup in order to use the commands necessary to simulate your design using ModelSim.

Files and Map Libraries	32
Step 1 — Create Work and Resource Libraries.....	33
Step 2 — Compile the Design	35
Step 3 — Load the Design for Simulation	36
Step 4 — Simulate the Design.....	37
Step 5 — Debug the Design.....	38

Files and Map Libraries

ModelSim must have access to several specific file types in order to simulate your design.

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

For detailed information about the files that ModelSim uses during system startup (including the *modelsim.ini* file), refer to “[System Initialization](#)”.

What is a Library?.....	32
Resource Libraries.....	32
Mapping the Logical Work to the Physical Work Directory.....	33

What is a Library?

A library is a location on your file system that contains data to be used for simulation. ModelSim relies on and can manipulate the data in one or more libraries for simulation. A library also helps to streamline simulation invocation.

ModelSim uses the following types of libraries:

- A local working library that contains the compiled version of your design.
- A resource library.

Resource Libraries

A resource library is typically static, serving as a parts source for your design. You can create your own resource libraries, or the libraries may be supplied by another design team or a third party (for example, a silicon vendor).

Examples of resource libraries:

- Shared information within your group.
- Vendor libraries.
- Packages.
- Previously compiled elements of your own working design.

Instead of compiling all design data each time you simulate, ModelSim makes use of pre-compiled resource libraries supplied in the installation tree. These libraries help to minimize errors during compilation and simulation startup. Also, if you make changes to a single Verilog

module in a given library, ModelSim recompiles only that module, rather than all modules in the design.

Related Topics

[Working Library Versus Resource Libraries](#)

[Library Window Contents](#)

[Working with Design Libraries](#)

[Verilog Resource Libraries](#)

[VHDL Resource Libraries](#)

[Creating a Library](#)

Mapping the Logical Work to the Physical Work Directory

VHDL uses logical library names that you map to ModelSim library directories. If you do not map libraries properly before invoking your simulation, the simulation will fail due to not loading the necessary components. Similarly, compilation does depend on proper library mapping.

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

Step 1 — Create Work and Resource Libraries

Before you can compile your source files, you must create a working library in which to store the compilation results.

You use the vlib command to create your working library. The contents of the library change as you update your design and recompile.

The vlib command creates a “flat” library type by default. Flat libraries condense library information into a small collection of files compared to the legacy library type. This remedies performance and capacity issues seen with very large libraries.

Restrictions and Limitations

- Makefile support for flat libraries is limited due to the lack of per-target file objects. However, the resulting makefile will trigger a build of all design units should any source file for any design unit be newer than the library. Optimized design units in flat libraries are also supported, with more precise dependency tracking.

Flows requiring the vmake command can revert to the legacy library type when you do any of the following:

- Specify “-type directory” in the vlib command.
- Set the [DefaultLibType](#) variable in your *modelsim.ini* file to the value 0.
- Set the shell environment variable MTI_DEFAULT_LIB_TYPE to the value 0.
- Use braces ({}) for cases where the path contains multiple items that need to be escaped, such as when the pathname contains spaces or backslash characters. For example:

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

Prerequisites

- Know the paths to the directories that contain your design files and resource libraries.
- Start ModelSim.

Procedure

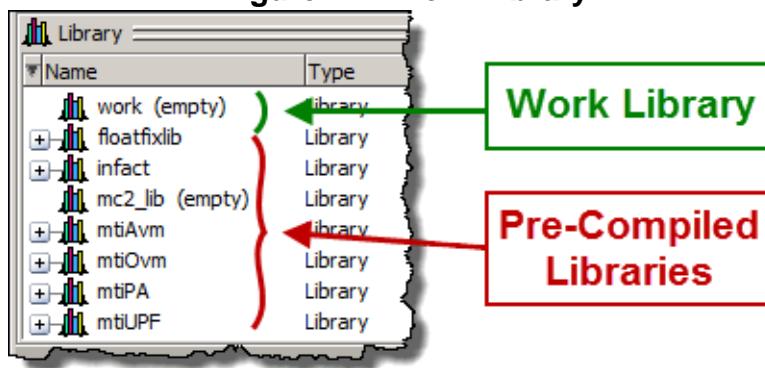
1. Choose **File > Change Directory** from the main menu to open the Browse For Folder dialog box.
2. Navigate to the directory where your source files are located.
3. Create the Logical Work Library with the vlib command in one of the following ways:
 - Enter the vlib command in a shell or the Transcript window:
vlib work
 - Choose **File > New > Library** from the main menu.
4. Map one or more user provided libraries between a logical library name and a directory with the vmap command:

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

Results

Creates a library named *work*, places it in the current directory and displays the work library in the Structure window ([Figure 1-2](#)).

Figure 1-2. Work Library



Related Topics

- [Working Library Versus Resource Libraries](#)
- [Working with Design Libraries](#)
- [Map a Logical Name to a Design Library](#)
- [Getting Started with Projects](#)
- [Creating a Library](#)

Step 2 — Compile the Design

Use the language-specific compiler command to compile your design files into your working directory.

- Verilog and SystemVerilog — Compile with the vlog command.
- VHDL — Compile with the vcom command.

Prerequisites

- Create the *work* library and map required resource libraries to the *work* library. Refer to “[Step 1 — Create Work and Resource Libraries](#)” for more information.

Procedure

Depending on the language used to create your design, use one of the following ModelSim commands to compile the design:

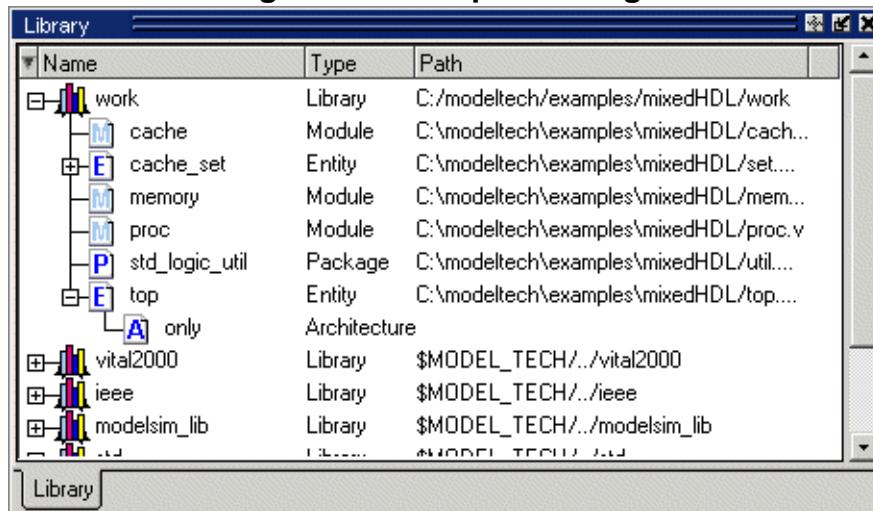
If your source files are written in ...	Enter the following in the Transcript window ...
Verilog and/or SystemVerilog	You can compile Verilog files in any order. For example: vlog gates.v and2.v cache.v memory.v

If your source files are written in ...	Enter the following in the Transcript window ...
VHDL	The vcom command compiles VHDL units in the order 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. For example: vcom v_and2.vhd util.vhd set.vhd

Results

By default, compilation results are stored in the *work* library. (Figure 1-3)

Figure 1-3. Compiled Design



Related Topics

[Verilog Compilation](#)

[Compilation and Simulation of VHDL](#)

[Auto-Generate the Compile Order](#)

Step 3 — Load the Design for Simulation

Use the vsim command to load the design, as defined by specifying the names of any top-level modules (many designs contain only one top-level module).

Prerequisites

- Create the *work* library and map required resource libraries to the *work* library. Refer to [Step 1 — Create Work and Resource Libraries](#) for more information.
- Compile the design. Refer to [Step 2 — Compile the Design](#).

Procedure

Enter the following command on the command line:

vsim testbench globals

where testbench and globals are the two top level modules.

Results

The simulator loads the top-level modules then iteratively loads the instantiated moduels and UDPs in the design hierarchy. This links the design together by connecting the ports and resolving hierarchical references.

Note

 You can incorporate actual delay values to the simulation by applying standard delay format (SDF) back-annotation files to the design.

Related Topics

[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.

Prerequisites

- Have a basic understanding of the following commands, which you commonly use to run a simulation:
 - [add wave](#)
 - [bp](#)
 - [force](#)
 - [run](#)
 - [step](#)

Procedure

Add stimulus to the design, using any of the following methods.

- Language-based test bench.
- Tcl-based ModelSim interactive commands. For example, [force](#) and [bp](#).
- VCD files / commands.

Refer to “[Creating a VCD File](#)” and “[Using Extended VCD as Stimulus](#).”

- Third-party test bench generation tools.

Related Topics

[Verilog and SystemVerilog Simulation](#)

[VHDL Simulation](#)

Step 5 — Debug the Design

You can debug your design from the ModelSim GUI.

Procedure

Use any or all of the following commands to begin interactively debugging your simulation:

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

General Modes of Operation

You can use the three general modes of operation to interact with ModelSim: GUI mode, command line mode, and batch mode.

The ModelSim User's Manual focuses primarily on the graphical user interface (GUI) mode of operation—interacting with your simulation by working in the ModelSim desktop with windows, menus, and dialog boxes. However, ModelSim also has a command line mode and batch mode for compiling and simulating a design.

The following table provides short descriptions of the modes.

Table 1-1. General Modes for ModelSim

Mode	Characteristics
GUI	<p>You can invoke this mode:</p> <ul style="list-style-type: none">• by specifying vsim from the OS command or shell prompt• by specifying vsim -gui from the OS command or shell prompt• by specifying vsim -i from the OS command or shell prompt• from a Windows desktop icon <p>It is recommended for viewing waveforms and graphic-based debugging.</p>
Command Line Mode	<p>You can invoke this mode using the -c argument of the vsim command: vsim -c</p> <p>This mode is non-interactive and displays no GUI.</p> <p>Supports all commands that are not GUI-based.¹</p> <p>It is recommended for DO file based simulations, and executing commands from a prompt.</p>
Batch Mode Simulation	<p>You can invoke this mode using the -b argument of the vsim command: vsim -b</p> <p>This mode invokes using a batch script. It is non-interactive and displays no GUI or command line. Most commands and command options are supported for use in the script.¹</p> <p>It is recommended for large, high-performance simulations.</p>

1. Refer to the [Supported Commands](#) table in the Command Reference Manual to see which commands are available for use with vsim -c and vsim -batch.

Command Line Mode	40
Batch Mode Simulation.....	43

Command Line Mode

Command line simulations are initiated from a Windows or shell command prompt and can be either interactive or non-interactive.

Most command line simulations operate in non-interactive mode. For example, when a DO file is being processed or a stdin redirect is present. Otherwise, the simulator operates in interactive mode—for example, when a DO file script requires input from the user to continue.

Note

 You can use CTRL-C to terminate batch simulation in both the shell and Windows environments.

Startup Variable Flow	40
Here-Document Flow	41
I/O Redirection Flow	41
Supported Commands for Command Line Mode	42

Startup Variable Flow

In command line mode, ModelSim runs any startup command specified by the Startup variable in the *modelsim.ini* file. If you invoke vsim with the -do “command_string” option, the DO file executed in this manner overrides any startup command in the *modelsim.ini* file.

If you invoke vsim from within a vsim invocation, some arguments are ignored because it is too late for them to be applied. Arguments that are ignored include the following:

-batch	-dpicpppath	-mc2
-c	-elab_cont	-mvchome
-cppinstall	-geometry	-name
-cpppath	-gui	-sync
-colormap	-i	-visual
-display	-lic_*	
-dpicppinstall	-load_elab	

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 of the project file (<Project_Root_Dir>/<Project_Name>.mpf).

Related Topics

[Startup](#)

[vsim \[ModelSim Command Reference Manual\]](#)

Here-Document Flow

You can use the “here-document” technique to enter a string of commands in a shell or Windows command window. You invoke vsim and redirect standard input using the exclamation character (!) to initiate and terminate a sequence of commands.

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

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

The file *test.do* can run until completion or contain commands that return control of the simulation to the command line and wait for user input. You can also use this technique to run multiple simulations.

I/O Redirection Flow

You can use a script with output and input redirection to and from user specified files. The script can be set up to run interactively or non-interactively.

For example:

```
vsim -c counter <infile >outfile
```

where “counter” is the design top, “infile” represents a script containing various ModelSim commands, and the angle brackets (< >) are redirection indicators.

Use the [batch_mode](#) command to verify that you are in Command Line Mode. stdout returns “1” if you specify batch_mode while you are in Command Line Mode (vsim -c) or Batch Mode (vsim -batch).

DO Files Generated from Transcript Files

ModelSim creates a transcript file during simulation that contains stdout messages. You can use this transcript file as the basis for a DO file if you invoke the transcript on command after the design loads, which writes all of the commands you invoke to the transcript file.

Run the following command:

```
vsim -c top
```

After reviewing the library and design loading messages you can then run the commands:

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

These commands result in a transcript file, which you can use for command input if you re-simulate *top*. Be sure to remove the quit -f command from the transcript file if you want to remain in the simulator.

You should rename a transcript file that you intend to use as a DO file. If you do not rename the file, ModelSim overwrites it the next time you run vsim. Also, simulator messages are already commented out with the pound sign (#), but any messages generated from your design (and subsequently written to the transcript file) causes the simulator to pause. A transcript file that contains only valid simulator commands works fine; use a pound sign to comment out anything else.

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

Related Topics

[Default stdout Messages](#)

[Stats](#)

[vsim command \[ModelSim Command Reference Manual\]](#)

Supported Commands for Command Line Mode

GUI-based commands are not available for use with vsim -c.

Refer to the [Supported Commands](#) table to see which commands you can use with vsim -c.

Batch Mode Simulation

Batch Mode is an operational mode to perform simulations without invoking the GUI.

Execute the simulations by invoking scripted files from a Windows command prompt or Linux®¹ shell. Batch mode does not provide for interaction with the design during simulation. The simulation run typically sends data to (standard output) stdout, which you can redirect to a log file.

Simulating with Batch Mode can yield faster simulation times, especially for simulations that generate a large amount of textual output. Refer to “[Saving Batch Mode Simulation Data](#)” for information about saving transcript data.

The commands within a DO file script for Batch Mode simulation are similar to those available for Command Line Mode (vsim -c). However, you cannot use all commands or command options with vsim -batch. Refer to the Commands chapter in the Reference Manual to see which commands you can use with vsim -batch.

You can enable batch mode with either of the following methods:

- **vsim** -batch — Enable batch mode for individual simulations with the -batch argument.
- **BatchMode** *modelsim.ini* variable — Enable this variable to turn on batch mode for all your simulations. If you set this variable to 0 (default), vsim runs as if you specified the vsim -i option. Transcript data goes to stdout by default. Automatically create a log file by enabling the **BatchTranscriptFile** *modelsim.ini* variable.

Note

 You receive a warning message if you specify vsim -batch with the -c, -gui, or the -i options, and -c, -gui, and -i are ignored. If you enable the **BatchMode** variable, vsim bypasses the variable if you specify the -batch, -c, -gui, or -i options to vsim.

Saving Batch Mode Simulation Data	43
Simulator Control Variables.....	44

Saving Batch Mode Simulation Data

Capture simulation data during batch mode for post-simulation analysis.

1. Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

Procedure

1. Determine how you want to view or save the transcript information.

If you want to ...	Do the following:
Send tool output and your code output to stdout	This is the default behavior. vsim -batch ...
Send tool output to a log file	Add the following arguments to your vsim command line: vsim -batch -nostdout -logfile filename ...
Redirect the tool output and your code output to an external file	Use file redirection with your vsim command line: vsim -batch ... > filename
Send the tool output and your code output to stdout and tool output to a log file (not recommended)	Add the following arguments to your vsim command line: vsim -batch -logfile filename ...
Always send output to a logfile without command line arguments	Enable the BatchTranscriptFile <i>modelsim.ini</i> variable

2. Determine how you want vsim to generate the transcript information.

If you want the information to be ...	Do the following:
Unbuffered, showing immediate results	Add the -syncio argument to your vsim command line.
Buffered, thus not available until the buffer is full or the simulation is complete	Add the -nosyncio argument to your vsim command line.

Related Topics

[BatchMode](#)

Simulator Control Variables

Control the batch-mode simulation using *modelsim.ini* variables and Tcl variables.

Available *modelsim.ini* variables are as follows:

AccessObjDebug	IgnoreSVAError	StdArithNoWarnings
BreakOnAssertion	IgnoreSVAFatal	UserTimeUnit
CheckpointCompressMode	IgnoreSVAInfo	PrintSimStats
ClassDebug	IgnoreSVAWarning	WildcardFilter

DefaultForceKind	IgnoreWarning	WLFCompress
DefaultRadix	IterationLimit	WLFFilename
DelayFileOpen	NoQuitOnFinish	WLFMCL
ForceSigNextIter	NumericStdNoWarnings	WLFOptimize
GCThreshold	OnBreakDefaultAction	WLFSIZElimit
IgnoreError	OnErrorDefaultAction	WLFTimeLimit
IgnoreFailure	PathSeparator	WLFUseThreads
IgnoreNote	RunLength	

Available Tcl variables are as follows:

now	library	architecture
delta	entity	resolution

Related Topics

[modelsim.ini Variables](#)

Default stdout Messages

By default, the simulator sends information about the simulator, commands executed, start time, end time, warnings, errors, and other data to stdout.

Tool Statistics Messages 46

Tool Statistics Messages

Each time you enter a command, the tool prints and sends this information to the Transcript window and/or a logfile.

The tool displays the data similarly to the following format:

```
1 # vsim topopt -c -do "run -all; quit -f" -warning 3053
2 # Start time: 18:06:45 on May 13,2014
3 # // Questa Sim-64
4 # // Version <information>
5 # Loading sv_std.std
6 # Loading work.top(fast)
7 # Loading work.pads(fast)
8 # ** Warning: (vsim-3053) test.sv(2): Illegal output or inout port
connection for "port 'AVSS'".
9 # Region: /top/pads
10 # run -all
11 # 0: Z=1, AVSS=0
12 # quit -f
13 # End time: 18:06:45 on May 13,2014, Elapsed time: 0:00:00
14 # Errors: 0, Warnings: 1
```

- Line 1 — The command with arguments.
- Line 2 — The time and date the command was executed.
- Line 3 — Executable information.
- Line 4 — Release information: including the number and letter release, the executable type, such as compiler (vlog, vcom), OS version, and build date.
- Lines 5 through 12 — Logged messages.
- Line 13 — The time and date the command finished, and elapsed time.
- Line 14 — The total number of errors and warnings in the following format: Errors: [number], Warnings [number], Suppressed Errors: [number], Suppressed Warnings: [number]. For zero suppressed errors and warnings, the corresponding count message is not displayed.

Definition of an Object

Because ModelSim supports a variety of design languages (Verilog, VHDL, and SystemVerilog), the documentation and the interface use the word “object” to refer to any valid design element in those languages.

Table 1-2 summarizes language-specific elements that define an object.

Table 1-2. 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

Standards Supported

ModelSim supports most industry standards.

Standards documents are sometimes informally referred to as a Language Reference Manual (LRM). Elsewhere the documentation may refer to only the IEEE Std number.

ModelSim supports the following standards:

- VHDL
 - IEEE Std 1076-2008, *IEEE Standard VHDL Language Reference Manual*.
ModelSim supports the VHDL 2008 standard features, with a few exceptions. For detailed standard support information see the vhdl2008 technote available at `<install_dir>/docs/technotes/vhdl2008.note`, or from the GUI menu **Help > Technotes > vhdl2008**.

The vhdl2008migration technote addresses potential migration issues and mixing use of VHDL 2008 with older VHDL code.

- IEEE Std 1164-1993, *Standard Multivalue Logic System for VHDL Model Interoperability*
- IEEE Std 1076.2-1996, *Standard VHDL Mathematical Packages*

Any design developed with ModelSim is 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 1800-2012. *IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language*
- ModelSim supports both PLI (Programming Language Interface) and VCD (Value Change Dump).
- 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*

Text Conventions

This manual uses a set of textual conventions.

Table 1-3. 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
monospace type	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: File > Quit
UPPER CASE	denotes file types used by ModelSim (such as DO, WLF,INI, MPF, PDF.)

Chapter 2 Projects

Projects simplify the process of compiling and simulating a design and are useful for getting started with ModelSim.

What are Projects?	50
What are the Benefits of Projects?	50
Project Conversion Between Simulator Versions.....	51
Getting Started with Projects	52
Open a New Project	52
Add Source Files to the Project	54
Compile the Files	55
Change Compile Order	56
Auto-Generate the Compile Order	57
Grouping Files	57
Simulate a Design	58
The Project Window	60
Creating a Simulation Configuration	61
Organizing Projects with Folders.....	64
Adding a Project Folder	64
Set File Properties and Project Settings	66
File Compilation Properties	66
Project Settings	68
Setting Custom Double-click Behavior	69
Access Projects from the Command Line	69

What are Projects?

A project is a collection of files and user-defined settings for designs under specification or test. At a minimum, a project has 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 the following items:

- Source files or references to source files
- Other files, such as READMEs or other project documentation
- Local libraries
- References to global libraries
- Simulation configurations
- Folders

What are the Benefits of Projects?..... **50**

Project Conversion Between Simulator Versions..... **51**

What are the Benefits of Projects?

Projects offer benefits to both new and advanced users.

- Projects simplify interaction with ModelSim. For example, you do not need to understand the intricacies of compiler switches and library mappings
- Projects eliminate the need to remember the conceptual model of the design; the compile order is maintained for you in the project.

Note



Compile order is maintained for HDL-only designs.

- Projects remove the necessity to re-establish compiler switches and settings for each new session. Settings and compiler switches are stored in the project metadata as are mappings to source files.
- Projects allow you to share libraries without copying files to a local directory. For example, you can establish references to source files that are stored remotely or locally.
- Projects allow you to change individual parameters across multiple files. In previous versions you could only set parameters one file at a time.
- Projects enable "what-if" analysis. For example, you can copy a project, manipulate the settings, and rerun the simulation to observe the new results.

- Projects reload the initial settings from the project *.mpf* file every time you open the project.

Related Topics

[Creating a Simulation Configuration](#)

[Organizing Projects with Folders](#)

Project Conversion Between Simulator 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 canceling the operation.

Choosing to convert a project to a newer version creates a backup of the original project before the conversion occurs. The backup file is named *<project name>.mpf.bak*, and it appears in the same directory in which the original project is located.

Getting Started with Projects

Your initial setup, compilation, and simulation of a design consists of working with several windows and dialog boxes.

Open a New Project	52
Add Source Files to the Project	54
Compile the Files	55
Change Compile Order	56
Auto-Generate the Compile Order	57
Grouping Files	57
Simulate a Design.....	58

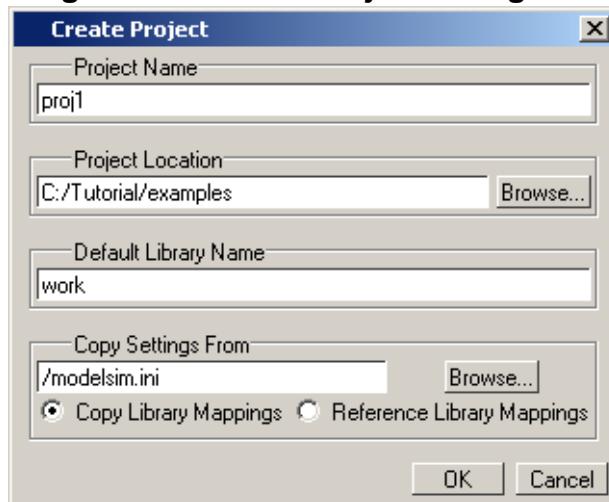
Open a New Project

To create a new ModelSim project, you open one from scratch and specify the details of its initial configuration.

Procedure

1. Select **File > New > Project** to create a new project. This opens the **Create Project** dialog box, as shown in [Figure 2-1](#).
2. Specify a project name, location, and default library name. You can generally leave the **Default Library Name** field set to "work" (as shown). The name you specify will be used to create a working library subdirectory within the Project Location. The **Copy Settings From** field allows you to import library settings from a selected .ini file instead of copying them directly into the project.

Figure 2-1. Create Project Dialog Box

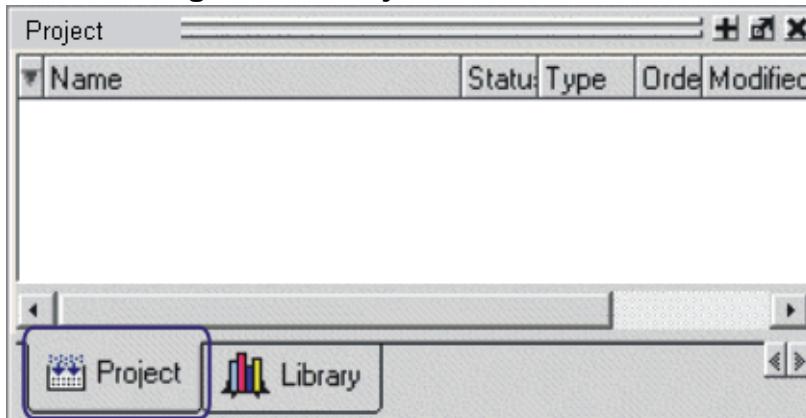


3. Click **OK**.

Results

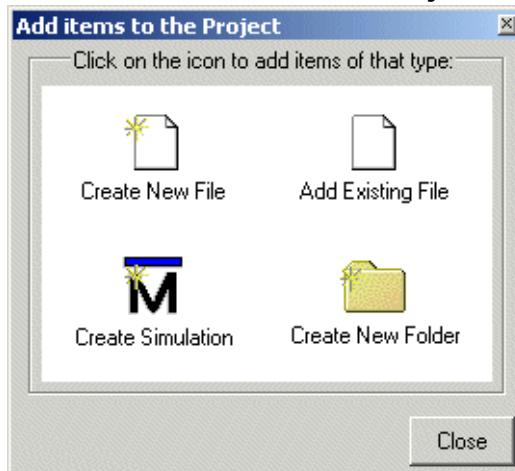
A blank Project window opens in the Main window (Figure 2-2)

Figure 2-2. Project Window Detail



and the Add Items to the Project dialog box opens. (Figure 2-3)

Figure 2-3. Add items to the Project Dialog



The name of the current project appears at the bottom bar of the Main window.

If you exit ModelSim with a project open, ModelSim automatically opens that same project upon startup.

You can open a different or existing project by selecting **File > Open** and choosing Project Files from the **Files of type** dropdown list.

To close a project file, right-click in the Project window and choose **Close Project**. This closes the Project window but leaves the Library window open. You cannot close a project while a simulation is in progress.

Add Source Files to the Project

Once you have created a project, you need to add the design files. You can either write and edit a new source file, or add a pre-existing file.

Procedure

1. Create a new project file
 - a. Choose **Project > Add to Project > New File** (the Project window must be active). This opens the Create Project File dialog box ([Figure 2-4](#)).

Figure 2-4. Create Project File Dialog

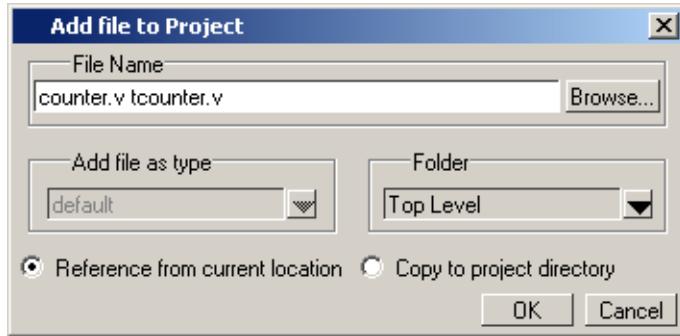


- b. Specify a name, file type, and folder location for the new file.

When you click OK, the file is listed in the Project window. If you double-click the name of the new file in the Project window, a Source editor window opens, where you can create source code.

2. Add an existing file.
 - a. Choose **Project > Add to Project > Existing File**.

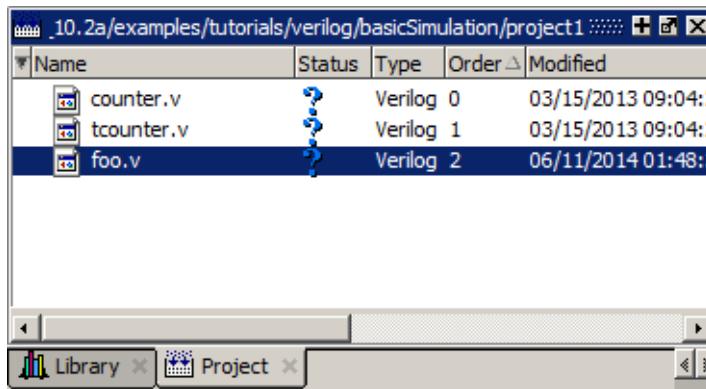
Figure 2-5. Add file to Project Dialog



- b. Click OK.

Results

The files are added to the Project window.



Tip

- i** You can send a list of all project filenames to the Transcript window by entering the command `project filenames`. This command works only when a project is open.

Compile the Files

The question marks in the Status column in the Project window indicate that either the files have not been compiled into the project or that the source has changed since the last compile.

Note

- Project metadata is 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.

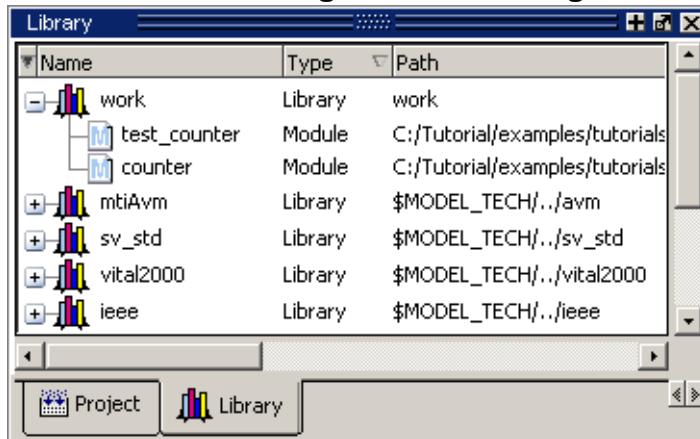
Procedure

Choose **Compile > Compile All** or right-click in the Project window and choose **Compile > Compile All**.

Results

Once compilation finishes, click the Library window, expand the library *work* by clicking the “+”, and you will see the compiled design units.

Figure 2-6. Click Plus Sign to Show Design Hierarchy



Change Compile Order

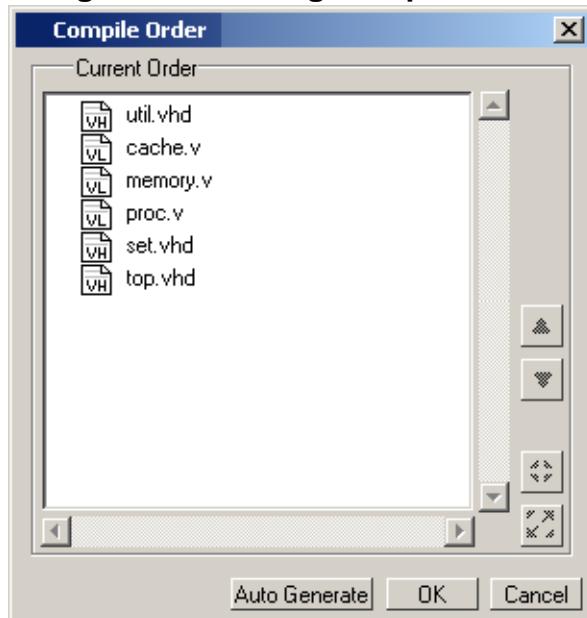
The Compile Order dialog box is functional for HDL-only designs. When you compile all files in a project, ModelSim by default compiles the files in the order in which they were added to the project.

You have two alternatives for changing the default compile order:

- Select and compile each file individually
- Specify a custom compile order

Procedure

1. Choose **Compile > Compile Order** from the main menu or from the context menu in the Project window.

Figure 2-7. 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-Generate the Compile Order

If you have an HDL-only design, you can automatically generate the compile order of its files.

When you click the **Auto Generate** button in the Compile Order dialog box (Figure 2-7), ModelSim 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) cannot be compiled for reasons other than dependency.

You can display files in the Project window in alphabetical or in compilation order (by clicking the column headings). Keep in mind that the order you see in the Project window is not necessarily the order in which the files will be compiled.

Grouping Files

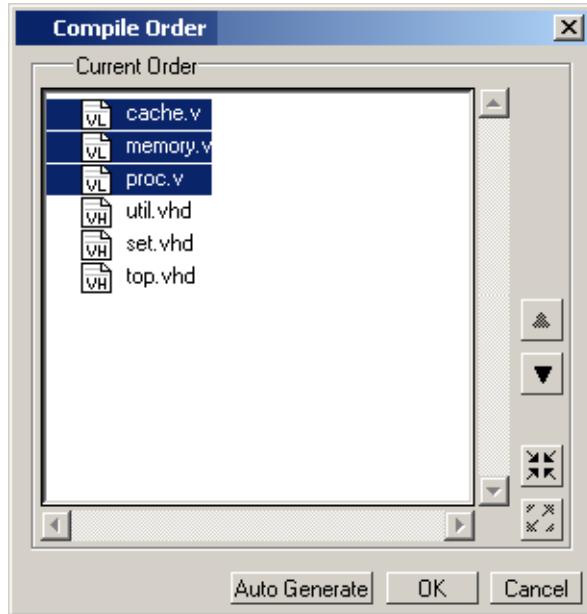
You can group two or more files in the Compile Order dialog so they are sent to the compiler at the same time.

For example, you might have one file with several Verilog define statements and a second file that is a Verilog module. Typically, you would want to compile these two files together.

Procedure

1. Select the files you want to group.

Figure 2-8. Grouping Files



2. Click the Group button.

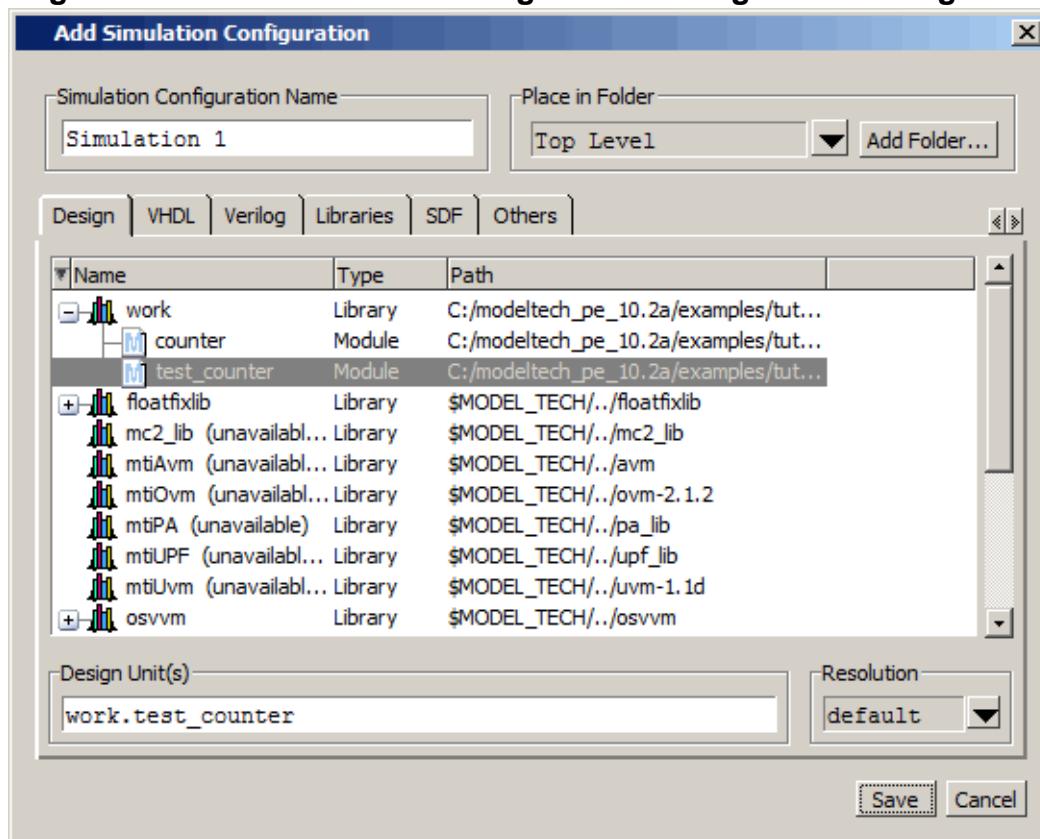
To ungroup files, select the group and click the Ungroup button.

Simulate a Design

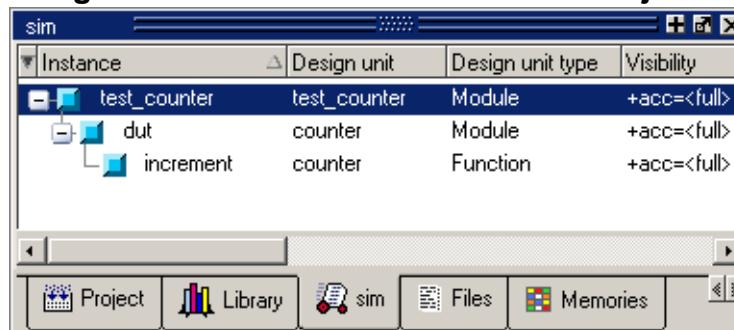
After you have finished compiling the files contained in your design, you can begin simulation.

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 choose **Simulate** from the popup menu.
- Choose **Simulate > Start Simulation** from the main menu to open the Add Simulation Configuration dialog box (Figure 2-9). Select a design unit in the Design tab. Set other options in the VHDL, Verilog, Libraries, SDF, and Others tabs. Click OK to start the simulation.

Figure 2-9. Add Simulation Configuration Dialog Box — Design Tab

A new Structure window, named *sim*, appears that shows the structure of the active simulation (Figure 2-10).

Figure 2-10. Structure Window with Projects

At this point, you can run the simulation and analyze your results. Typically, you would 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.

The Project Window

To access:

- New Project: **File > New > Project.**
- Saved Project: **File > Open > Files of Type > Project File (.mpf)**

The Project window contains information about the objects in your project. By default the window is divided into five columns. You can display this window to create a new project or to work on an existing project that you have saved

Figure 2-11. 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		

Objects

- Column titles
 - **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 has not 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 ModelSim compiles the file when you run 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 a column title and selecting or deselecting entries.

Usage Notes

You can sort the list by any of the five columns. Click 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 is used to sort the list, and whether the sort order is descending (down arrow) or ascending (up arrow).

Creating a Simulation Configuration

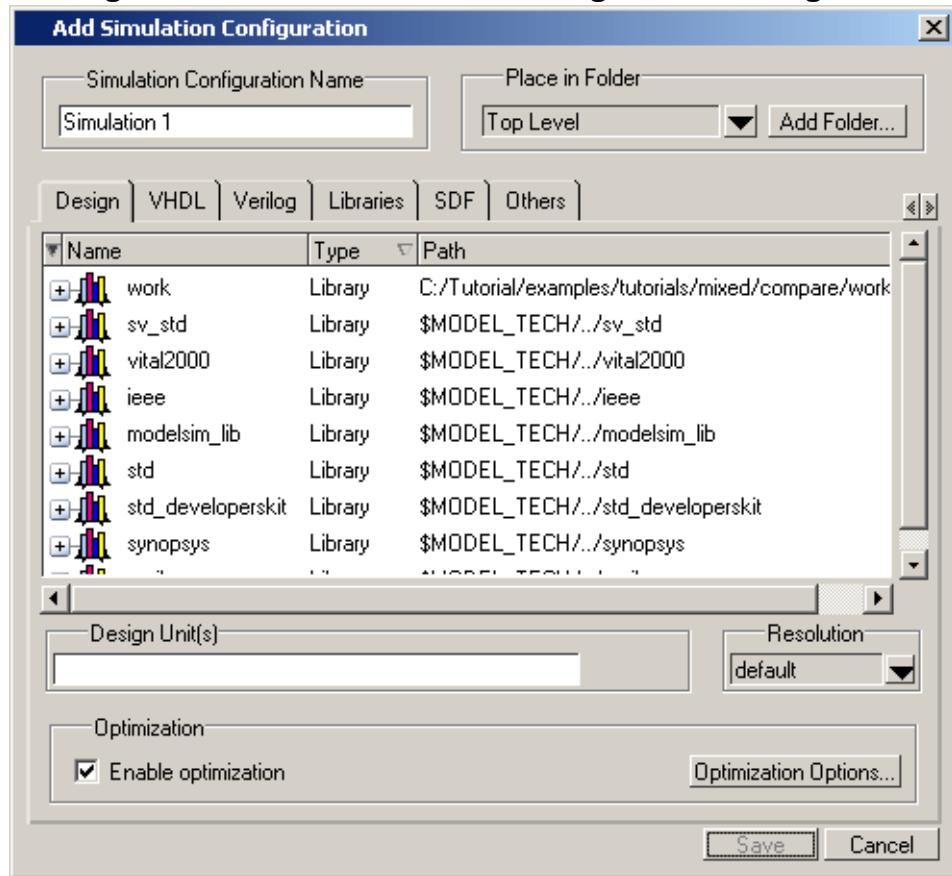
A Simulation Configuration associates a design unit(s) and its simulation options. Ordinarily, you would have to specify simulation options each time you load the design. With a Simulation Configuration, you specify the design and simulation options and then save the configuration with a name.

For example, assume you routinely load a particular design and you also have to specify the simulator resolution limit, generics, and SDF timing files. With a Simulation Configuration, you would specify the design and those options and then save the configuration and name it *top_config*. This name is then listed in the Project window where you can double-click it to load the design along with its options.

Procedure

1. Add a simulation configuration to the project by doing either of the following:
 - Choose **Project > Add to Project > Simulation Configuration** from the main menu.
 - Right-click the Project window and choose **Add to Project > Simulation Configuration** from the popup menu in the Project window.

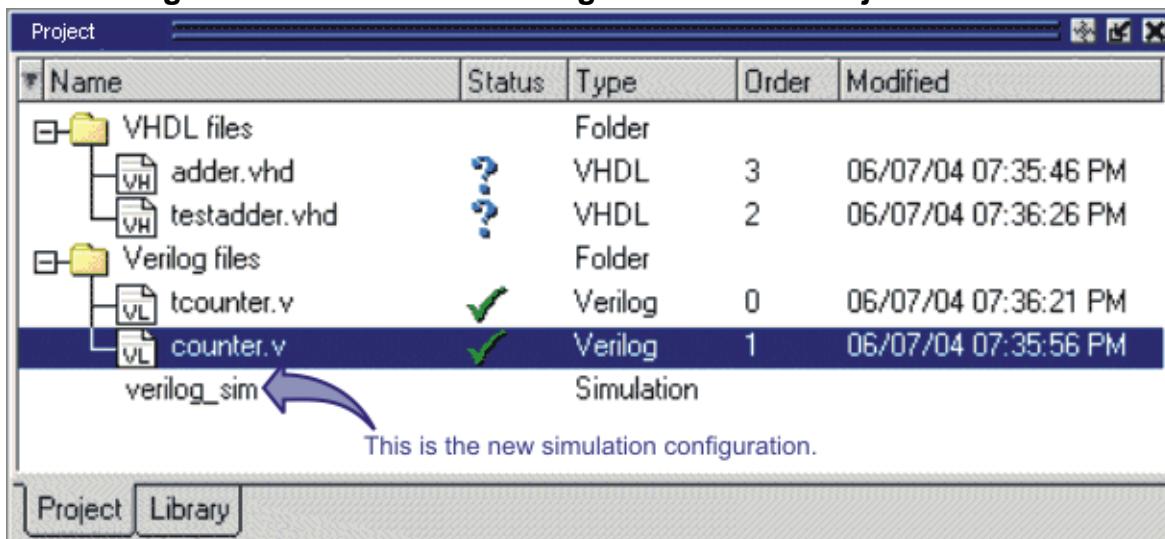
This displays the dialog box shown in [Figure 2-12](#).

Figure 2-12. Add Simulation Configuration Dialog Box

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 box to specify any required simulation options.
6. Click **OK**

Results

- The simulation configuration is added to the Project window, as shown in [Figure 2-13](#).
- As noted, the name of the new simulation configuration you have added is *verilog_sim*.
- To load the design, double-click on *verilog_sim*.

Figure 2-13. Simulation Configuration in the Project Window

Organizing Projects with Folders

Adding more files to a project makes it more difficult to locate the item you need. You can add one or more folders to a project and use them to organize the files that you have added.

Adding a Project Folder **64**

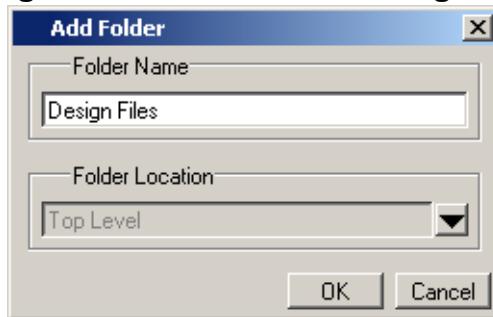
Adding a Project Folder

Project folders are similar to directories in that they are containers that allow you to organize multiple levels of folders and sub-folders. However, no actual project directories are created in the file system—the folders are present only within the project file.

Procedure

1. Choose **Project > Add to Project > Folder** or right-click in the Project window and choose **Add to Project > Folder**.

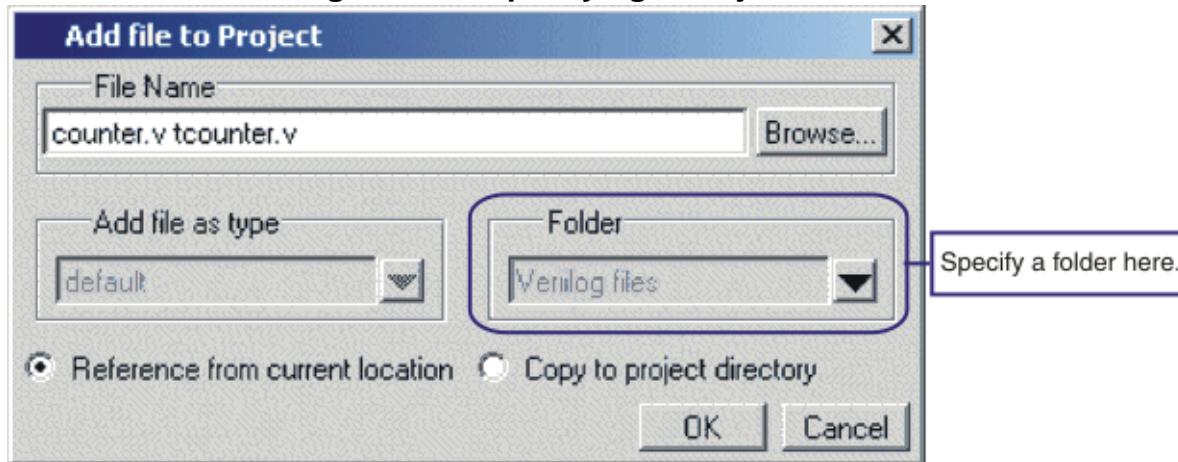
Figure 2-14. Add Folder Dialog Box



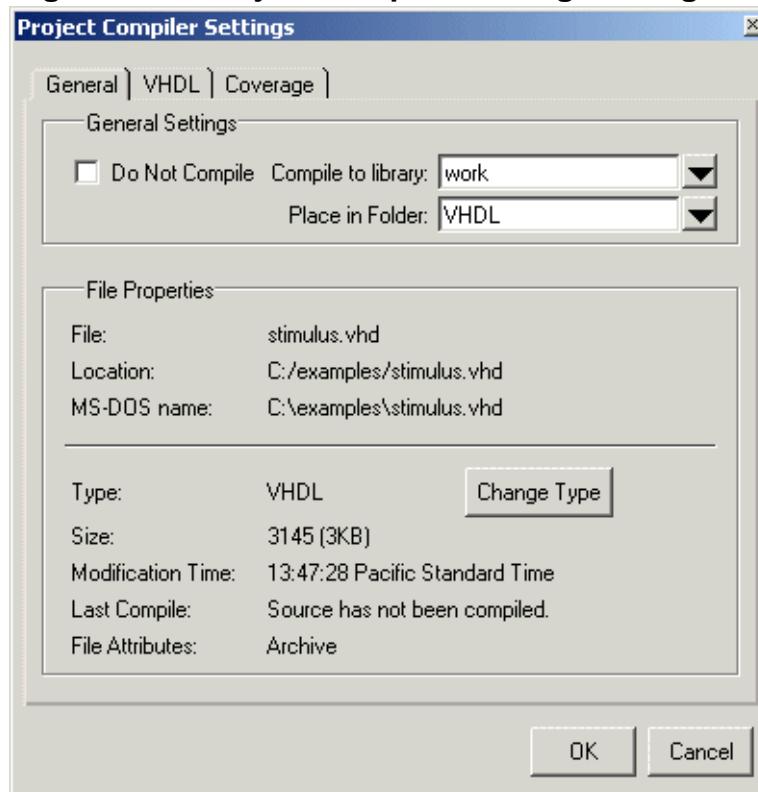
2. Specify the Folder Name, the location for the folder, and click **OK**. The folder will be displayed in the Project tab.

Examples

For example, when you add a file, you can select which folder to place it in.

Figure 2-15. Specifying a Project Folder

If you want to move a file into a folder later on, you can do so using the Properties dialog box for the file. To display this dialog box, right-click on the filename in the Project window and choose Properties from the context menu. This opens the Project Compiler Settings dialog box (Figure 2-16). Use the Place in Folder field to specify a folder.

Figure 2-16. Project Compiler Settings Dialog Box

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

Set 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	66
Project Settings	68
Setting Custom Double-click Behavior	69

File Compilation Properties

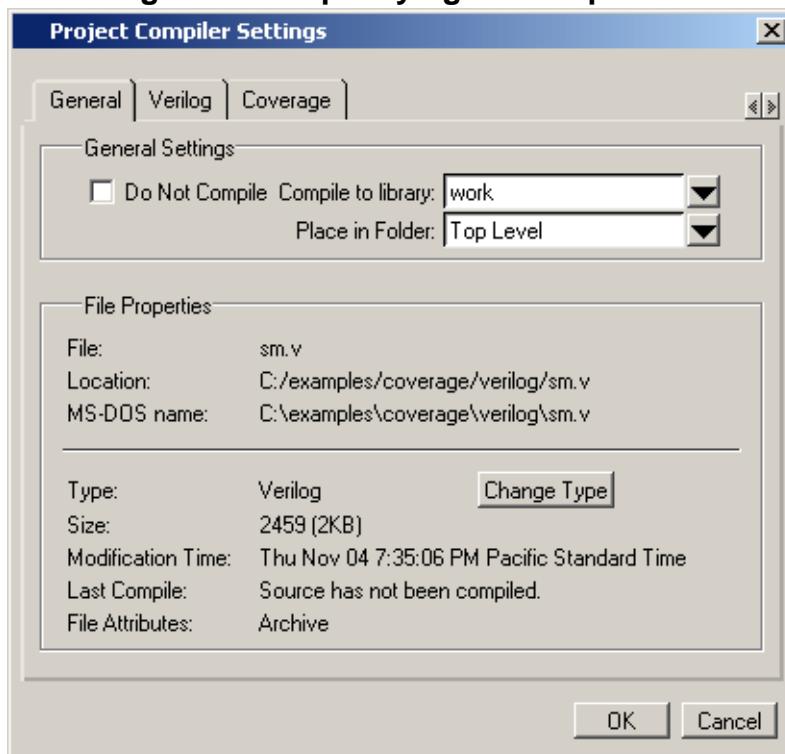
The VHDL and Verilog compiler commands (vcom and vlog, respectively) have numerous arguments that affect how a design is compiled and subsequently simulated. You can use the arguments of these commands to customize the settings on individual files or a group of files.

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 the file names, and choose **Properties** to display the Project Compiler Settings dialog box ([Figure 2-17](#)). The appearance of this dialog box can vary, depending on the number and the type of files you have selected. If you select a single VHDL or Verilog file, you will see the following tabs:

- General tab — properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed.
- Coverage tab
- VHDL or Verilog tab — 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 2-17. Specifying File Properties

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

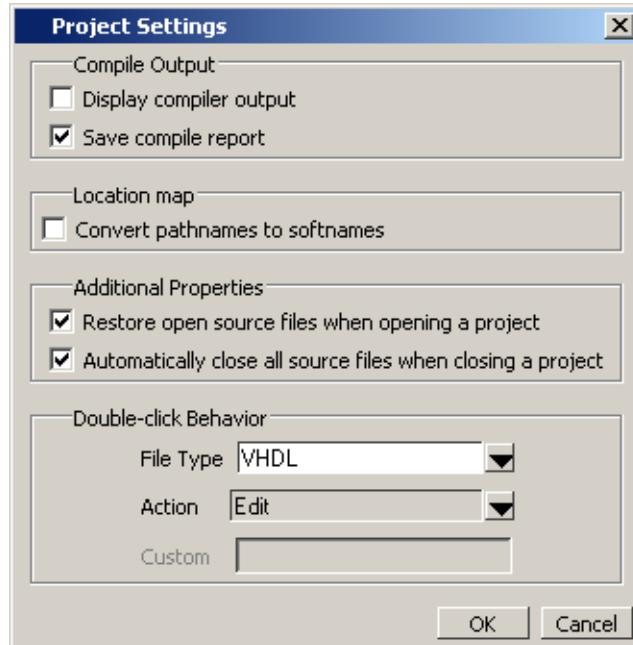
- If two or more files have different settings for the same option, the checkbox in the dialog will be inactive ("grayed out").
- If you change the option, you cannot change it back to a "multi- state setting" without canceling out of the dialog box.
- If you select a combination of VHDL and Verilog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

Project Settings

To modify project settings, right-click anywhere within the Project window and choose **Project Settings** from the popup menu. This opens the Project Settings Dialog Box.

The Project Settings Dialog Box allows you to select the compile output you want, the location map, what to do with source files when you open or close a project, and how the double-click action of your mouse will operate on specific file types.

Figure 2-18. Project Settings Dialog Box



Convert Pathnames to Softnames for Location Mapping 68

Convert Pathnames to Softnames for Location Mapping

If you are using a location map, you can convert relative pathnames, full pathnames, and pathnames with an environment variable into a soft pathname.

Tip

i The term soft pathname (or softname) denotes a pathname that is defined by location mapping, which uses the environment variable named MGC_LOCATION_MAP. A soft pathname contains an environment variable that identifies the source of the path instead of specifying the full absolute path to the physical location. This type of file specification identifies the source using the location map rather than the current operating environment.

Prerequisites

- Under the Location map section of the Project Settings dialog box (Figure 2-18), enable the checkbox for “Convert pathnames to softnames.”

Procedure

1. Right-click anywhere within the Project window and select **Project Settings**
2. Enable “Convert pathnames to softnames” in the Location map area of the **Project Settings** dialog box ([Figure 2-18](#)).

Results

When you enable the conversion, all pathnames currently in the project are converted to softnames, as are any that are added later.

During conversion, if there is no softname in the mgc location map that matches the entry, the pathname is converted to its absolute (hard) pathname. This conversion consists of removing the environment variable or the relative portion of the path.

Related Topics

[Using Location Mapping](#)

Setting Custom Double-click Behavior

Use the **Project Settings** dialog box to control the double-click behavior of the **Project** window.

Procedure

1. Select the desired File Type in the Double-click Behavior pane.
2. Select Custom from the Action dropdown.
3. In the Custom text box, enter a Tcl command that uses %f for filename substitution.

Examples

The following example shows how the Custom text box could appear:

```
notepad %f
```

This causes the double-click behavior to substitute %f with the filename that was clicked and then execute the string.

Access Projects from the Command Line

Generally, you access a project from within the ModelSim GUI. However, you can access a project from a standalone tool if you invoke it 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 3

Design Libraries

VHDL designs are associated with libraries, which are objects that contain compiled design units. Verilog and SystemVerilog designs simulated within ModelSim are compiled into libraries as well.

Design Library Overview	72
Working with Design Libraries	74
Verilog Resource Libraries.....	81
VHDL Resource Libraries	84
Importing FPGA Libraries.....	86
Protect Source Code	87

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; and Verilog modules and UDPs (user-defined primitives).

Design units are classified in two ways.

- **Primary design units** — Entities, package declarations, configuration declarations, modules, and UDPs. A primary design unit within a given library must have a unique name.
- **Secondary design units** — Consist of architecture bodies, and package bodies. A secondary design unit is 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 72

Working Library Versus Resource Libraries 72

Design Unit Information

Information about each design unit in a design library is stored as part of that library.

The following types of information are stored for each design unit in a design library:

- retargetable, executable code
- debugging information
- dependency information

Working Library Versus Resource Libraries

You can designate a design library as a working library or as a resource library, depending on how you want to use it for the current simulation.

You can use a design library in either of the following ways:

- A local working library that contains the compiled version of your design. Only one library can be the working library.
- A resource library.

The contents of your working library 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).

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 their search order (refer to [Verilog Resource Libraries](#) and [VHDL 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, so you do not need to declare it explicitly. It is also the library name used by the compiler as the default destination of compiled design units, so you do not need to map it. In other words, the *work* library is the default *working* library.

Caution

 Do not put any of your files or files from third party vendors into any QuestaSIM working library. In addition do not rename, edit, or change of the permissions of any QuestaSIM generated file within a working directory. These actions may cause problems such as clashing filenames, and problems in the future if any QuestaSIM generated file is updated for improved features.

Working with Design Libraries

The implementation of a design library is not defined within standard VHDL or Verilog. ModelSim implements design libraries as directories, which can have any legal name allowed by the operating system, with one exception: ModelSim does not support extended identifiers for library names.

Creating a Library.....	74
Library Size	75
Library Window Contents	76
Map a Logical Name to a Design Library.....	77
Move a Library	79
Setting Up Libraries for Group Use.....	79

Creating a Library

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.

Note

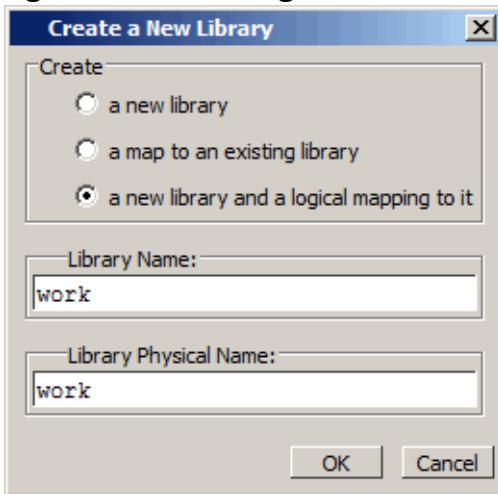
 When you create a project, ModelSim automatically creates a working design library.

Procedure

You can use either of the following methods to create a working design library:

- From the ModelSim prompt or from a UNIX/DOS prompt, use the **vlib** command:
vlib <directory_pathname>
- With the graphic interface, choose **File > New > Library**.

Either method displays the dialog box shown in [Figure 3-1](#).

Figure 3-1. Creating a New Library

Results

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 ([Figure 3-1](#)) or the `vlib` command. Do not try to create a library using UNIX, Linux, DOS, or Windows commands.

Related Topics

[Getting Started with Projects](#)

[modelsim.ini Variables](#)

Library Size

The `-smartdbgsym` argument of the `vcom` and `vlog` commands helps to reduce the size of debugging database symbol files generated at compile time from the design libraries. When you specify `-smartdbgsym`, most design units have their debugging symbol files generated on-demand by `vsim`.

By default, library size reduction is disabled so that a debugging symbol file database is generated for all design units. A companion `SmartDbgSym` variable in `modelsim.ini` allows you to enable or disable this capability for all simulations.

Related Topics

[vcom and vlog. \[ModelSim Command Reference Manual\]](#)

Library Window Contents

You can use either the graphic user interface (GUI) or the command line to view, delete, recompile, or edit the contents of a library. The GUI method is provided in the Library window.

The Library window ([Figure 3-2](#)) provides access to design units (configurations, modules, packages, entities, and architectures) in a library. Categories of information about the design units appear in columns to the right of the design unit name.

Figure 3-2. The Library Window—Design Unit Information in the Workspace

Name	Type	Path
work	Library	C:/modeltech/examples/mixedHDL/work
cache	Module	C:/modeltech/examples/mixedHDL/cache...
cache_set	Entity	C:/modeltech/examples/mixedHDL/set...
memory	Module	C:/modeltech/examples/mixedHDL/mem...
proc	Module	C:/modeltech/examples/mixedHDL/proc.v
std_logic_util	Package	C:/modeltech/examples/mixedHDL/util...
top	Entity	C:/modeltech/examples/mixedHDL/top...
only	Architecture	
vital2000	Library	\$MODEL_TECH/./vital2000
ieee	Library	\$MODEL_TECH/./ieee
modelsim_lib	Library	\$MODEL_TECH./modelsim_lib

The Library window provides a popup menu with various commands that you can display by clicking your right mouse button.

The context menu includes the following commands:

- **Simulate** — Loads the selected design unit(s) and opens Structure (sim) and Files windows. Related command line command is [vsim](#).
- **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 [Map a Logical Name to a Design Library](#)).
- **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](#).
- **Update** — Updates the display of available libraries and design units.

Map a Logical Name to a Design Library

VHDL uses logical library names that you can map 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.

For Verilog and SystemVerilog libraries, ModelSim searches for the mapping of a logical name in the following order:

- A *modelsim.ini* file.
- If the search does not find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, ModelSim searches the current working directory for a subdirectory that matches the logical name.

The compiler generates an error if you specify a logical name that does not resolve to an existing directory.

You can use the GUI, a command, or a project to assign a logical name to a design library. You can also map multiple logical names to the same design library.

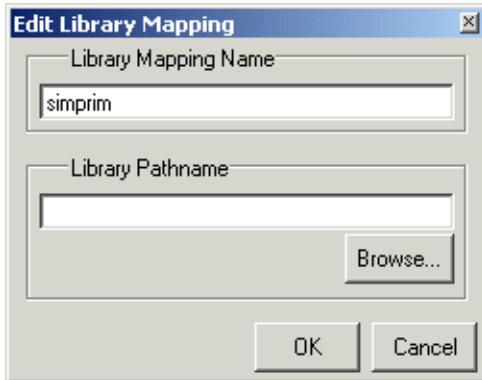
Mapping a Library with the GUI.....	77
Mapping a Library from the Command Line.....	78
Manual Mapping of the <i>modelsim.ini</i> File.....	78

Mapping a Library with the GUI

You can map a library with the GUI using the Edit Library Mapping dialog box.

Procedure

1. Select the library in the Library window.
2. Right-click on the library name, which displays a popup menu.
3. Choose Edit. This displays a dialog box where you can edit the mapping.

Figure 3-3. Edit Library Mapping Dialog Box

The dialog box includes these options:

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

Mapping a Library from the Command Line

Use the vmap command to map a library from the command line.

Procedure

1. Use the **vmap** command. For example:

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

2. You can invoke this command from UNIX, Linux, or DOS prompt or from the command line within ModelSim.
3. The vmap command adds the mapping to the library section of the *modelsim.ini* file.

Manual Mapping of the *modelsim.ini* File

You can map a library by manually modifying the *modelsim.ini* file.

Procedure

1. Open the *modelsim.ini* file with a text editor
2. Add a line under the [Library] section heading using the syntax:

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

To map more than one logical name to a single directory:

- a. Open the *modelsim.ini* file with a text editor

- b. Add a library logical name and pathname for the same library under the [Library] section heading using the syntax. For example:

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

In this example, you can use either the logical name **work** or **my_asic** in a **library** or **use** clause to refer to the same design library.

You can also create a UNIX symbolic link to the library using the ln -s command. For example:

ln -s <directory_pathname> <logical_name>

3. (optional) Use the **vmap** command 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>

Related Topics

[modelsim.ini Variables](#)

Move a Library

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

Setting Up Libraries for Group Use

By adding an “others” specification to your *modelsim.ini* file, you can establish a hierarchy of library mappings. These mappings can reside in other locations. In particular, you can identify a library in a common location that has been made available for group use.

If ModelSim 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” specification.

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 instructs ModelSim to look only in the specified *modelsim.ini* file for a library. It does not load any other part of the specified file.

If two libraries with the same name are mapped to two different locations—one in the current *modelsim.ini* file and the other specified by the others clause—then the mapping specified in the current *.ini* file takes effect.

Verilog Resource Libraries

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	81
Handling Sub-Modules with the Same Name	82
The LibrarySearchPath Variable	83

Library Search Rules

Because instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design.

Top-level modules are loaded from the library named work unless you prefix the modules with the <library> option. If they are not found in the work library, they are searched in the libraries specified with -Lf arguments, followed by libraries specified with -L arguments.

All other Verilog instantiations are resolved in the following order:

- Search libraries specified with -Lf arguments for the vlog or vsim commands 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 for the vlog or vsim commands 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.
- Search the libraries containing top design units that are not explicitly present in the set of -L/-Lf options.

Note

 The -libverbose argument for the [vsim](#) commands provides verbose messaging about library search and resolution operations. Using -libverbose=prlib prints out the -L or -Lf setting used to locate each design unit.

Related Topics

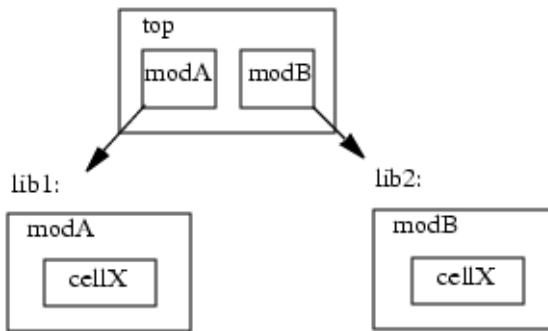
[SystemVerilog Multi-File Compilation](#)

Handling Sub-Modules with the Same Name

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 multiple libraries have sub-modules that have the same name but with different definitions.

This may happen if you are using vendor-supplied libraries. For example, consider the design configuration shown in [Figure 3-4](#).

Figure 3-4. Sub-Modules with the Same Name



The normal library search rules do not work 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 resolve 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
```

The LibrarySearchPath Variable

The LibrarySearchPath variable in the *modelsim.ini* file (in the [vlog] section) defines a space-separated list of resource library paths and/or library path variables. This behavior is identical to the **-L** argument for the vlog command.

```
LibrarySearchPath = <path>/lib1 <path>/lib2 <path>/lib3
```

The default for [LibrarySearchPath](#) is:

```
LibrarySearchPath = mtiAvm mtiOvm mtiUvm mtiUPF
```

Related Topics

[LibrarySearchPath](#)

[vlog. \[ModelSim Command Reference Manual\]](#)

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. The scope does not extend to the next design unit in the file.

Tip

 Note that the **library** clause does not 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, use **vcom -work** and specify the name of the desired target library.

Predefined Libraries	84
Alternate IEEE Libraries Supplied	85
Regenerating Your Design Libraries	85

Predefined Libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard**, **env**, and **textio**. 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*. Do not modify any contents of this predefined library.

A VHDL **use** clause selects 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 should contain the following declarations:

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

To specify referencing of all declarations in a library or package, 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 you have compiled it. There is no limit to the number of libraries that you can reference , but only one library is modified during compilation.

Related Topics

[The TextIO Package](#)

Alternate IEEE Libraries Supplied

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

- *ieepure* — Contains only IEEE approved packages (accelerated for ModelSim).
- *ieee* — (default) 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.

Regenerating Your Design Libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. You do this by using the `-refresh` argument to either the `vcom` or `vlog` command. Check the installation README file to see if your libraries require an update.

By default, using `-refresh` updates the current work library. An important feature of using the `-refresh` argument is that it rebuilds the library image without using source code. This means that you can rebuild models delivered as compiled libraries (without source code) 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.

Restrictions and Limitations

You do not need to regenerate the `std`, `ieee`, `vital22b`, and `verilog` libraries. Also, you cannot use the `-refresh` argument to update libraries that were built before Release 4.6.

You can 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 cannot specify a file name.

Procedure

1. Identify the HDL of the library you want to regenerate: VHDL or Verilog.

2. Determine whether you want to regenerate design units in the work library from the GUI or from the command line:

If you want to update from...	Do the following...
The GUI (VHDL or Verilog)	Choose Library > Regenerate from the main menu.
The command line (VHDL)	Use vcom with the -refresh argument.
The command line (Verilog)	Use vlog with the -refresh argument.

3. (optional) To update a library other than *work* from the command line, use either the vcom or vlog command with the -work <library> argument to regenerate that library. For example, if you have a library named *mylib* that contains both VHDL and Verilog design units, enter the following two commands:

vcom -work mylib -refresh

vlog -work mylib - refresh

Related Topics

[Library Window Contents](#)

[vcom, and vlog. \[ModelSim Command Reference Manual\]](#)

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.

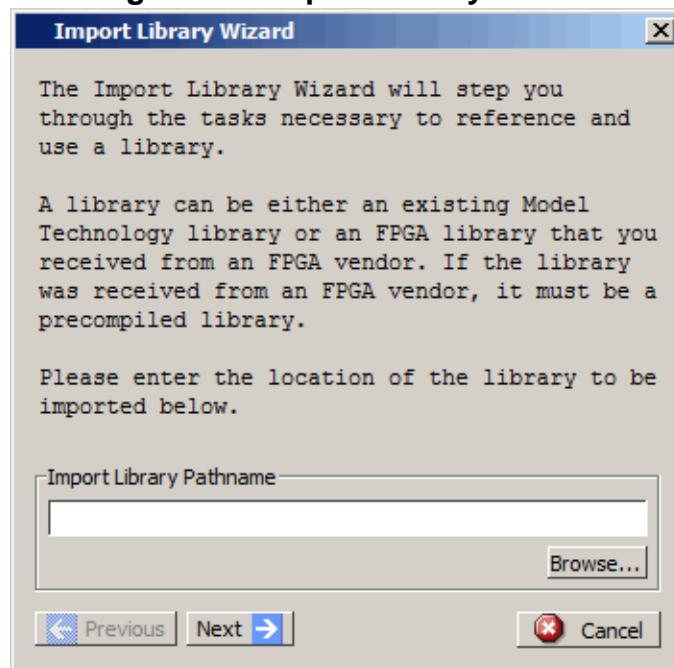
Prerequisites

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

Procedure

1. Select **File > Import > Library** to open the Import Library Wizard. ([Figure 3-5](#))

Figure 3-5. Import Library Wizard



2. Follow the instructions in the wizard to complete the import.

Protect 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 4

VHDL Simulation

ModelSim enables you to compile, optimize, load, and simulate VHDL designs.

Mixed-Language Support	89
Basic VHDL Usage.....	89
Compilation and Simulation of VHDL	91
Usage Characteristics and Requirements	97
The TextIO Package	107
VITAL Usage and Compliance.....	112
VHDL Utilities Package (util).....	114
Modeling Memory	118
VHDL Access Object Debugging	129

Mixed-Language Support

This version of ModelSim supports simulation of mixed-language designs, allowing you to simulate designs written in VHDL, Verilog, and SystemVerilog. While design units must be entirely of one language type, any design unit can instantiate design units from another language. Any instance in the design hierarchy can be a design unit from another language, without restriction.

The basic flow for simulating mixed-language designs is:

1. Compile your HDL source with vcom (VHDL) or vlog (Verilog and SystemVerilog) following order-of-compile rules. Note that VHDL and Verilog observe different rules for case-sensitivity.
2. Simulate your design with the vsim command.
3. Run the simulation and perform any debug processes.

Basic VHDL Usage

Using a VHDL design with ModelSim consists of running the vcom and vsim commands to compile, load, and simulate. Note that you need to be familiar with any setup requirements for running these commands, such as using the vlib command to create a design library.

The following steps summarize the basic VHDL usage process:

1. Compile your VHDL code into one or more libraries using the [vcom](#) command. Refer to [Compilation of a VHDL Design—the vcom Command](#) for more information.
2. Load your design with the [vsim](#) command. Refer to [Simulation of a VHDL Design—the vsim Command](#).
3. Simulate the loaded design, then debug as needed.

Compilation and Simulation of VHDL

The basic operations for using VHDL with ModelSim are establishing a library for compilation results, compilation, and simulation.

Creating a Design Library for VHDL	91
Compilation of a VHDL Design—the vcom Command	91
Simulation of a VHDL Design—the vsim Command	96

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.

Procedure

Use the [vlib command](#) to create a new library. For example:

vlib work

Results

Running the vlib command creates a library named work. By default, compilation results are stored in the work library.

Caution

 The work library is actually a subdirectory named work. This subdirectory contains a special file named `_info`. Do not use a system command to create a VHDL library as a directory—always use the vlib command.

Related Topics

[Design Libraries](#)

Compilation of a VHDL Design—the vcom Command

ModelSim compiles one or more VHDL design units with a single invocation of the vcom command, which functions as the VHDL compiler.

As a default, design units compile in the order 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 either manually order the design units yourself on the command line, or you can use the `-autoorder` argument. An `-autoorder` compilation determines the proper order of VHDL

design units independent of the order that you listed the files on the command line. Compilation proceeds in a scan phase followed by a refresh phase. Without the argument, you must list VHDL files in their proper compilation order.

You can simulate a design written with any of the following versions of VHDL, but you must compile units from each version separately:

- 1076-1987
- 1076-1993
- 1076-2002
- 1076-2008

The **vcom** command compiles using 1076-2002 rules by default; use the -87, -93, or -2008 arguments to compile units written with version 1076-1987, 1076-1993, or 1076-2008 respectively. You can change the default by modifying the **VHDL93** variable in the *modelsim.ini* file (see [modelsim.ini Variables](#) for more information).

Note

 Not all VHDL 1076-2008 constructs are currently supported. From the main window, select **Help > Technotes > vhdl2008** for more information.

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 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.

Note

 This differs from the Verilog and SystemVerilog languages, both of 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 use either of the following methods to convert uppercase letters to lowercase:
 - 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 are addressed in the following ways:
 - **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. Treating VHDL design units as lower case provides compatibility with previous releases, and provides consistent filenames in the file system for make files and scripts.
 - **Verilog packages compiled with -mixedsvvh** — These packages are not affected by VHDL uppercase conversion.
 - **VHDL packages compiled with -mixedsvvh** — These packages are 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 do not have the original case unless you use vcom -lower to compile the source. Port and Generic names in the mtiInterfaceListT structure convert 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 the lowercase name is 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. In these examples, 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 contains 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 to be of a subtype with a range 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) using arguments to the **vcom** command. Or, you can use the [NoRangeCheck](#) and [NoIndexCheck](#) variables in the [vcom] section of the *modelsim.ini* file to specify not to perform checks. Refer to [modelsim.ini Variables](#) for more information.

Generally, disable these checks only after the design is known to be error-free. If you run a simulation with range checking disabled, any scalar values that are out of range display the value in the following format: ?(N) where N is the current value. For example, the range constraint for STD_ULONGIC is 'U' to '-'; if the value is reported as ?(25), the value is out of range because the type STD_ULONGIC value internally is between 0 and 8 (inclusive). Values that are out of range may indicate that an error in the design is not being caught because range checking was disabled.

Range checks in ModelSim are 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. 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 is 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.
- When single-stepping to a subprogram call that 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 use the *mti_inhibit_inline* attribute to disable inlining for individual design units (a package, architecture, or entity) and subprograms. 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;
```

Use the same method to inhibit inlining for entities and architectures.

Simulation of a VHDL Design—the vsim Command

A VHDL design is ready for simulation after you compile it with vcom. You can then use the vsim command to invoke the simulator with the name(s) of the configuration or entity/architecture pair.

Note

 This section discusses invoking simulation from the command line (in Linux or Windows). Alternatively, you can use a project to simulate (refer to [Getting Started with Projects](#)) or use the Start Simulation dialog box (choose **Simulate > Start Simulation** from the main menu).

The following example uses the **vsim** command to begin simulation on a design unit that has an entity named my_asic and an architecture named structure:

```
vsim my_asic structure
```

Timing Specification

The vsim command annotates a design using VITAL-compliant models with timing data from an SDF file. You can specify delay by using the vsim command with the -sdfmin, -sdftyp, or -sdfmax arguments.

The following example annotates maximum timing values for the design unit named my_asic by using an SDF file named f1.sdf in the current work directory:

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

By default, the timing checks within VITAL models are enabled (refer to [VITAL Usage and Compliance](#)). 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. Setting this generic to FALSE disables the actual calls to the timing checks and anything else 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.

Usage Characteristics and Requirements

ModelSim supports the use of VHDL in compliance with the *IEEE Standard VHDL Language Reference Manual* (IEEE Std 1076), which was originally adopted in 1987. This standard has undergone several revisions, each of which is identified by a suffix indicating the year of its approval by the IEEE. There are considerations in using VHDL with ModelSim that are not explicitly covered by the Language Reference Manual (LRM).

Differences Between Supported Versions of the VHDL Standard	97
Naming Behavior of VHDL for Generate Blocks	100
Simulator Resolution Limit for VHDL	101
Default Binding	102
Delta Delays	103

Differences Between Supported Versions of the VHDL Standard

The four different versions of the VHDL standard (IEEE Std 1076) have names that reflect the year each version was approved by the IEEE: 1076-1987, 1076-1993, 1076-2002, and 1076-2008. The default language version supported for ModelSim is 1076-2002.

If your VHDL design was written according to the 1987, 1993, or 2008 version, you may need to update your code or instruct ModelSim to use rules for a 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

Incompatibilities Among Versions of the VHDL Standard

The following is a list of VHDL standard incompatibilities that may cause problems when compiling a design.

Tip

 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". If you have VHDL-93 programs which use PROTECTED as an identifier, you 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" function**— 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, passing files as parameters produces the following warning message:

```
"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** — Compile each package header and body 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 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 will not see any difference in simple variable/signal assignments such as:

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

But you get unexpected results if: you have a function that takes an unconstrained array as a parameter, you then pass a concatenation expression as a formal argument to this parameter, and the body of the function makes assumptions about the direction or bounds of the parameter. This can 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 get an error message like the following:

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

- **'FOREIGN attribute** — In the 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 that depends on one of these sizes can behave incorrectly between standards. This situation occurs most commonly in test suites that check VHDL functionality, though it is 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
```

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), ...
```

The default names appear in the GUI to identify each block. Use the block's default 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 is 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 is 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, and would have appeared as:

```
g1_1, g1_2, ...
```

All previously-generated scripts using this old format should work by default, but you can use the **GenerateFormat** and **OldVhdlForGenNames** *modelsim.ini* variables to ensure that the old and current names are mapped correctly.

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, the representation of time units is referred to as precision or timescale.

Overriding the Default 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
```

Take care when 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 a Resolution Value 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 the vsim command. The advantage of this default binding at load time is that it provides more flexibility for compile order, in that VHDL entities do not 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 can help 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 with which to bind, 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. To counter the IEEE flaw, 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 a configuration declaration contains library and use clauses, use them.

If none of these methods are successful, ModelSim then does the following:

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

- If performing default binding at load time, searches 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 an appropriate binding cannot be made between an entity and an architecture, default port, and generic maps, ModelSim issues an error or warning. You can disable normal default binding methods and require a user specified binding by setting the [RequireConfigForAllDefaultBinding](#) variable in the *modelsim.ini* file to 1 (true), or by specifying the -ignoredefaultbind argument to [vcom](#).

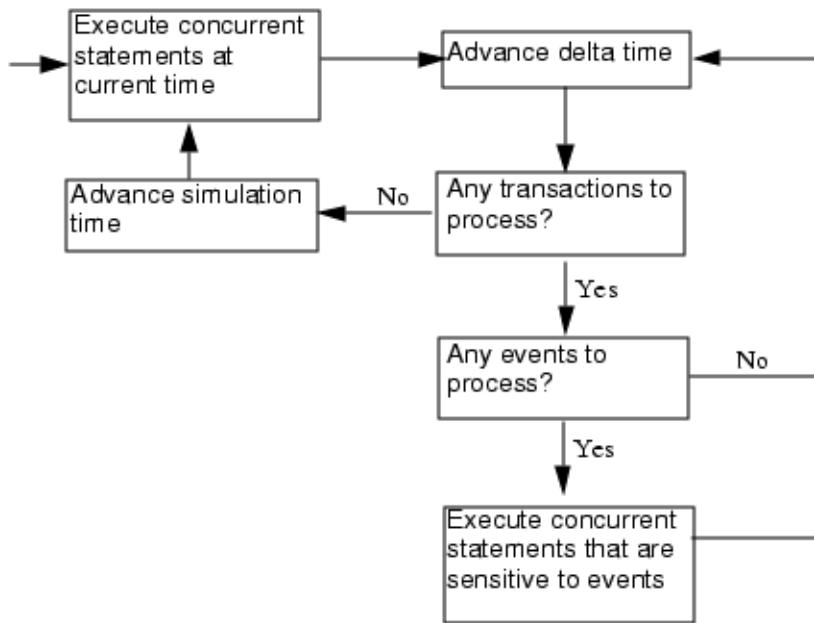
When you specify the [RequireConfigForAllDefaultBinding](#), ModelSim requires you to provide a configuration specification or component configuration in order to bind an entity with an architecture. You must explicitly bind all components in the design through either configuration specifications or configurations. If an explicit binding is not fully specified, defaults for the architecture, port maps, and generic maps are used.

Delta Delays

Event-based simulators such as ModelSim can 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."

[Figure 4-1](#) illustrates the process for VHDL designs. This process continues until the end of simulation time.

Figure 4-1. 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, there are two synchronous processes, one triggered with *clk* and the other with *clk2*. Consider the unexpected situation of the signals changing 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 that 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 value of *inp* appears at *s1* in the same simulation cycle.

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

- Insert a delay at every output
- Use the same clock
- Insert a delta delay

To insert a delta delay, modify the code as follows:

```
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 to observe your signals in the [Wave Window](#) or [List Window](#) (refer to the GUI Reference Manual for more information on these windows). 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 stops the simulation and issues an error message.

The iteration limit default value is 10 million (10000000).

If you receive an iteration limit error, first increase the iteration limit and try to continue simulation, and then try single stepping to attempt to determine which instances in the design may be oscillating.

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.

The TextIO Package

The TextIO package for VHDL is defined in 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	107
STD_INPUT and STD_OUTPUT Within ModelSim	108
TextIO Implementation Issues	108
Alternative Input/Output Files	111
The TEXTIO Buffer	111
Input Stimulus to a Design	111

Syntax for File Declaration

The syntax supported for Text IO can vary according to the version of IEEE Std 1076 you are using.

For IEEE Std 1076-1987, the supported syntax for a file declaration is the following:

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

where "file_logical_name" must be a string expression.

For newer versions of IEEE Std 1076, supported syntax for a file declaration is the following:

```
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, when you declare a file in an architecture, process, or package, the file opens when you start the simulator, and closes when you exit the simulation. When you declare a file in a subprogram, the file opens when the subprogram is called and closes when execution RETURNS from the subprogram.

Alternatively, you can delay the opening of files until the first read or write by setting the [DelayFileOpen](#) variable in the *modelsim.ini* file. Also, you can control the number of concurrently open files with the [ConcurrentFileLimit](#) variable. These variables help you manage a large number of files during simulation. See [modelsim.ini Variables](#) for more details.

STD_INPUT and STD_OUTPUT Within ModelSim

STD_INPUT is a file_logical_name that refers to characters that you enter interactively from the keyboard, and STD_OUTPUT refers to text that displays on the screen. The syntax supported for STD_INPUT and STD_OUTPUT for Text IO can vary according to the version of IEEE Std 1076 you are using.

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.

For IEEE Std 1076-1987, TextIO package contains the following file declarations:

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

For newer versions of IEEE Std 1076, TextIO package contains these file declarations:

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

TextIO Implementation Issues

ModelSim does not fully implement all facets of the TextIO package, which can result in ambiguous results.

WRITE Procedures for 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");
```

causes 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 cannot 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 defines the value to be a STRING and calls the WRITE procedure, and it also 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 io_utils package, 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 often occur when using the TextIO package, because WRITELINE de-allocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

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

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

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

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

The ENDLINE Function

The ENDLINE function — described in the IEEE 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 ENDLINE function has been removed from the TextIO package. The following test can be substituted for this function:

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

The ENDFILE Function

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

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

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

Alternative Input/Output Files

To use the TextIO package to read and write to your own files, you declare an input or output file of type TEXT.

The following examples show how to do this 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";
```

After making these declarations, you then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

The TEXTIO Buffer

Flushing of the TEXTIO buffer depends on whether VHDL files are open for writing.

You can turn the [UnbufferedOutput](#) variable in the *modelsim.ini* file on (1) or off (0, default) to control the status.

Input Stimulus to a Design

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.

The ModelSim installation includes a VHDL test bench as an example. Check for this file in your installation directory:

```
<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.

<http://www.ieee.org>

VITAL Source Code	112
VITAL 1995 and 2000 Packages.....	112
VITAL Compliance	113
Compiling and Simulating with Accelerated VITAL Packages	113

VITAL Source Code

The ModelSim installation includes the source code for VITAL packages.

The source code for VITAL packages is in the following directories:

```
/<install_dir>/vhdl_src/vital2.2b
    /vital1995
    /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, run the following vmap command:

```
vmap ieee <install_dir>/vital1995
```

Alternatively, you can 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

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

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.

Compiling and Simulating with Accelerated VITAL Packages

When you run the vcom command, ModelSim 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 — Performs 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.

If you do not want to use the built-in VITAL routines (when debugging for instance), invoke vcom with the -novital argument. The -novital switch affects calls to VITAL functions only from the design units currently being compiled. Pre-compiled design units referenced from the current design units will still call the built-in functions unless they too are compiled with the -novital argument.

- To exclude all VITAL functions, use -novital all. For example:
vcom -novital all design.vhd
- To exclude selected VITAL functions, use one or more -novital <fname> arguments. For example:

```
vcom -novital VitalTimingCheck -novital VitalAND design.vhd
```

VHDL Utilities Package (util)

The util package contains various VHDL utilities that you can run as ModelSim commands. The package is part of the modelsim_lib library, which is located in the /modeltech tree of your installation directory and is mapped in the default *modelsim.ini* file.

To include the utilities in this package, add the following lines 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;
```

Arguments

None

Return Values

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

Related functions

- [to_real\(\)](#)
- [to_time\(\)](#)

Examples

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

```
resval := get_resolution;
```

the value returned to resval is 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 enables 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 enables 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 enables 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. You can set the variable `ForceSigNextIter` in the `modelsim.ini` file 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 enables 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 simulator resolution determines the precision of the converted value.

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\(\)](#)

Examples

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 want the returned value to be in units of femtoseconds (fs), enter the function as follows:

```
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 simulator resolution determines the precision of the converted value. For example, if you convert 5.9 to a time and the simulator resolution is 1 ps, then the time value is 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\(\)](#)

Examples

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

Modeling memory presents some challenges which careful planning can address.

The challenges include the following common problems with simulation:

- Memory allocation errors, which typically mean 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..... **118**

Effects on Performance by Canceling Scheduled Events..... **128**

Examples of Different Memory Models

You should avoid using VHDL signals to model memory. For large memories especially, the run time for a VHDL model using a signal is many times longer than the run time for a VHDL model using variables in the memory process or as part of the architecture. A signal also uses much more memory.

The first example in this section 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

To implement these models, you need functions that convert vectors to integers. To use them, you may 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, and VHDL2002 Architectures

The VHDL code for the examples demonstrating the approaches to modeling memory is provided below.

- [Example 4-1](#) 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. This example shows a second process that initializes the memory—you could add other processes to create a multi-ported memory.

- [Example 4-2](#) is a package (named conversions) that is included by the memory model in [Example 4-1](#).
- [Example 4-3](#) is provided for completeness—it shows protected types using VHDL 2002. Note that using protected types offers no advantage over shared variables.

Example 4-1. 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  

begin  

shared variable ram : ram_type;  

-----  

begin  

memory:  

process (cs)  

    variable address : natural;  

    begin  

        if rising_edge(cs) then  

            address := sulp_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 := sulp_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 := sulp_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 4-2. Conversions Package

```

library ieee;
use ieee.std_logic_1164.all;

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

package body conversions is

    function sulp_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 sulp_to_natural argument exceeds
                natural range"
            severity error;
        for i in x'range loop
            n := n * 2;
            case x(i) is
                when '1' | 'H' => n := n + 1;
                when '0' | 'L' => null;
                when others      => failure := true;
            end case;
        end loop;

        assert not failure
            report "sulp_to_natural cannot convert indefinite
                std_ulogic_vector"
            severity error;

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

    function natural_to_sulp(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_sulp;

```

```
end natural_to_sulv;  
end conversions;
```

Example 4-3. 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;
```

```

        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_sp1  : 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.
-----  

spram1 : 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_sp1);

-----  

-- 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;

```

Effects on Performance by Cancelling Scheduled Events

Simulation performance typically suffers if events are scheduled far into the future but then canceled before they take effect. Cancelling scheduled events before they take effect acts as a memory leak and slows down simulation.

In VHDL, cancellation of scheduled events can occur in several ways. The most common ways are waits with time-out clauses and projected waveforms in signal assignments.

The following shows a wait with a time-out:

```
signal 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 canceled but not deleted, and a new event is scheduled at 10ms + 10ns. The canceled events are not reclaimed until time 10ms is reached and the canceled event is processed. As a result, there are 500000 (10ms/20ns) canceled but undeleted events. Once 10ms is reached, memory no longer increases because the simulator is 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;
```

VHDL Access Object Debugging

VHDL does not have an object-oriented modeling capability, but VHDL variables of access type enable you to use ModelSim to log and display dynamic simulation data. You enable this logging by specifying `vsim -accessobjdebug`.

When logging a VHDL variable of an access type, ModelSim also automatically logs any designated objects that the variable value points to as the simulation progresses. By default, these objects are unnamed, in accordance with the VHDL LRM (IEEE Std-1076). When you enable logging, each object is given a unique generated name that you can manipulate as a design pathname, but the name is not rooted at any particular place in the design hierarchy. Various windows in the GUI (such as the Wave window, Objects window, Locals window, Watch window, and Memory window) can display both the access variable and any logged designated objects.

Tip

 You can use the `examine` and the `describe` commands in the normal manner for variables and objects displayed in a ModelSim window.

In general, the automatically logged designated objects have a limited lifespan, which corresponds to the VHDL allocator "new." This allocator creates a designated object at a particular time, and the `deallocate()` procedure destroys the designated object at a particular time, as the simulation runs. Each designated object receives its unique name when the new allocation occurs; the name is unique over the life of the simulation.

Terminology and Naming Conventions	129
VHDL Access Type	130
Limitations	131
Default Behavior—Logging and Debugging Disabled	132
Logging and Debugging Enabled	132
The <code>examine</code> and <code>describe</code> Commands	134

Terminology and Naming Conventions

Using VHDL access type variables for logging dynamic data entails various names and descriptors.

- access variable — A VHDL variable declared to be of an access type. An access variable can be either shared or not shared.

NOTE: The VHDL LRM defines “access value” to mean the value of such a variable. This value can be either NULL, or it can denote (point to) some unnamed object, which is the “designated object” and is referred to as an “access object.” That is, when an access variable has a value that is not NULL, then it points to an access object.

- access object — The term "access object" means the designated object of an access variable. An access object is created with the VHDL allocator "new," which returns the access value. This value is then assigned to an access variable, either in an assignment statement or an association element in a subprogram call.
- AIID — The access instance identifier. Each access object gets a unique identifier, its access instance identifier, which is named in the manner of the class instance identifier (CIID) for SystemVerilog (which is also known as a handle—refer to [SystemVerilog Class Debugging](#)).
- DOID — dynamic object identifier. The name of a VHDL access object. The terms DOID and AIID are interchangeable. Access object names have two different forms, depending on whether or not the vsim-accessobjdebug command is in effect. Refer to [Default Behavior—Logging and Debugging Disabled](#) and [Logging and Debugging Enabled](#).
- deep logging — If an access variable is logged, then the DOID of any access object that it points to during the simulation is also logged automatically. Any embedded access type subelements of an access type are also logged automatically. Similarly, logging an access object by name (its access instance identifier) logs not only the access object itself, but any embedded access objects (if the outer access object is of a composite type that contains a subelement of an access type).
- prelogging — The logging of an access object by name, even if you have not declared it (that is, it does not yet exist at the time an "add log" command is issued, but you can still log it by name). This produces useful results only if you use a DOID (dynamic object identifier) that matches the name of an access object that will exist at some future simulation time.

VHDL Access Type

Once you declare an access type, you can declare an access variable within a process or subprogram. When creating dynamic data in VHDL, the usual strict rules apply to assignment of newly constructed objects to an access type.

For instance, there is no implicit casting, and no such thing as an access that can point to anything (such as a void * in C).

For example, you can use any VHDL subtype "foo" to declare an access type that is a pointer to objects of type foo. (This can be a fully constrained type, but it is also legal to point to an unconstrained or partially constrained type.) Subtype foo is called the designated subtype, and the base type of the designated subtype is called the designated type. The designated type of an access type cannot be a file type or a protected type. Note that composite types cannot contain elements that are of file types or protected types, so if the designated type of an access type is a composite type, it will not have any file type or protected type sub-elements.

Lifespan of an Access Object

You construct a dynamic access object in VHDL with a "new" operator and destroy it with a "de-allocate" procedure. Dynamic access objects are referenced only through pointers declared by the HDL author. An access object can be assigned a value of NULL, the value of another compatible access type object, or the result of the new operator that constructs a compatible object. The only way to track an access object is during this lifespan; before and after the lifespan, only the access variable is available.

Restrictions and Requirements

- Beginning with VHDL 2002, shared variables technically must be of a protected type and cannot be of an access type, but ModelSim does not enforce this restriction, allowing access variables to be shared variables. This presents a different set of implementation considerations, because shared variables are context tree items, while non-shared variables (local PROCESS statement variables, local subprogram variables, and class VARIABLE subprogram formals, in general) are debug section objects and not context tree items.
- You cannot point to an elaborated object of the same type as a dynamic object—access types point only to objects constructed by new. (There is no address_of operator.)
- According to the formal definition, dynamic objects have no simple name. That means logging and debugging requires the generation of an internal, authoritative name for the table of contents of any logging database.
- You can declare only VHDL variables (ordinary or shared) as access types, not signals or constants. This access variable has a value of either the literal NULL (which means there is no designated object), or an AIID, which is a pointer to the designated object (called the access object). An access variable is of an access type, and an access object is of the designated type of that access type (not of an access type itself in general). Note that an access variable, when it is not NULL, always points to an access object. Conversely, an access object, when pointed to, is pointed to by an access variable. However, an access object does not have to be pointed to by an access variable, except when it is originally created with "new". That is, while it is not a good idea to "orphan" an access object, it is possible. A simulator can deallocate such an orphaned access object by using some garbage collection method, but is not required to do so. ModelSim does not deallocate orphaned access objects.

Limitations

Access object debugging has some limitations.

It is not possible to log a variable (access variable or not) that is declared in the declarative region of a FUNCTION or PROCEDURE. This is not really a limitation of access object debug, but it is a general limitation. Only shared variables and variables declared in a PROCESS declarative region can be logged (whether access variables or not).

The List window can display the value of an access variable, but cannot display the corresponding access objects.

Access objects, which are of type STD.STANDARD.STRING, are not logged if variables of type STD.TEXTIO.LINE are logged. Thus, “deep logging” of variables of type LINE does not occur.

Default Behavior—Logging and Debugging Disabled

By default, logging access objects by name is turned off. Access variables can be logged and displayed in the various display windows, but the access object(s) they point to are not logged. You can display the value of an access variable (the "name" of the access object it points to) but commands cannot use the value to reference the access variable.

Tip

 Default behavior is applied by either of the following methods:

- In modelsim.ini ([vsim] section), set AccessObjDebug = 0
 - Run vsim -noaccessobjdebug (overrides AccessObjDebug variable).
-

You can use and update the value of the access object by using the VHDL keyword “all” as a suffix to the access variable name.

Examples

- Declare an access variable “v1” that designates some access object. The value of v1 displays as [10001]. This name is for display only—it cannot be used as input to any command that expects an object name. However, it is a unique identifier for any access object that the design may produce. Note that this value replaces any hexadecimal address-based value displayed in previous versions of ModelSim.
- Use variable v1 with the VHDL keyword “all” as an argument to the examine command, which returns the current value of the access object. This essentially dereferences the object.

```
examine v1.all
```

Logging and Debugging Enabled

Logging an access variable logs both the variable value and any access object that the variable points to during the simulation.

Tip

 You can use either of the following methods to apply access object logging and debugging behavior:

- In modelsim.ini, set AccessObjDebug = 1.
 - Run vsim -accessobjdebug (overrides AccessObjDebug variable).
-

When logging is enabled for a VHDL access variable, display-only names (such as [10001]) take on a different form that includes:

- the initial character, @
- the name of the access type or subtype
- another @
- a unique integer N that represents the sequence number (starting with 1) of the objects of that designated type that were created with the VHDL allocator called new.

Displaying Objects in ModelSim Windows

When an access variable displays in the Wave window, the wave trace is not expandable (there is no "+" next to the variable name). When the access variable points to an access object, such that a DOID (such as @ptr@1) appears in the values column of the Wave window, you can then right-click to add the access object under the cursor pointer. This allows adding composite type access objects to the Wave window.

Tip

 An alternative method would be to use the add wave command with the DOID of the access object. For example:

```
add wave @ptr@1
```

Example

Logged access variables take the following form:

```
@ptr@1
```

Related Topics

[Waveform Analysis](#)

[Wave Window](#)

The examine and describe Commands

You can use the examine and describe commands to obtain a description of a variable's characteristics.

Use the examine command with a declared access variable to obtain a display of the current value of its access object. The returned value depends on whether access logging is enabled, or disabled.

- Disabled — The returned value of the access object is its display-only DOID (as per [Default Behavior—Logging and Debugging Disabled](#)).
- Enabled — The returned value of the access object is the logged name that you assigned (as per [Logging and Debugging Enabled](#)).

Tip

 You can also use the [describe](#) command with an access variable (for example, `describe v1.all`). The describe command returns a more qualitative description of the variable's characteristics.

You can use the examine command to obtain a variety of access object values, depending on the data type of the access object. In particular, this command returns object values for the following VHDL data types:

- [Integer](#)
- [String](#)
- [Record](#)

The examples in the following tables show how to use access variables of these three types to specify arguments to the examine command, with access object logging disabled and enabled. Each example uses an access variable named `v1`, declared as one of the data types, and an access object named `@ptr@1`.

Integer

[Table 4-1](#) shows examples of how to use `v1` and `@ptr@1` as arguments to the examine command to obtain the current value of the access object, `@ptr@1`, which is an integer. In the examples, the current integer value is 5. Note that an error results when attempting to use `@ptr@1` as an examine argument with access object logging disabled.

Table 4-1. Using the examine Command to Obtain VHDL Integer Data

Command	Value Returned with Logging Disabled (<code>vsim -noaccessobjdebug</code>)	Value Returned with Logging Enabled (<code>vsim -accessobjdebug</code>)
<code>examine v1</code>	[10001]	<code>@ptr@1</code>

Table 4-1. Using the examine Command to Obtain VHDL Integer Data (cont.)

Command	Value Returned with Logging Disabled (vsim -noaccessobjdebug)	Value Returned with Logging Enabled (vsim -accessobjdebug)
examine v1.all	5	5
examine @ptr@1	<i>error</i>	5

String

Table 4-2 shows examples of how to use v1 and @ptr@1 as arguments to the examine command to obtain the current value of the access object, @ptr@1, which is a string. In the examples, the value of the entire string is abcdef. Note that specifying an index of 4 in the string obtains the fourth character of the string, d. Also, note that an error results when attempting to use @ptr@1 as an examine argument with access object logging disabled

Table 4-2. Using the examine Command to Obtain VHDL String Data

Command	Value Returned with Logging Disabled (vsim -noaccessobjdebug)	Value Returned with Logging Enabled (vsim -accessobjdebug)
examine v1	[10001]	@ptr@1
examine v1.all	"abcdef"	"abcdef"
examine v1(4)	'd'	'd'
examine v1.all(4)	'd'	'd'
examine @ptr@1	<i>error</i>	"abcdef"
examine @ptr@1(4)	<i>error</i>	'd'

Record

A VHDL record is composite data type, consisting of multiple fields (also referred to as elements) each of which contains its own separate data. Record fields may be of the same or of different types.

Table 4-3 shows examples of using the examine command on a record object with an integer field (f1) and a string field (f2). In the examples, the current value of integer field f1 is 5, and the current value of string field f2 is abcdef. Note that an error results when attempting to use @ptr@1 as an examine argument with access object logging disabled.

Table 4-3. Using the examine Command to Obtain VHDL Record Data

Command	Value Returned with Logging Disabled (vsim -noaccessobjdebug)	Value Returned with Logging Enabled (vsim -accessobjdebug)
examine v1	[10001]	@ptr@1

Table 4-3. Using the examine Command to Obtain VHDL Record Data (cont.)

Command	Value Returned with Logging Disabled (vsim -noaccessobjdebug)	Value Returned with Logging Enabled (vsim -accessobjdebug)
examine v1.all	{5, "abcdef"}	{5, "abcdef"}
examine v1.f1	5	5
examine v1.all.f1	5	5
examine @ptr@1.f1	<i>error</i>	5

Related Topics[describe](#)[examine](#)

Chapter 5

Verilog and SystemVerilog Simulation

Introduction to the process of compiling and simulating Verilog and SystemVerilog designs with ModelSim.

Mixed-Language Support - Verilog Top	137
Standards, Nomenclature, and Conventions.....	138
Basic Verilog Usage	141
Verilog Simulation.....	159
Cell Libraries	186
SystemVerilog System Tasks and Functions.....	191
Compiler Directives.....	205
Unmatched Virtual Interface Declarations.....	207
Verilog PLI and SystemVerilog DPI	209
SystemVerilog Class Debugging.....	210
Autofindloop and the Autofindloop Report	243

Mixed-Language Support - Verilog Top

This version of ModelSim supports simulation of mixed-language designs, which allows you to simulate designs that are written in VHDL, Verilog, and SystemVerilog. 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.

The basic flow for simulating mixed-language designs is:

1. Compile your HDL source with vcom (VHDL) or vlog (Verilog and SystemVerilog) following order-of-compile rules. Note that VHDL and Verilog observe different rules for case-sensitivity.
2. Simulate your design with the vsim command.
3. Run the simulation and perform any debug processes.

Standards, Nomenclature, and Conventions

SystemVerilog is an extension of IEEE Std 1364 for the Verilog HDL and improves the productivity, readability, and re-usability 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.

ModelSim implements the Verilog and SystemVerilog languages as defined by the following standards:

- IEEE 1364-2005 and 1364-1995 (Verilog)
- IEEE 1800-2012, 1800-2009 and 1800-2005 (SystemVerilog)

Note

 ModelSim supports partial implementation of SystemVerilog IEEE Std 1800-2012. 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

The SystemVerilog standard specifies extensions for a higher level of abstraction for modeling and verification with the Verilog hardware description language (HDL).

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 1800-2012.

Note

 The term “Language Reference Manual” (or LRM) refers to the current IEEE standard for Verilog or SystemVerilog.

Supported Variations in Source Code	139
for Loops	139
Naming Macros with Integers	139

Supported Variations in Source Code

ModelSim enables you 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 you to use Verilog syntax that omits any or all three specifications of a for loop — initialization, termination, increment. These allowed omissions are also allowed in C.

Note

 If you use this variation, a suppressible warning (2252) is displayed, which you can change to an error with the vlog -pedanticerrors command.

The following examples show the missing for loop specifications this variation allows:

- Missing initializer (in order to continue where you left off):

```
for ( ; incr < foo; incr++) begin ... end
```

- Missing incrementer (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
```

Naming Macros with Integers

The vlog command compiles macros named with integers in addition to identifiers.

For example:

```
`define 11 22
`define q(s) `s
module defineIdent;
    string s2 = `q(`11);
    int i = `11;
    initial begin
        $display("i: %d\n", i);
        #10;
        $display("s2: %s\n", s2);
    end
endmodule
```

The following compiler directives also accept integer names as well as IEEE-1800 Language Reference Manual macro names:

```
'define
'else
'elsif
'endif
'ifdef
'undef
```

You can disable this functionality by specifying [vlog -pedanticerrors](#).

Basic Verilog Usage

Basic Verilog usage include the steps of compiling, optimizing, loading, and simulating.

Generally you perform the steps in the following order:

1. Compile your Verilog code into one or more libraries with the [vlog](#) command. See [Verilog Compilation](#) for details.
2. Load your design with the [vsim](#) command. Refer to [Verilog Simulation](#).
3. Simulate the loaded design and debug as needed.

Verilog Compilation	142
Initializing enum Variables	145
Incremental Compilation	145
Library Usage.	148
SystemVerilog Multi-File Compilation	150
Verilog-XL Compatible Compiler Arguments	152
Verilog Configurations	156
Verilog Generate Statements	158

Verilog Compilation

Compiling your Verilog design for the first time is a two-step process.

1. Create a working library with the [vlib](#) command, or select **File > New > Library**.
2. Compile the design with the [vlog command](#), or select **Compile > Compile**.

Creating a Working Library	142
Invoking the Verilog Compiler	142
Verilog Case Sensitivity	143
Parsing SystemVerilog Keywords	143
Recognizing SystemVerilog Files by File Name Extension	144

Creating a Working Library

Before you can compile your design, you must create a library in which to store the compilation results, which is compatible across all platforms.

Procedure

Use the [vlib](#) command or select **File > New > Library** to create a new library.

For example, the command **vlib work** 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.

Invoking the Verilog Compiler

The Verilog compiler compiles Verilog source code into retargetable, executable code. You can then simulate your design on any supported platform without having to recompile your design.

Prerequisites

Create a working library.

Procedure

Enter the [vlog](#) command or choose **Compile > Compile** from the main menu to invoke the Verilog compiler.

As the design compiles, the compiler also generates the resulting object code for modules and user-defined primitives (UDPs) 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.

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 (you can use any combination of suffixes). Compilation only works on referenced definitions. Compilation does accept compressed SystemVerilog source files (.gz extension, compressed with zlib).

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.

Verilog compilation parses SystemVerilog keywords when either of the following situations exists:

- Any file in 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 1800-2012.

The following reserved keywords have been added since IEEE Std 1800-2009: implements, interconnect, nettype, and soft.

Older SystemVerilog code can use words as identifiers that are now 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 causes 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 the older keywords are recognized.

Recognizing SystemVerilog Files by File Name Extension

If you use the `-sv` argument with the `vlog` command, 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, ModelSim assumes that only files with the extension `.sv`, `.svh`, or `.svp` are SystemVerilog.

File extensions of include files

If you do not use the `-sv` argument with `vlog` when specifying a file that uses an ``include` statement to specify an include file, the compilation will ignore the file extension of the include file and assume the language to be the same as the file containing the ``include`. For example, if you do not use the `-sv` argument:

- If `a.v` includes `b.sv`, then `b.sv` is read as a Verilog file.
- If `c.sv` includes `d.v`, then `d.v` is 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, where the default setting is:

```
; 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 SystemVerilog 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:

- Any Verilog source read in is added to the current compilation unit.
- A compilation unit ends at the close of a SystemVerilog file.

Note

 Keep all Verilog files in the same compilation unit to maintain backward compatibility with legacy Verilog designs.

Initializing enum Variables

By default, to initialize enum variables ModelSim uses the default value of the base type instead of the leftmost value.

You can use the following methods to change the default behavior so that ModelSim sets the initial value of an enum variable to the left most value:

- Run **vlog -enumfirstinit** when compiling and run **vsim -enumfirstinit** 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) because compilation resolves all module and UDP instantiations and external hierarchical references when the simulator loads the design.

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 5-1. 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:

```
vlog testbench.sv
# Top level modules:
# testbench

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 design units 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. When you use the -incr argument, the compiler automatically determines which modules have changed, and generates code only for those design units.

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
```

If 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 design unit. 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 design units.

Note

 Changes to dependency files (such as include files or packages specified by the design unit/module code) will be analyzed and will cause dependent design unit/module files, as well as the dependency files, to be recompiled when changes are made to them.

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 design units 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

You must compile all modules and UDPs in a Verilog design 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.

Because compilation does not determine instantiation bindings at compile time, you must instruct the simulator to search your libraries when loading the design. Simulation loads the top-level modules from the library named work unless you prefix the modules with the <library>.

option. If simulation does not find them in the work library, it searches in the libraries specified with -Lf arguments followed by libraries specified with -L arguments.

Please refer to [Library Search Rules](#) for more information on how to search your libraries.

Related Topics

[Library Search Rules](#)

[Handling Sub-Modules with the Same Name](#)

SystemVerilog Multi-File Compilation

ModelSim allows you to compile multiple SystemVerilog files at a time.

Declarations in Compilation Unit Scope 150

Macro Definitions and Compiler Directives in Compilation Unit Scope 150

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. It is important to understand how the simulator defines compilation units 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 SystemVerilog source file on the command line. 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.

When you use 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 the [vlog -sfcu](#) command.

Macro Definitions and Compiler Directives in Compilation Unit Scope

According to the IEEE Std 1800-2005, the visibility of macro definitions and compiler directives spans the lifetime of a single compilation unit.

By default, this means the definitions of macros and the 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 how vlog handles compilation units.

Note

 Compiler directives revert to their default values at the end of a compilation unit.

If you specify a compiler directive as an option to the compiler, it uses this setting for all compilation units present in the current compilation.

When you are in MFCU mode, despite a unit scope starting in a Verilog file, the simulator will pull in built-in macros and directives (including, `undefineall, SV_COV_*, mtiTypename*) upon transition from a Verilog file to a SystemVerilog file.

Verilog-XL Compatible Compiler Arguments

The Verilog compiler supports arguments that are equivalent to Verilog-XL arguments, simplifying 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	152
Verilog-XL uselib Compiler Directive	153

Arguments Supporting Source Libraries

The Verilog compiler provides arguments that support source libraries in the same manner as Verilog-XL.

Note that source libraries are very different from the libraries that the ModelSim compiler uses to store compilation results.

The compiler searches source libraries after compiling the source files on the command line. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules vlog compiles from source libraries may in turn have additional unresolved references that trigger another search of the source libraries. The compiler repeats this process until it resolves all references or until there are no longer any new unresolved references. The compiler searches source libraries in the order they appear on the command line.

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

Related Topics

[vlog \[ModelSim Command Reference Manual\]](#)

Verilog-XL uselib Compiler Directive

The `uselib compiler directive is a method of source library management that you can use as an alternative to the -v, -y, and +libext compiler arguments. It has the advantage that a design may reference different modules having the same name.

To compile designs that contain `uselib directive statements, use the -compile_uselibs argument (described below) with the [vlog](#) command.

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
```

Because the `uselib directives are embedded in the Verilog source code, you have 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.

Because the `uselib directive allows a design to reference multiple modules that have the same name, the source libraries referenced by the `uselib directive must be compiled independently.

You should compile each source library into its own object library. The compilation of the code containing the `uselib directives records only which object libraries to search for each module instantiation when the simulator loads the design.

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 ignores 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 that you can define 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 you use `-compile_uselibs`, ModelSim determines the directory into which to compile the object libraries into 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 you can instantiate two different modules that have the same name 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 instantiation 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 can 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. Because *srtr.v* is compiled after *dut.v*, the `uselib directive is still in effect. When *srtr* loads, it uses 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 support for configurations. Configurations specify how a design is “assembled” during the elaboration phase of simulation. A configuration consists of two pieces: the library mapping and the configuration itself. 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 the -includir argument:

```
library lib1 src_dir/*.v -includir ../include_dir2, ..., my_includir;
```

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.

-
- Tip** You can use vlog -mfcu to compile macros for all files in a given testbench. Any macros already defined before the -libmap argument appears are still defined for use by the -libmap files. That is, -mfcu macros are applied to the other libraries in library mapping files.
-

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 156

Configurations and the Library Named work

ModelSim treats the library named “work” in a special way 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
```

Related Topics

[Working Library Versus Resource Libraries](#)

Verilog Generate Statements

ModelSim implements the rules for generate statements adopted for Verilog 2005. Most of the 2005 rules are backwards compatible, but there is one key difference related to name visibility.

Name Visibility in Generate Statements 158

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 is excluded in the second example.

Verilog Simulation

A Verilog design is ready for simulation after you compile it with vlog . You can then invoke the simulator with the names of the top-level modules (many designs contain only one top-level module).

For example, if your top-level modules are “testbench” and “globals”, then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default, all modules and UDPs are loaded from the library named work. You can specify Modules and UDPs from other libraries with the vsim -L or -Lf arguments (refer to [Library Search Rules](#) 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)	159
Modules Without Timescale Directives	160
Multiple Timescale Directives.....	161
Choosing the Resolution for Verilog	162
Event Ordering in Verilog Designs	163
Debugging Event Order Issues	167
Signal Segmentation Violations	168
Negative Timing Checks	171
Force and Release Statements in Verilog	183
Verilog-XL Compatible Simulator Arguments.....	183
Using Escaped Identifiers	185

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.

You can specify time units and precision 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. The simulator issues an elaboration error in this situation. You should make sure that all modules having delays also have timescale directives so that the timing of the design operates as you intend.

You can use the following vsim arguments to suppress elaboration errors or reduce them to warnings, but you risk improper design behavior and reduced performance.

- Use the **vsim +nowarnTSCALE** or **-suppress** arguments to ignore the error.
- Use the **-warning** argument to reduce the severity to a warning.

-timescale Option

Use the **-timescale** option with vlog 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 quotation marks in the example above are required because the argument contains white space.

Design units that do not have a timescale set in the HDL source, or with vlog -timescale, will generate an error similar to the following:

```
# ** Error (suppressible): (vsim-3009) [TSCALE] - Module 'top2' does not
have a timeunit/timeprecision specification in effect, but other modules
do.
#   Time: 0 ps  Iteration: 0  Instance: /top2 File: t2.sv
# Loading work.dut2(fast)
```

You can suppress the error, causing vsim to use the simulator time resolution.

Multiple Timescale Directives

A design can have multiple timescale directives. Separately compiled modules can also have different timescales. The simulator compares the timescale of all of the modules in a design, and uses the smallest timescale as the simulator resolution.

The timescale directive takes effect where it appears in a source file and applies to all Verilog source files (.v files) that follow in the same vlog command.

Note

 For SystemVerilog source files (.sv files), this requires that you use either the -mfcu argument or the -mfcu=macro argument with the vlog command.

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 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 simulation handles the delays in the module 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. The simulator determines the module's precision by the `timescale directive. ModelSim does not override the module's precision.
- -t is set to 1 ns — The delay is rounded to 13. The simulator determines the module's precision 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, setting the time precision unnecessarily small can degrade performance.

Event Ordering in Verilog Designs

Event-based simulators such as ModelSim can process multiple events at a given simulation time. The Verilog language definition does not provide you control of the processing order of simultaneous events. Some designs rely on a particular event order, and these designs may behave differently than you expect.

Event Queues	163
Controlling Event Queues with Blocking or Non-Blocking Assignments	165

Event Queues

Section 11 of IEEE Std 1364-2005 defines several event queues that determine how 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.

You cannot control event order in the active event queue. Active events are processed in any order, and new active events are added in any order. The example below illustrates potential ramifications of this situation.

Assume that these four statements are in the event queue:

- 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

Table 5-1 and **Table 5-2** 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 $\langle name \rangle (old \rightarrow new)$ where $\langle name \rangle$ indicates the reg being updated and new is the updated value.\

Table 5-1. 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
clk(0 → 1)	4, 2
4	2
2	p2(1 → 0)
p2(1 → 0)	3
3	clk(1 → 0)
clk(1 → 0)	<empty>

Table 5-2. 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 that you can assign an event to a particular queue. You do this with 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 nonzero 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.)

You should use non-blocking assignments only for outputs of flip-flops. This ensures that outputs of flip-flops do not change until after all flip-flops have been evaluated. If you attempt to use non-blocking assignments in combinational logic paths to remove race conditions, you may only cause more problems.

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
```

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

Debugging Event Order Issues

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

ModelSim helps you track down event order dependencies with the following [vlog](#) compiler arguments: -compat, -hazards, and -keep_delta.

Hazard Detection	167
Hazard Detection and Optimization Levels	167

Hazard Detection

The -hazards argument for the vsim command detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes.

To enable hazard detection you must invoke [vlog](#) with the -hazards argument when you compile your source code, and you must invoke vsim with the -hazards argument when you simulate.

ModelSim 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. The simulator calls this a READ/WRITE hazard if the simulator executed the read first.
- WRITE/READ — This hazard is similar to a READ/WRITE hazard except that the simulator executes the write first.

The simulator 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.

Note

 Using the -hazards argument implicitly enables the -compat argument. As a result, using -hazards may affect your simulation results.

Hazard Detection and Optimization Levels

The optimization level you use in a simulation can have an affect on hazard detection results.

Some optimizations change the read/write operations performed on a variable if the transformation is determined to produce similar 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, optimizations reduce the number of false hazards by eliminating unnecessary reads and writes, but optimizations can produce additional false hazards. The following are some 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.
- Hazards caused by simultaneous forces are not detected.

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 code attempts to initialize a property of *obj*, but *obj* has not been constructed. The code is missing the following information:

```
C obj = new;
```

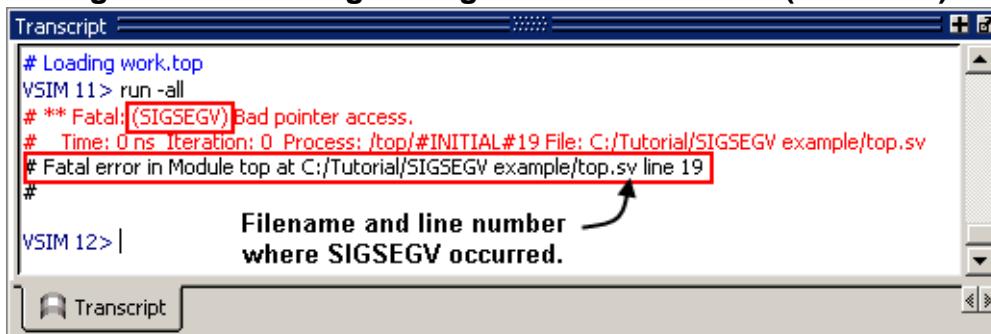
The **new** operator performs three distinct operations:

- 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 is nothing to reference. The simulator sets the variable to the value **null** and the SIGSEGV fatal error occurs.

To debug a SIGSEGV error, first look in the transcript. [Figure 5-1](#) shows an example of a SIGSEGV error message in the Transcript window.

Figure 5-1. Fatal Signal Segmentation Violation (SIGSEGV)



The screenshot shows the ModelSim Transcript window. The log output includes:

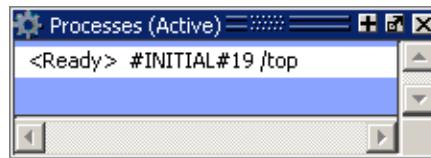
```
# Loading work.top
VSIM 11> run -all
# ** Fatal:[SIGSEGV] Bad pointer access.
# Time: 0 ns Iteration: 0 Process: /top/#INITIAL#19 File: C:/Tutorial/SIGSEGV example/top.sv
# Fatal error in Module top at C:/Tutorial/SIGSEGV example/top.sv line 19!
#
VSIM 12>
```

An annotation with a blue arrow points from the text "Filename and line number where SIGSEGV occurred." to the line "# Fatal error in Module top at C:/Tutorial/SIGSEGV example/top.sv line 19!".

The Fatal error message identifies the filename and line number of the code violation (in this example, the file is *top.sv* and the line number is 19).

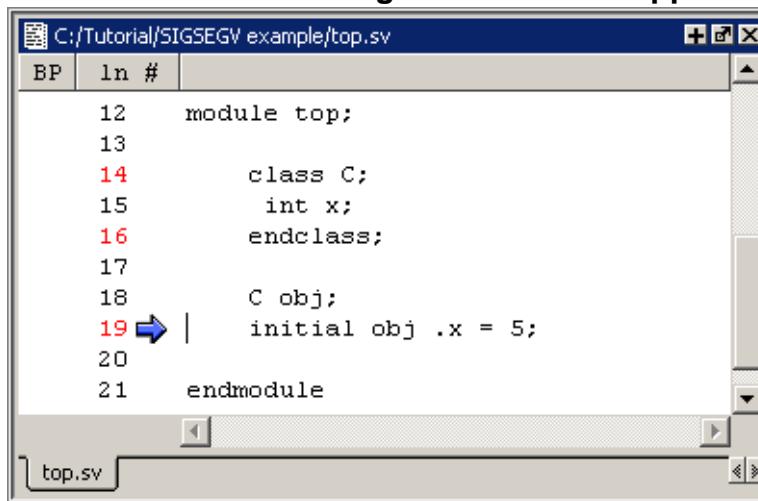
ModelSim sets the active scope to the location of the error. In the Processes window, the current process is highlighted ([Figure 5-2](#)).

Figure 5-2. Current Process Where Error Occurred



Double-click the highlighted process to open a Source window. A blue arrow points to the statement where the simulation stopped executing ([Figure 5-3](#)).

Figure 5-3. Blue Arrow Indicating Where Code Stopped Executing



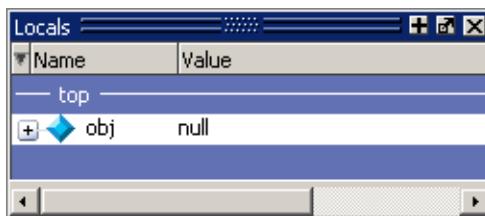
The screenshot shows the ModelSim Source window for the file "C:/Tutorial/SIGSEGV example/top.sv". The code is:

```
12 module top;
13
14 class C;
15     int x;
16 endclass;
17
18 C obj;
19 initial obj .x = 5;
20
21 endmodule
```

A blue arrow points to the line "initial obj .x = 5;" which corresponds to line 19 in the code.

Next, look for *null* values in the ModelSim Locals window ([Figure 5-4](#)), which displays data objects declared in the local (current) scope of the active process.

Figure 5-4. Null Values in the Locals Window



The screenshot shows the ModelSim Locals window. The window title is "Locals". It contains a table with two columns: "Name" and "Value". There is one entry: "top" under "Name" and "obj" under "Value". The "Value" column for "obj" contains the word "null". The window has standard scroll bars on the right side.

Name	Value
top	
obj	null

The *null* value in [Figure 5-4](#) 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).

Negative timing syntax is defined in the IEEE Standard for Verilog Hardware Description Language, specifically the chapter “Timing Checks.”

The negative timing check algorithm is enabled by default. To explicitly enable the algorithm, use the `+delayed_timing_checks` with the `vsim` command. If you want to disable the functionality, add the `+no_autodtc` to the `vsim` command line.

vsim Arguments Related to Timing Checks	171
Commands Supporting Negative Timing Check Limits	172
Negative Timing Constraint Algorithm	177
Using Delayed Inputs for Timing Checks	181
Re-evaluation of Zero-Delay Output Schedules	182

vsim Arguments Related to Timing Checks

The `vsim` command supports several timing check-related arguments.

- `vsim +delayed_timing_checks` — (on by default) Instructs the simulator to automatically detect cells with negative timing checks.
- `vsim +no_autodtc` — Disables the default behavior of the `+delayed_timing_checks` argument.
- `vsim +no_neg_tchk` — Forces all negative timing check limits to a zero value.
- `vsim +ntc_warn` — Enables messaging for negative timing checks.
- `vsim +notimingchecks` — Removes all timing check entries from the design as it is parsed.
- `vsim -glsnegtchk <solver_level>` — Specifies the negative timing check solver.

Commands Supporting Negative Timing Check Limits

By default, the simulator supports negative timing check limits in Verilog \$setuphold and \$recrrem system tasks.

Using the +no_neg_tchk 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 to be evaluated correctly. These timing checks specify delayed versions of the input ports. The delayed versions are used for functional evaluation. The following sections describe the correct syntax for \$setuphold and \$recrrem.

\$setuphold	173
\$recrrem	175
Timing Check Syntactical Conventions	176

\$setuphold

The \$setuphold check determines whether signals obey the timing constraints.

Usage

```
$setuphold ( reference_event , data_event , timing_check_limit , timing_check_limit , [ notifier  
] , [ stamptime_condition ] , [ checktime_condition ] , [ delayed_reference ] , [ delayed_data  
] );
```

Arguments

- **reference_event**
(required) Specifies a transition in a reference signal that establishes the reference time for tracking timing violations on the data_event. Because \$setuphold combines the functionality of the \$setup and \$hold system tasks, the reference_event sets the upper bound event for \$setup and the lower bound event for \$hold.
- **data_event**
(required) Specifies 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.
- **timing_check_limit** (both instances are required)
Specifies a constant expression or specparam that specifies the minimum interval between:
 - First instance — The data_event and the clk_event. Any change to the data signal within this interval results in a timing violation.
 - Second instance — The interval between the clk_event and the data_event. Any change to the data signal within this interval results in a timing violation.
- **notifier**
(optional) Specifies a register whose value is updated whenever a timing violation occurs. The notifier can be used to define responses to timing violations.
- **stamptime_condition**
(optional) Conditions the data_event for the setup check and the reference_event for the hold check. This alternate method of conditioning precludes specifying conditions in the reference_event and data_event arguments.
- **checktime_condition**
(optional) Conditions the data_event for the hold check and the reference_event for the setup check. This alternate method of conditioning precludes specifying conditions in the reference_event and data_event arguments.
- **delayed_reference**
(optional) Specifies a net that is continuously assigned the value of the net specified in the reference_event. The delay is determined by the simulator and may be nonzero depending on all the timing check limits.

- `delayed_data`

(optional) Specifies 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 nonzero depending on all the timing check limits.

\$recrem

The \$recrem timing check determines whether signals obey the timing constraints.

Usage

```
$recrem ( reference_event , data_event , timing_check_limit , timing_check_limit , [ notifier ] ,  
[ stamptime_condition ] , [ checktime_condition ] , [ delayed_reference ] , [ delayed_data ] )  
;
```

Arguments

- **reference_event**
(required) Specifies an asynchronous control signal with an edge identifier to indicate the release from an active state.
- **data_event**
(required) Specifies a clock or gate signal with an edge identifier to indicate the active edge of the clock or the closing edge of the gate.
- **timing_check_limit** (both instances are required)
Specifies a minimum interval between:
 - First instance — 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.
 - Second instance — 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.
- **notifier**
(optional) Specifies a register whose value is updated whenever a timing violation occurs. The notifier can be used to define responses to timing violations.
- **stamptime_condition**
(optional) Conditions the data_event for the removal check and the reference_event for the recovery check. This alternate method of conditioning precludes specifying conditions in the reference_event and data_event arguments.
- **checktime_condition**
(optional) Conditions the data_event for the recovery check and the reference_event for the removal check. This alternate method of conditioning precludes specifying conditions in the reference_event and data_event arguments.
- **delayed_reference**
(optional) Specifies a net that is continuously assigned the value of the net specified in the reference_event. The delay is determined by the simulator and may be nonzero depending on all the timing check limits.

- delayed_data

(optional) Specifies 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 nonzero depending on all the timing check limits.

Timing Check Syntactical Conventions

Your \$setuphold() or \$recrem() timing checks must follow the LRM defined syntax exactly. The simulator behaves in the following ways based on your commands.

The two timing_check_limit values are your delayed reference and delayed data values, respectively, which can be negative values. In all cases, you must ensure that the sum of these two values is greater than zero (0). If sum does not meet this requirement, the simulator silently sets any negative values to zero (0) during elaboration or SDF annotation. You can force the simulator to show a warning (vsim-3616) with the +ntc_warn argument to the vsim command.

```
** Warning: (vsim-3616) cells.v(x): Instance 'dff0' - Bad $setuphold
constraints: 5 ns and -6 ns. Negative limit(s) set to zero.
```

The internal timing check algorithm determines the proper delay values; a negative hold requires the shifting of your DATA signal, and a negative setup requires the shifting of your CLOCK. In rare cases, typically due to bad SDF values, the timing check algorithm cannot create convergence. Use the +ntc_warn argument to the vsim command to enable additional warning messages.

The LRM does not permit specifying a reference_event or data_event condition using the &&& operator and also specifying a stamptime_condition or checktime_condition. When this does occur, the simulator issues a warning and ignores the condition defined in either event. For example, in the task:

```
$setuphold(posedge clk &&& cond1, posedge d, 10, -5, notifier, cond2, ,
dclk, dd);
```

the condition “cond1” is ignored.

The delayed_reference 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_reference and delayed_data nets in place of the normal reference 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_reference 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.

Negative Timing Constraint Algorithm

The negative timing constraint algorithm of the simulator looks for 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 changes the timing check limits to force convergence. Basically, the algorithm sets the smallest negative \$setup/\$recovery limit. If a negative \$setup/\$recovery does not exist, then the algorithm zeros the smallest negative \$hold/\$removal limit to zero. After zeroing a negative limit, the delay calculation procedure is repeated. If the delays do not converge, the algorithm sets another negative limit to zero, repeating the process until convergence is found.

Example 5-2. Timing Check Example 1

Given this timing check,

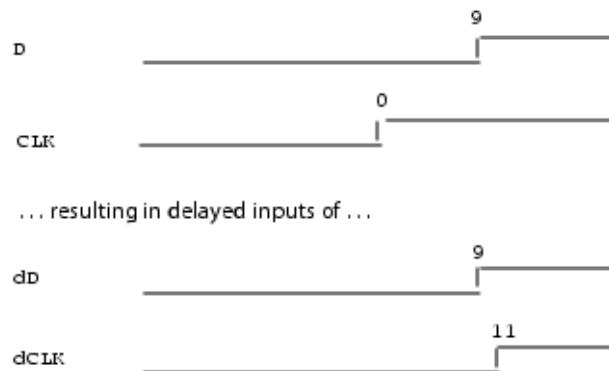
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
```

dCLK is the delayed version of the input *CLK* and *dD* is the delayed version of *D*. 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:



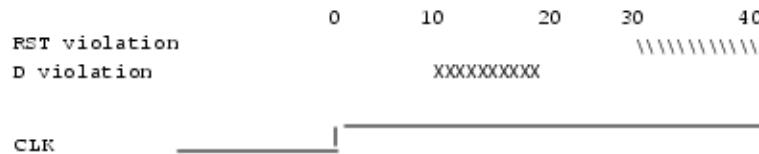
Because the posedge *CLK* transition is delayed by the amount of the negative setup limit (plus one time unit to prevent race conditions) no timing violation is reported and the new value of *D* is latched.

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.

Example 5-3. Timing Check Example 2

Consider the following timing checks specified relative to CLK:

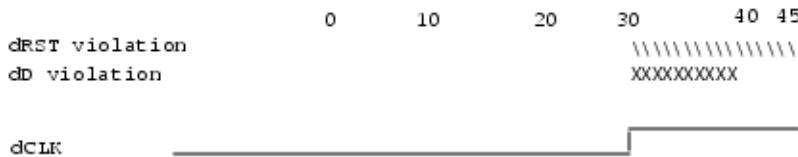
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);  
$setuphold(posedge CLK, negedge RST, -30, 45, notifier,,, dCLK, dRST);
```



To solve the timing checks specified relative to *CLK* the following delay values are necessary:

	Rising	Falling
$dCLK$	31	31
dD	20	20
$dRST$	0	0

The simulator's intermediate delay solution shifts the violation regions to overlap the reference events.



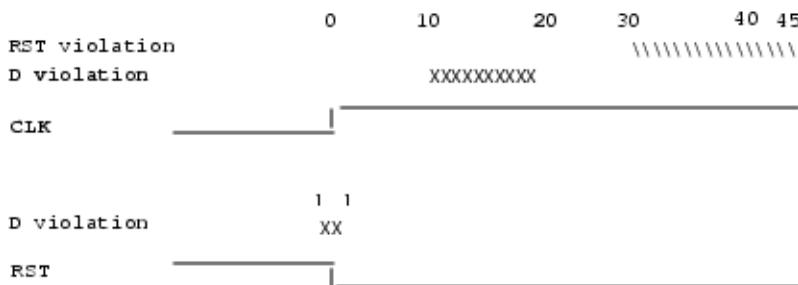
Notice that no timing is specified relative to negedge *CLK*, but the *dCLK* falling delay is set to the *dCLK* rising delay to minimize pulse rejection on *dCLK*. Pulse rejection that occurs due to delayed input delays is reported by:

```
"WARNING[3819] : Scheduled event on delay net dCLK was cancelled"
```

Example 5-4. Timing Check Example 3

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:

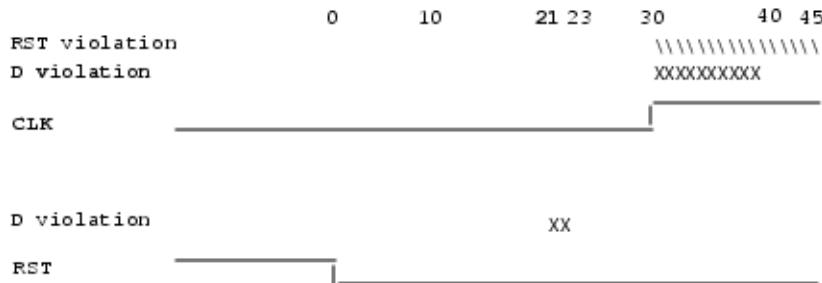
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
$setuphold(posedge CLK, negedge RST, -30, 45, notifier,,, dCLK, dRST);
$setuphold(negedge RST, D, 1, 1, notifier,,, dRST, dD);
```



As illustrated earlier, to solve timing checks on *CLK*, delays of 20 and 31 time units were necessary on *dD* and *dCLK*, respectively.

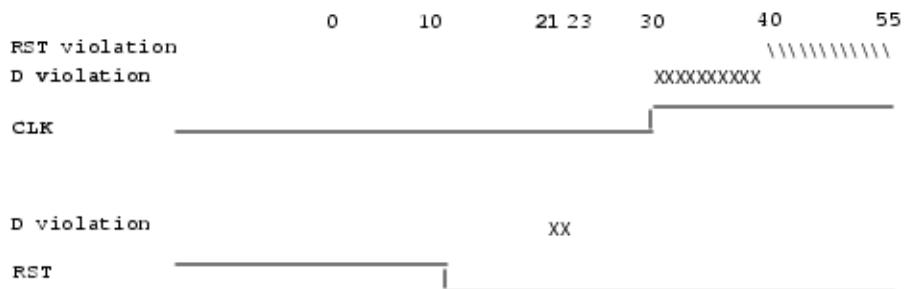
	Rising	Falling
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	0

The simulator's intermediate delay solution is:



Note that 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



If a timing check in the design was set to zero because a delay solution was not found, a summary message is issued:

```
# ** Warning: (vsim-3316) No solution possible for some delayed timing
check nets. 1 negative limits were zeroed. Use +ntc_warn for more info.
```

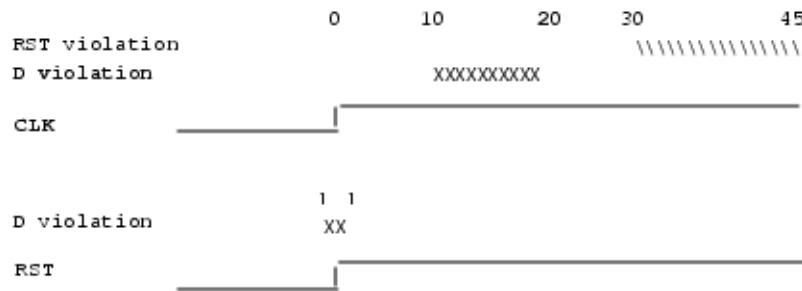
Invoking **vsim** with the **+ntc_warn** option identifies the timing check that is being zeroed.

Example 5-5. Timing Check Example 4

Finally consider the case where the *RST* and *D* timing check is specified on the posedge *RST*.

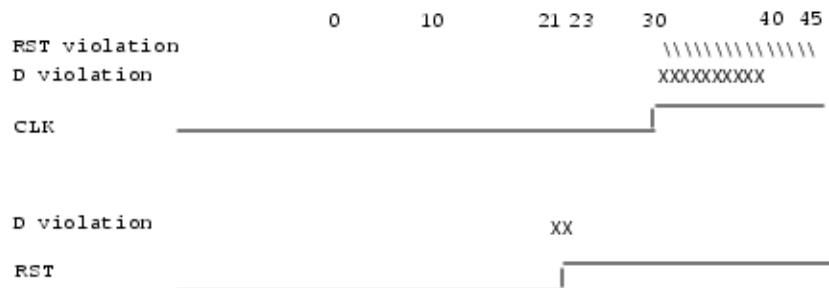
```
$setuphold(posedge CLK, D, -10, 20, notifier,, dCLK, dD);
$setuphold(posedge CLK, negedge RST, -30, 45, notifier,, dCLK, dRST);
```

```
$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
<i>dD</i>	20	20
<i>dRST</i>	20	10



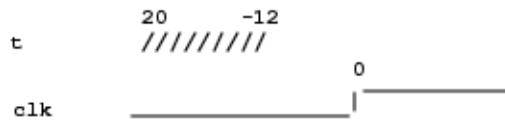
Using Delayed Inputs for Timing Checks

By default, the simulator performs timing checks on inputs specified in the timing check. You can use the `+delayed_timing_checks` argument with the `vsim` command to perform timing checks on the delayed inputs.

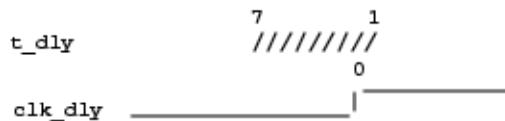
Example timing check:

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

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



When the timing check is performed on the delayed inputs, 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. The ModelSim simulator supports both methods. By default, timing checks are performed on the delayed inputs. You can disable this behavior with the `+no_autodtc` switch.

Re-evaluation of Zero-Delay Output Schedules

Minimize the use of zero-delay output schedules of gate-level cells without requiring the use of the `ifnone` construct with the `vsim -glspath simuconddc` argument.

Add the `-glspath simuconddc` argument/value pair to the `vsim` command line when:

- You are simulating a full timing gate-level simulation with a cell library containing conditional timing specifications.
- During simulation, if your timing was not met due to a zero-delay output schedule when no timing path was active.

This functionality is often useful for combinational cells that have detailed timing, which is not exhaustively specified. It is also most effective with simple state-dependent paths.

When you add the -glspath simucondc argument/value pair, the simulator re-evaluates specify path conditions if the following requirements are met:

- A module's specify block has state-dependent specify paths with no ifnone timing paths.
- For an output change, no specify path is active due to no conditions being evaluated true. This would normally result in the output being scheduled with zero-delay.
- More than one input change occurred simultaneously to cause the output to change.
- Due to simultaneous input changes, conditions for the timing paths were not evaluated true.

Force and Release Statements in Verilog

The Verilog Language Reference Manual IEEE Std 1800-2009, section 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 cannot be a variable.

Related Topics

[force \[ModelSim Command Reference Manual\]](#)

Verilog-XL Compatible Simulator Arguments

The simulator supports several arguments that are equivalent to Verilog-XL arguments make it easier to port a design to the ModelSim simulator.

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

the ModelSim simulator recognizes and maintains Verilog escaped identifier syntax.

All object names inside the simulator appear identical to their names in original HDL source files.

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) should be composed of components with valid escaped identifier syntax.

The `modelsim.ini` variable [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.” Signal Spy function arguments are also parsed in “generous mode.”

Tcl and Escaped Identifiers [185](#)

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]`. In this example, backslashes are also 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 `<install_dir>/docs/tcl_help_html/TclCmd` directory.

Cell Libraries

Many ASIC and FPGA vendors' Verilog cell libraries are compatible with ModelSim Verilog.

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 [186](#)

Delay Modes [187](#)

SDF Timing Annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files.

Related Topics

[Standard Delay Format \(SDF\) Timing Annotation](#)

Delay Modes

Verilog models can contain both distributed delays and path delays. Distributed delays appear on primitives, UDPs, and continuous assignments; path delays are the port-to-port delays specified in specify blocks. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with no distributed delays specified.

The following code shows a simple two-input AND gate cell, where no distributed delay is specified for the AND primitive.

```
module and2(y, a, b);
    input a, b;
    output y;
    and(y, a, b);
    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule
```

For cells such as this, the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, though more complex cells may require nonzero distributed delays to work properly.

Delay Modes and the Verilog Standard	187
Distributed Delay Mode	190
Path Delay Mode	190
Unit Delay Mode	190
Zero Delay Mode	190

Delay Modes and the Verilog Standard

The Verilog standard (LRM, IEEE Std 1364-2005) states that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (Section 14.4). This is the default behavior of the ModelSim simulator.

However, you can specify alternate delay modes using compiler directives and arguments to the [vlog](#) command:

- [Distributed Delay Mode](#)
- [Path Delay Mode](#)
- [Unit Delay Mode](#)
- [Zero Delay Mode](#)

Tip
 Delay mode arguments to the vlog command take precedence over delay mode directives in the source code.

Note that these directives and arguments are compatible with Verilog-XL. However, using these modes results in behavior that is not clearly defined by the Verilog standard—the delays that are set to zero can vary from one simulator to another (some simulators zero out only some delays).

[Example 5-6](#) shows the 2-input AND gate cell using a different compiler directive to apply each delay mode. In particular, ModelSim does the following:

- The `delay_mode_zero directive sets both the continuous assignment delay (assign #2 c = b) and the primitive delay (and #3 (y, a,c)) to zero.
- The `delay_mode_unit directive converts both of these nonzero delays (continuous assignment and primitive) to 1.

Example 5-6. Delay Mode Directives in a Verilog Cell

The following instances of a 2-input AND gate cell (and2_1, and2_2, and2_3, and2_4) use compiler directives to apply each delay mode.

```
`delay_mode_zero
module and2_1(y, a, b);
    input a, b;
    output y;
    wire c;
    assign #2 c = b;

    and #3(y, a, c);
    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule

`delay_mode_unit
module and2_2(y, a, b);
    input a, b;
    output y;
    wire c;
    assign #2 c = b;

    and #3(y, a, c);
    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule

`delay_mode_distributed
module and2_3(y, a, b);
    input a, b;
    output y;
    wire c;
    assign #2 c = b;

    and #3(y, a, c);
    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule

`delay_mode_path
module and2_4(y, a, b);
    input a, b;
    output y;
    wire c;
    assign #2 c = b;

    and #3(y, a, c);
    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule
```

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 nonzero 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.

SystemVerilog System Tasks and Functions

SystemVerilog system tasks and functions are built into the simulator, although some designs depend on user-defined system tasks implemented with the various programming and procedural interfaces.

If the simulator issues warnings regarding undefined system tasks or functions, then it is likely that these tasks or functions are defined by an interface application that must be loaded by the simulator.

ModelSim supports:

- Most SystemVerilog system tasks and functions defined in SystemVerilog IEEE Std 1800-2012
- Several system tasks and functions that are specific to ModelSim
- Several non-standard, Verilog-XL system tasks

IEEE Std 1800-2012 System Tasks and Functions	191
Using the \$typename Data Query Function	195
Task and Function Names Without Round Braces ‘()’	196
Verilog-XL Compatible System Tasks and Functions	198
String Class Methods for Matching Patterns	202

IEEE Std 1800-2012 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 1800-2012.

Note

 You can use the [change](#) command to modify local variables in Verilog and SystemVerilog tasks and functions.

Utility System Tasks and Functions

Table 5-3. Utility System Tasks and Functions

Simulator control tasks	Simulation time functions	Timescale tasks	Data query functions
\$finish	\$realtime	\$printtimescale	\$bits
\$stop	\$stime	\$timeformat	\$isunbounded
\$exit	\$time		\$typename

Table 5-4. Utility System Functions

Conversion functions	Array querying functions	Bit vector system functions
\$bitstoreal	\$dimensions	countbits
\$bitstoshortreal	\$left	countones
\$realtobits	\$right	\$onehot
\$shortrealtobits	\$low	\$onehot0
\$itor	\$high	\$isunknown
\$rtoi	\$increment	
\$signed	\$size	
\$unsigned		
\$cast		

Table 5-5. Utility System Math Functions

Math Functions			
\$clog2	\$floor	\$acos	\$cosh
\$ln	\$ceil	\$atan	\$tanh
\$log10	\$sin	\$atan2	\$asinh
\$exp	\$cos	\$hypot	\$acosh
\$sqrt	\$tan	\$sinh	\$atanh
\$pow	\$asin		

Table 5-6. Utility System Analysis Tasks and Functions

Probabilistic distribution functions	Stochastic analysis tasks and functions	PLA modeling tasks	Miscellaneous tasks and functions
\$dist_chi_square	\$q_add	\$async\$and\$array	\$system
\$dist_erlang	\$q_exam	\$async\$nand\$array	
\$dist_exponential	\$q_full	\$async\$or\$array	
\$dist_normal	\$q_initialize	\$async\$nor\$array	
\$dist_poisson	\$q_remove	\$async\$and\$plane	
\$dist_t		\$async\$nand\$plane	
\$dist_uniform		\$async\$or\$plane	

Table 5-6. Utility System Analysis Tasks and Functions (cont.)

Probabilistic distribution functions	Stochastic analysis tasks and functions	PLA modeling tasks	Miscellaneous tasks and functions
\$random		\$async\$nor\$plane \$sync\$and\$array \$sync\$nand\$array \$sync\$or\$array \$sync\$nor\$array \$sync\$and\$plane \$sync\$nand\$plane \$sync\$or\$plane \$sync\$nor\$plane	

Input/Output System Tasks and Functions

Table 5-7. Input/Output System Tasks and Functions

Display tasks	Value change dump (VCD) file tasks
\$display	\$dumpall
\$displayb	\$dumpfile
\$displayh	\$dumpflush
\$displayo	\$dumplimit
\$monitor	\$dumpoff
\$monitorb	\$dumpon
\$monitorh	\$dumpvars
\$monitoro	
\$monitoroff	
\$monitoron	
\$strobe	
\$strobeb	
\$strobeh	
\$strobeo	
\$write	

Table 5-7. Input/Output System Tasks and Functions (cont.)

Display tasks	Value change dump (VCD) file tasks
\$writeb	
\$writeh	
\$writeo	

Table 5-8. Input/Output System Memory and Argument Tasks

Memory load tasks	Memory dump tasks	Command line input
\$readmemb	\$writememb	\$test\$plusargs
\$readmemh	\$writememh	\$value\$plusargs

Table 5-9. Input/Output System File I/O Tasks

File I/O tasks		
\$fclose	\$fmonitoro	\$fwriteo
\$fdisplay	\$open	\$rewind
\$fdisplayb	\$fread	\$sdf_annotation
\$fdisplayh	\$fscanf	\$sformatf
\$fdisplayo	\$fseek	\$sscanf
\$feof	\$fstrobe	\$swrite
\$ferror	\$fstrobeb	\$swriteb
\$fflush	\$fstrobeh	\$swriteh
\$fgetc	\$fstrobeo	\$swriteo
\$fgets	\$ftell	\$ungetc
\$fmonitor	\$fwrite	
\$fmonitorb	\$fwriteb	
\$fmonitorh	\$fwriteh	

Other System Tasks and Functions

Table 5-10. Other System Tasks and Functions

Timing check tasks	Random number functions	Other functions
\$hold	\$urandom	\$root
\$nochange	\$urandom_range	\$unit

Table 5-10. Other System Tasks and Functions (cont.)

Timing check tasks	Random number functions	Other functions
\$period		
\$recovery		
\$setup		
\$setuphold		
\$skew		
\$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. 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.

Using the \$typename Data Query Function

The type name string returned by \$typename() does not include class, struct and enum members, nor any class extensions.

You can override this default behavior using any of the following predefined macros as the optional second argument to \$typename():

- `mtiTypenameExpandSuper — Extensions are included in type name.
- `mtiTypenameExpandMembers — Class, struct and enum members are included.
- `mtiTypenameExpandAll — Members and extensions are both included.

Example Usage

```
$typename(a, `mtiTypenameExpandAll);
```

The various forms of \$typename() output for a parametrized class: vector, which extends another parametrized class: vector_base, both of which are defined in the module scope: typename_parameterized_class, are shown in the following:

- \$typename(a) will return:

```
class typename_parameterized_class/vector #(10, reg, 0)
```

- \$typename(a, `mtiTypenameExpandSuper) will return:

```
class typename_parameterized_class/vector #(10, reg, 0) extends
  class typename_parameterized_class/vector_base #(reg)
```

- \$typename(a, `mtiTypenameExpandMembers) will return:

```
class typename_parameterized_class/vector #(10, reg, 0){reg b;
  reg$[9:0] a;}
```

- \$typename(a, `mtiTypenameExpandAll) will return:

```
class typename_parameterized_class/vector #(10, reg, 0){reg b;
  reg$[9:0] a;} extends class typename_parameterized_class/
vector_base #(reg){reg b;}
```

Task and Function Names Without Round Braces ‘()’

Strict compliance with the Language Reference Manual IEEE Std 1364 requires that all hierarchical task and function names have parentheses “()” following the name to call the task or function. In ModelSim 10.3 and later you may use hierarchical task and function names without parentheses.

The compiler uses the following rules for interpreting task and function names without parentheses:

1. Non class tasks/functions (static or non static) are interpreted as a search in the scope of the function and not a function call.
2. Non-static class methods are treated as a function call.
3. Static class methods are treated as a lookup in the function scope.
4. Once a function call is made for a hierarchical name, all subsequent function names are treated as function calls whether the type of function is static or non-static.

Examples

```

module top;
    class CTest1 ;
        string s;
        static function CTest1 g();
            static CTest1 s = new();
            CTest1 t = new();
            $display ("hello_static" );
            return t;
        endfunction
        function CTest1 f();
            static string s;
            CTest1 t = new();
            $display ("hello_auto" );
            return t;
        endfunction
    endclass;
    CTest1 t1 = new();

    initial t1.g.s.f.g.s="hello";

endmodule

```

In the above code, the dotted name:

`t1.g.s.f.g.s`

is interpreted by the fourth rule above as:

`t1.g.s.f().g().s`

The first *g* is treated as a scope lookup, since it is a static function. Since *f* is an automatic function, it is treated as a function call. The next *g* is treated as a function call *g()* since according to rule 4, once an automatic function gets called, all subsequent names in the list which are Function names, whether static or automatic, are treated as function calls.

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.....	198
Supported Tasks and Functions Not Described in IEEE Std 1364.....	198
Extensions to Supported System Tasks	201
Unsupported Verilog-XL System Tasks	201

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  
$sreadmemb  
$sreadmemh
```

Supported Tasks and Functions Not Described in IEEE Std 1364

Some system tasks are also provided for compatibility with Verilog-XL, though they are not described in the IEEE Std 1364.

The \$deposit system task sets a Verilog net to the specified value. variable is the net to be changed; value is the new value for the net. The value remains until there is a subsequent driver transaction or another \$deposit task for the same net. This system task operates identically to the ModelSim force -deposit command.

```
$deposit(variable, value);
```

The \$disable_warnings system task instructs the simulator 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, The simulator disables warnings for the entire simulation.

```
$disable_warnings("<keyword>" [,<module_instance>...]);
```

Disabling warnings with \$disable_warnings does not prevent notifier toggling by default. If you want to prevent notifier toggling when the \$disable_warnings system task is in effect, pass the -no_notifier_on_disable argument to vsim.

The \$enable_warnings 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, the simulator enables warnings for the entire simulation.

```
$enable_warnings("<keyword>" [, <module_instance>...]);
```

The \$system 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("command");
```

where the return value of the \$system function is a 32-bit integer that is set to the exit status code of the underlying OS process.

```
$system("ls -l");
```

Note

 There is a known issue in the return value of the \$system 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.

The \$systemf 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.

```
$systemf(list_of_args)
```

Note

 There is a known issue in the return value of the \$systemf 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.

The \$test\$plusargs system function tests for the presence of a specific plus argument on the simulator's command line.

```
$test$plusargs("plus argument")
```

It returns 1 if the plus argument is present; otherwise, it returns 0. For example, to test for +verbose:

```
if ($test$plusargs("verbose"))
$display("Executing cycle 1");
```

Extensions to Supported System Tasks

Additional functionality has been added to the \$fopen, \$setuphold, and \$recrem system tasks.

New Directory Path With \$fopen **201**

Negative Timing Checks With \$setuphold and \$recrem **201**

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 [CreateDirFor FileAccess](#) *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 CreateDirFor FileAccess 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.

Related Topics

[Commands Supporting Negative Timing Check Limits](#)

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.

The \$input system task reads commands from the specified filename. The equivalent simulator command is do <filename>.

```
$input("filename")
```

The \$list 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.

```
$list[(hierarchical_name)]
```

The \$reset system task resets the simulation back to its time 0 state. The equivalent simulator command is restart.

```
$reset
```

The \$restart 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>.

```
$restart("filename")
```

The \$save system task saves the current simulation state to the file specified by filename. The equivalent simulator command is checkpoint <filename>.

```
$save("filename")
```

The \$scope system task sets the interactive scope to the scope specified by hierarchical_name. The equivalent simulator command is environment <pathname>.

```
$scope(hierarchical_name)
```

The \$showscopes system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is show.

```
$showscopes
```

The \$showvars system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is show.

```
$showvars
```

String Class Methods for Matching Patterns

This group of functions are not a part of the SystemVerilog LRM. However, the ModelSim simulator supports their use, unless you include the -pedanticerrors argument to vlog, in which case you will receive an error.

The regular expressions for these functions use Perl pattern syntax.

- search() — This function searches for a pattern in the string and returns the integer index to the beginning of the pattern.

```
search(string pattern);
```

where pattern must be a string. For example:

```
integer i;
string str = "ABCDEFGHIJKLM";
i = str.search("HIJ");
printf("%d \n", i);
```

results in printing out “8”.

- `match ()` — This function processes a regular expression pattern match, returning a 1 if the expression is found or a 0 if the expression is not found or if there is an error in the regular expression.

```
match (string pattern);
```

where pattern must be a regular expression. For example:

Example 5-7. `match()` Example

```
integer i;
string str;
str = "ABCDEFGHIJKLM";
i = str.match("CDE");
```

results assigning the value 1 to integer **i** because the pattern CDE exists within string **str**.

- `prematch()` — This function returns the string before a match, based on the result of the last `match()` function call.

```
prematch();
```

Based [Example 5-7](#), the following:

```
str1 = str.prematch();
```

would be assigned the string “AB”

- `postmatch()` — This function returns the string after a match, based on the result of the last `match()` function call.

```
postmatch();
```

Based on [Example 5-7](#), the following:

```
str2 = str.postmatch();
```

would be assigned the string “FGHIJKLM”

- `thismatch()` — This function returns matched string, based on the result of the last `match()` function call.

```
thismatch();
```

Based on [Example 5-7](#), the following:

```
str3 = str.thismatch();
```

would be assigned the string “CDE”

- `backref()` — This function returns matched patterns, based on function call in [Example 5-7](#).

```
backref(integer index);
```

where index is the integer number of the expression being matched (indexing starts at 0).
For example:

```
integer i;
string str, patt, str1, str2;
str = "12345ABCDE"
patt = "([0-9]+) ([a-zA-Z .]+)";
i = str.match(patt);
str1 = str.backref(0);
str2 = str.backref(1);
```

results in assigning the value “12345” to the string str1 because of the match to the expression “[0-9]+”. It also results in assigning the value “ABCDE” to the string str2 because of the match to the expression “[a-zA-Z .]+”.

You can specify any number of additional Perl expressions in the definition of patt and then call them using sequential index numbers.

Compiler Directives

The ModelSim simulator support of SystemVerilog includes all of the compiler directives defined in the IEEE Std 1800, 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, resetting them back to their default settings (this information is not provided in the IEEE Std 1800):

```
`celldefine
`default_decay_time
`default_nettpe
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`protect
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macros:

```
MODEL_TECH
QUESTA
SV_COV_ASSERTION
SV_COV_CHECK
SV_COV_ERROR
SV_COV_FSM_STATE
SV_COV_HIER
SV_COV_MODULE
SV_COV_NOCOV
SV_COV_OK
SV_COV_OVERFLOW
SV_COV_PARTIAL
SV_COV_RESET
SV_COV_START
SV_COV_STATEMENT
SV_COV_STOP
SV_COV_TOGGLE
VLOG_DEF
mtiTypenameExpandAll
mtiTypenameExpandMembers
mtiTypenameExpandSuper
```

IEEE Std 1364 Compiler Directives..... 206

Verilog-XL Compatible Compiler Directives	206
--	------------

IEEE Std 1364 Compiler Directives

The simulator supports SystemVerilog compiler directives, which are described in detail in the IEEE Std 1364.

```
`celldefine
`default_nettype
`define
`else
`elsif
`endcelldefine
`endif
`ifdef
`ifndef
`include
`line
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef
```

Verilog-XL Compatible Compiler Directives

The ModelSim simulator supports a number of compiler directives that are compatible 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 nonzero 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
```

Unmatched Virtual Interface Declarations

The [1800-2012 SV] LRM does not address the relationship between interfaces as design elements and virtual interfaces as types. The ModelSim flow allows virtual interfaces to exist even when the underlying interface design unit does not exist, even in the design libraries.

When no matching interface exists, a virtual interface necessarily has a null value throughout simulation, as any incompatible assignment causes an error. In all cases of accessing data during simulation through such a virtual interface, an error results due to dereferencing a null virtual interface.

Situations exist in which types from such references can participate in the design without requiring a dereference of the virtual interface pointer. This is extremely rare in practice, but due to the simulator's overall elaboration and simulation flow, it is not possible for ModelSim to determine whether such type references will actually be exercised during simulation. For these cases, the following argument enables vsim to elaborate the design:

```
vsim -permit_unmatched_virtual_intf
```

Tip

-  When using the `-permit_unmatched_virtual_intf` argument, ensure that no simulation time operations occur through unmatched virtual interfaces.
-

Related Topics

[vsim \[ModelSim Command Reference Manual\]](#)

Verilog PLI and SystemVerilog DPI

ModelSim supports the use of several interfaces.

The interfaces include:

- Verilog PLI (Programming Language Interface)
- 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.

Extensions to SystemVerilog DPI..... [209](#)

Extensions to SystemVerilog DPI

The ModelSim simulator supports various extensions to the SystemVerilog DPI.

- SystemVerilog DPI extension to support automatic DPI import tasks and functions.
You can specify the automatic lifetime qualifier to a DPI import declaration so 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 of import functions:

```
import DPI-C cfoo = task automatic foo(input int p1);
import DPI-C context function automatic int foo (input int p1);
```

SystemVerilog Class Debugging

Debugging your design starts with an understanding of how the design is put together; the hierarchy, the environments, the class types. The ModelSim debug environment gives you a number of avenues for exploring your design, finding the areas of the design that are causing trouble, pinpointing the specific part of the code that is at fault, making the changes necessary to fix the code, then running the simulation again.

This section describes the steps you take to enable the class debugging features and the windows and commands that display information about the classes in your design.

Enabling Class Debug	210
The Class Instance Identifier	211
Logging Class Types and Class Instances	212
Working with Class Types	213
Working with Class Instances	219
Working with Class Path Expressions	225
Conditional Breakpoints in Dynamic Code	228
Stepping Through Your Design	229
The Run Until Here Feature	230
Command Line Interface	231
Class Instance Garbage Collection	241

Enabling Class Debug

You can enable visibility of class instances in your design in two ways.

Procedure

Use either of the following methods:

- Use the vsim command with the -classdebug argument.
- Set the **ClassDebug** *modelsim.ini* variable to 1.

The Class Instance Identifier

The Class Instance Identifier (CIID or Handle) is a unique name for every class instance created during a simulation.

The CIID format is

`@<class-type>@<n>`

where `<class_type>` is the name of the class and `<n>` is the nth instance of that class. For example:

`@packet@134`

is the 134th instance of the class type packet.

You can use the class type name alone in the CIID if the class type name is unique in the design. If the class type name is not unique, the full path to the type declaration is necessary.

You can use the CIID in commands such as `examine`, `describe`, `add wave`, and `add list`.

Note

 A CIID is unique for a given simulation. Modifying a design, or running the same design with different parameters, randomization seeds, or other configurations that change the order of operations, may result in a class instance changing. For example, `@packet@134` in one simulation run may not be the same `@packet@134` in another simulation run if the design has changed.

Obtaining the CIID with the examine Command.	211
Obtaining the CIID With a System Function	211

Obtaining the CIID with the examine Command

You can use the `examine -handle` command to return the CIID to the transcript.

Procedure

Enter the following command at the command line:

`examine -handle <filename>`

Obtaining the CIID With a System Function

You can use the built in system function `$get_id_from_handle(class_ref)` to obtain the string representing the class instance id for the specified class reference.

Procedure

Add the `$get_id_from_handle(class_ref)` function to your design to return the CIID of the class item referenced by *var*.

```
myclass var;
initial begin
    #10
    var = new();
    $display( "%t : var = %s", $time, $get_id_from_handle(var) );
end
```

Results

```
10 : var = @myclass@1
```

Logging Class Types and Class Instances

You must log class variables, class types, or class instances in order to view them in the Wave and List windows, and to view them post-simulation. The data recorded depends on the type of class object you log.

- Log the class variable to create a record of all class objects the variable references from the time they are assigned to the variable to when they are destroyed. For example:

log sim:/top/simple

You can find the correct syntax for the class variable by dragging and dropping the class variable from the Objects window into the Transcript.

- Log a class type to create a contiguous record of each instance of that class type from the time the instance first comes into existence to the time the instance is destroyed with the `log -class` command. For example:

log -class sim:/mem_agent_pkg::mem_item

Refer to [Finding the Class Type Syntax](#) for more information.

- Log a specific instance of a class until it is destroyed by specifying the class identifier for the specific class instance. For example:

log @myclass@7

Refer to [The Class Instance Identifier](#) for more information about finding and specifying a class instance identifier.

- Log a Class Path Expression. Refer to [Working with Class Path Expressions](#) for more information.

Working with Class Types

You can view the class types in your design in the Class Tree, Class Graph, Structure, and other windows.

Authoritative and Descriptive Class Type Names	214
Finding the Class Type Syntax	214
Viewing Class Types in the GUI.....	216

Authoritative and Descriptive Class Type Names

ModelSim maintains two representations for class names: the authoritative class type name and the descriptive class type name. This name mapping is specifically to support parameterized class specializations.

Authoritative Class Type Names	214
Descriptive Class Type Names	214

Authoritative Class Type Names

Authoritative names are used in most places in the user interface. They are also used as input to commands that take a class type argument.

Authoritative names offer a short, well-formed name for a parameterized class specialization. Authoritative names end with "`__n`" where '`n`' is an integer. For example:

```
/pkg::mypclass__6.
```

Descriptive Class Type Names

Descriptive names more closely resemble the class definition, can be very long and difficult to read and parse.

Descriptive names are used in error messages and are shown in some places in the GUI such as in the class tree window. For example:

```
/pkg::mypclass #( class inputclass, 128, class report__2 ).
```

The [classinfo descriptive](#) command translates an authoritative name to a descriptive name. For example:

```
classinfo descriptive /pkg::mypclass__6
```

returns:

```
# Class /pkg::mypclass__6 maps to /pkg::mypclass #( class inputclass, 128,  
class report__2 )
```

In this example, one of the parameters in the descriptive name is also a specialization of a parameterized class.

Finding the Class Type Syntax

The <class_type> can be specified using the specific class type name or any path that resolves to the class type.

For example: `@packet@134` can also be specified as `@/test_pkg::packet@134`, assuming the class `packet` is defined in `/test_pkg`.

You can use the [classinfo types](#) -n command to determine whether or not a type name is unique and return the requisite full class type name to the transcript. For example, the following command returns all the shortest usable names for all class type names containing the string “foo”:

```
classinfo types -n *foo*
```

```
# my_foo
# foo2
# /top/mod1/foo
# /top/mod2/foo
```

In the output, `my_foo` and `foo2` are unique class types. However, the last two entries show that there are two distinct class types with the name 'foo'; one defined in `mod1` and the other in `mod2`. To specify an instance of type 'foo', the full path of the specific “foo” is required, for example `@/top/mod2/foo@19`.

You can also find the correct syntax for a class type by dragging and dropping the class type from the Structure window into the Transcript window.

Viewing Class Types in the GUI

You can view class types in several windows, including the Structure, Class Tree, and Class Graph windows.

The Class Tree Window	216
The Class Graph Window.....	216
The Structure Window	217

The Class Tree Window

The Class Tree window displays the class inheritance tree in various forms. You can expand objects to see parent/child relationships, properties, and methods. You can organize by extended class (default) or base class.

The Class Tree window can help with an overview of your environment and architecture. It also helps you view information about an object that is both a base and extended class. ([Figure 5-5](#))

Figure 5-5. Classes in the Class Tree Window

The screenshot shows the 'Class Tree' window with the following data:

Class	Type	File	Unique Id	Scope
semaphore	Class	std.sv	semaphore	
Methods				
f(x) new	Function			
f(x) post_randomize	Function			
f(x) pre_randomize	Function			
f(x) constraint_mode	Function			
Properties				
x= p1	chandle			
x= keyCount	int			
std/mailbox	Param Class			
mailbox #(class Packet)	Class	std.sv	mailbox_1	
Methods				
Properties				
x= items	Queue			
x= maxItems	int			
x= read_awaiting	chandle			
x= write_awaiting	chandle			
x= qtd	chandle			
read_semaphore	semaphore	std.sv	semaphore	
write_semaphore	semaphore	std.sv	semaphore	

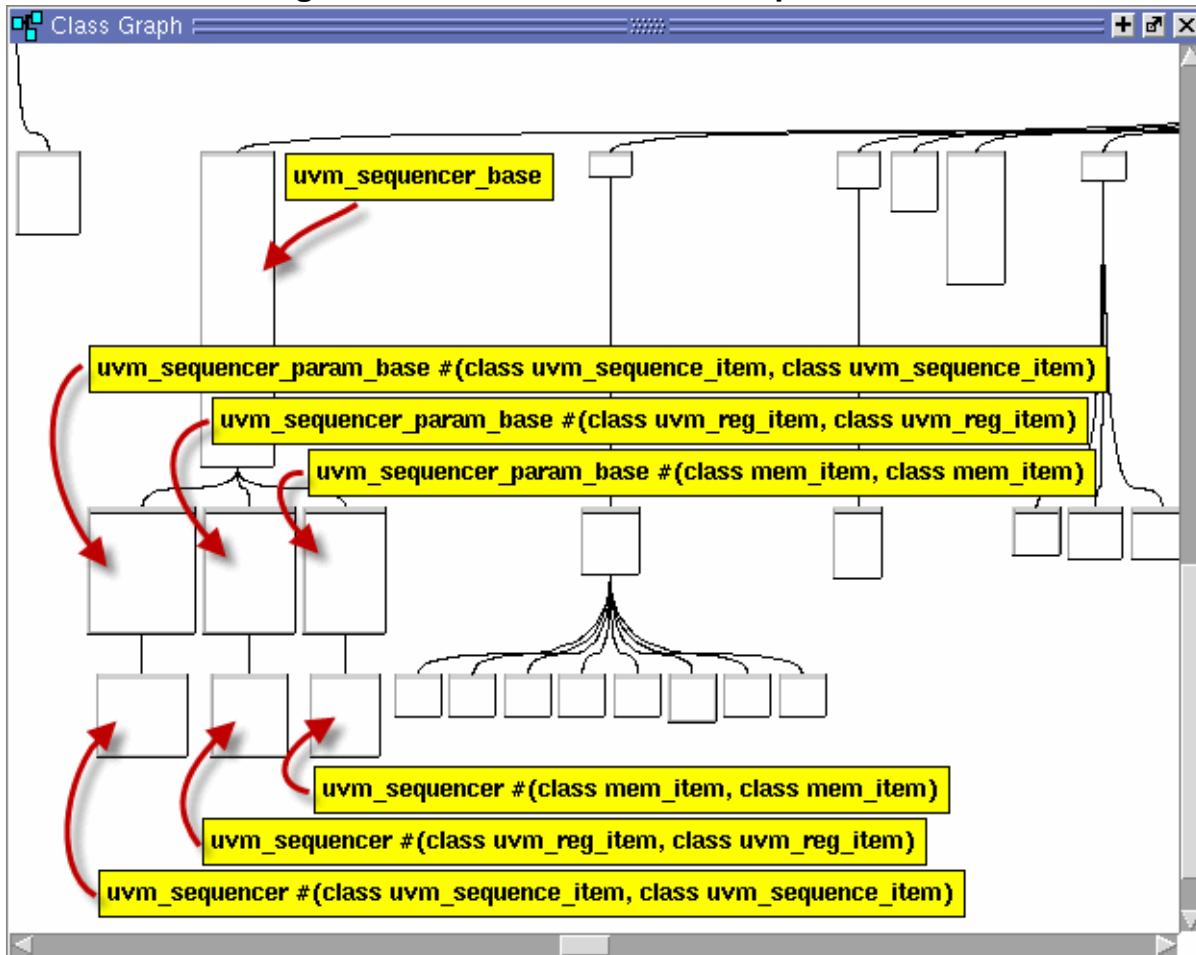
Refer to [Class Tree Window](#) in the GUI Reference Manual for more information.

The Class Graph Window

The Class Graph window displays interactive relationships between SystemVerilog classes in a graphical form and includes extensions of other classes and related methods and properties.

You can organize by extended class (default) or by base class. Figure 5-6 shows an example of using this window to display all the relationships among the classes in your design.

Figure 5-6. Class in the Class Graph Window



Refer to [Class Graph Window](#) in the GUI Reference Manual for more information.

The Structure Window

The Structure window displays the class types in your design. You must select a class type in the Structure window to view that class type's instances in the Class Instances window.

Figure 5-7. Classes in the Structure Window

Instance	Design unit	Design unit type
top	top(fast)	Module
tb_pkg	tb_pkg(fast)	ViPackage
my_print_accessors	tb_pkg(fast)	Function
my_print_resources	tb_pkg(fast)	Function
go	tb_pkg(fast)	Function
mode_t	tb_pkg(fast)	ViTypeDef
transaction	transaction	SVClass
new	tb_pkg(fast)	Function
type_id	tb_pkg(fast)	ViTypeDef
get_type	tb_pkg(fast)	Function
get_object_type	tb_pkg(fast)	Function
create	tb_pkg(fast)	Function
get_type_name	tb_pkg(fast)	Function
__m_uvm_field_automation	tb_pkg(fast)	Function
#ublk#128476951#31	tb_pkg(fast)	Statement
__local_type____	tb_pkg(fast)	ViTypeDef
convert2string	tb_pkg(fast)	Function
do_record	tb_pkg(fast)	Function
config_object	config_object	SVClass
driver	driver	SVClass

Working with Class Instances

Viewing class instances is helpful for finding class, OVM, and UVM components or subtypes that have been instantiated.

You can use the Class Instances window, the classinfo report, or the classinfo instances command to see how many instances have been created. You can search through the list of components or transactions for an object with a specific value in the Objects window.

The Class Instances Window	219
Viewing Class Instances in the Wave Window	221
The Locals Window	223
The Watch Window	223
The Call Stack Window	224

The Class Instances Window

The Class Instances window displays information about all instances of a selected class type that exist at the current simulation time.

You can open the Class Instances window by choosing **View > Class Browser > Class Instances** from the main menu or by specifying `view classinstances` on the command line. (Figure 5-8)

Figure 5-8. The Class Instances Window

Name	Value	Kind
- @mem_item@9	{mem_item} 905 @uvm_object_string_pool...	Class Instance
super	{mem_item} 905 @uvm_object_string_pool...	SVClass(uvm_seq...
super	{mem_item} 905 @uvm_object_string_pool...	SVClass(uvm_tran...
m_sequence_id	-1	Int
m_use_sequence_inf...	1	Protected Bit
m_depth	-1	Protected Int
m_sequencer	{m_sequencer} 632 @uvm_report_handler@...	Class Instance
m_parent_sequence	{m_mem_seq} 499 @uvm_object_string_poo...	Class Instance
print_sequence_info	0	Bit
m_client_str		Protected String
m_client	null	Class Instance
m_rh	null	Class Instance
issued1	0	Static Bit
issued2	0	Static Bit
report_id	MEM_ITEM	String
instruction	READ	Enum
address	00000000	Packed Array
data_to_dut	0000000000000000	Packed Array
data_to_dut_valid	0	Bit
latency	00000000	Packed Array
data_from_dut	0000000000000000	Packed Array
choose_read_address	0	Int
addresses_written_list	{0:0000000000000000} {1:00000000011001...}	Static Associative ...
instructions_sent	2	Static Int
type_name	mem_item	Static String
+ @mem_item@8	{m_out_item} 901 @uvm_object_string_pool...	Class Instance
+ @mem_item@6	{mem_monitor item} 893 @uvm_object_string...	Class Instance
sim:/mem_agent_pkg::mem_item (fixed)		

You must enable the class debug feature to use the Class Instances window. Refer to [Enabling Class Debug](#) for more information.

The Class Instances window is dynamically populated when you select SystemVerilog classes 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. Refer to [The classinfo Commands](#) for more information about verifying the current state of a class instance.

Once you have chosen the design unit you want to observe, you can lock the Class Instances window on that design unit by choosing **File > Environment > Fix to Current Context when the Class Instances window is active**. from the main menu

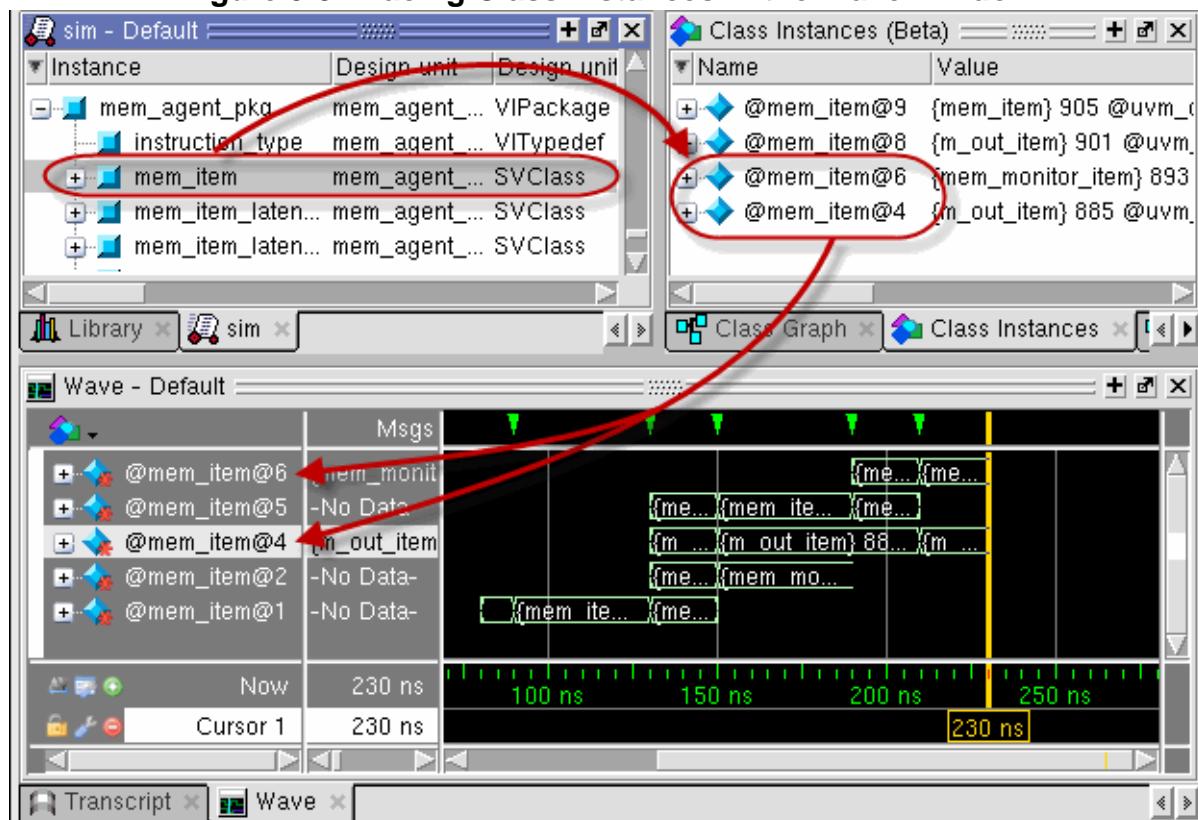
Viewing Class Instances in the Wave Window

There is a suggested workflow for logging SystemVerilog class objects in the Wave window.

Workflow

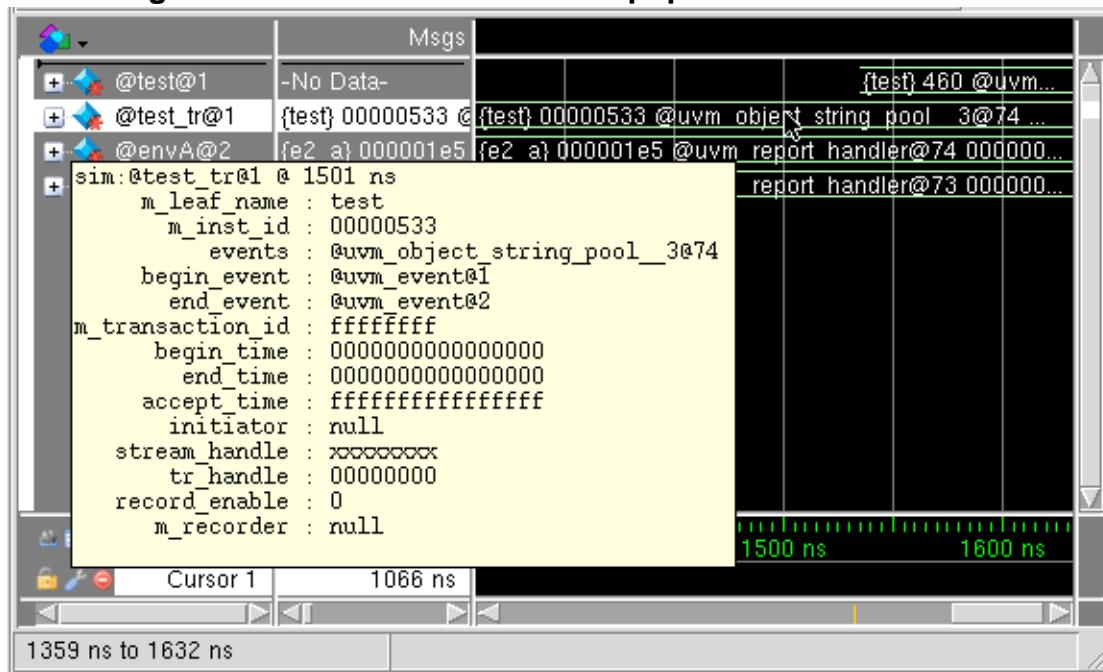
1. Log the class objects you are interested in viewing (refer to [Logging Class Types and Class Instances](#) for more information)
2. In the Structure Window, select a design unit or testbench System Verilog class type that contains the class instances you want to see. The class type will be identified as a System Verilog class object in the Design Unit column. All currently existing class instances associated with that class type or testbench item are displayed in the Class Instances window. (Open the Class Instances window by selecting **View > Class Browser > Class Instances** from the menus or use the [view class instances](#) command.)
3. Place the class objects in the Wave window, once they exist, by doing one of the following:
 - Drag a class instance from the Class Instances window or the Objects window and drop it into the Wave window (refer to [Figure 5-9](#)).
 - Select multiple objects in the Class Instances window, 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. The group of class instances are arranged with the most recently created instance at the top. You can change the order of the class instances to show the first instance at the top of the window by selecting **View > Sort > Ascending**.

Figure 5-9. Placing Class Instances in the Wave Window



You can hover the mouse over any class waveform to display information about the class variable (Figure 5-10).

Figure 5-10. Class Information Popup in the Wave Window



The Locals Window

The Locals window displays data objects that are immediately visible at the current execution point of the selected context.

Clicking in the objects window or Structure window might make you lose the current context. The Locals window is synchronized with the Call-Stack window and the contents are updated as you move through the design.

Related Topics

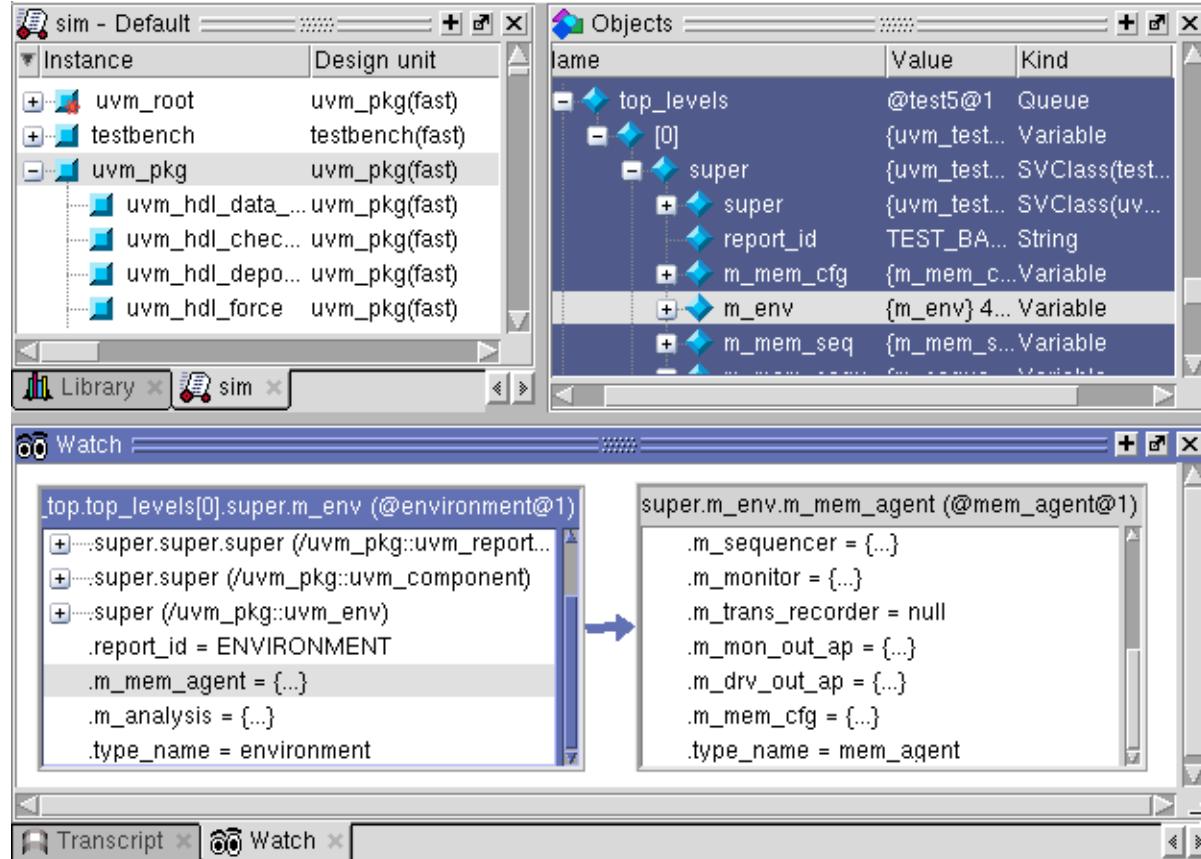
[Locals Window \[ModelSim GUI Reference Manual\]](#)

The Watch Window

The Watch window displays signal or variable values at the current simulation time. It enables you to view a subset of local or class variables when stopped on a breakpoint.

Use the Watch window when the Locals window is crowded. You can drag and drop objects from the Locals window into the Watch window ([Figure 5-11](#)).

Figure 5-11. Class Viewing in the Watch Window



Refer to [Watch Window](#) in the GUI Reference Manual for more information.

The Call Stack Window

The Call Stack window enables you to view your design when you are stopped at a breakpoint. You can go up the call stack to see the locals context at each stage of your design.

Related Topics

[Call Stack Window \[ModelSim GUI Reference Manual\]](#)

Working with Class Path Expressions

A class path expression is a hierarchical path through a class hierarchy.

Class path expressions:

- allow you to view class properties in the Wave and Watch windows, and return data about class properties with the `examine` command. You can see how the class properties change over time even when class references within the path expression change values.
- can be added to the Wave window.
- can be expanded inline in the Wave window without having to add class objects to the Wave window individually.
- can be cast to the legal types for the expression. In the Wave window, the casting options are restricted to the set of types of objects actually assigned to the references.
- are automatically logged once the expression is added to the Wave window.

Class Path Expression Syntax	225
Adding a Class Path Expression to the Wave Window	226
Class Path Expression Values	226
Casting a Class Variable to a Specific Type	226
Class Objects vs Class Path Expressions	228
Disabling Class Path Expressions	228

Class Path Expression Syntax

Class path expressions require a specific syntax.

For example, a correct path expression is written as follows:

```
/top/myref.xarray[2].prop
```

where

- *myref* is a class variable
- *xarray* is an array of class references
- *prop* is a property in the *xarray* element class type

In this example the expression allows you to watch the value of *prop* even if *myref* changes to point to a different class object, or if the reference in element [2] of *xarray* changes.

Adding a Class Path Expression to the Wave Window

You can add a class path expression to the Wave window with the add wave command.

For example:

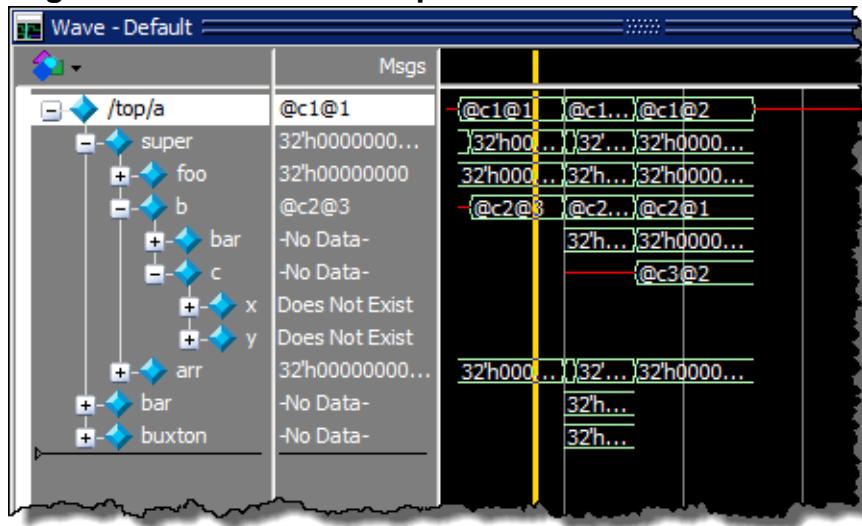
```
add wave /top/myref.ref_array[0].prop
```

Class Path Expression Values

A class path expression can have one of several possible values.

- The expression can have a standard value of the type of the leaf element in the expression.
- The expression can have a value of ‘Null’ if the leaf element is a class reference and its value is null.
- The expression can have a value of ‘Does Not Exist’ if an early part of the expression has a null value. In the earlier example, `/top/myref.xarray[2].prop`, if `myref` is null then `prop` does not exist.

Figure 5-12. Class Path Expressions in the Wave Window



Casting a Class Variable to a Specific Type

You can cast a class variable to any of the class types that have been assigned to that class variable. The default is the declared type of the class variable.

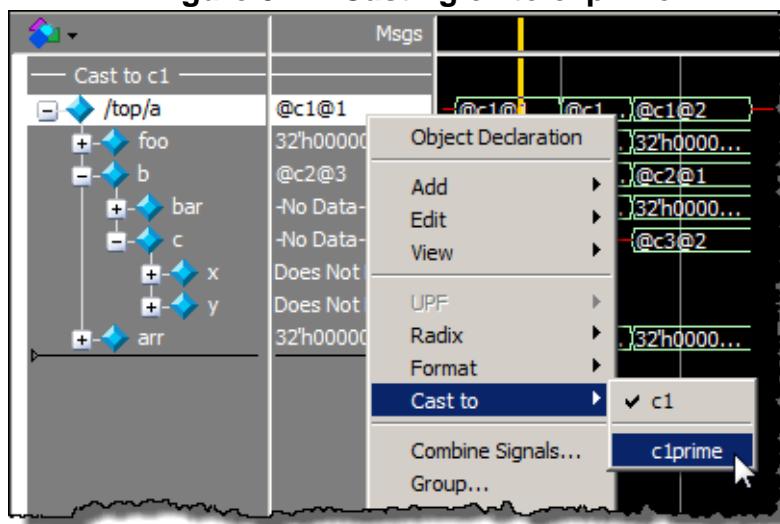
Figure 5-13. /top/a Cast as c1 and c1prime



Procedure

1. Right-click (RMB) the class variable waveform to display a popup menu and choose **Cast to**.
2. Right-click over the name/value of the class reference in the Pathnames or the Values Pane of the Wave window to open a popup menu.
3. Choose **Cast to > <class_type>**. The current value will have check mark next to it. ([Figure 5-14](#))

Figure 5-14. Casting c1 to c1prime



Class Objects vs Class Path Expressions

By default, the user interface interprets paths that include class references as path expressions. There are cases where the interpreted object is what is desired and not the path expression.

For example,

```
add wave /top/myref.prop
```

adds the class path expression to the wave window. The expression is evaluated regardless of what class object is referenced by *myref*.

Using the -obj argument to the **add wave** command causes the command to interpret the expression immediately and add the specific class object to the Wave window instead of the class path expression. For example:

```
add wave -obj /top/myref.prop
```

adds the current class object and property to the Wave window, in this case, @*myref*@19.prop. @*myref*@19 is the specific object at the time the command was executed.

Disabling Class Path Expressions

Setting the MTI_DISABLE_PATHEXPR environment variable disables the interpretations of all class path expressions. This is equivalent to the behavior in version 10.2 and earlier.

Conditional Breakpoints in Dynamic Code

Set a breakpoint or a conditional breakpoint at any place in your source code.

Examples

- Conditional breakpoint in dynamic code

```
bp mem_driver.svh 60 -cond {this.id == 9}
```

- Stop on a specific instance ID.

- Enter the command:

```
examine -handle
```

- Drag and drop the object from the Objects window into the Transcript window, which adds the full path to the command.

```
examine -handle
{sim:/uvm_pkg::uvm_top.top_levels[0].super.m_env.m_mem_agent.m_driver}
```

- Press Enter

Returns the class instance ID in the form @<class_type>@<n>:

```
# @mem_driver@1
```

- Enter the class instance ID as the condone in the breakpoint.

```
bp mem_driver.svh 60 -cond {this == @mem_driver@1}
```

- Stop on a more complex condition:

```
bp bfm.svh 50 {
    set handle [examine -handle this];
    set x_en_val [examine this.x_en_val];
    if {($handle != "@my_bfm@7") || ($x_en_val != 1)}{
        stop
    } else {
        run -continue
    }
}
```

Refer to [Setting Conditional Breakpoints](#) or more information about conditional breakpoints.

Stepping Through Your Design

Stepping through your design is helpful once you have pinpointed the area of the design where you think there is a problem. In addition to stepping to the next line, statement, function or procedure, you have the ability to step within the current context (process or thread). This is helpful when debugging class based code since the next step may take you to a different thread or section of your code rather than to the next instance of a class type.

For example:

Table 5-11. Stepping Within the Current Context.



Step the simulation into the next statement, remaining within the current context.



Step the simulation over a function or procedure remaining within the current context. Executes the function or procedure call without stepping into it.



Step the simulation out of the current function or procedure, remaining within the current context.

Refer to [Step Toolbar](#) in the GUI Reference Manual for a complete description of the stepping features.

The Run Until Here Feature

To quickly and easily run to a specific line of code, you can use the ‘Run Until Here’ feature. When you invoke Run Until Here, the simulation runs from the current simulation time and stops on the specified line of code.

The simulator stops at the specified time, unless:

- It encounters a breakpoint.
- The **Run Length** preference variable causes the simulation run to stop.
- It encounters a bug.

To specify **Run Until Here**, right-click 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.

Refer to [Run Until Here](#) for more information.

Command Line Interface

You can enter commands on the command line in the Transcript window to interact with class instances.

This allows you to work with data for class types, their scopes, paths, names, and related information. You can call SystemVerilog static functions and class functions with the call command. Commands also help you find the proper name syntax for referencing class-based objects in the GUI.

Class Instance Values	231
Class Instance Properties	231
Calling Functions	232
The classinfo Commands.....	234

Class Instance Values

The examine command returns current values for classes or variables to the transcript while debugging. The examine command can help you debug by displaying the name of a class instance or the field values for a class instance before setting a conditional breakpoint.

Examples

- Print the current values of a class instance.
examine /ovm_pkg::ovm_test_top
- Print the values when stopped at a breakpoint within a class.
examine this
- Print the unique ID of a specific class instance using the full path to the object.
examine -handle /ovm_pkg::ovm_test_top.i_btn_env
- Print the unique handle of the class object located at the current breakpoint.
examine -handle this
- Print the value of a specific class instance.
examine @mem_item@9

Class Instance Properties

Use the describe command to display data members, properties, methods, tasks, inheritance, and other information about class instances, and print it in the transcript window.

- Display data for the class instance **@questa_messagelogger_report_server@1**

describe @questa_messagelogger_report_server@1

Returns:

```
# class /questa_uvm_pkg::questa_messagelogger_report_server extends
/uvm_pkg::uvm_report_server
#     static /questa_uvm_pkg::questa_messagelogger_report_server
m_q;
#     function new;
#     static function message_logger;
#     function compose_message;
#     function process_report;
#     static function get;
#     static function init;
# endclass
```

- Display data for the class type *mailbox_1*

describe mailbox_1

Returns:

```
class /std::mailbox::mailbox_1
#     Queue items;
#     int maxItems;
#     chandle read_awaiting;
#     chandle write_awaiting;
#     chandle qtd;
#     /std::semaphore  read_semaphore;
#     /std::semaphore  write_semaphore;
#     function new;
#     task put;
#     function try_put;
#     task get;
#     function try_get;
#     task peek;
#     function try_peek;
#     function post_randomize;
#     function pre_randomize;
#     function constraint_mode;
# endclass
```

Calling Functions

The call command calls SystemVerilog static functions and class functions directly from the Vsim command line in live simulation mode and Verilog interface system tasks and system functions.

Function return values are returned to the Transcript window as a Tcl string, where is it returns the class instance ID when a function returns a class reference.

Call a static function or a static 0 time task from the command line, for example:

```
call /ovm_pkg::ovm_top.find my_comp
call @ovm_root@1.find my_comp
call @ovm_root@1.print_topology
call /uvm_pkg::factory.print
```

The classinfo Commands

The classinfo commands give you high level information about the class types and class instances in your design.

Finding the Full Path and Name of a Class Type	234
Determining the Current State of a Class Instance	235
Finding All Instances of a Class Type	235
Reporting Statistics for All Class Instances	236
Reporting Class Instance Statistics for a Simulation Run.....	237
Reporting Active References to a Class Instance	237
Finding Class Type Inheritance	238
Listing Classes Derived or Extended From a Class Type	239
Analyzing Class Types.....	240

Finding the Full Path and Name of a Class Type

The classinfo descriptive command returns the descriptive class type name, given the authoritative class type name.

The authoritative class type name (for example, myclass_9) has a corresponding descriptive name that may be more useful in determining the actual class type and the details of its specialization. This command allows you to see the mapping from the authoritative name to the descriptive name.

Prerequisites

Specify the -classdebug argument with the `vsim` command.

Procedure

Enter the classinfo descriptive command for the desired class type.

classinfo descriptive <class_type>

Examples

- Display the descriptive class type name for `/std::mailbox::mailbox_1`

classinfo descriptive /std::mailbox::mailbox_1

which returns:

```
# Class /std::mailbox::mailbox_1 maps to mailbox #(class uvm_phase)
```

Related Topics

[Authoritative and Descriptive Class Type Names](#)

[classinfo descriptive \[ModelSim Command Reference Manual\]](#)

Determining the Current State of a Class Instance

The classinfo find command searches the currently active dataset for the state of the specified Class Instance Identifier; whether it exists, has not yet been created, or has been destroyed. You can specify an alternate dataset for the search and save the results of the search to a text file or to the transcript as a tcl string.

Procedure

Enter the classinfo find command with the desired class instance.

classinfo find <class_instance>

Results

Examples

- Verify the existence of the class instance @mem_item@87

classinfo find @mem_item@87

which returns:

@mem_item@87 exists

or

@mem_item@87 not yet created

or

@mem_item@87 has been destroyed

Related Topics

[classinfo find \[ModelSim Command Reference Manual\]](#)

Finding All Instances of a Class Type

The classinfo instances command reports the list of existing class instances for a specific class type. This can be useful in determining what class instances to log or examine. It can also help in debugging problems resulting from class instances not being cleaned up as they should be and causing run-away memory usage.

Procedure

Enter the classinfo instances command with the desired class type.

classinfo instances <classname>

Examples

- List the currently active instances of the class type *mem_item*.

classinfo instances mem_item

which returns:

```
# @mem_item@140
# @mem_item@139
# @mem_item@138
# @mem_item@80
# @mem_item@76
# @mem_item@72
# @mem_item@68
# @mem_item@64
```

Related Topics

[classinfo instances \[ModelSim Command Reference Manual\]](#)

Reporting Statistics for All Class Instances

The classinfo report command prints detailed statistics about class instances.

The report includes:

- full relative path
- class instance name
- total number of instances of the named class
- maximum number of instances of a named class that existed simultaneously at any time in the simulation
- current number of instances of the named class

The columns may be arranged, sorted, or eliminated using the command arguments.

Procedure

Enter the classinfo report command at the command line.

classinfo report

Examples

- Create a report of all class instances in descending order in the Total column. Print the Class Names, Total, Peak, and Current columns. List only the first six lines of that report.

classinfo report -s dt -c ntpc -m 6

Returns:

# Class Name	Total	Peak	Current
# uvm_pool_11	318	315	315
# uvm_event	286	55	52
# uvm_callback_iter_1	273	3	2
# uvm_queue_3	197	13	10
# uvm_object_string_pool_1	175	60	58
# mem_item	140	25	23

Related Topics

[classinfo report \[ModelSim Command Reference Manual\]](#)

Reporting Class Instance Statistics for a Simulation Run

The classinfo stats command reports statistics about the total number of class types and total, peak, and current class instance counts during the simulation.

Procedure

Enter the classinfo stats command at the command line.

classinfo stats

Examples

- Display the current number of class types, the maximum number, peak number and current number of all class instances.

classinfo stats

Returns:

```
# class type count          451
# class instance count (total) 2070
# class instance count (peak) 1075
# class instance count (current) 1058
```

Related Topics

[classinfo stats \[ModelSim Command Reference Manual\]](#)

Reporting Active References to a Class Instance

The classinfo trace command displays the active references to the specified class instance. This is useful in debugging situations where class instances are not being destroyed as expected because something in the design is still referencing the class instance. Finding those references can lead to uncovering bugs in managing these class references, which can lead to large memory savings.

Procedure

Enter the classinfo trace command with the desired class instance.

```
classinfo trace <class_instance>
```

Examples

- Return the first active reference to @my_report_server@1

```
classinfo trace @my_report_server@1
```

Returns:

```
# top.test.t_env.m_rh.m_srvr
```

Related Topics

[classinfo trace \[ModelSim Command Reference Manual\]](#)

Finding Class Type Inheritance

The classinfo ancestry command shows the inheritance of a specific class type. With some designs and methodologies, class hierarchy can become quite deep. This command shows all of the super classes of a class type, back to its base class.

Procedure

Enter the classinfo ancestry command with the desired class type.

```
classinfo ancestry <class_type>
```

Examples

- Return the inheritance for *mem_item*.

```
classinfo ancestry mem_item
```

Returns:

```
# class /mem_agent_pkg::mem_item extends /
uvm_pkg::uvm_sequence_item
#   class /uvm_pkg::uvm_sequence_item extends /
uvm_pkg::uvm_transaction
#     class /uvm_pkg::uvm_transaction extends /uvm_pkg::uvm_object
#       class /uvm_pkg::uvm_object extends /uvm_pkg::uvm_void
#         class /uvm_pkg::uvm_void
```

Related Topics

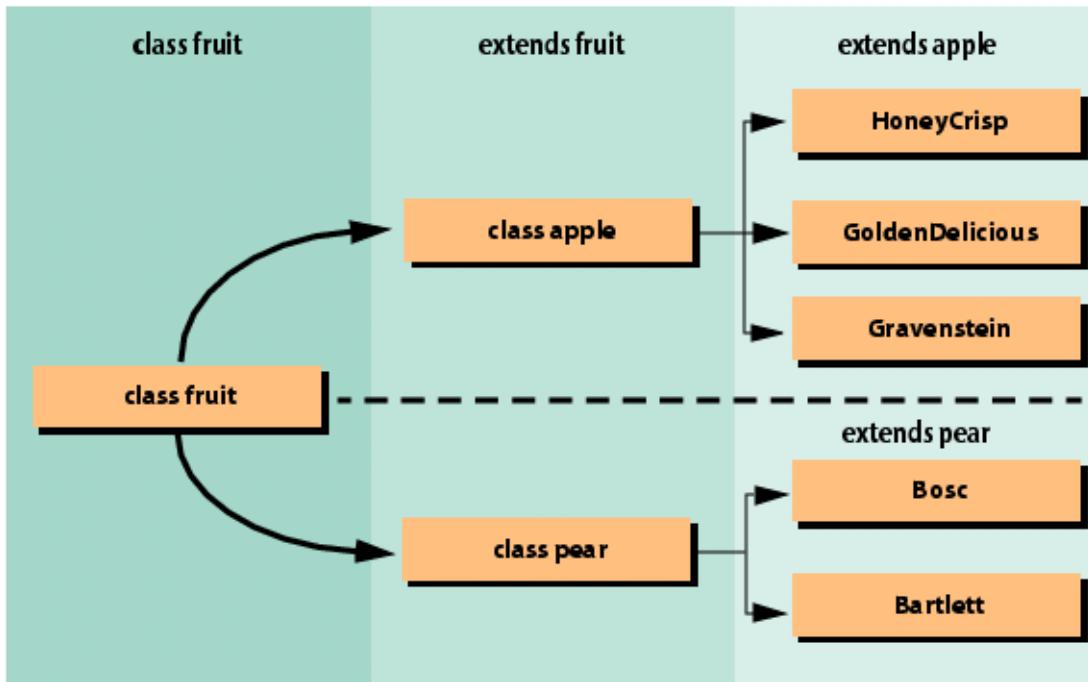
[classinfo ancestry \[ModelSim Command Reference Manual\]](#)

Listing Classes Derived or Extended From a Class Type

The classinfo command lists the classes derived from the specified class type. When one class (X) extends another class (Y), class X inherits the characteristics of class Y. Class X, therefore, is a class Y. Class X is also a class X, of course. Class Y, however, is not a class X.

Consider a simple example of a class called Fruit (Figure 5-15). Class Apple extends Fruit, and class Pear extends Fruit. Further, classes HoneyCrisp, GoldenDelicious, and Gravenstein extend Apple. The classes Bosc and Bartlett extend Pear.

Figure 5-15. Extensions for a Class Type



Asking the question [classinfo isa Apple] would return Apple, HoneyCrisp, GoldenDelicious, and Gravenstein. Asking [classinfo isa Pear] would return Pear, Bosc, and Bartlett. And finally, [classinfo isa Fruit] would return Fruit, Apple, Pear, HoneyCrisp, GoldenDelicious, Gravenstein, Bosc, and Bartlett. This command could be useful for determining all the types extended from a particular methodology sequencer, for example.

Examples

- Find all extensions for the class type *mem_item*.

```
classinfo isa mem_item
```

Returns:

```
# /mem_agent_pkg::mem_item
# /mem_agent_pkg::mem_item_latency4_change_c
# /mem_agent_pkg::mem_item_latency2_change_c
# /mem_agent_pkg::mem_item_latency6_change_c
# /mem_agent_pkg::mem_item_latency_random_c
```

Analyzing Class Types

The classinfo types command searches for and analyses class types by matching a regular expression. Returns the inheritance hierarchy for classes, class extensions, and determines the full path of class types.

Procedure

Enter the classinfo types command with the desired class type.

```
classinfo types <class_type>
```

Examples

- List the full path of the class types that do not match the pattern *uvm*. The scope and instance name returned are in the format required for logging classes and for setting some types of breakpoints,

```
classinfo types -x *uvm*
```

Returns:

```
# /environment_pkg::test_predictor
# /environment_pkg::threaded_scoreboard
# /mem_agent_pkg::mem_agent
# /mem_agent_pkg::mem_config
# /mem_agent_pkg::mem_driver
```

Related Topics

[classinfo types \[ModelSim Command Reference Manual\]](#)

Class Instance Garbage Collection

As your simulation run progresses, it creates and destroys class instances and stores the data in memory. Though a class instance ceases to be referenced, the data for that instance is retained in memory. The garbage collector (GC) deletes all un-referenced class objects from memory.

Default Garbage Collector Settings	241
Changing the Garbage Collector Configuration.....	242
Running the Garbage Collector	243

Default Garbage Collector Settings

Automatic execution of the garbage collector is dependent upon how your design is simulated.

Table 5-12. Garbage Collector Modes

Mode	Modelsim.ini Variable	vsim argument
Class debug disabled	ClassDebug = 0	vsim -noclassdebug (default)
Class debug enabled	ClassDebug = 1	vsim -classdebug

The default settings for execution of the garbage collector are optimized to balance performance and memory usage for either mode. The garbage collector executes when one of the following events occurs, depending on the mode:

- After the total of all class objects in memory reaches a specified size in Megabytes.
- At the end of each run command.
- After each step operation.

GC Settings in Class Debug Disabled Mode

- Memory threshold = 100 megabytes
- At the end of each run command: Off
- At the end of each step command: Off

GC Settings in Class Debug Enabled Mode

- Memory threshold = 5 megabytes
- At the end of each run command: On
- At the end of each step command: Off

Changing the Garbage Collector Configuration

You can change the default garbage collector settings for the current simulation in the Garbage Collector Configuration dialog box, on the command line, via modelsim.ini variables, or with vsim command arguments.

Refer to the following table for garbage collector commands, modelsim.ini variables and vsim command arguments:

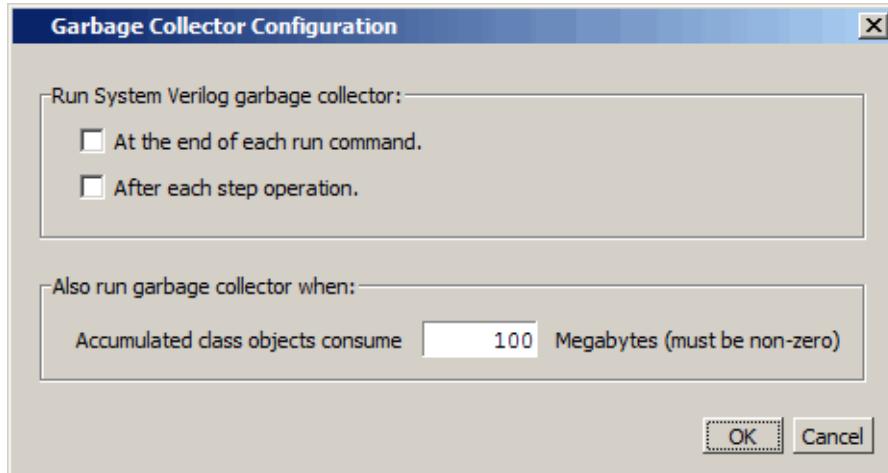
Table 5-13. CLI Garbage Collector Commands and INI Variables

Action	Commands	INI Variable	INI Variable
Set Memory Threshold	gc configure -threshold <value>	“GCThreshold” on page 605 or “GCThresholdClassDebug” on page 606	vsim -gcthreshold <value>
Execute after each run command	gc configure -onrun 0 1		vsim -gconrun/-nogconrun
Execute after each step command	gc configure -onstep 0 1		vsim -gconstep/-nogconstep

Procedure

1. To open the Garbage Collector Configuration dialog, choose **Tools > Garbage Collector > Configure** from the main menu to open the dialog box (Figure 5-16).

Figure 5-16. Garbage Collector Configuration



2. The default settings are loaded automatically and set based on whether you have specified the -classdebug or the -noclassdebug argument with the **vsim** command.

Related Topics

[gc configure \[ModelSim Command Reference Manual\]](#)

[GCThreshold](#)

[GCThresholdClassDebug](#)

[vsim \[ModelSim Command Reference Manual\]](#)

Running the Garbage Collector

You can run the garbage collector at any time.

Procedure

Enter `gc run` at the command line.

Autofindloop and the Autofindloop Report

When a simulation enters a zero delay loop, and the iteration count exceeds the iteration limit setting, autofindloop (`vsim -autofindloop`) attempts to detect the loop and to create a report describing it. When possible, the report identifies the active processes and event and signal activity.

Design content, optimization, and design visibility can have an impact on the content and detail of the reported results. For behavioral designs, the best practice is to preserve line numbers with `+acc=l`. For gate-level designs, having net visibility can provide additional information allowing primitive process drivers to be identified.

The autofindloop feature is implemented in the simulation kernel. It has no overhead associated with it until it reaches the iteration limit. Exceeding the limit causes a process trace to be performed until either the loop is identified, the simulation escapes the execution loop, or a findloop step limit is exceeded. You can modify the default findloop step limit of 500 by setting the `MTI_FINDLOOP_STEP_LIMIT` environment variable before invoking `vsim`.

The autofindloop report includes the following information:

- Active processes – Processes activated for evaluation.
- Wakeup Events – Events scheduled with a distributed delay.
- Wakeup Event Master Processes – Active processes that are members of a wakeup activation block.
- Signal activity:
 - Driving nets – Nets that are driven by an associated process.
 - Signal – Nets that are activated due to a value change during loop execution.

Wakeup events that appear in the zero delay loop are zero delay events. Any non-zero delay event that appears in the loop output is denoted as special, and terminates the loop analysis. This action indicates that the simulation iteration limit is not large enough for the design.

The report uses indentation to associate information for display with the active processes. An indented block can include the source file, the line number, and the driving nets. The report also uses indentation to indicate the importance of loop activity; events and processes appear leftmost in the output, and signal activity is indented further to denote reduced importance.

The autofindloop feature has some limitations, including:

- Vopt optimizations can affect the quality of the autofindloop output.
- There is no causality tracing to refine the reported loop activity.
- autofindloop does not require or use a generated QFD debug database.

Examples

The following design has a zero delay loop which is caused by `Tvco_4` having an initial value of 0:

```
1 `timescale 1 ps / 1 ps
2
3 module top;
4
5 wire o;
6 reg some_clk, clk2;
7 real Tvco_4 = 1 / 4;
8
9 initial begin
10 $monitor ($time, " : %b %g", o , Tvco_4);
11 #0 clk2 = 0;
12 #10 some_clk = 0;
13 end
14
15 dut d1(clk2, o );
16 always @ (clk2)
17 #(Tvco_4) some_clk = clk2;
18
19 always @ (some_clk)
20 #(Tvco_4) clk2 = ~clk2;
21
22 endmodule
23
24 module dut (input i, output y);
25
26 mycell u1 (i, y);
27
28 endmodule
29
30 module mycell (input i, output o);
31
32 buf b (o, i);
33
34 endmodule
```

The following report shows the autofindloop output for this design when simulated with full visibility (+acc). Note that this output optimizes the `dut` and `u1` instances into a continuous assignment, and denotes the implicit wire connection to `clk2`. The process associated with

\$monitor is not activated, because of its lower priority and the activation mechanism used by unoptimized system tasks:

```
##### Autofindloop Analysis #####
##### Loop found at time 0 ns #####
# Active process: /top/#ALWAYS#19 @ sub-iteration 0
#   Source: test.v:19
# Wakeup process: /top/#ALWAYS#19 @ sub-iteration 1
#   Source: test.v:19
# Active process: /top/#IMPLICIT-WIRE(clk2)#15 @ sub-iteration 2
#   Source: test.v:15
# Wakeup process: /top/#ALWAYS#16 @ sub-iteration 3
#   Source: test.v:16
# Active process: /top/#ALWAYS#19 @ sub-iteration 4
#   Source: test.v:19
##### END OF LOOP #####
# ** Error (suppressible): (vsim-3601) Iteration limit 10000000 reached at
time 0 ns.
```

The following report shows the autofindloop output for the same design, with optimizations and preserved line numbers (+acc=l). Native code optimizations implement the \$monitor system task in this case. Also, optimizations cause the native code implementation of the dut and u1 instances, and the use of wakeup events arrays for process delays:

```
##### Autofindloop Analysis #####
##### Loop found at time 0 ns #####
# Active process: /top/#ALWAYS#19 @ sub-iteration 0
#   Source: test.v:19
# Wakeup Event Array : @ sub-iteration 1
#   Member process: /top/#ALWAYS#19 @ sub-iteration 1
#   Source: test.v:19
# Active process: /top/#ALWAYS#16 @ sub-iteration 2
#   Source: test.v:16
# Active process: Native Verilog Function @ sub-iteration 2
#   Source: test.v:10
# Wakeup process: Native Verilog Function @ sub-iteration 3
#   Source: test.v:0
# Wakeup Event Array : @ sub-iteration 3
#   Member process: /top/#ALWAYS#16 @ sub-iteration 3
#   Source: test.v:16
# Active process: /top/#ALWAYS#19 @ sub-iteration 4
#   Source: test.v:19
##### END OF LOOP #####
# ** Error (suppressible): (vsim-3601) Iteration limit 10000000 reached at
time 0 ns.
```

The following report shows the autofindloop output for the same design fully optimized (no acc settings).

```
##### Autofindloop Analysis #####
##### Loop found at time 0 ns #####
# Active process: Native Verilog @ sub-iteration 0
#   Source: test.v:0
# Wakeup Event Array : @ sub-iteration 1
#   Member process: Native Verilog @ sub-iteration 1
#     Source: test.v:0
# Active process: Native Verilog @ sub-iteration 2
#   Source: test.v:0
# Active process: Native Verilog Function @ sub-iteration 2
#   Source: test.v:0
# Wakeup process: Native Verilog Function @ sub-iteration 3
#   Source: test.v:0
# Wakeup Event Array : @ sub-iteration 3
#   Member process: Native Verilog @ sub-iteration 3
#     Source: test.v:0
# Active process: Native Verilog @ sub-iteration 4
#   Source: test.v:0
##### END OF LOOP #####
# ** Error (suppressible): (vsim-3601) Iteration limit 10000000 reached at
time 0 ns.
```

The following portion of a report shows the autofindloop output for a gate-level design with net and line visibility (+acc=nl). In the case of gate-level designs, active processes can map to gate primitive instances. Net visibility then provides driving net names and values, and also provides additional signal activity to offer additional context to the loop activity:

```
# Active process: /MBIST_uSRAM_tb/E1/u0/iaddr_0_/ms_imqb @ sub-iteration
2
#   Source: f.v:102
# Driving net: /MBIST_uSRAM_tb/E1/u0/iaddr_0_/mq_b to St1
# Active process: /MBIST_uSRAM_tb/E1/u0/iaddr_0_/ms_imqb @ sub-iteration
3
#   Source: f.v:102
# Driving net: /MBIST_uSRAM_tb/E1/u0/iaddr_0_/mq_b to St1
#   Signal: /MBIST_uSRAM_tb/E1/u0/iaddr_0_/q_b @ sub-iteration 3 at Value
St1 (f.v:17)
#   Signal: /MBIST_uSRAM_tb/E1/u0/iaddr_0_/mq_b @ sub-iteration 3 at
Value St1 (f.v:17)
```

Note

 Line number information for gate primitives currently references their parent instance.

Chapter 6

Mixed-Language Simulation

ModelSim allows you to simulate designs that are written in VHDL, Verilog, and SystemVerilog. While design units must be entirely of one language type, any design unit may instantiate other 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.....	249
Different Compilers with Common Design Libraries	250
The SystemVerilog bind Construct in Mixed-Language Designs	253
Simulator Resolution Limit.....	262
Runtime Modeling Semantics.....	263
Mapping Data Types	264
VHDL Instantiating Verilog or SystemVerilog.....	276
Verilog or SystemVerilog Instantiating VHDL.....	281
Sharing User-Defined Types.....	284

Basic Mixed-Language Flow

Simulating mixed-language designs with ModelSim consists of a few basic steps. The actions you take for each step in the flow depend on which languages your design units are written in and where they are in the design hierarchy.

1. Compile HDL source code using either the [vcom](#) or [vlog command](#). Compile all modules in the design following order-of-compile rules.
2. Simulate the design with the [vsim](#) command.
3. Run and debug your design.

Different Compilers with Common Design Libraries

Depending on the language of your design units, you need to use the appropriate compilation command before running a simulation on the mixed-language design.

- VHDL source code is compiled by using the vcom command. ModelSim stores the resulting compiled design units (entities, architectures, configurations, and packages) in the working library.
- Verilog/SystemVerilog source code is compiled by using the vlog command. ModelSim stores the resulting design units (modules and UDPs) in 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). Refer to [Design Libraries](#) for more information about library management.

Case Sensitivity	250
Hierarchical References	251

Case Sensitivity

Note that VHDL and Verilog observe different rules for case sensitivity.

- VHDL is not case-sensitive. For example, clk and CLK are regarded as the same name for the same signal or variable.
- Verilog (and SystemVerilog) are case-sensitive. For example, clk 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.

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 they 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, as follows:

```
<<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 DOWNTO 0)>>;
ALIAS test_pin IS <<SIGNAL .tb.dut.i0.tp : std_logic_vector(3 DOWNTO 0)>>;
...
test_pin(3) <= '1';
Q(0) <= test_pin(0);
```

To use this capability, use the vcom command to 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>>
```

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 that any restrictions on VHDL external names apply to the hierarchical references in the bind statement (that is, the target must be a VHDL object.) Because binds are done after all other instances within a scope, there should be no ordering issues.

The 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. This is especially useful for binding SystemVerilog assertions to your, VHDL, Verilog and mixed-language 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, the following basic steps show how to bind a SystemVerilog assertion module to a VHDL design:

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.

Syntax of bind Statement	253
Allowed Bindings	253
Hierarchical References to a VHDL Object from a Verilog/SystemVerilog Scope	254
Mapping of Types	256
Port Mapping with VHDL and Verilog Enumerated Types	256
VHDL Instance Mapping	258

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>
```

Allowed Bindings

The following list provides examples of bindings you can make.

- Bind to all instances of a VHDL entity.

```
bind e bind_du inst(p1, p2);
```

- Bind to all instances of a VHDL entity and 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);
```

Note

 For ModelSim DE and ModelSim PE, the SystemVerilog bind construct supports only simple names as actual expressions. Therefore, you cannot use bit-selects, part-selects, concatenations, or any expression (apart from simple names) as your actual expression. Also, you cannot use hierarchical references as actual expressions when the target of bind is VHDL, however, this does work when the target of bind is Verilog.

Actual expressions 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.

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. VHDL functions, procedures, and types are not supported, and you cannot read VHDL process variables.

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 anywhere 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 is 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.

Signal Spy 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. Refer to the [Signal Spy](#) chapter for more information on the use of Signal Spy.

Mapping of Types

All SystemVerilog data types supported at the SystemVerilog-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 SystemVerilog-VHDL mixed-language boundary for direct instantiation.

Related Topics

[Mapping Data Types](#)

Port Mapping with VHDL and Verilog 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, you can use the bind construct for port mapping of VHDL enumerated types to Verilog port vectors.

Port mapping is supported for both input and output ports. ModelSim first converts the integer value of the enum to a bit vector before connecting to a Verilog formal port. Note that you cannot connect an 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.

Table 6-1 shows the allowed VHDL types for port mapping to SystemVerilog port vectors.

Table 6-1. VHDL Types Mapped To SystemVerilog Port Vectors

bit	std_logic	vl_logic
bit_vector	std_logic_vector	vl_logic_vector

Example of Binding to VHDL Enumerated Types

Consider an example of using SystemVerilog assertions to monitor a VHDL finite state machine that uses enumerated types. With ModelSim, you can use the bind statement to map VHDL enumerated types directly to SystemVerilog enumerated types.

The following steps show how to follow the same type-sharing rules, which are applicable for direct instantiations at the SystemVerilog-VHDL mixed-language boundary (refer to [Sharing User-Defined Types](#)).

Example 6-1. SystemVerilog Assertions Monitor a VHDL Finite State Machine

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;
```

The following procedure shows how to use this at the mixed-language boundary of SystemVerilog and VHDL.

1. Compile this package using the `-mixedsvvh` argument for the `vcom` command:

```
vcom -mixedsvvh pack.vhd
```

2. Make the package available to the design in either of the following ways:

- o 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;
...
```

- o Import this package to 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
...
```

3. Assume 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. To do this, define the following covergroup in SystemVerilog:

```
...
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;
...
```

4. As with monitoring VHDL components, you create a wrapper containing the bind statement to connect the SystemVerilog Assertions 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
```

5. Use either of the following to perform the actual binding in ModelSim:

- instantiation
- loading of multiple top modules into the simulator with the vsim command:

```
vsim interleaver_tester interleaver_binds
```

Related Topics

[Sharing User-Defined Types](#)

VHDL Instance Mapping

You can use the SystemVerilog bind statement to tie an assertion to VHDL design units that are defined by generate or configuration statements.

[Example 6-2](#) shows how to use the bind statement in a SystemVerilog wrapper module to connect a SystemVerilog cover directive to a VHDL component.

Example 6-2. Using the Bind Statement with VHDL Component and SystemVerilog Assertion

Consider the following VHDL code that uses nested generate statements:

```
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 (SVA) uses a cover directive to define 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, named SVA, to the specific instance defined by the generate and configuration statements.

Tip

-  You can control the format of generate statement labels by using the [GenerateFormat](#) variable in the modelsim.ini file.
-

Separate 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.

Note that when the bind statement is in the compilation unit scope, the bind becomes effective only 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. As a result, 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 the [vlog](#) command if bind statements are found in the compilation unit scope.

The -cuname argument for vlog gives a user-defined name to a specified compilation \$unit package (which, in the absence of -cuname, is some internally generated name). You must provide this named compilation unit package as the top-level design unit with the vsim command in order to force elaboration.

Tip

-  If you are using the [vlog -R](#) commands to compile and simulate the design, ModelSim handles this binding issue automatically.
-

The vlog -cuname argument is used only in conjunction with the vlog -mfcu argument, which instructs the compiler to treat all files within a compilation command line as a single compilation unit.

[Example 6-3](#) shows how to use vlog -cuname and -mfcu arguments to elaborate a bind statement contained in its own file.

Example 6-3. Using vlog -cuname and -mfcu Arguments to Ensure Proper Elaboration

Consider the following 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
```

Next, bind that assertion module to the following 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
```

using the **bind** statement contained separately 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 module 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 you compile the *bind.sv* file by itself, such as with vlog bind.sv, you will receive a Warning like this one:

```
** Warning: (vlog-2650) 'bind' found in a compilation unit scope. Please
use -cuname to ensure that 'bind' gets elaborated.
```

To fix this problem, use the -cuname argument with the vlog command, as follows:

```
vlog -cuname bind_pkg -mfcu bind.sv
```

Then enter the following command to simulate the design:

```
vsim testbench bind_pkg
```

Simulator Resolution Limit

In a mixed-language design with only one top-level design unit, the resolution for simulation time 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 (refer to [Simulator Resolution Limit \(Verilog\)](#) for details).

In the case of a mixed-language design with multiple tops, the following algorithm is used:

- If VHDL modules are present, then the Verilog resolution is ignored. An error is issued if the Verilog resolution is finer than the chosen one.
- If VHDL modules are present, then the Verilog resolution is ignored. An error is issued if the Verilog resolution is finer than the chosen one.
- 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 for each language of a mixed-language design.

To achieve this compliance, the sequence of operations in one simulation iteration (that is, delta cycle) is as follows:

1. Signal updates are made
2. HDL processes are run

The above scheduling semantics are required to satisfy the HDL LRM. All processes triggered by an event on an HDL signal shall wake up at the end of the current delta.

Hierarchical References to SystemVerilog 263

Hierarchical References to SystemVerilog

Hierarchical references to SystemVerilog properties and sequences are supported with the following restrictions.

- Clock and disable iff expressions cannot have a formal.
- Method 'matched' is not supported on a hierarchically referenced sequence

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.

A VHDL instantiation of Verilog may associate VHDL signals and values with Verilog ports and parameters. Likewise, a Verilog instantiation of VHDL may associate Verilog nets and values with VHDL ports and generics.

The following sections describe data type mappings for mixed-language designs in ModelSim:

Verilog and SystemVerilog to VHDL Mappings	264
VHDL To Verilog and SystemVerilog Mappings.....	268

Verilog and SystemVerilog to VHDL Mappings

Verilog or SystemVerilog instantiations of VHDL may associate Verilog nets and values with VHDL ports and generics.

Table 6-2 shows the mapping of data types from SystemVerilog to VHDL.

Table 6-2. SystemVerilog-to-VHDL Data Type Mapping

SystemVerilog Type	VHDL Type		Comments
	Primary mapping	Secondary mapping	
bit	bit	std_logic	2-state scalar data type
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

Table 6-2. SystemVerilog-to-VHDL Data Type Mapping (cont.)

SystemVerilog Type	VHDL Type		Comments
	Primary mapping	Secondary mapping	
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
byte unsigned	integer		2-state data type, 8-bit unsigned integer or ASCII character
enum	enum		SystemVerilog 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 6-3. 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_ulegic
natural	std_logic_vector	vl_ulegic_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 6-4. Verilog States Mapped to std_logic and bit

Verilog	std_logic	bit
HiZ	'Z'	'0'
Sm0	'L'	'0'
Sm1	'H'	'1'
SmX	'W'	'0'
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

VHDL instantiations of Verilog or SystemVerilog may associate VHDL ports and generics with Verilog nets and values.

Table 6-5 summarizes the mapping of data types from VHDL to SystemVerilog.

Table 6-5. 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-dimensional arrays	multi-dimensional arrays		multi-dimensional arrays of supported types

Mapping VHDL Generics to Verilog Types

Table 6-6 shows the mapping of VHDL Generics to Verilog types.

Table 6-6. 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 6-9:

Table 6-7. 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 6-8:

Table 6-8. 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

Table 6-8. Mapping VHDL std_logic Type to Verilog States (cont.)

VHDL std_logic	Verilog State
'L'	Pu0
'H'	Pu1
'_'	StX

VHDL Generics at a VHDL-SystemVerilog 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-SystemVerilog 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 6-9. 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 6-10](#) 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 6-10. 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)

Table 6-10. Mapping Table for SystemVerilog-style Declarations (cont.)

Type of Verilog Formal	Type of VHDL Actual
real, shortreal	real integer
string	string

In addition to the mapping in [Table 6-10](#), 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 SystemVerilog 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 6-10](#) for a list of allowed mappings).

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	276
Component Declaration for VHDL Instantiating Verilog	276
vgencomp Component Declaration when VHDL Instantiates Verilog.....	277
Modules with Bidirectional Pass Switches.....	278
Modules with Unnamed Ports	279

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, enum, and struct. (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

The vgencomp command generates a component declaration according to the following rules.

- Generic Clause

The **vgencomp** command generates a generic clause 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

The `vgencomp` command generates a port clause 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	281
Entity and Architecture Names and Escaped Identifiers.....	282
Named Port Associations.....	283
Generic Associations	283

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.

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.

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

Related Topics

[Library Usage](#)

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.

Sharing User-Defined Types

You can use shared VHDL packages for both VHDL and Verilog/SystemVerilog. Each usage takes a different VHDL construct.

Using a Common VHDL Package **284**

Using a Common SystemVerilog Package..... **287**

Using a Common VHDL Package

With the “import” construct of SystemVerilog, you can implement user-defined types (records, enums, aliases, subtypes, types, and multi-dimensional arrays) and constants from VHDL in a SystemVerilog design. (Any other user-defined types are 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
 - Unconstrained arrays.
 - Subtypes of all supported types
 - Alias of records, enums and arrays only
 - Types (static ranges only)
-

ModelSim supports VHDL constants of all types currently supported at the VHDL-SystemVerilog mixed language boundary as shown in [Table 6-5](#).

Deferred constants are not supported. Only static expressions are supported as constant values.

[Table 6-11](#) shows the mapping of literals from VHDL to SystemVerilog.

Table 6-11. 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 6-12](#) lists all supported types inside VHDL Records.

Table 6-12. 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>
```

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
```

Import this package into 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. (Any other user-defined types are 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

 You must use the vcom -mixedsvvh option when compiling the common package, and 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
-

[Table 6-13](#) lists all supported types inside SystemVerilog structures.

Table 6-13. Supported Types Inside SystemVerilog Structure

SystemVerilog Type	VHDL Type	Comments
bit	bit	bit types
logic, reg	std_logic	multi-valued types

Table 6-13. Supported Types Inside SystemVerilog Structure (cont.)

SystemVerilog Type	VHDL Type	Comments
enum	enum	SystemVerilog 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 SystemVerilog-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>

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 SystemVerilog-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
```


Chapter 7

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*.

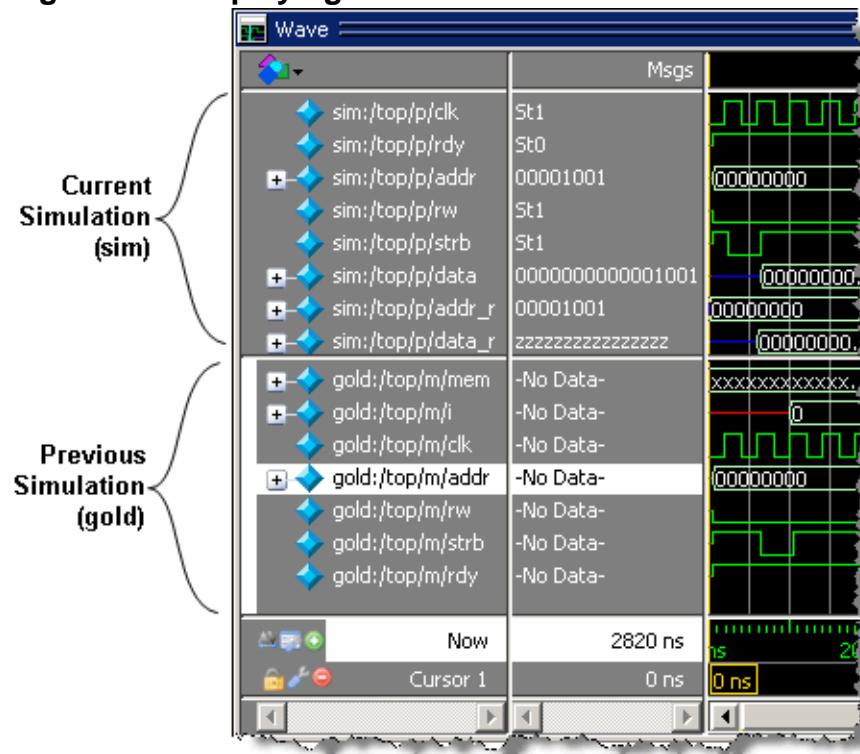
One common example of a dataset is a wave log format (WLF) file. In particular, you can save any ModelSim simulation to a wave log format (WLF) file for future viewing or comparison to a current simulation. You can also view a wave log format file during the currently running simulation.

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 7-1](#) 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 7-1. 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	293
Dataset Structure.....	299
Managing Multiple Datasets.....	301
Collapsing Time and Delta Steps	303
Virtual Objects.....	305

Saving a Simulation to a WLF File

If you add objects to the debugging windows in the graphic interface, 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 disable the creation of a WLF file with the **vsim +questa_mvc_core+wlf_disable** command.

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.

Also, datasets can be saved at intervals, each with unique filenames, with the **dataset snapshot** command. See “[Saving at Intervals with Dataset Snapshot](#)” for GUI instructions.

Note

 If you do not use either the **dataset save** or **dataset snapshot** command, 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.

Saving at Intervals with Dataset Snapshot	293
Saving Memories to the WLF.....	295
WLF File Parameter Overview	295
Limiting the WLF File Size.....	297
Opening Datasets	298

Saving at Intervals with Dataset Snapshot

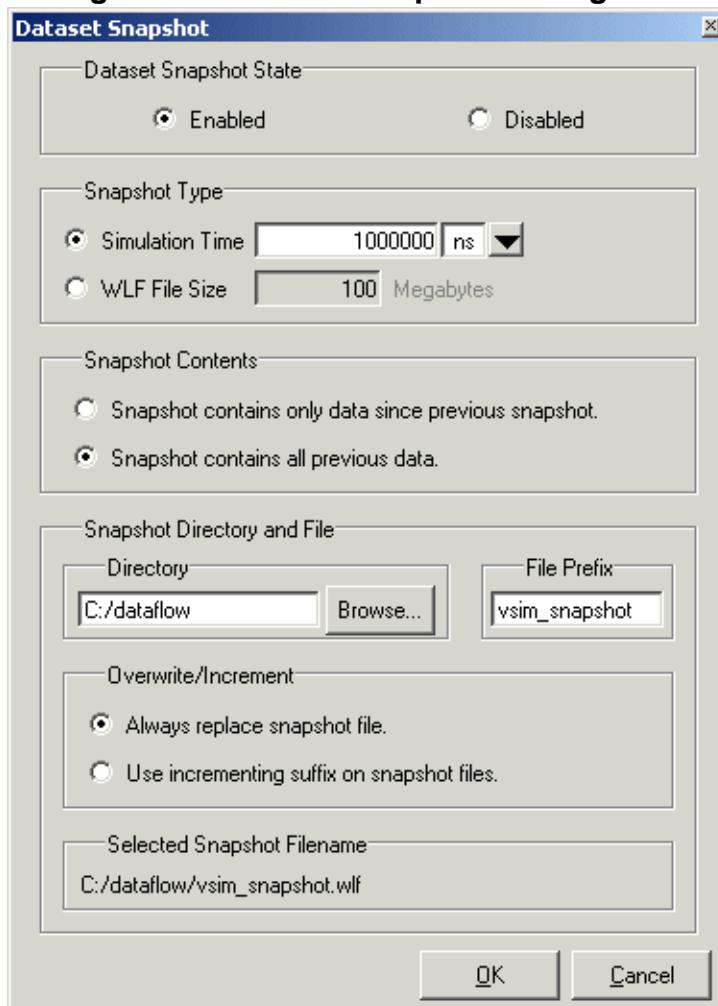
Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic “snapshots” of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Procedure

1. Log objects of interest with the **log** command.
2. Select the Wave window to make it active.

3. Select **Tools > Dataset Snapshot** to open the Dataset Snapshot dialog box ([Figure 7-2](#)).
4. Select **Enabled** for the Dataset Snapshot State.
5. Set the simulation time or the wlf file size.
6. Choose whether the snapshot will contain only data since previous snapshot or all previous data.
7. Designate the snapshot directory and file.
8. Choose whether to replace the existing snapshot file or use an incrementing suffix if a file by the same name exists.
9. Click the OK button to create the dataset snapshot.

Figure 7-2. Dataset Snapshot Dialog Box



You can customize the datasets either to contain all previous data, or only the data since the previous snapshot. You can also set the dataset to overwrite previous snapshot files, or increment the names of the files with a suffix.

Saving Memories to the WLF

By default, memories are not saved in the WLF file when you issue a “log -r /*” command.

Procedure

1. To get memories into the WLF file you will need to explicitly log them. For example:

log /top/dut/I0/mem

2. If you want to use wildcards, then you will need to remove memories from the WildcardFilter list. To see what is currently in the WildcardFilter list, use the following command:

set WildcardFilter

If “Memories” is in the list, reissue the set WildcardFilter command with all items in the list except “Memories.” For details, refer to [Using the WildcardFilter Preference Variable](#) in the Command Reference Manual.

Note

 For post-process debug, you can add the memories into the Wave or List windows but the Memory List window is not available.

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 7-1. WLF File Parameters

Feature	modelsim.ini	modelsim.ini Default	vsim argument
WLF Cache Size	WLFCacheSize = <n>	0 (no reader cache)	
WLF Collapse Mode	WLFCollapseModel = 0 1 2	1 (-wlfcollapsedelta)	-nowlfcollapse - wlfcollapsedelta - wlfcollapsetime
WLF Compression	WLFCompress = 0 1	1 (-wlfcollapse)	-wlfcollapse - nowlfcollapse
WLF Delete on Quit	WLFDeleteOnQuit = 0 1	0 (-wlfdelteonquit)	-wlfdelteonquit - nowlfdelteonquit
WLF File Lock	WLFFileLock = 0 1	0 (-nowlflock)	-wlflock -nowlflock
WLF File Name	WLFFilename=<filename>	<i>vsim.wlf</i>	-wlf <filename>

Table 7-1. WLF File Parameters (cont.)

Feature	modelsim.ini	modelsim.ini Default	vsim argument
WLF Index	WLFIIndex 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	-wlftlim <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

You can easily limit the WLF file size by setting a simulation control variable or with a **vsim** command switch.

Limit the WLF file size 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.

Opening Datasets

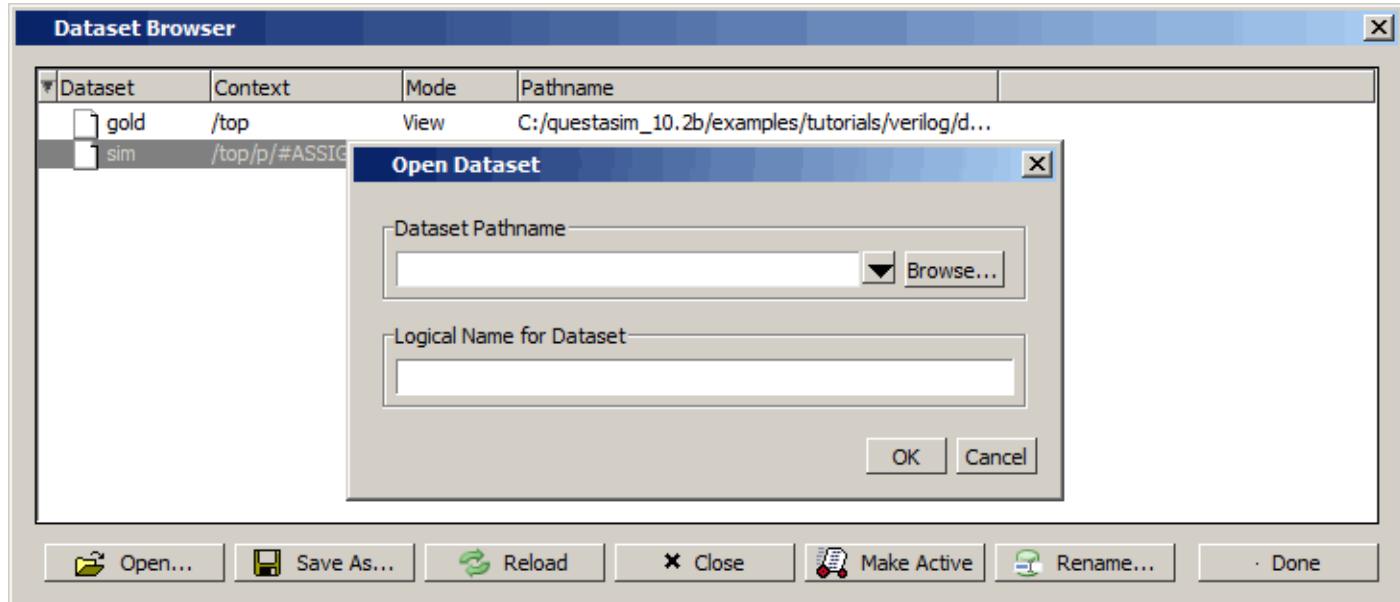
ModelSim allows you to open existing datasets.

Procedure

To open a dataset, do one of the following:

- Select **File > Open** to open the Open File dialog box 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 box ([Figure 7-3](#)).

Figure 7-3. 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 box includes the following options:

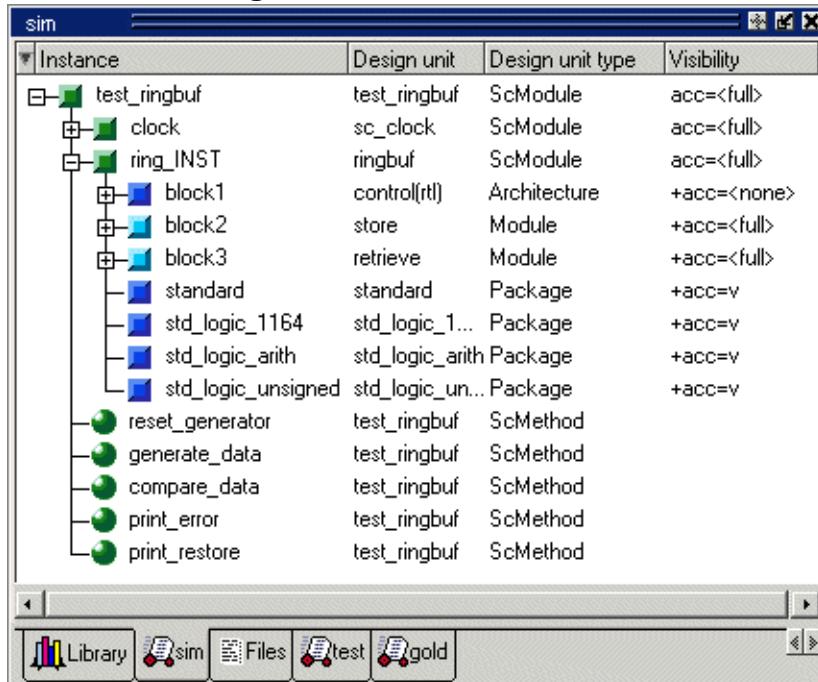
- **Dataset Pathname** — Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset** — This is the name by which the dataset will be referred. By default this is the name of the WLF file.

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 7-4. 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 Window Columns [299](#)

Structure Window Columns

Structural information about datasets is presented in the Structure window.

[Table 7-2](#) lists the columns displayed in each Structure window, by default.

Table 7-2. Structure Tab Columns

Column name	Description
Instance	the name of the instance
Design unit	the name of the design unit

Table 7-2. Structure Tab Columns (cont.)

Column name	Description
Design unit type	the type (for example, Module, Entity, and so forth) of the design unit

You can hide or show columns by right-clicking a column name and selecting the name on the list.

Managing Multiple Datasets

ModelSim allows you to manage multiple datasets using menu selections from the graphic interface or from the command line.

Managing Multiple Datasets in the GUI	301
Managing Multiple Datasets from the Command Line	301
Restricting the Dataset Prefix Display.....	303

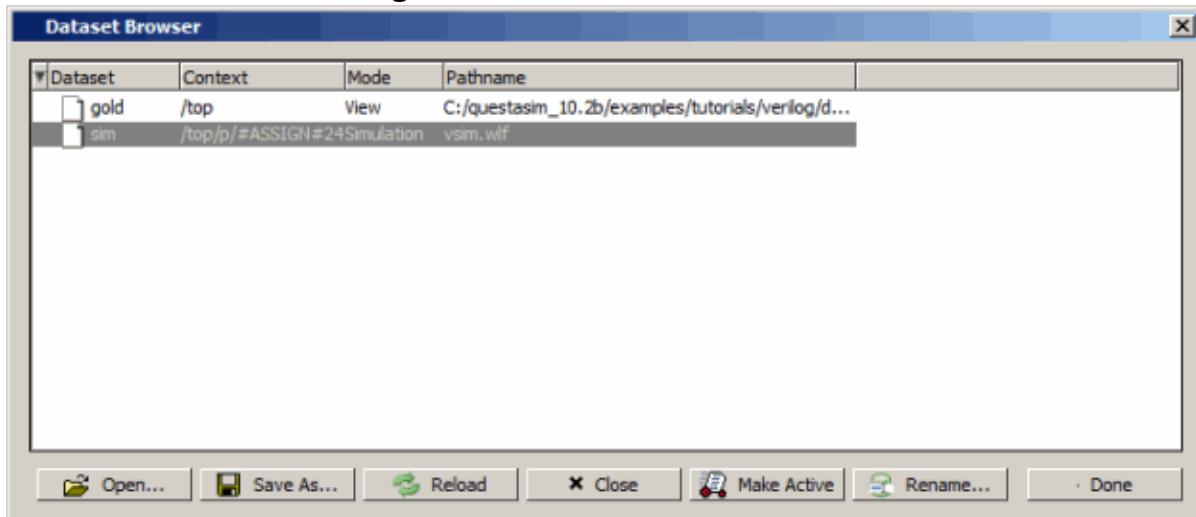
Managing Multiple Datasets in the GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**.

Procedure

1. Open the Dataset Browser by selecting **File > Datasets**.

Figure 7-5. The Dataset Browser



2. From the Dataset Browser you can open a selected dataset, save it, reload it, close it, make it the active dataset, or rename it.

Managing Multiple Datasets from the 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.

Procedure

1. 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.

2. 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**
3. 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.

4. 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.

Procedure

1. To change the value of this variable, do the following:
 2. Choose Tools > Edit Preferences... from the main menu.
 3. In the Preferences dialog box, click the By Name tab.
 4. Scroll to find the Preference Item labeled Main and click [+] to expand the listing of preference variables.
 5. Select the `DisplayDatasetPrefix` variable then click the Change Value... button.
 6. 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)
 7. Click OK; click OK.
8. 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.

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 7-3. vsim Arguments for Collapsing Time and Delta Steps

vsim argument	effect	modelsim.ini setting
<code>-nowlfcollapse</code>	All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.	<code>WLFCollapseMode = 0</code>

Table 7-3. vsim Arguments for Collapsing Time and Delta Steps (cont.)

vsim argument	effect	modelsim.ini setting
-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
-wlfcollapsetime	Same as delta collapsing but at the timestep granularity.	WLFCollapseMode = 2

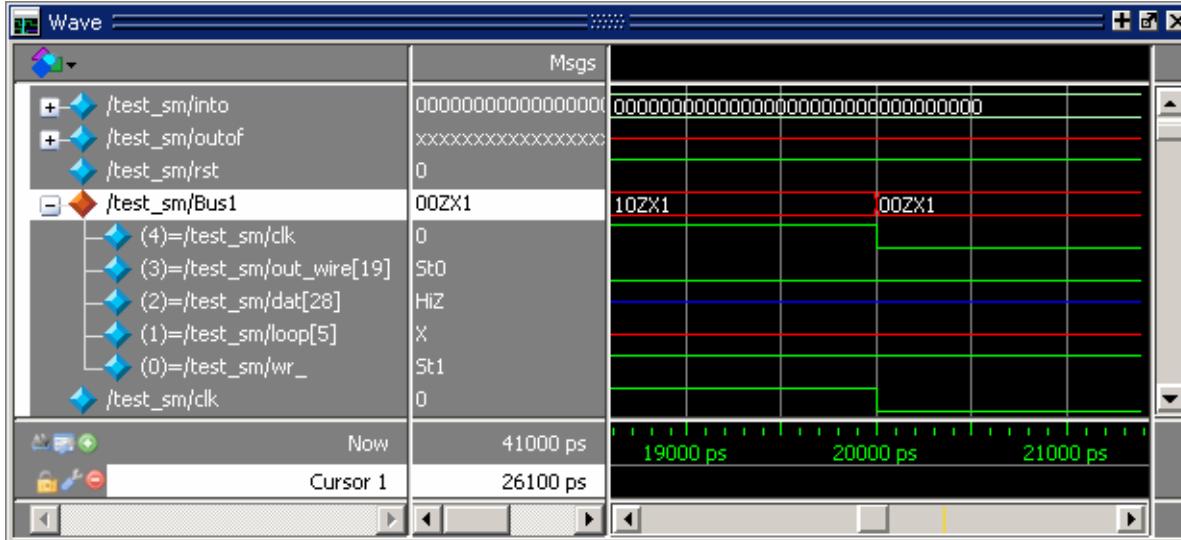
When a run completes that includes single stepping or hitting a breakpoint, all events are flushed to the WLF file regardless of the time collapse mode. It's possible that single stepping through part of a simulation may yield a slightly different WLF file than just running over that piece of code. If 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.

Virtual objects are indicated by an orange diamond as illustrated by Bus1 in [Figure 7-6](#):

Figure 7-6. Virtual Objects Indicated by Orange Diamond



Virtual Signals	305
Virtual Functions	306
Virtual Regions	307
Virtual Types	307

Virtual Signals

Virtual signals are aliases for combinations or subelements of signals written to the WLF file by the simulation kernel. They can be displayed in the Objects, List, Watch, and Wave windows, accessed by the examine command, and set using the force command.

You can create virtual signals using the **Wave or List > Combine Signals** menu selections or by using the [virtual signal](#) command. Once created, virtual signals can be dragged and dropped from the Objects pane to the Wave, Watch, and List windows. In addition, you can create virtual signals for the Wave window using the Virtual Signal Builder (refer to [Using the Virtual Signal Builder](#)).

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The virtual signal command has an -install <region> option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in order to reconstruct the

original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The [virtual hide](#) command can be used to hide the display of the broken-down bits if you do not want them cluttering up the Objects window.

If the virtual signal has elements from more than one WLF file, it will be automatically installed in the virtual region *virtuals:/Signals*.

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the scalar elements of the first two virtual signals.

The definitions of virtuals can be saved to a DO 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.

Virtual functions 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 8

Waveform Analysis

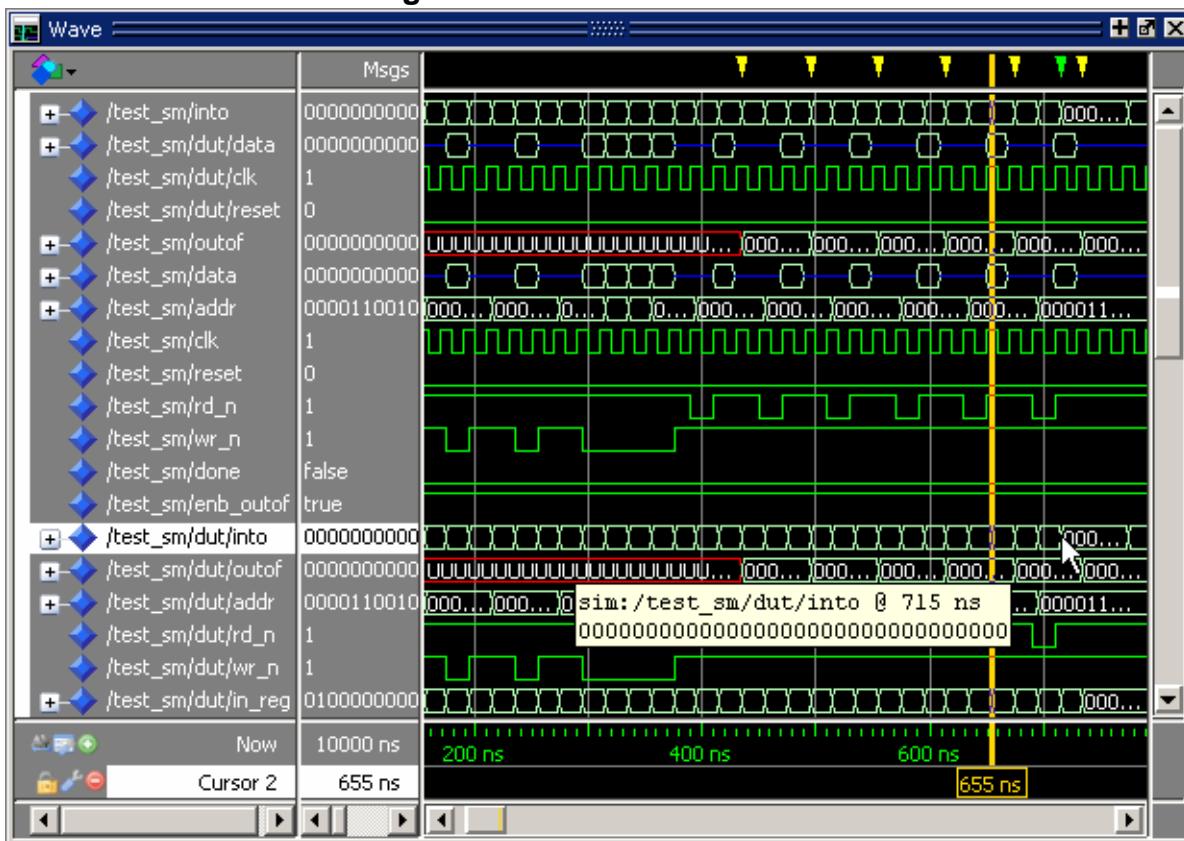
The Wave window is the most commonly used tool for analyzing and debugging your design after simulation. It displays all signals in your design as waveforms and signal values and provides a suite of graphical tools for debugging.

Wave Window Overview	310
Objects You Can View	311
Adding Objects to the Wave Window	312
Inserting Signals in a Specific Location	313
Working with Cursors	315
Adding Cursors	317
Editing Cursor Properties	317
Jump to a Signal Transition	318
Measuring Time with Cursors in the Wave Window	318
Syncing All Active Cursors	319
Linking Cursors	319
Understanding Cursor Behavior	320
Shortcuts for Working with Cursors.....	321
Two Cursor Mode.....	322
Expanded Time in the Wave Window	323
Expanded Time Terminology	323
Recording Expanded Time Information	324
Viewing Expanded Time Information in the Wave Window	325
Customizing the Expanded Time Wave Window Display	327
Expanded Time Display Modes	329
Switching Between Time Modes	330
Expanding and Collapsing Simulation Time	330
Expanded Time with examine and Other Commands	331
Zooming the Wave Window Display	332
Zooming with the Menu, Toolbar and Mouse	332
Saving Zoom Range and Scroll Position with Bookmarks	333
Editing Bookmarks	334
Searching in the Wave Window	335
Searching for Values or Transitions	335
Search with the Expression Builder	337
Filtering the Wave Window Display	340
Formatting the Wave Window	341
Setting Wave Window Display Preferences	342

Formatting Objects in the Wave Window	345
Dividing the Wave Window	348
Splitting Wave Window Panes	349
Wave Groups	350
Creating a Wave Group	351
Deleting or Ungrouping a Wave Group	354
Adding Items to an Existing Wave Group	354
Removing Items from an Existing Wave Group	354
Miscellaneous Wave Group Features	355
Composite Signals or Buses	356
Creating Composite Signals through Menu Selection	356
Saving the Window Format	357
Exporting Waveforms from the Wave window	359
Exporting the Wave Window as a Bitmap Image.....	359
Printing the Wave Window to a Postscript File	359
Printing the Wave Window on the Windows Platform	360
Saving Waveform Sections for Later Viewing.....	361
Viewing System Verilog Interfaces	364
Working with Virtual Interfaces	365
Combining Objects into Buses	367
Extracting a Bus Slice.....	367
Wave Extract/Pad Bus Dialog Box.....	368
Splitting a Bus into Several Smaller Buses	369
Using the Virtual Signal Builder	370
Creating a Virtual Signal	371
Miscellaneous Tasks	374
Examining Waveform Values.....	374
Displaying Drivers of the Selected Waveform.....	374
Sorting a Group of Objects in the Wave Window	375
Creating and Managing Breakpoints.....	376
Signal Breakpoints	377
File-Line Breakpoints	380
Saving and Restoring Breakpoints	382

Wave Window Overview

The Wave window opens in the Main window. Like all other windows, it may 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 tabs.

Figure 8-1. The Wave Window

For more information about the graphic features of the Wave window, refer to [Wave Window](#) in the GUI Reference Manual.

Objects You Can View

The list below identifies the types of objects that you can view in the Wave window. Each object type is indicated by its own color-coded shape (such as a diamond or a triangle).

- **VHDL objects** (dark blue diamond) —
signals, aliases, process variables, shared variables
- **Verilog and SystemVerilog objects** (light blue diamond) —
nets, registers, variables, named events, interfaces, classes
- **Virtual objects** (orange diamond) —
virtual signals, buses, functions Refer to [Virtual Objects](#) for more information.

Related Topics

[Using the WildcardFilter Preference Variable](#)

Adding Objects to the Wave Window

You can add objects to the Wave window with mouse actions, menu selections, commands, and with a window format file.

Table 8-1. Add Objects to the Wave Window

To Add Using ...	Do the Following:
Mouse Actions	<ul style="list-style-type: none"> Drag and drop objects into the Wave window from the Structure, Processes, Memory, List, Objects, Source, or Locals windows. When objects are dragged into the Wave window, the add wave command is echoed in the Transcript 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.
Menu Selections	<ul style="list-style-type: none"> Add > window — Add objects to the Wave window or Log file. Add Selected to Window Button — Add objects to the Wave, Dataflow, Schematic, List, or Watch windows. Refer to Add Selected to Window Button in the GUI Reference Manual for more information. <p>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:</p> <ul style="list-style-type: none"> 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. Right-click a Verilog virtual interface waveform and select Add Wave > <interface_name/*> from the popup menu.
Commands	<p>Use the add wave command to add objects from the command line. For example:</p> <pre>VSIM> add wave /proc/a</pre> <p>Adds signal <i>/proc/a</i> to the Wave window.</p> <pre>VSIM> add wave -r /*</pre> <p>Adds all objects in the design to the Wave window.</p> <p>Refer to “Using the WildcardFilter Preference Variable” in the Command Reference Manual for information on controlling the information that is added to the Wave window when using wild cards.</p>
A Window Format File	Select File > Load and specify a previously saved format file. Refer to Saving the Window Format for details on how to create a format file.

Inserting Signals in a Specific Location

New signals are inserted above the Insertion Point Bar located at the bottom of the Pathname Pane. You can change the location of the Insertion Point Bar by using the Insertion Point Column of the Pathname Pane.

Restrictions and Limitations

By default, new signals are added above the Insertion Point Bar. You can change the default location for insertion by setting the **PrefWave(InsertMode)** preference variable to one of the following:

- **insert** — (default) Places new object(s) above the Insertion Pointer Bar.
- **append** — Places new object(s) below the Insertion Pointer Bar.
- **top** — Places new object(s) at the top of the Wave window.
- **end** — Places new object(s) at the bottom of the Wave window.

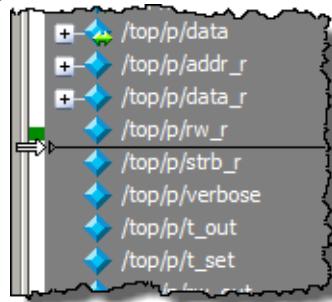
Prerequisites

There must be at least one signal in the Wave window.

Procedure

1. Click the vertical white bar on the left-hand side of the active Wave window to select where signals should be added. (Figure 8-2)
2. Your cursor will change to a double-tail arrow and a green bar will appear. Clicking the vertical white bar next to a signal places the Insertion Point Bar below the indicated signal. Alternatively, you can Ctrl+click the white bar to place the Insertion Point Bar below the indicated signal.

Figure 8-2. Insertion Point Bar



3. Select an instance in the Structure (sim) window or an object in the Objects window.
4. Use the hot key Ctrl+w to add all signals of the instance or the specific object to the Wave window in the location of the Insertion Point Bar.

Related Topics

[Insertion Point Bar and Pathname Pane.](#)

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 settings.

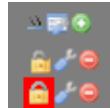


Table 8-2 summarizes common cursor actions you can perform with the icons in the toolbox, or with menu selections.

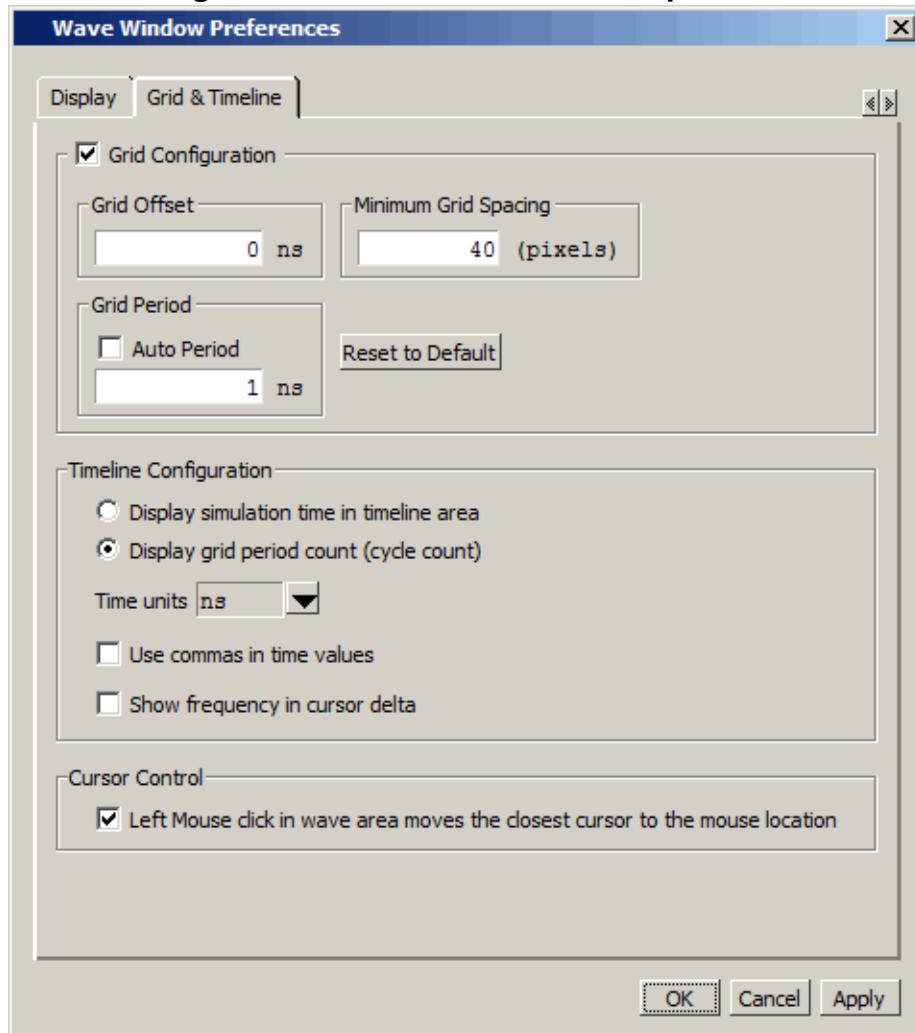
Table 8-2. Actions for Cursors

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 > Window Preferences > Display Tab
	Edit grid and timeline properties	Wave > Wave Preferences > Grid and Timeline Tab	Tools > Window 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
NA	Zoom between Cursors	Debug Toolbar Tab only (refer to the GUI Reference Manual)	Debug Toolbar Tab only.
NA	Two Cursor Mode	Wave > Mouse Mode > Two Cursor Mode	Wave > Mouse Mode > Two Cursor Mode

The **Toggle leaf names <-> full names** icon allows you to switch from displaying full pathnames (the default) to displaying leaf or short names in the Pathnames Pane. 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 box to the Grid & Timeline tab (Figure 8-3).

Figure 8-3. 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 change the 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 (default: ns) does not reflect 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	317
Editing Cursor Properties	317
Jump to a Signal Transition	318
Measuring Time with Cursors in the Wave Window	318
Syncing All Active Cursors	319
Linking Cursors	319
Understanding Cursor Behavior	320
Shortcuts for Working with Cursors	321
Two Cursor Mode	322

Adding Cursors

To add cursors when the Wave window is active you can do one of the following.

Procedure

1. Click the Insert Cursor icon.
2. Choose **Add > To Wave > Cursor** from the menu bar.
3. Press the “A” key while the mouse pointer is located in the cursor pane.
4. Right click in the cursor pane and select **New Cursor @ <time> ns** to place a new cursor at a specific time.

Editing Cursor Properties

After adding a cursor, you can alter its properties by using the Cursor Properties dialog box.

Procedure

1. Right-click the cursor you want to edit and select **Cursor Properties**. (You can also use the **Edit this cursor** icon in the cursor toolbox)

2. From the Cursor Properties dialog box, alter any of the following properties:
 - **Cursor Name** — The name that appears in the Wave window.
 - **Cursor Time** — The time location of the cursor.
 - **Cursor Color** — The color of the cursor.
 - **Locked Cursor Color** — The color of the cursor when it is locked to a specific time location.
 - **Lock cursor to specified time** — Disables relocation of the cursor.

Jump 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 located in the Debug Toolbar Tab. Refer to the following table.

Table 8-3. Find Previous and Next Transition Icons

	Find Previous Transition locate the previous signal value change for the selected signal
	Find Next Transition locate the next signal value change for the selected signal

These actions will not work on locked cursors.

Related Topics

[Debug Toolbar Tab.](#)

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 8-4).

Figure 8-4. 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.

Procedure

1. Right-click the time value of the active cursor in any window and select Sync All Active Cursors from the popup menu ([Figure 8-5](#)).

Figure 8-5. Sync All Active Cursors



2. 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.

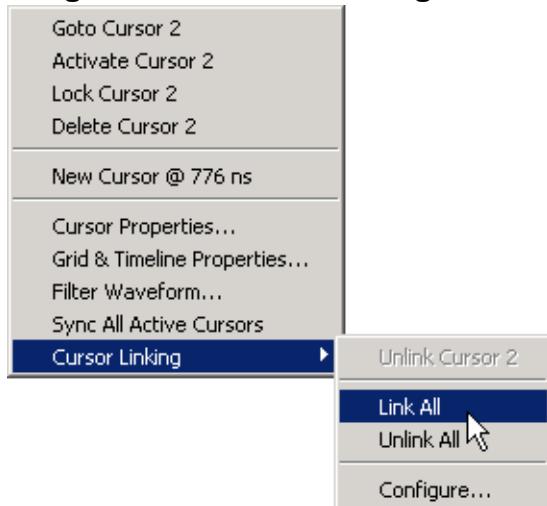
Linking Cursors

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.

Procedure

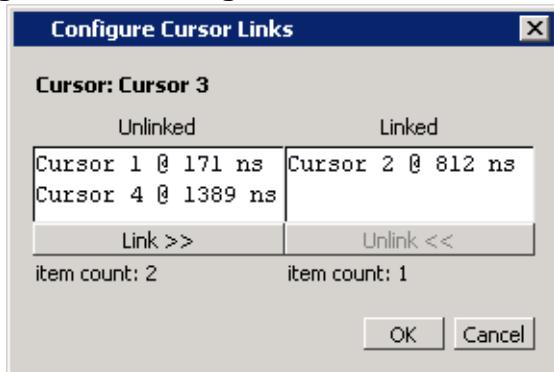
1. You can link all displayed cursors by right-clicking the time value of any cursor in the timeline, as shown in [Figure 8-6](#), and selecting **Cursor Linking > Link All**.

Figure 8-6. Cursor Linking Menu



2. 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 8-7).

Figure 8-7. Configure Cursor Links Dialog



Understanding Cursor Behavior

The following list describes how cursors behave when you click in various panes of the Wave window unless you are in Two Cursor Mode:

- 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.

Two Cursor Mode

Two Cursor Mode places two active cursors in the Wave window. Where default Wave window cursor behavior is for the closest cursor to snap to the location of the mouse when the left mouse button is pressed, in Two Cursor Mode the left mouse button controls movement of the first cursor and the middle mouse button controls the second cursor regardless of the proximity of the pointer to the closest cursor. Additional cursors may be added but are locked upon insertion.

Enable Two Cursor Mode	322
Additional Mouse Actions	322

Enable Two Cursor Mode

You can enable Two Cursor Mode by selecting **Wave > Mouse Mode > Two Cursor Mode**, or by selecting the Two Cursor Mode button in the Debug Toolbar Tab.



You can return to standard Wave Window behavior by selecting **Wave > Mouse Mode >** and choosing one of the other menu picks or by selecting a different button in the Debug Toolbar Tab.

Additional Mouse Actions

Both cursors snap to the position of the mouse pointer when the mouse button controlling the cursor is released. Holding down a button and dragging changes the action from cursor placement to zooming in or out in the waveform pane:

Table 8-4. Two Cursor Zoom

Mouse Action		
Down-Right or Down-Left	Zoom Area (In)	
Up- Right	Zoom Out	
Up-Left	Zoom to Fit	

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

To zoom with 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.

Expanded Time in the Wave Window

When analyzing a design using ModelSim, you can see a value for each object at any time step in the simulation. If logged in the `.wlf` file, the values at any time step prior to and including the current simulation time are displayed in the Wave window or by using the `examine` command.

Some objects can change values more than once in a given time step. These intermediate values are of interest when debugging glitches on clocked objects or race conditions. With a few exceptions (viewing delta time steps with the `examine` command), the values prior to the final value in a given time step cannot be observed.

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	323
Recording Expanded Time Information	324
Viewing Expanded Time Information in the Wave Window	325
Customizing the Expanded Time Wave Window Display	327
Expanded Time Display Modes	329
Switching Between Time Modes	330
Expanding and Collapsing Simulation Time	330
Expanded Time with <code>examine</code> and Other Commands	331

Expanded Time Terminology

The following list provides definitions of the basic terms used when discussing expanded time in the Wave window.

- **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 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.

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.

Table 8-5. Recording Delta and Event Time Information

vsim command argument	modelsim.ini setting	effect
<code>-nowlfcollapse</code>	<code>WLFCollapseMode = 0</code>	Saves multiple value changes of an object during a single time step or single delta cycle. All events for each logged signal are recorded to the <i>.wlf</i> file in the exact order they occur in the simulation.
<code>-wlfcollapsesdelta</code>	<code>WLFCollapseMode = 1</code> (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.
<code>-wlfcollapsetime</code>	<code>WLFCollapseMode = 2</code>	Similar to delta collapsing but at the simulation time step granularity.

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.

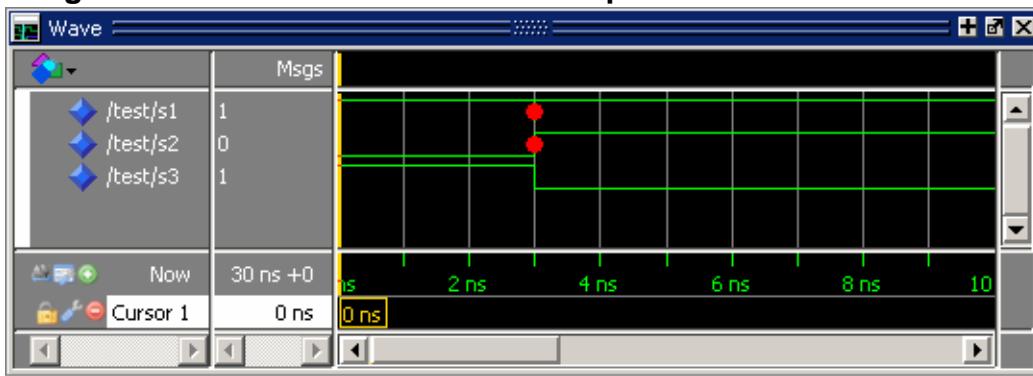
Viewing Expanded Time Information in the Wave Window

Expanded time information is displayed in the Debug Toolbar Tab, 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.

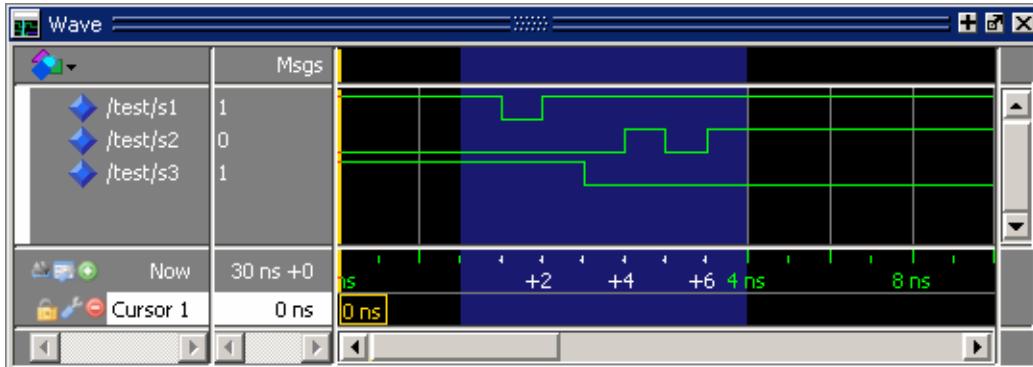
- **Expanded Time Buttons**— The Expanded Time buttons are displayed in the Debug Toolbar Tab in both the undocked Wave window the Main window when the Wave window is docked. It contains three exclusive toggle buttons for selecting the Expanded Time mode (refer to [Toolbar Selections for Expanded Time 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, 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 8-8](#) 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 8-8. Waveform Pane with Collapsed Event and Delta Time



[Figure 8-9](#) 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.

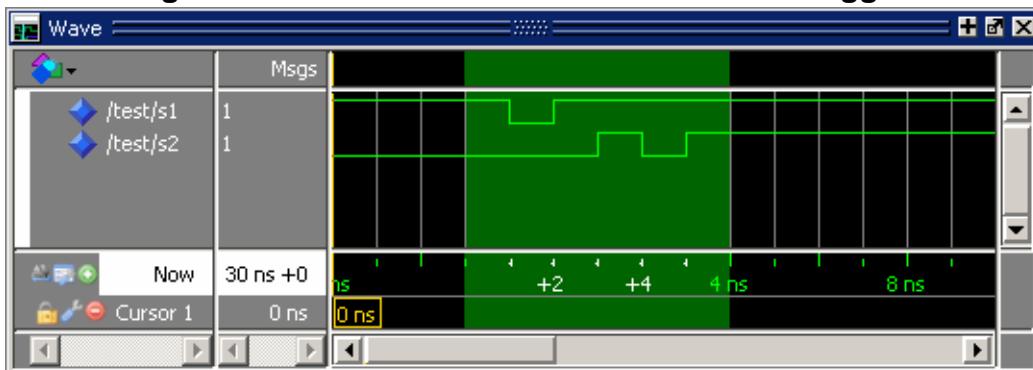
Figure 8-9. Waveform Pane with Expanded Time at a Specific Time

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.

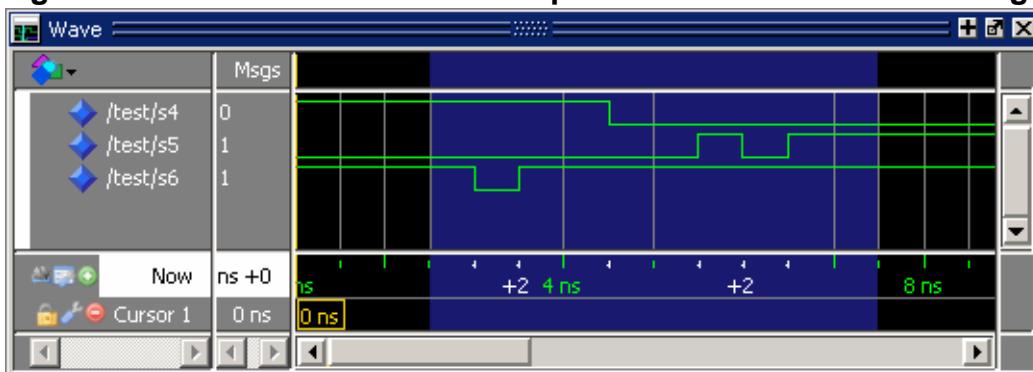
Labels on the time axis in the expanded section indicate the order of events from all of the objects added to the Wave window. If an object that had an event at a particular time but it is not in the viewable area of the Waveform panes, then there will appear to be no events at that time.

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 8-9, had not been added to the Wave window, the result would be as shown in Figure 8-10.

Figure 8-10. Waveform Pane with Event Not Logged

Now the first event on s2 occurs at event time 3ns + 2 instead of event time 3ns + 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 3ns + 3, with no event visible at 3ns + 2.

Figure 8-11 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 8-11. 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. Refer to [Expanded Time Viewing in the List Window](#) in the GUI Reference Manual.

You can display any number of disjoint expanded times or expanded ranges of times.

Related Topics

[Debug Toolbar Tab.](#)

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:

Procedure

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`.

Expanded Time Display Modes

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.....	329
Toolbar Selections for Expanded Time Modes.....	329
Command Selection of Expanded Time Mode	330

Menu Selections for Expanded Time Display Modes

The following table shows the menu selections for initiating expanded time display modes.

Table 8-6. 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 8-6](#) 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 Debug Toolbar Tab 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, the [Debug Toolbar Tab](#) (described in the GUI Reference Manual) includes four buttons 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.

Related Topics

[Debug Toolbar Tab.](#)

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.

Procedure

Use the following procedure:

To expand or collapse simulation time with...	Do the following:
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.,
Toolbar Selections	The Debug Toolbar Tab (described in the <i>GUI Reference Manual</i>) includes four buttons for expanding and collapsing simulation time in the Wave window: Expand Full, Expand Cursor, Collapse Full, and Collapse Cursor.
Commands	<p>There are six commands for expanding and collapsing simulation time in the Wave window.</p> <ul style="list-style-type: none">• wave expand all• wave expand range• wave expand cursor• wave collapse all• wave collapse range• wave collapse cursor <p>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 affect all Waveform panes in the Wave window to which the command applies.</p>

Expanded Time with examine and Other Commands

The Wave window can expand time to show delta delays. You can use the examine, searchlog, and seetime commands to manipulate expanded time data.

- **examine** — The -event <event> option to the [examine](#) command behaves in the same manner as the -delta <delta> option. When the -event option is used, the event time given will refer to the event time relative to events for all signals in the objects dataset at the specified time. This may be misleading as it may not correspond to event times displayed in the List or Wave windows.
- **searchlog** — The -event <event> option to the [searchlog](#) command behaves in the same manner as the -delta <delta> option.

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. You can also save a specific zoom range and scroll position with Wave window bookmarks.

Zooming with the Menu, Toolbar and Mouse.....	332
Saving Zoom Range and Scroll Position with Bookmarks	333
Editing Bookmarks	334

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 [Debug Toolbar Tab](#) (described in more detail in the GUI Reference Manual):



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 between Cursors

zoom window in or out to show the range between the last two active cursors



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 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

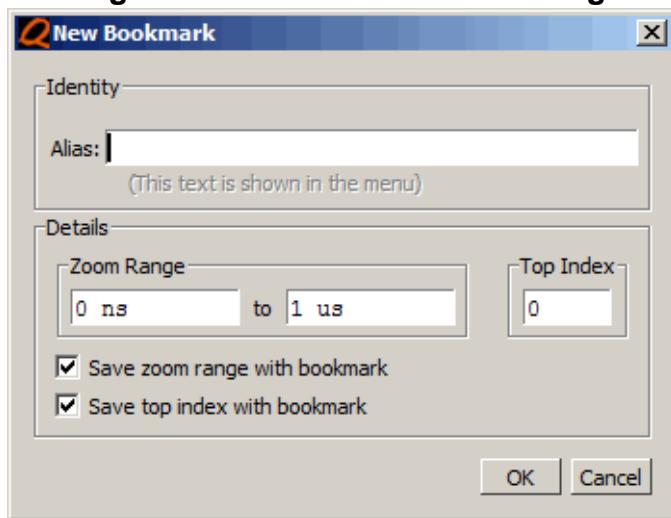
Bookmarks save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name and then access the named bookmark from the Bookmark menu. Bookmarks are saved in the Wave format file and are restored when the format file is read.

To add a bookmark, follow these steps:

Procedure

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 > To Wave > Bookmark**. If the Wave window is undocked, select **Add > Bookmark**.

Figure 8-12. New Bookmark Dialog



3. Give the bookmark a name and click OK.
4. The table below summarizes actions you can take with bookmarks.

Table 8-7. 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 > B ookmarks > <select bookmark then > Delete>	bookmark delete wave

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 Window

The Wave window provides two methods for locating objects:

1. Finding signal names:
 - o Select **Edit > Find**.
 - o Click the **Find** toolbar button (binoculars icon) in the [Home Toolbar Tab](#) (described in the GUI Reference Manual) when the Wave window is active
 - o Use the [find](#) command.

The first two of these options will open a Find mode toolbar at the bottom of the Wave window. By default, the “Search For” option is set to “Name.” For more information, refer to [Find and Filter Functions](#) in the GUI Reference Manual.

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 window.

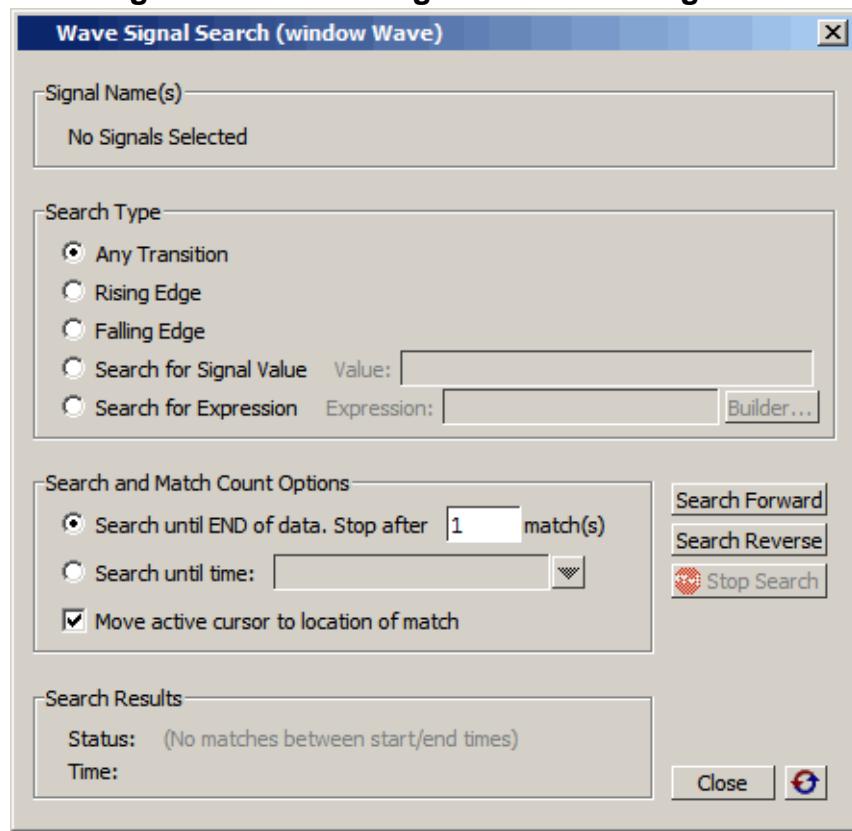
Wave window searches can be stopped by clicking the “Stop Drawing” or “Break” toolbar buttons.

Searching for Values or Transitions	335
Search with the Expression Builder.....	337

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 Wave Signal Search dialog appears.

Figure 8-13. Wave Signal Search Dialog Box



Search with the Expression Builder

The Expression Builder is a feature of the Wave Signal Search dialog box. You can use it to create a search expression that follows the GUI_expression_format, save an expression to a Tcl variable and use it in the Expression Builder to perform a search, and search for when a signal reaches a particular value.

Using the Expression Builder for Expression Searches	337
Saving an Expression to a Tcl Variable.....	339
Searching for a Particular Value	339
Evaluating Only on Clock Edges	339

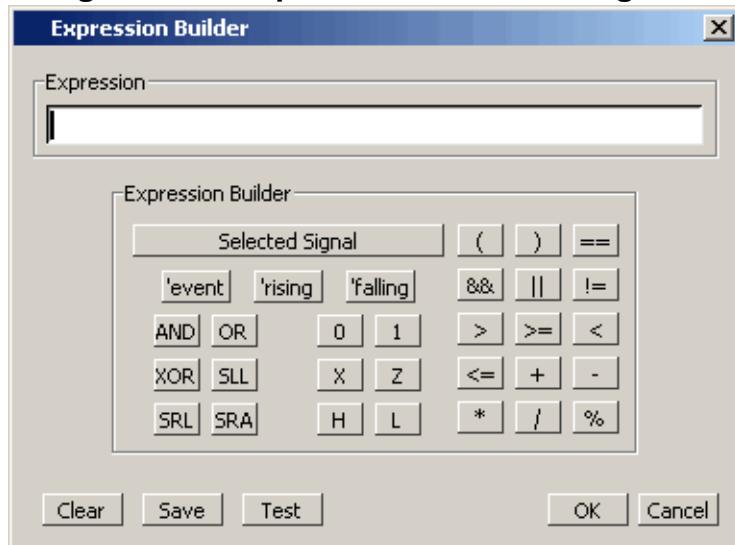
Using the Expression Builder for Expression Searches

You can create a search expression that follows the GUI_expression_format.

Procedure

1. Choose **Wave > 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 8-14](#)

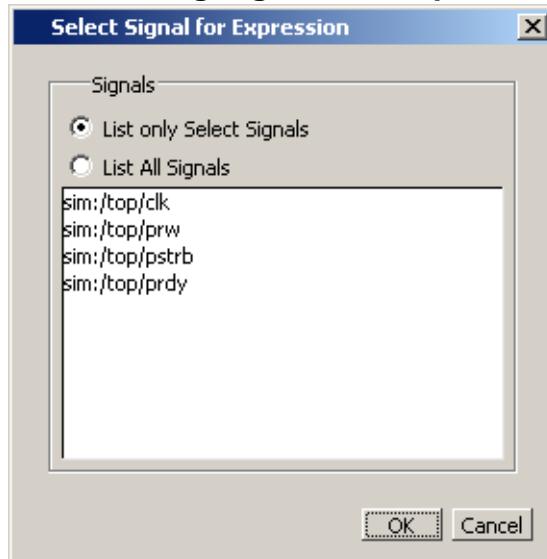
Figure 8-14. Expression Builder Dialog Box



4. 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.

5. In addition, you can use the **Selected Signal** button to create an expression from signals you select from the associated Wave window. For example, instead of typing in a signal name, you can select signals in a Wave window and then click **Selected Signal** in the Expression Builder. This displays the Select Signal for Expression dialog box shown in Figure 8-15.

Figure 8-15. Selecting Signals for Expression Builder



6. 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.
7. Once you have selected the signals you want displayed in the Expression Builder, click OK.
8. Other buttons add operators of various kinds, or you can type them in. (Refer to “[Expression Syntax](#)” in the *Command Reference Manual* for more information.)

Related Topics

[GUI_expression_format](#).

Saving an Expression to a Tcl Variable

Clicking the **Save** button in the Expression Builder 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 a Particular Value

You can use the Expression Builder to search for when a signal reaches a particular value.

Procedure

1. Select a signal of interest in the Wave window.
2. Choose **Wave > Signal Search** from the main menu to open the Wave Signal Search dialog box.
3. Select **Search for Expression** radio button.
4. Click the **Builder** button to open the Expression Builder.
5. Click the **Selected Signal** button to open the **Select Signal for Expression** dialog box.
6. Click the **List only Selected Signals** radio button.
7. Highlight the desired signal and click the **OK** button. This closes the **Select Signal for Expression** dialog box and places the selected signal in the **Expression** field of the **Expression Builder**.
8. Click the == button.
9. Click the value buttons or type a value.
10. Click **OK** to close the **Expression Builder**.
11. Click the **Search Forward** or the **Search Reverse** button to perform the search.

Evaluating Only on Clock Edges

You can use the **Expression Builder** to evaluate search expressions only on clock edges.

Procedure

1. Select the clock signal in the Wave window.
2. Choose **Wave > Signal Search** from the main menu to open the Wave Signal Search dialog box.
3. Select **Search for Expression** radio button.
4. Click the **Builder** button to open the Expression Builder.
5. Click the **Selected Signal** button to open the **Select Signal for Expression** dialog box.
6. Click the **List All Signals** radio button.
7. Highlight the desired signal you want to search and click the **OK** button. This closes the **Select Signal for Expression** dialog box and places the selected signal in the **Expression** field of the **Expression Builder**.
8. Click '**rising**'. You can also select the falling edge or both edges. Or, click the **&&** button to AND this condition with the rest of the expression.
9. Click the **Search Forward** or the **Search Reverse** button to perform the search.

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.

Procedure

1. To activate the filtering function:
2. Select **Edit > Find** in the menu bar (with the Wave window active) or click the **Find** icon in the **Home Toolbar Tab**  (described in the *GUI Reference Manual*). This opens a “Find” toolbar at the bottom of the Wave window.
3. Click the binoculars icon in the Find field to open a popup menu and select **Contains**. This enables the filtering function.

Related Topics

[Find and Filter Functions](#)

Formatting the Wave Window

The primary tool for formatting the Wave Window to fit your environment is the Wave Window Preferences dialog box.

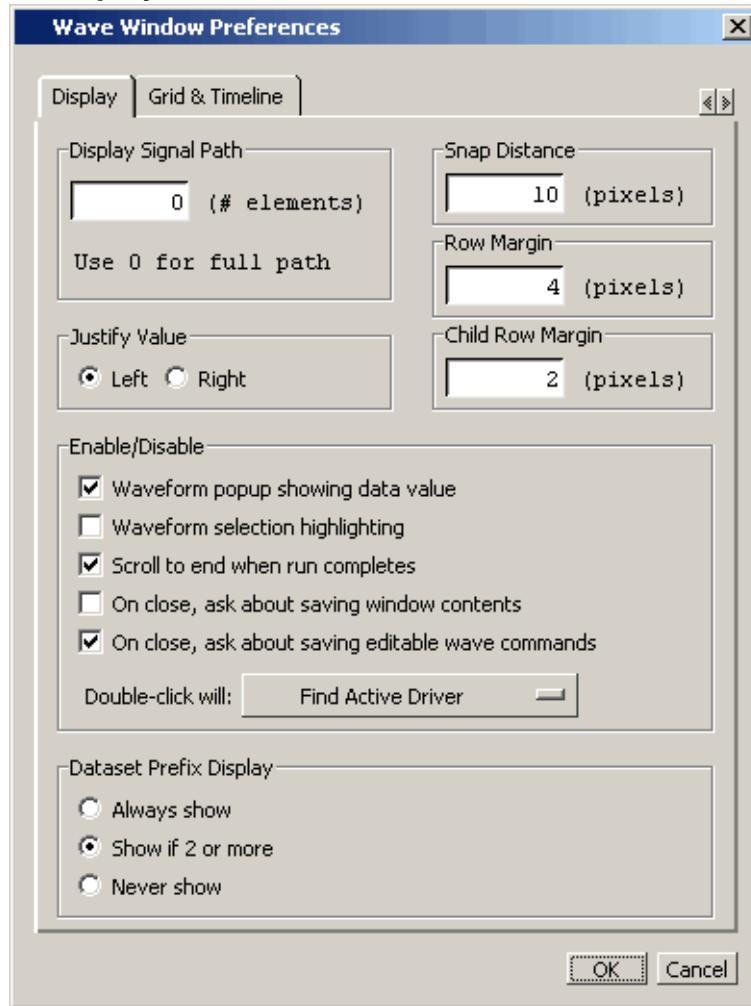
Setting Wave Window Display Preferences	342
Formatting Objects in the Wave Window.....	345
Dividing the Wave Window	348
Splitting Wave Window Panes	349

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 menu selections open the **Wave Window Preferences** dialog (Figure 8-16).

Figure 8-16. Display Tab of the Wave Window Preferences Dialog Box



Hiding/Showing Path Hierarchy	342
Double-Click Behavior in the Wave Window	343
Setting the Timeline to Count Clock Cycles	343

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.

Zero specifies the full path, 1 specifies the leaf name, and any other positive number specifies the number of path elements to be displayed ([Figure 8-16](#)).

Double-Click Behavior in the Wave Window

You can set the default behavior for double-clicking a waveform in the Wave window.

Procedure

1. In the **Wave Window Preferences** dialog box, select the **Display** tab.
2. In the Enable/Disable section, click the button after “**Double-click will:**” and choose one of the following actions from the popup menu:
 - **Do Nothing** — Double-clicking on a waveform does nothing.
 - **Show Drivers in Dataflow** — Double-clicking a waveform 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.
 - **Find Immediate Driver** — Double-clicking a waveform traces to the immediate driver for that signal.
 - **Find Active Driver** — Double-clicking a waveform 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.
 - **Find Root Cause** — Double-clicking a waveform traces the event for the specified signal and time back to the root cause of the event.
 - **Find All Drivers** — Double-clicking a waveform traces to all drivers for the event.

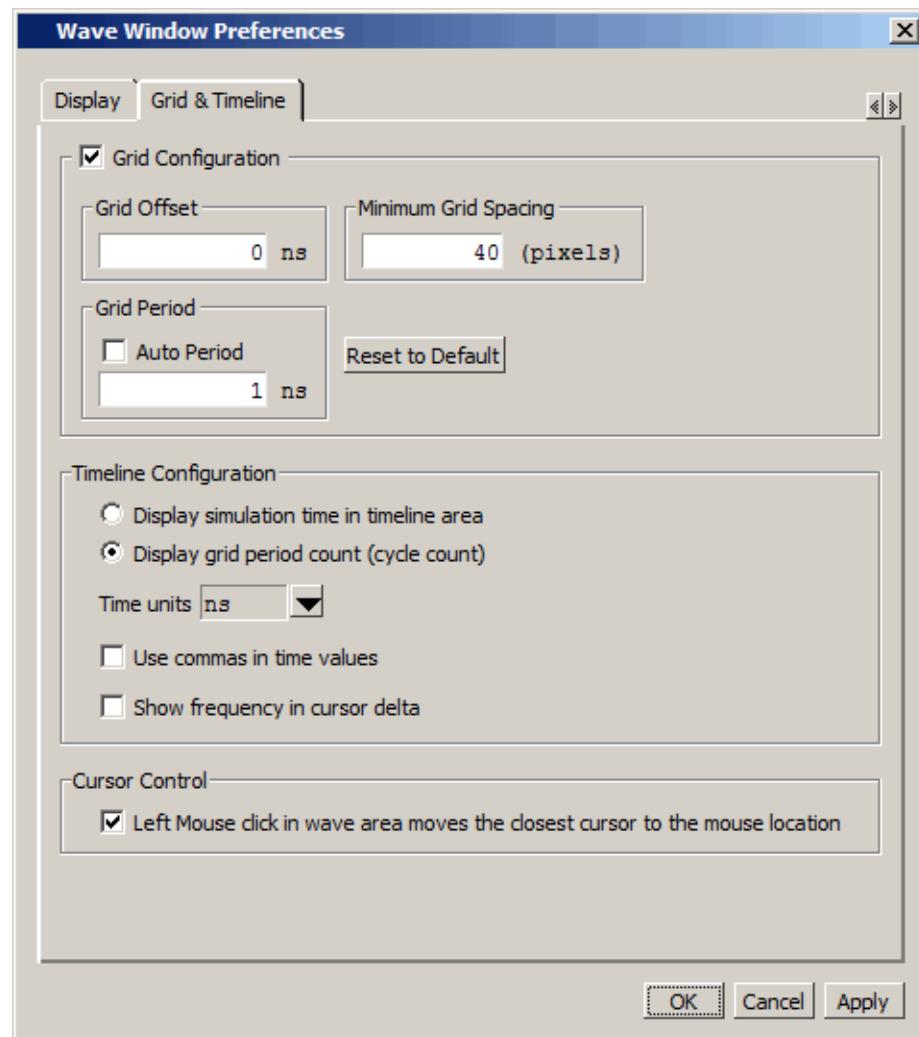
Setting the Timeline to Count Clock Cycles

You can set the timeline of the Wave window to count clock cycles rather than elapsed time.

Procedure

1. 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 box.
2. In the dialog, select the **Grid & Timeline** tab.
3. Enter the period of your clock in the Grid Period field and select “Display grid period count (cycle count)” ([Figure 8-17](#)).

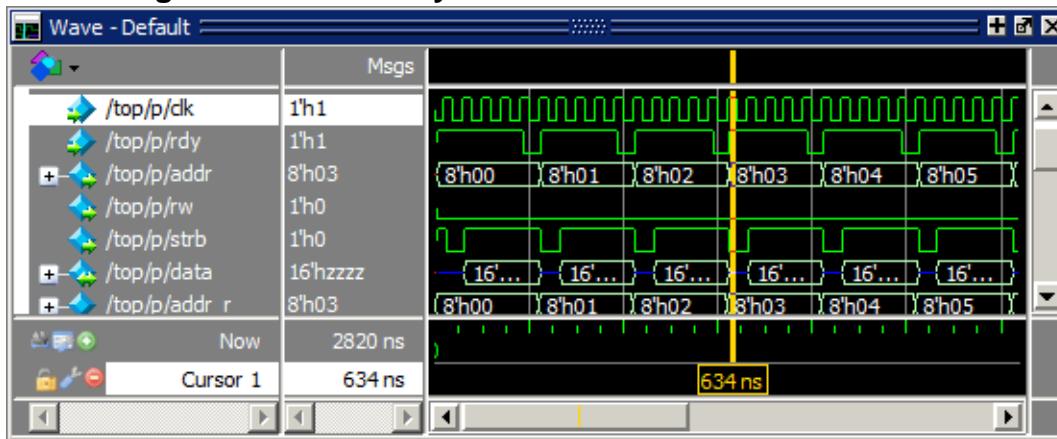
Figure 8-17. Grid and Timeline Tab of Wave Window Preferences Dialog Box



Results

The timeline will now show the number of clock cycles, as shown in Figure 8-18.

Figure 8-18. Clock Cycles in Timeline of Wave Window

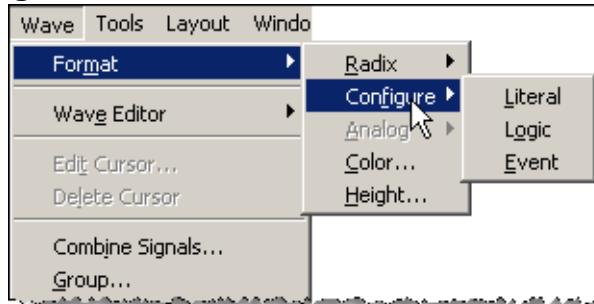


Formatting Objects in the Wave Window

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 8-19](#)).

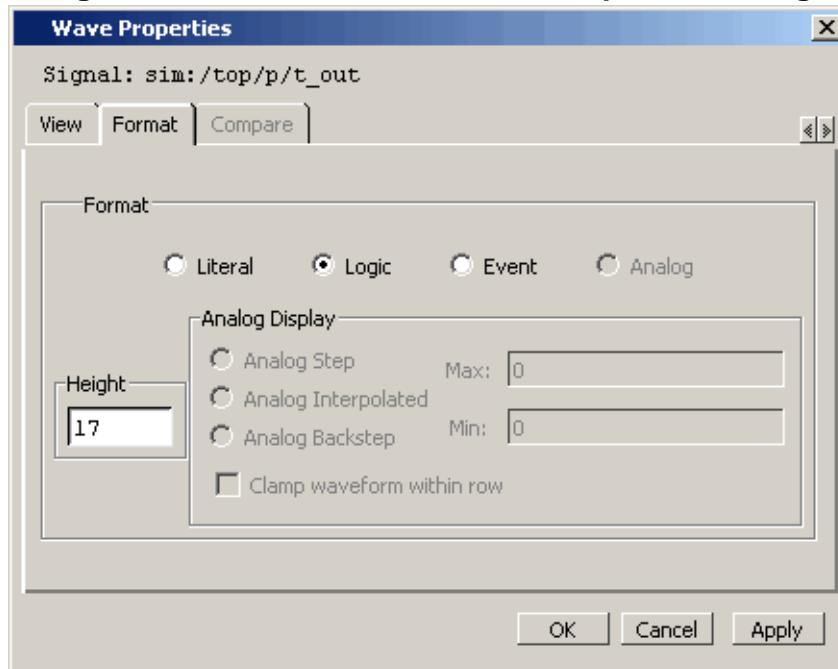
Figure 8-19. Wave Format Menu Selections



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 8-20](#)).

Figure 8-20. Format Tab of Wave Properties Dialog



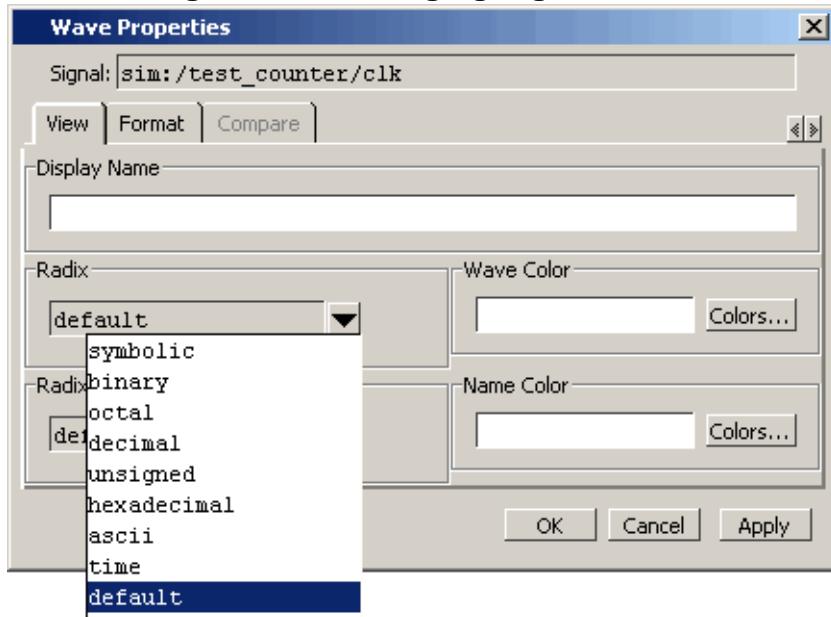
Changing Radix (base) for the Wave Window 346

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 8-21](#)).

Figure 8-21. Changing Signal Radix



The default radix is hexadecimal, which means the value pane lists the hexadecimal values of the 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 347

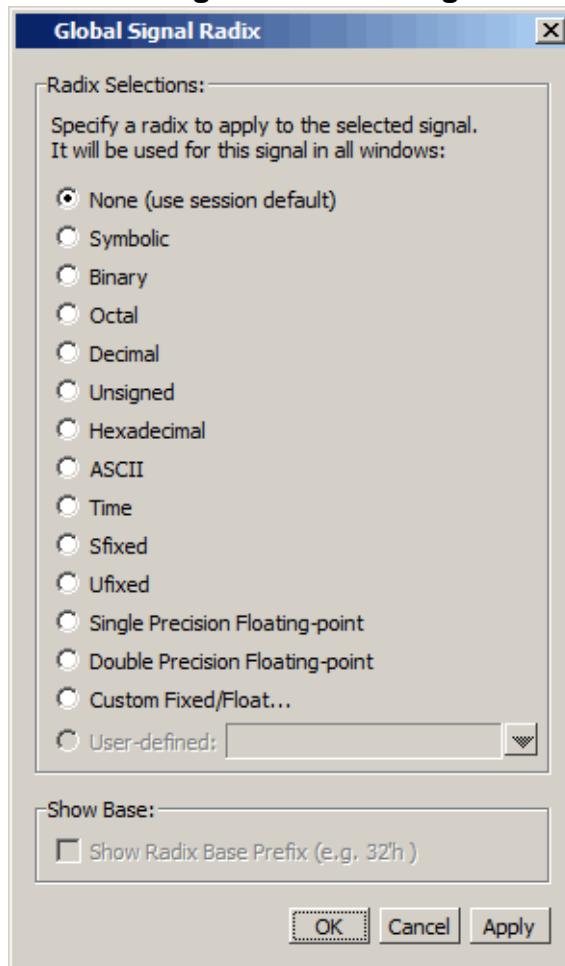
Setting the Global Signal Radix for Selected Objects

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.

Procedure

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 8-22. Global Signal Radix Dialog in Wave Window



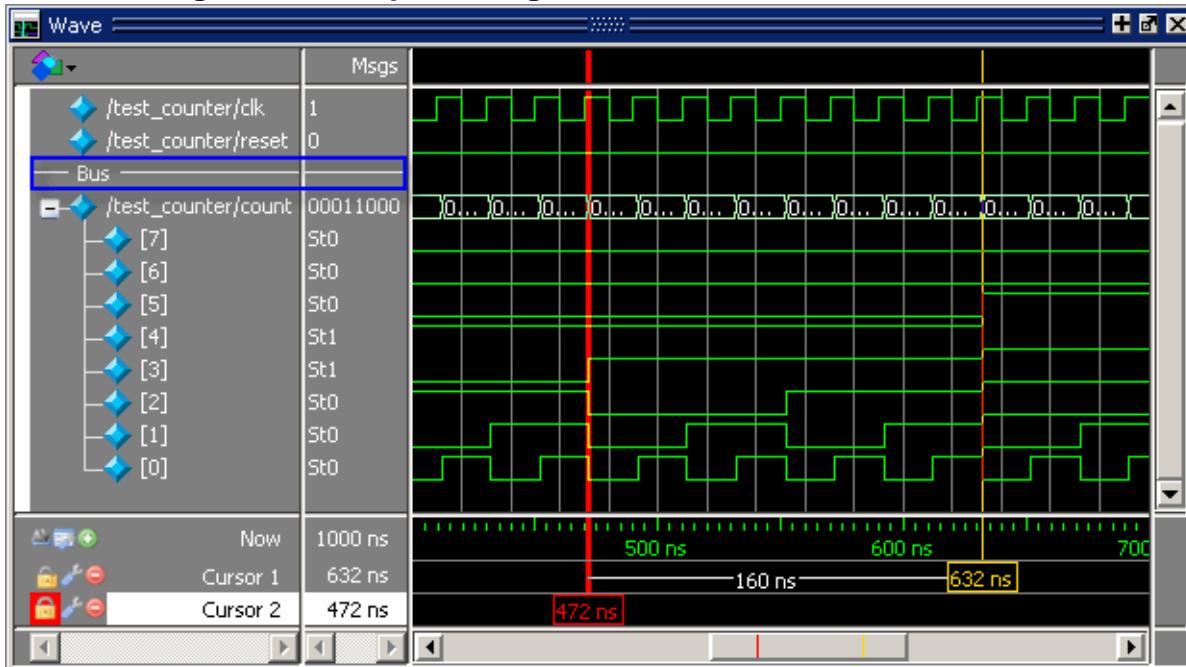
Sfixed and Ufixed indicate “signed fixed” and “unsigned fixed,” respectively. To display an object as Sfixed or Ufixed the object must be an array of std_ulogic elements between 2 and 64 bits long with a descending range. The binary point for the value is implicitly located between the 0th and -1st elements of the array. The index range for the type need not include 0 or -1, for example (-4 downto -8) in which case the value will be

extended for conversion, as appropriate. If the type does not meet these criteria the value will be displayed as decimal or unsigned, respectively.

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 8-23. Separate Signals with Wave Window Dividers



The following procedure shows how to insert a divider.

Procedure

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.
5. You can also insert dividers with the **-divider** argument to the [add wave](#) command.

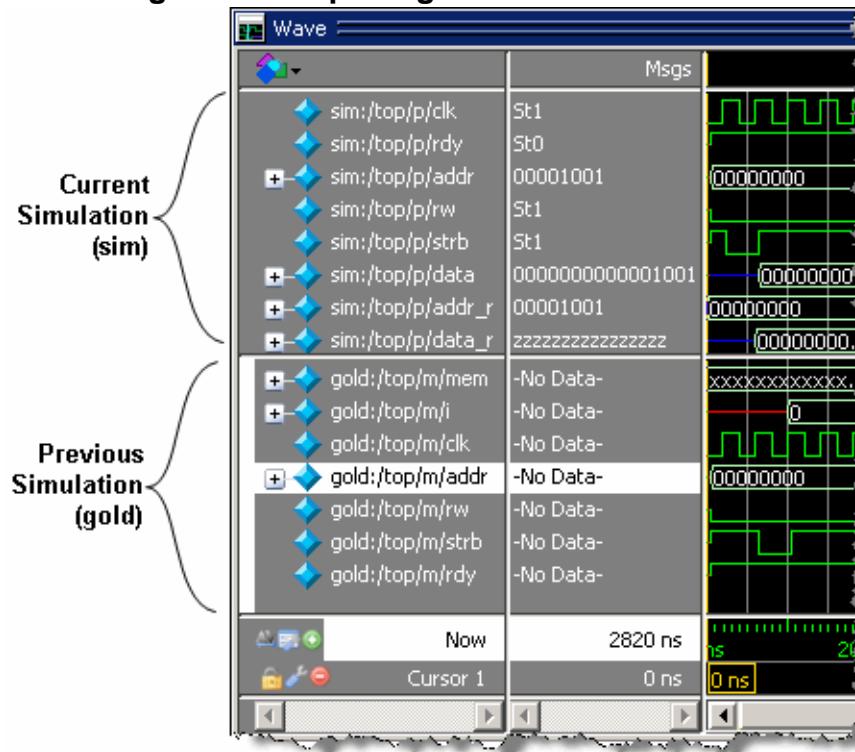
Splitting Wave Window Panes

The pathnames, values, and waveform panes of the Wave window display can be split to accommodate signals from one or more datasets.

Procedure

1. To split the window, select Add > Window Pane.
2. 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."
3. 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.

Figure 8-24. Splitting Wave Window Panes



Related Topics

[Recording Simulation Results With Datasets](#)

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.
- You can create a group that contains the input signals to the process that drives a specified signal.

Creating a Wave Group	351
Deleting or Ungrouping a Wave Group.....	354
Adding Items to an Existing Wave Group.....	354
Removing Items from an Existing Wave Group.....	354
Miscellaneous Wave Group Features	355

Creating a Wave Group

There are three ways to create a wave group.

Grouping Signals through Menu Selection	351
Adding a Group of Contributing Signals	352
Grouping Signals with the add wave Command.....	353
Grouping Signals with a Keyboard Shortcut	353

Grouping Signals through Menu Selection

If you have already added some signals to the Wave window, you can create a group of signals using the following procedure.

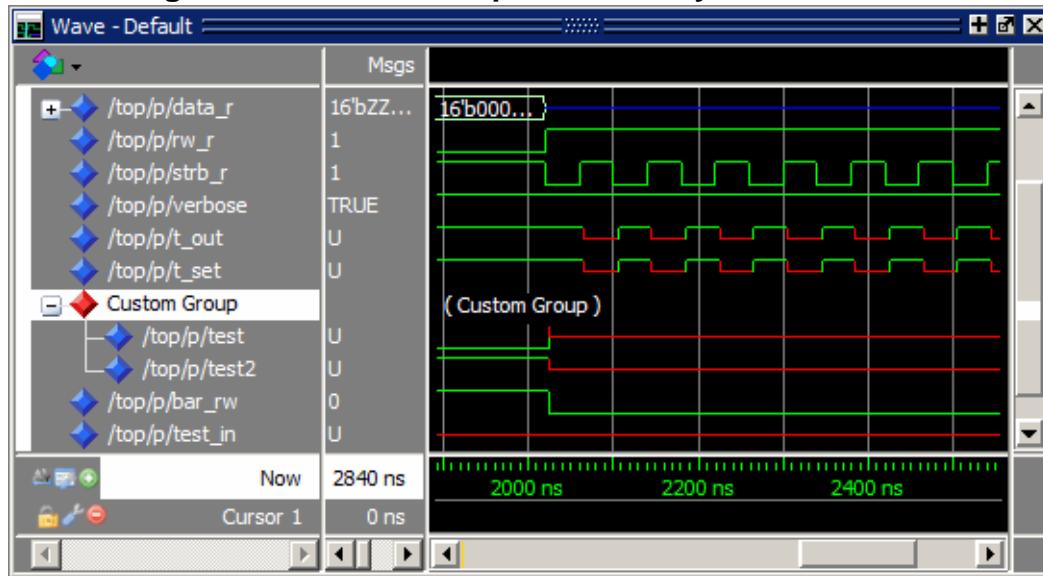
Procedure

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 8-25](#)), with the name specified in the dialog box.

Figure 8-25. Wave Groups Denoted by Red Diamond



Adding a Group of Contributing Signals

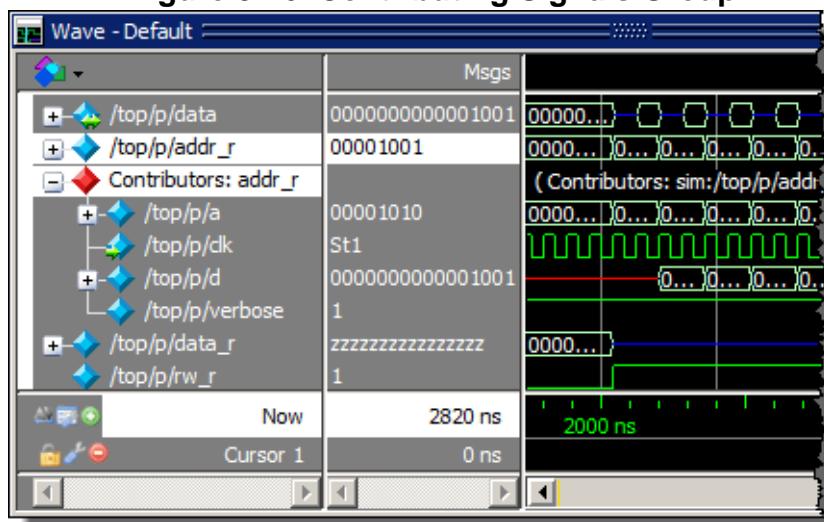
You can select a signal and create a group that contains the input signals to the process that drives the selected signal.

Procedure

1. Select a signal for which you want to view the contributing signals.
2. Click the **Add Contributing Signals** button in the Wave toolbar.

Results

A group with the name `Contributors:<signal_name>` is placed below the selected signal in the Wave window pathnames pane ([Figure 8-26](#)).

Figure 8-26. Contributing Signals Group

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 have 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 356

Creating Composite Signals through Menu Selection

If you have already added some signals to the Wave window, you can create a composite signal or bus using the following procedure.

Procedure

1. Select signals to combine:
 - Shift-click signal pathnames to select a contiguous set of signals, records, and/or buses.
 - Control-click 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 buses 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.

Related Topics

[Virtual Signals](#)

[Virtual Objects](#)

[Using the Virtual Signal Builder](#)

[Concatenation of Signals or Subelements](#)

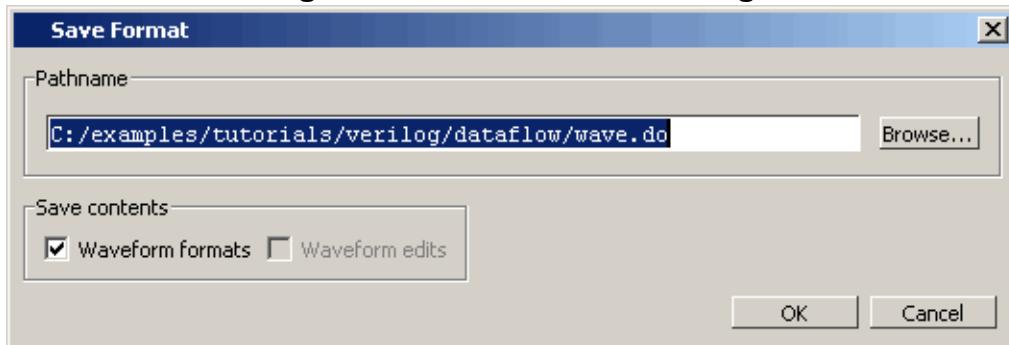
Saving the Window Format

By default, all Wave window information is lost once you close the window. If you want to restore the window to a previously configured layout, you must save a window format file with the following procedure.

Procedure

1. Add the objects you want to the Wave 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 8-27](#)), where you can save waveform formats in a *.do* file.

Figure 8-27. Save Format Dialog



4. To use the format file, start with a blank Wave window and run the DO file in one of two ways:
 - Invoke the **do** command 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.

5. 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>

6. 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.....	359
Printing the Wave Window to a Postscript File	359
Printing the Wave Window on the Windows Platform	360
Saving Waveform Sections for Later Viewing	361

Exporting the Wave Window as a Bitmap Image

You can export the current view of the Wave window to a Bitmap (.bmp) image with the following procedure.

Procedure

1. Select **File > Export > Image** from the Main menus
2. Complete the **Save Image** dialog box.

Results

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 with the following procedure.

Procedure

1. Select **File > Print Postscript** from the Main menus.
2. Complete the Write Postscript dialog box.

The Write Postscript dialog box allows you to control the amount of information exported.

- **SignalSelection** — Allows you to select which signals are exported
- **TimeRange** — 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 with the following procedure.

Procedure

1. Select **File > Print** from the Main menus.
2. Complete 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 Waveform Sections for Later Viewing

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.

Saving Waveforms Between Two Cursors	361
Viewing Saved Waveforms	362
Working With Multiple Cursors	363

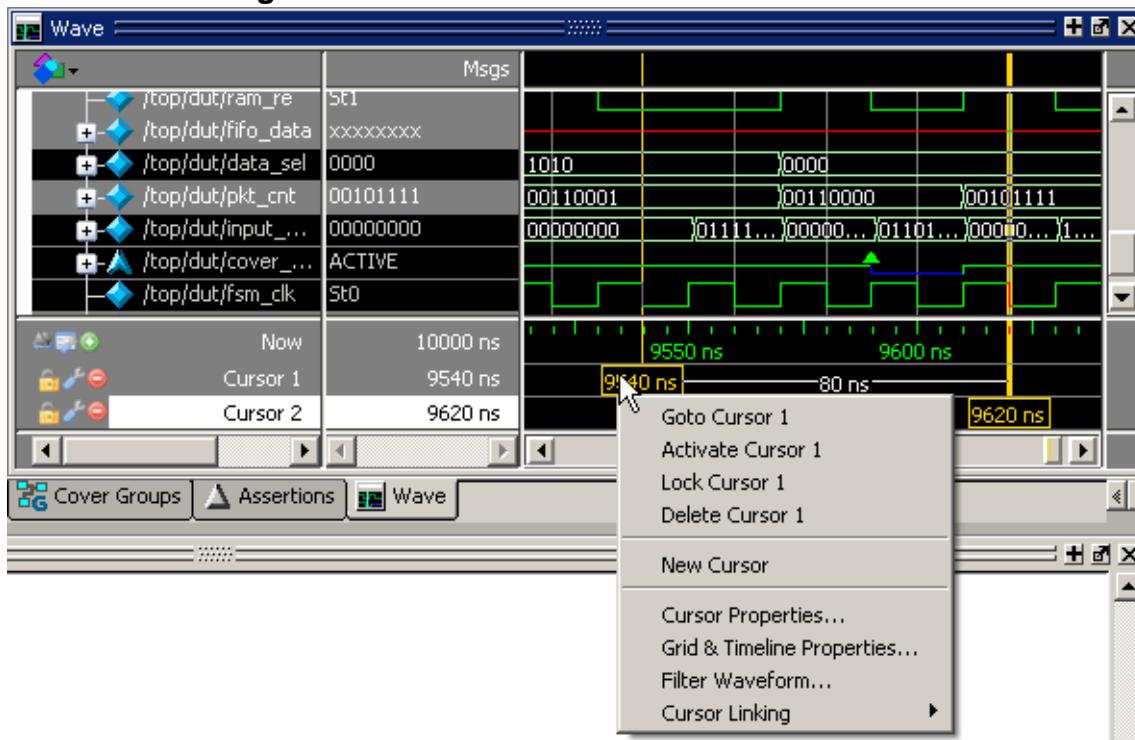
Saving Waveforms Between Two Cursors

You can save a waveform section between two cursors.

Procedure

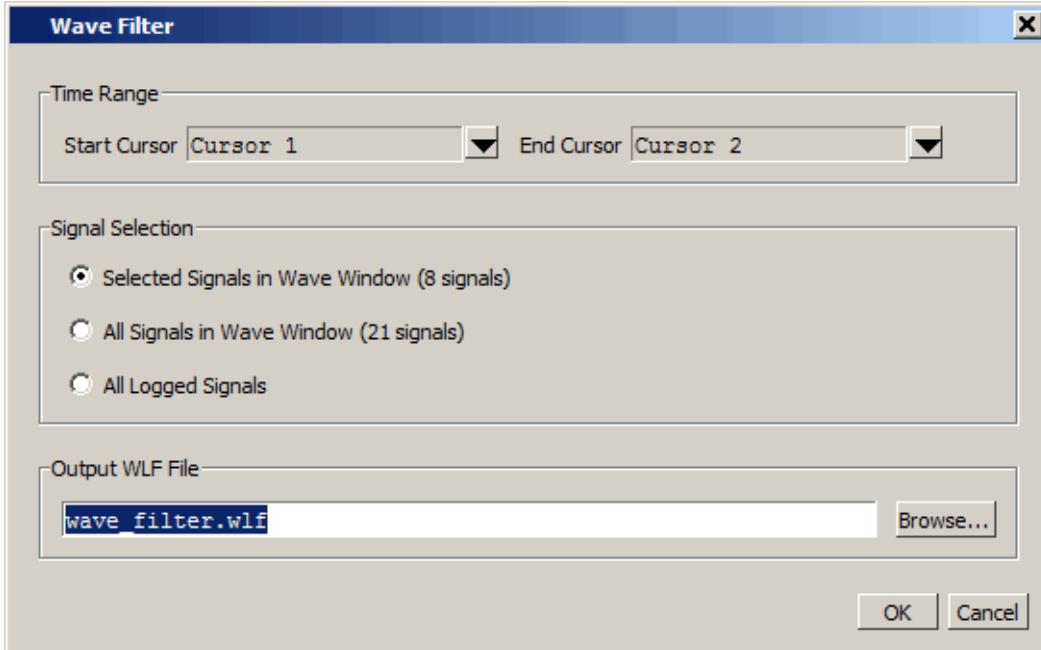
1. Place the first cursor (Cursor 1 in [Figure 8-28](#)) at one end of the portion of simulation time you want to save.
2. Click the **Insert Cursor** icon to insert a second cursor (Cursor 2). 
3. Move Cursor 2 to the other end of the portion of time you want to save. Cursor 2 is now the active cursor, 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 8-28. Waveform Save Between Cursors



5. Select **Filter Waveform** to open the **Wave Filter** dialog box. (Figure 8-29)

Figure 8-29. Wave Filter Dialog



6. Select **Selected Signals in Wave Window** to save the selected objects or signals. You can also choose to save all waveforms displayed in the Wave window between the specified start and end time or all of the logged signals.
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

Call up and view saved waveform sections with the following procedure.

Procedure

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 8-30)

Figure 8-30. 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.

Viewing System Verilog Interfaces

You can log and display scalar and array virtual interface values in the Wave and List windows.

Working with Virtual Interfaces [365](#)

Working with Virtual Interfaces

You can perform the following actions with virtual interfaces:

- Log the virtual interface with the log command. For example:

log /test2/virt

- Add a virtual interface to the List window with the add list command.
- Add a virtual interface to the Wave window with the add wave command. For example:

add wave /test2/virt

Adding Virtual Interface References to the Wave Window 365

Adding Virtual Interface References to the Wave Window

You can add the real interfaces that are referenced by a virtual interface.

Procedure

1. Right-click the portion of the virtual interface waveform you are interested in.
2. Select **Add wave <virtual_interface>/***.

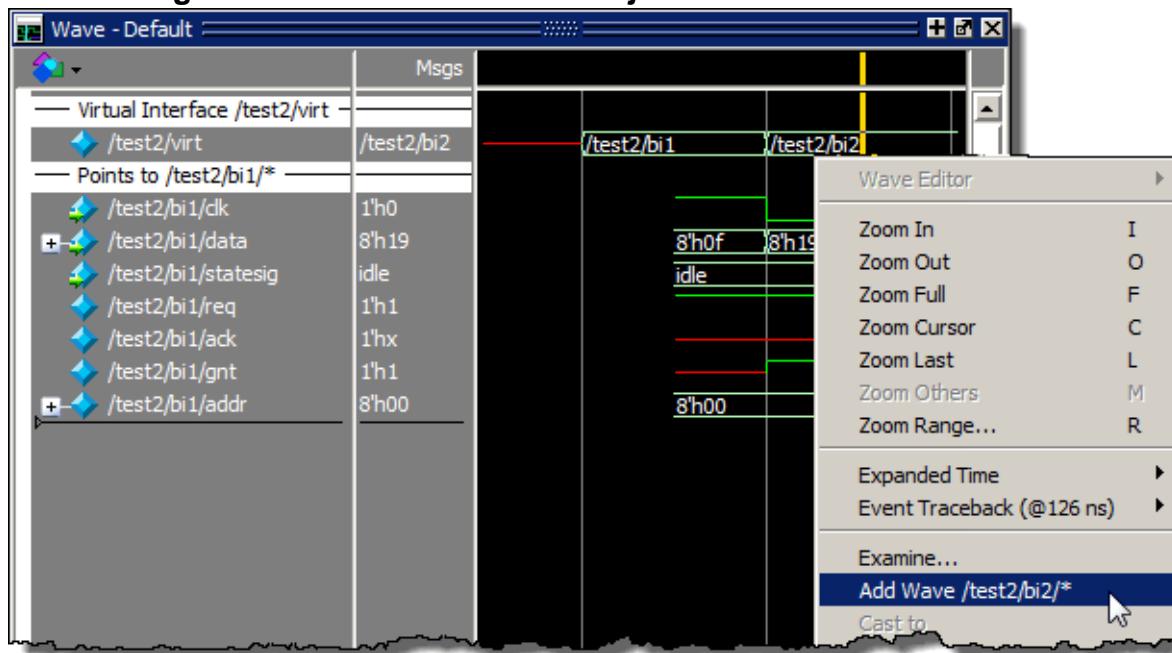
Results

The real interface objects are added to the Wave window and logged from the time they are added.

Examples

Figure 8-31 shows the virtual interface */test2/virt* logged in the Wave window with the real interface */test2/bi1/** added at 75 ns. The nets, array and so forth in the interface */test2/bi2/** are about to be added.

Figure 8-31. Virtual Interface Objects Added to Wave Window



Combining Objects into Buses

You can combine signals in the Wave 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.

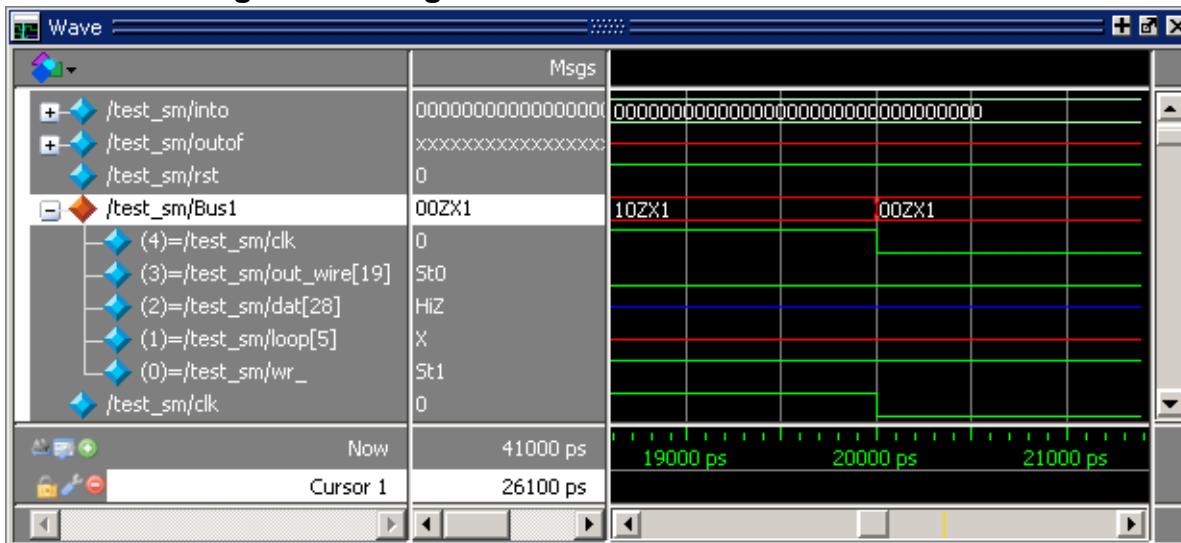
A virtual compare signal (the result of a comparison simulation) is not supported for combination with any other signal.

To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave 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 8-32. Signals Combined to Create Virtual Bus



Extracting a Bus Slice	367
Wave Extract/Pad Bus Dialog Box	368
Splitting a Bus into Several Smaller Buses	369

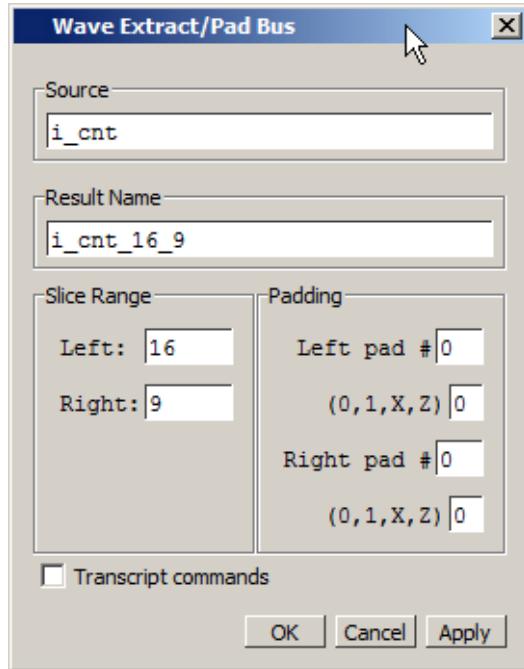
Extracting a Bus Slice

You can create a new bus containing a slice of a selected bus using the following procedure. This action uses the virtual signal command.

Procedure

1. In the Wave window, locate the bus and select the range of signals that you want to extract.
2. Select **Wave > Extract/Pad Slice** (Hotkey: Ctrl+e) to display the [Wave Extract/Pad Bus Dialog Box](#).

Figure 8-33. Wave Extract/Pad Bus Dialog Box



By default, the dialog box is prepopulated with information based on your selection and will create a new bus based on this information.

This dialog box also provides you options to pad the selected slice into a larger bus.

3. Click OK to create a group of the extracted signals based on your changes, if any, to the dialog box.

The new bus, by default, is added to the bottom of the Wave window. Alternatively, you can follow the directions in “[Inserting Signals in a Specific Location](#).”

Wave Extract/Pad Bus Dialog Box

Use the **Wave > Extract/Pad Slice** menu selection to open the Wave Extract/Pad Bus dialog box.

The features of the **Wave Extract/Pad Bus** dialog box ([Figure 8-33](#)) are as follows:

- **Source** — The name of the bus from which you selected the signals.

- **Result Name** — A generated name based on the source name and the selected signals. You can change this to a different value.
- **Slice Range** — The range of selected signals.
- **Padding** — These options allow you to create signal padding around your extraction.
 - **Left Pad / Value** — An integer that represents the number of signals you want to pad to the left of your extracted signals, followed by the value of those signals.
 - **Right Pad / Value** — An integer that represents the number of signals you want to pad to the right of your extracted signals, followed by the value of those signals.
- **Transcript Commands** — During creation of the bus, the virtual signal command to create the extraction is written to the Transcript window.

Splitting a Bus into Several Smaller Buses

You can split a bus into several equal-sized buses using the following procedure. This action uses the virtual signal command.

Procedure

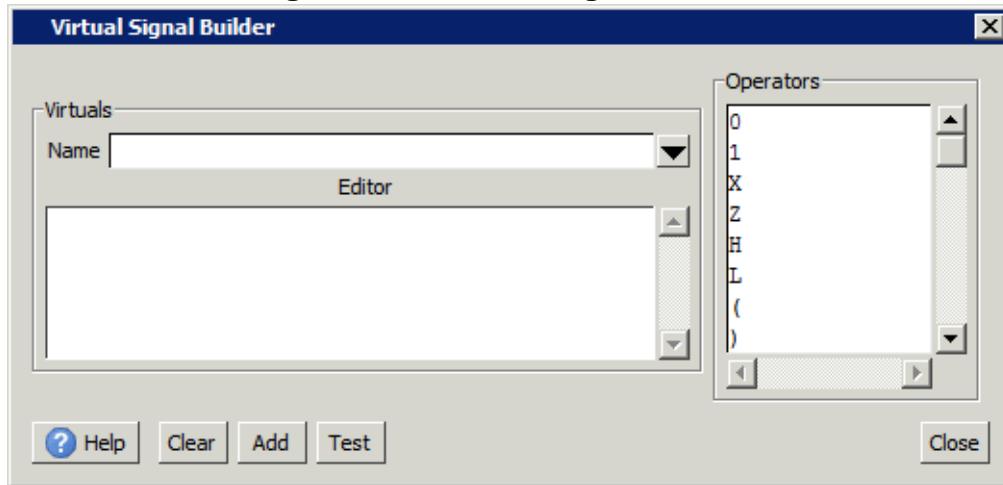
1. In the Wave window, select the top level of the bus you want to split.
2. Select **Wave > Split Bus** (Hotkey: Ctrl+p) to display the Wave Split Bus dialog box.
3. Edit the settings of the Wave Split dialog box
 - **Source** — (cannot edit) Shows the name of the selected signal and its range.
 - **Prefix** — Specify the prefix to be used for the new buses.
The resulting name is of the form: <prefix><n>, where n increments for each group.
 - **Split Width** — Specify the width of the new buses, which must divide equally into the bus width.

Using the Virtual Signal Builder

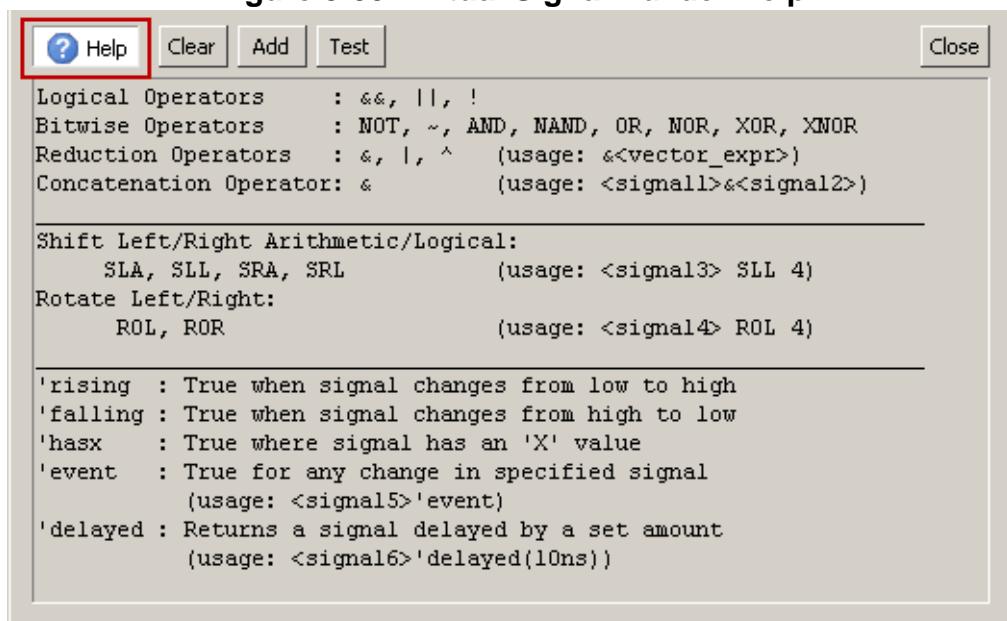
You can create, modify, and combine virtual signals and virtual functions and add them to the Wave window with the Virtual Signal Builder dialog box. Virtual signals are also added to the Objects window and can be dragged to the List, and Watch windows once they have been added to the Wave window.

The Virtual Signal Builder dialog box 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 8-34](#))

Figure 8-34. Virtual Signal Builder



- The Name field allows you to enter the name of the new virtual signal or select an existing virtual signal from the drop down list. Use alpha, numeric, and underscore characters only, unless you are using VHDL extended identifier notation.
- The Editor field is a regular text box. You can enter text directly, copy and paste, or drag a signal from the Objects, Locals, Source , or Wave window and drop it in the Editor field.
- The Operators field allows you to select from a list of operators. Double-click an operator to add it to the Editor field.
- The Help button provides information about the Name, Clear, and Add Text buttons, and the Operators field ([Figure 8-35](#)).

Figure 8-35. Virtual Signal Builder Help

- The Clear button deletes the contents of the Editor field.
- The Add button places the virtual signal in the Wave window in the default location. Refer to [Inserting Signals in a Specific Location](#) for more information.
- The Test button tests the syntax of your virtual signal.

Creating a Virtual Signal **371**

Creating a Virtual Signal

Use the following procedure to create a virtual signal with the Virtual Signal Builder.

Prerequisites

- An active simulation or open dataset.
- An active Wave window with objects loaded in the Pathname pane

Procedure

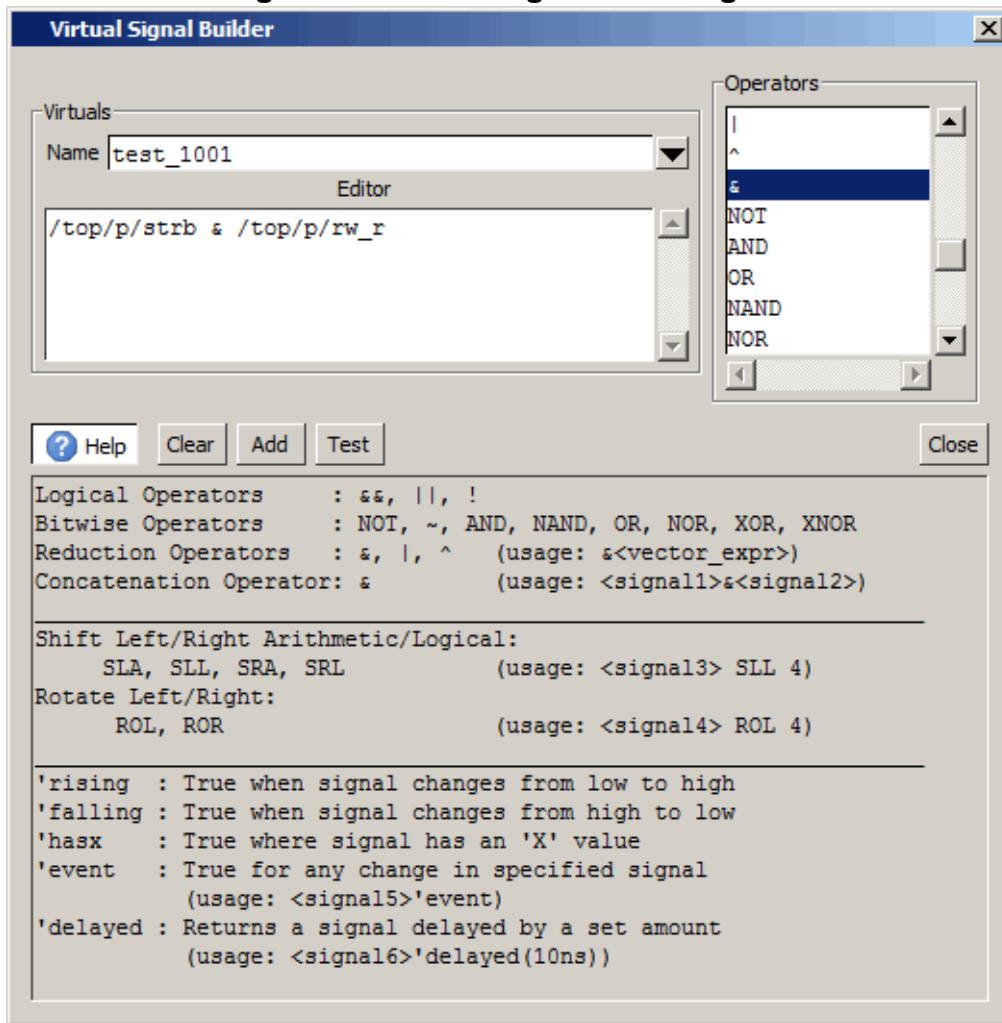
1. Select **Wave > Virtual Builder** from the main menu to open the Virtual Signal Builder dialog box.
2. Drag one or more objects from the Wave or Object window into the **Editor** field.
3. Modify the object by double-clicking on items in the **Operators** field or by entering text directly.

Tip

Select the Help button then place your cursor in the Operator field to view syntax usage for some of the available operators. Refer to [Figure 8-34](#)

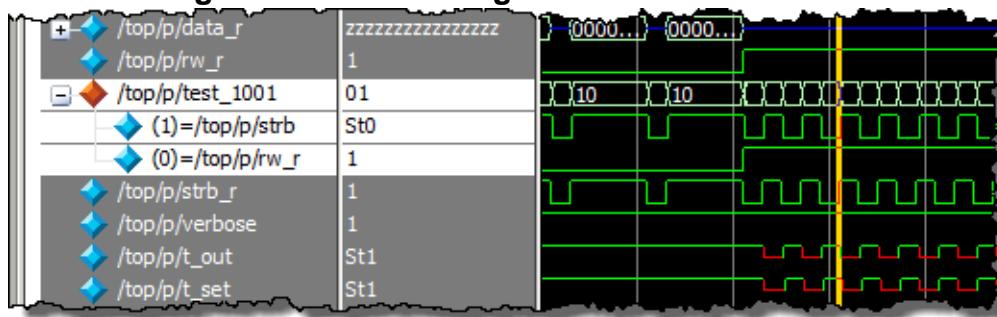
4. Enter a string in the **Name** field. Use alpha, numeric, and underscore characters only, unless you are using VHDL extended identifier notation.
5. Select the **Test** button to verify the expression syntax is parsed correctly.
6. Select **Add** to place the new virtual signal in the Wave window at the default insertion point. Refer to [Inserting Signals in a Specific Location](#) for more information.

Figure 8-36. Creating a Virtual Signal.



Results

The virtual signal is added to the Wave window and the Objects window. An orange diamond marks the location of the virtual signal in the wave window. ([Figure 8-37](#))

Figure 8-37. Virtual Signal in the Wave Window

Related Topics

- [Virtual Objects](#)
- [Virtual Signals](#)
- [GUI_expression_format. See also the virtual signal](#)
- [virtual function](#)

Miscellaneous Tasks

The Wave window allows you to perform a wide variety of tasks, from examining waveform values, to displaying signal drivers and readers, to sorting objects.

Examining Waveform Values	374
Displaying Drivers of the Selected Waveform	374
Sorting a Group of Objects in the Wave Window	375

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 would 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.

Displaying Drivers of the Selected Waveform

You can display the drivers of a signal selected in the Wave window in the Dataflow.

Procedure

1. 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](#))
2. This operation opens the Dataflow window and displays the drivers of the signal selected in the Wave window. A Wave pane also opens in the Dataflow window to show the selected signal with a cursor at the selected time. The Dataflowwindow shows the signal(s) values at the Wave pane cursor position.

Related Topics

[Double-Click Behavior in the Wave Window](#)

Sorting a Group of Objects in the Wave Window

You can easily sort objects in the Wave window.

Procedure

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.

Signal Breakpoints	377
File-Line Breakpoints	380
Saving and Restoring Breakpoints	382

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. 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	377
Setting Signal Breakpoints with the GUI	377
Modifying Signal Breakpoints	378

Setting Signal Breakpoints with the when Command

ModelSim allows you to set a breakpoint with a simple command line instruction.

Procedure

Use the [when](#) command to set a signal breakpoint from the VSIM> prompt.

Examples

The command:

```
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.

Related Topics

[when](#)

Setting Signal Breakpoints with the GUI

Signal breakpoints are most easily set in the Objects and Wave windows.

Procedure

Right-click a signal and select **Insert Breakpoint** from the context menu.

Results

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.

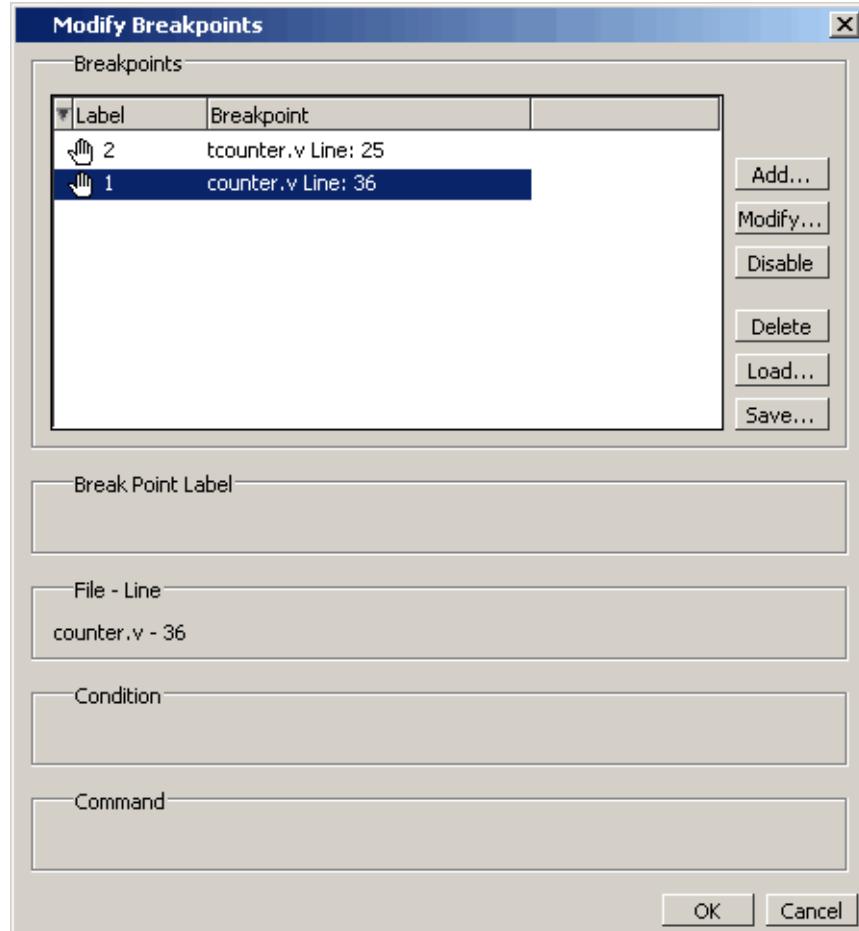
Modifying Signal Breakpoints

You can easily modify the signal breakpoints you have created.

Procedure

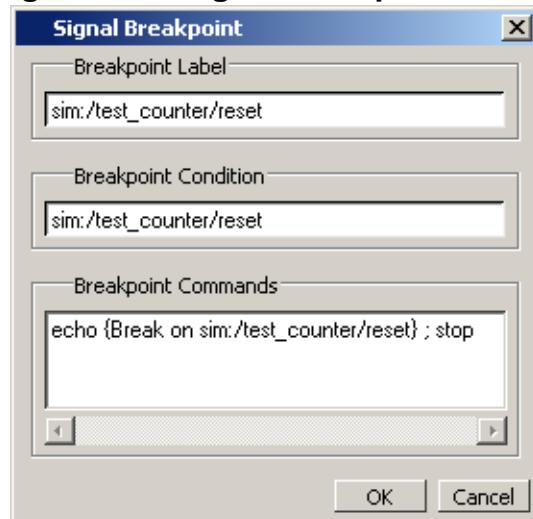
1. Select **Tools > Breakpoints** from the Main menus.
2. This will open the Modify Breakpoints dialog ([Figure 8-38](#)), which displays a list of all breakpoints in the design.

Figure 8-38. Modifying the Breakpoints Dialog



3. When you select a signal breakpoint from the list and click the Modify button, the Signal Breakpoint dialog ([Figure 8-39](#)) opens, allowing you to modify the breakpoint.

Figure 8-39. Signal Breakpoint Dialog



File-Line Breakpoints

You set file-line breakpoints on executable lines in your source files.

When the simulator encounters a line with a breakpoint, it stops and the Source window opens to show the line with the breakpoint. You can change this behavior by editing the PrefSource(OpenOnBreak) variable.

Setting File-Line Breakpoints Using the bp Command	380
Setting File-Line Breakpoints Using the GUI	380
Modifying a File-Line Breakpoint	381

Setting File-Line Breakpoints Using the bp Command

ModelSim allows you to set a file-line breakpoint with a simple command line instruction.

Procedure

Use the **bp** command to set a file-line breakpoint from the VSIM> prompt.

Examples

The command

bp top.vhd 147

sets a breakpoint in the source file *top.vhd* at line 147.

Related Topics

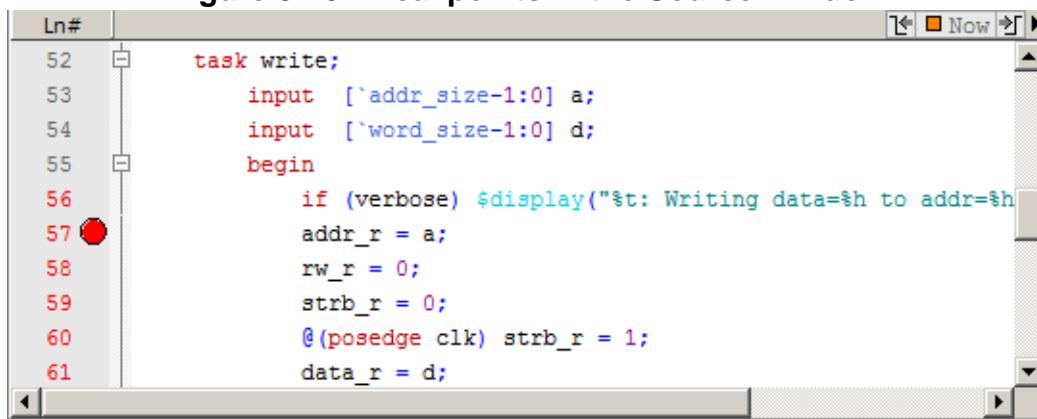
[Simulator GUI Preferences](#)

Setting File-Line Breakpoints Using the GUI

File-line breakpoints are most easily set using your mouse in the Source window.

Procedure

1. 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 8-40](#)).

Figure 8-40. Breakpoints in the Source Window

```
Ln#      task write;
52        input  [`addr_size-1:0] a;
53        input  [`word_size-1:0] d;
54        begin
55            if (verbose) $display("%t: Writing data=%h to addr=%h", $time, d, a);
56            addr_r = a;
57            rw_r = 0;
58            strb_r = 0;
59            @(posedge clk) strb_r = 1;
60            data_r = d;
61
```

2. 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.
3. 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.

Related Topics

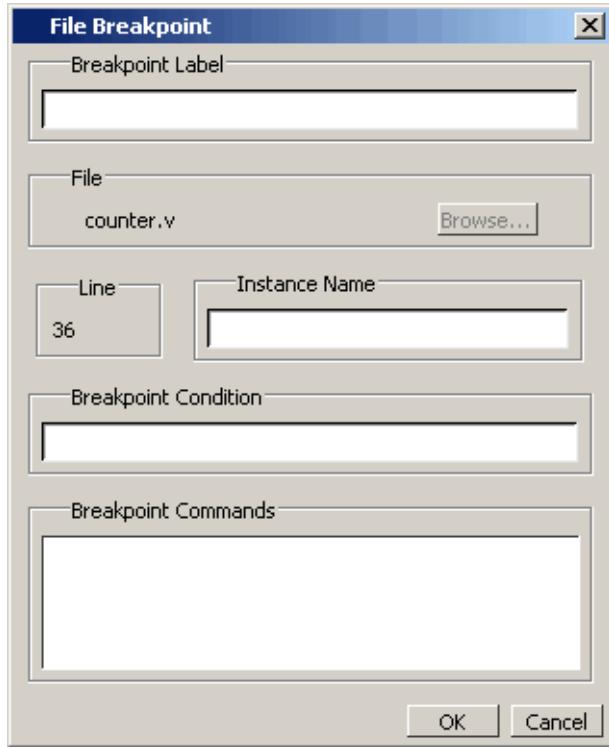
[Source Window](#)

Modifying a File-Line Breakpoint

You can easily modify a file-line breakpoints.

Procedure

1. Select **Tools > Breakpoints** from the Main menus. This will open the Modify Breakpoints dialog ([Figure 8-38](#)), which displays a list of all breakpoints in the design.
2. When you select a file-line breakpoint from the list and click the Modify button, the File Breakpoint dialog ([Figure 8-41](#)) opens, allowing you to modify the breakpoint.

Figure 8-41. File Breakpoint Dialog Box

Saving and Restoring Breakpoints

Command line instructions allow you to save and restore breakpoints.

Procedure

1. 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. The syntax is:

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

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

Results

The file created is primarily a list of **add list** or **add wave** 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.

Chapter 9

Debugging with the Dataflow Window

This chapter discusses how to use the Dataflow window for tracing signal values, browsing the physical connectivity of your design, and performing post-simulation debugging operations.

Dataflow Window Overview	383
Dataflow Usage Flow	385
Common Tasks for Dataflow Debugging	389
Dataflow Concepts	403
Dataflow Window Graphic Interface Reference	408

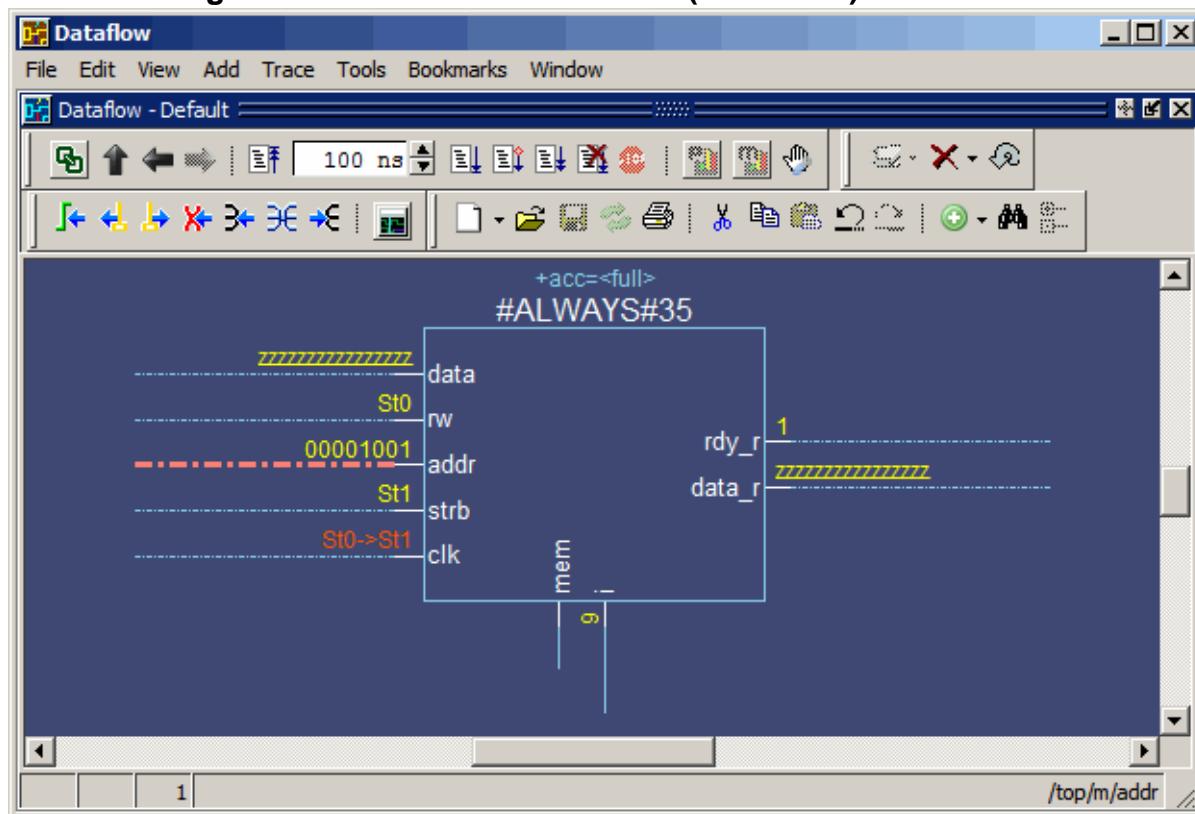
Dataflow Window Overview

The Dataflow window allows you to explore the “physical” connectivity of your design.

Note

 This version of ModelSim has limited Dataflow functionality. Many of the features described below will operate differently. The window will show only one process and its attached signals or one signal and its attached processes, as displayed in [Figure 9-1](#).

Figure 9-1. The Dataflow Window (undocked) - ModelSim



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. For post-simulation debugging, a database is created at design load time, immediately after elaboration, and used later.

Note

 The -postsimdataflow option must be used with the `vsim` command for the Dataflow window to be available for post simulation debug operations.

Live Simulation Debug Flow.....	385
Post-Simulation Debug Flow Details	387

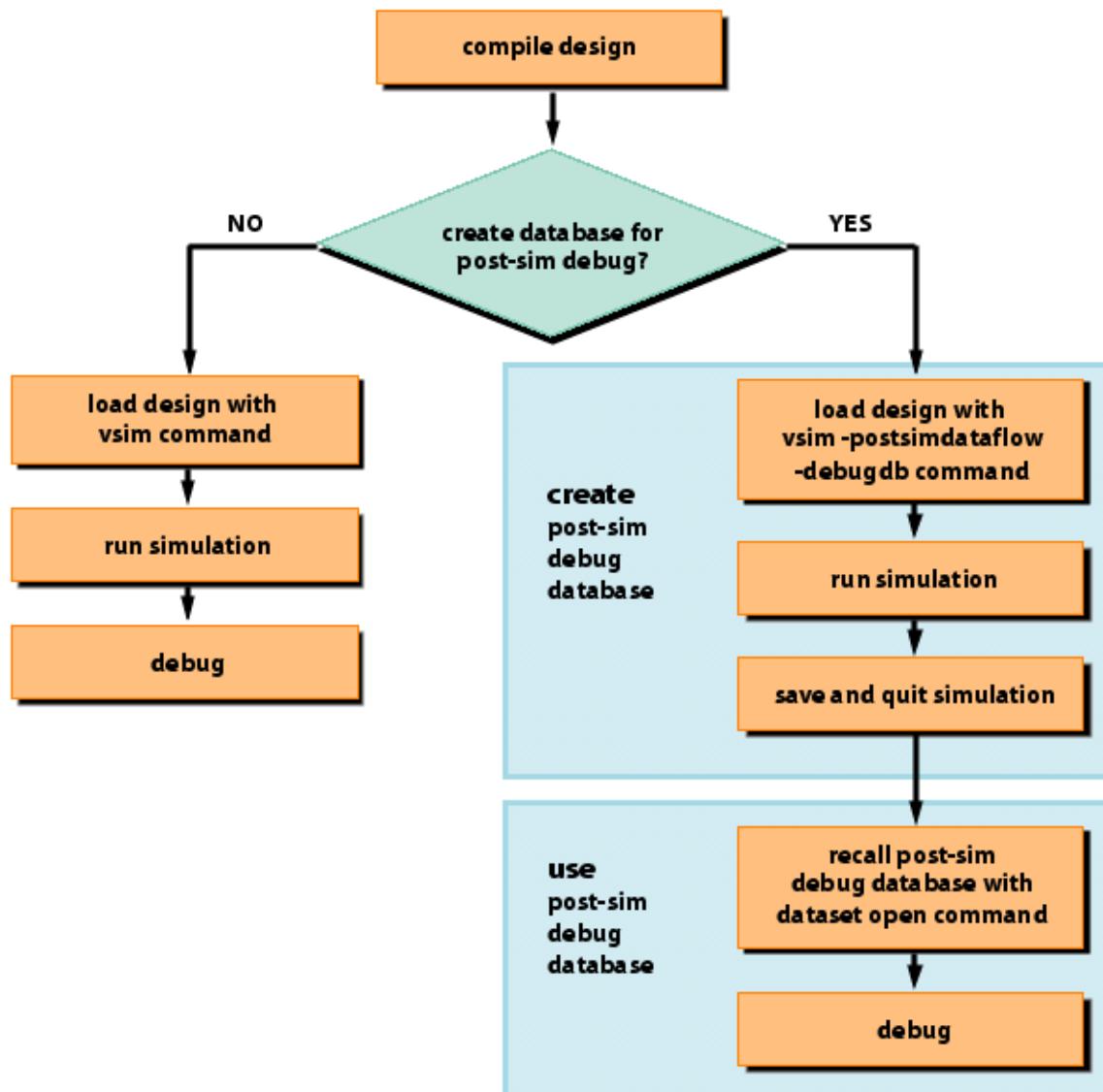
Live Simulation Debug Flow

The usage flow for debugging the live simulation is as follows.

Procedure

1. Compile the design using the `vlog` and/or `vcom` commands.
2. Load the design with the `vsim` command:
`vsim <design_name>`
3. Run the simulation.
4. Debug your design.
5. [Figure 9-2](#) illustrates the current and post-sim usage flows for Dataflow debugging.

Figure 9-2. 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.

- Create the Post-Sim Debug Database** [387](#)
Use the Post-Simulation Debug Database [388](#)

Create the Post-Sim Debug Database

Use the following procedure to create a post-simulation debug database.

Procedure

1. Compile the design using the `vlog` and/or `vcom` commands.
2. Load the design with the following commands:

```
vsim -postsimdataflow -debugdb=<db_pathname> -wlf <db_pathname>
add log -r /*
```

By default, the Dataflow window is not available for post simulation debug operations. You must use the `-postsimdataflow` argument with the `vsim` command to make the Dataflow window available during post-sim debug.

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.
5. You only need to use the `-debugdb=<db_pathname>` argument for the `vsim` command 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.
6. 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

You can use the saved dataset to view objects and trace connectivity. Use the following procedure to open a saved dataset.

Procedure

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 debugging using the Dataflow window include adding objects to the window, exploring the connectivity of the design, tracing events, finding objects, and tracing paths between nets.

Add Objects to the Dataflow Window	389
Exploring the Connectivity of the Design	392
Explore Designs with the Embedded Wave Viewer	396
Tracing Events	398
Tracing the Source of an Unknown State (StX)	399
Finding Objects by Name in the Dataflow Window.....	400
Automatically Tracing All Paths Between Two Nets	401

Add 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 AddSelected to Window Button in the **Standard** toolbar and select **Add to Dataflow**. Refer to [Add Selected to Window Button](#) in the *GUI Reference Manual* for more information.
- Use the [add dataflow](#) command.

The **Add > To Dataflow** menu offers four commands that will add objects to the window:

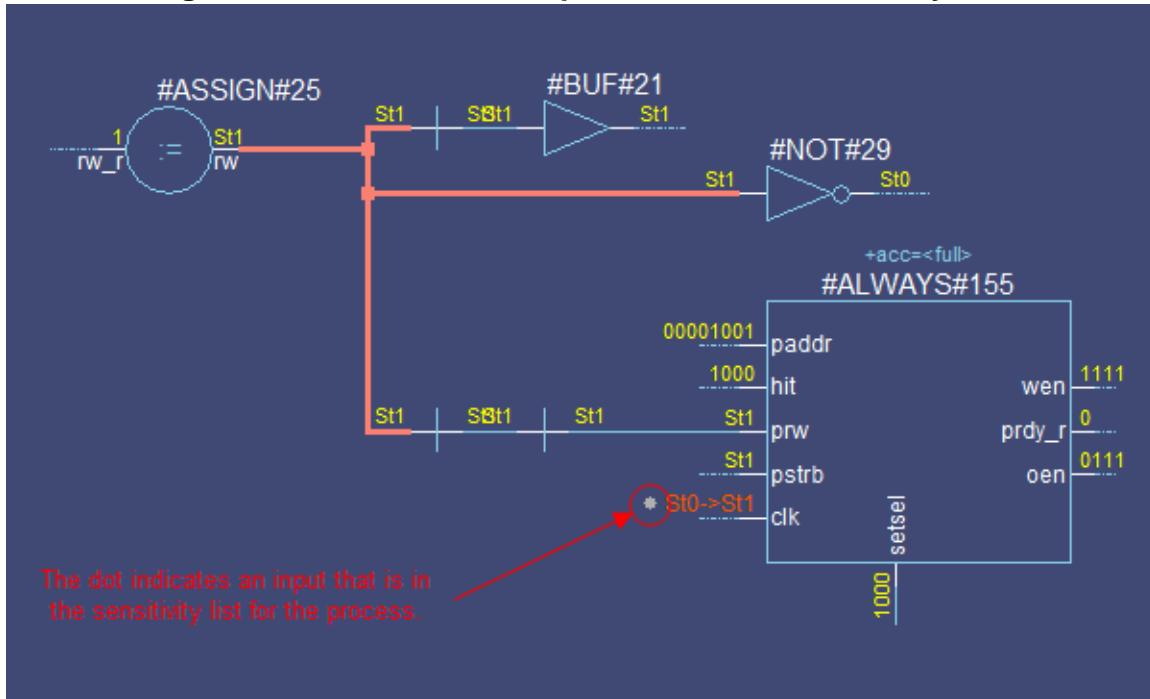
- **View region** — clear the window and display all signals from the current region
- **Add region** — display all signals from the current region without first clearing the window
- **View all nets** — clear the window and display all signals from the entire design
- **Add ports** — add port symbols 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 9-3](#), 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 9-3. Dot Indicates Input in Process Sensitivity List

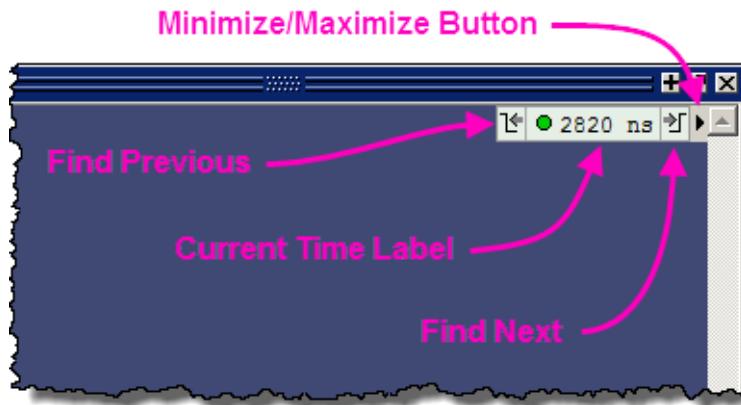


The Dataflow window displays values at the current “active time,” which is set a number of different ways:

- with the selected cursor in the Wave window,
- with the selected cursor in the Dataflow window’s embedded Wave viewer,
- with the Current Time label in the Source or Dataflow windows.

Figure 9-4 shows the CurrentTime label in the upper right corner of the Dataflow window. (This label is turned on by default. If you want to turn it off, select **Dataflow > Preferences** to open the [Dataflow Options Dialog](#) and check the “Current Time label” box.) Refer to [Current Time Label](#) in the *GUI Reference Manual* for more information.

Figure 9-4. CurrentTime Label in Dataflow Window



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 9-1](#).

Table 9-1. Icon and Menu Selections for Exploring Design Connectivity

	Expand net to all drivers display driver(s) of the selected signal, net, or register	Right-click in the Dataflow window > Expand Net to Drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Right-click in the Dataflow window > Expand Net
	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.

Analyzing a Scalar Connected to a Wide Bus.....	392
Control the Display of Readers and Nets.....	394
Controlling the Display of Redundant Buffers and Inverters.....	395
Track Your Path Through the Design.....	395

Analyzing a Scalar Connected to a Wide Bus

During design analysis you may need to trace a signal to a reader or driver through a wide bus. To prevent the Dataflow window from displaying all of the readers or drivers of the bus follow this procedure:

1. You must be in a live simulation; you can not perform this action post-simulation.
2. Select a scalar net in the Dataflow window (you must select a scalar)
3. Right-click and select one of the **Expand > Expand Bit ...** options.

After internally analyzing your selection, the dataflow will then show the connected net(s) for the scalar you selected without showing all the other parts of the bus. This saves in processing time and produces a more compact image in the Dataflow window as opposed to using the **Expand > Expand Net ...** options, which will show all readers or drivers that are connected to any portion of the bus.

Control 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 new will appear as a solid line. To draw the undisplayed readers or drivers, double-click the dashed line.

Limiting the Display of Readers	394
Limit the Display of Readers and Drivers.....	394

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:

Procedure

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.
7. 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.

Limit the Display of Readers and Drivers

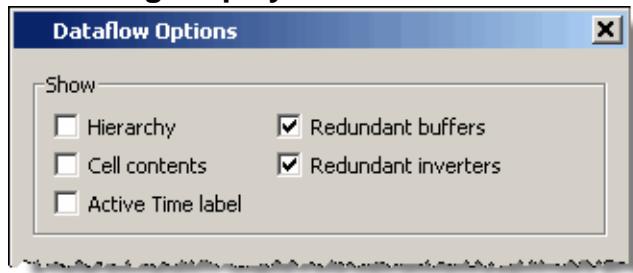
To restrict the expansion of readers and/or drivers to the hierarchical boundary of a selected signal select Dataflow > Dataflow Options to open the **Dataflow Options** dialog box then check **Stop on port** in the **Miscellaneous** field.

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 9-5)

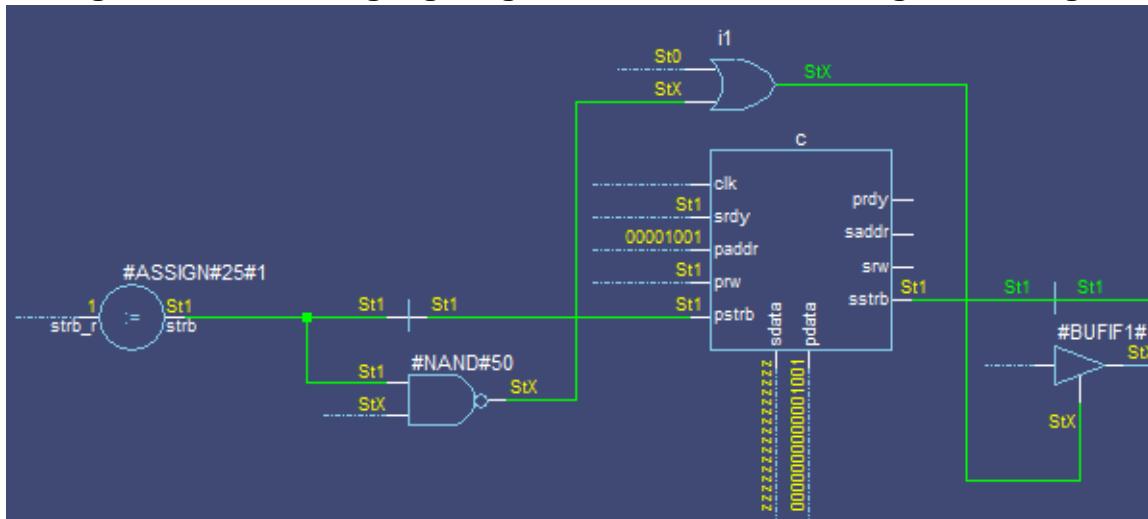
Figure 9-5. Controlling Display of Redundant Buffers and Inverters



Track 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 9-6. Green Highlighting Shows Your Path Through the Design

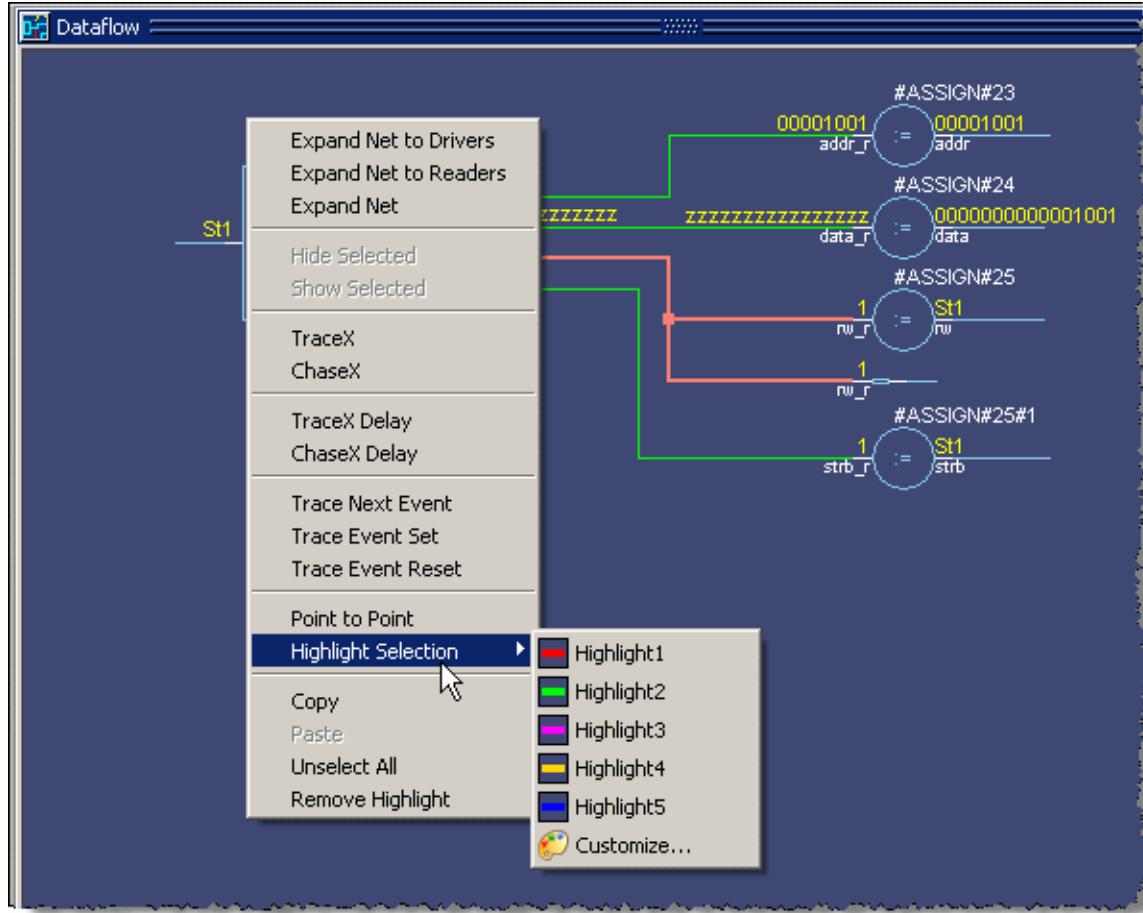


You can clear this highlighting using the **Dataflow > Remove Highlight** menu selection or by clicking the **Remove All Highlights** icon in the toolbar. If you click and hold the **Remove All**

Highlights icon a drop down 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 9-7).

Figure 9-7. 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.

Explore 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.

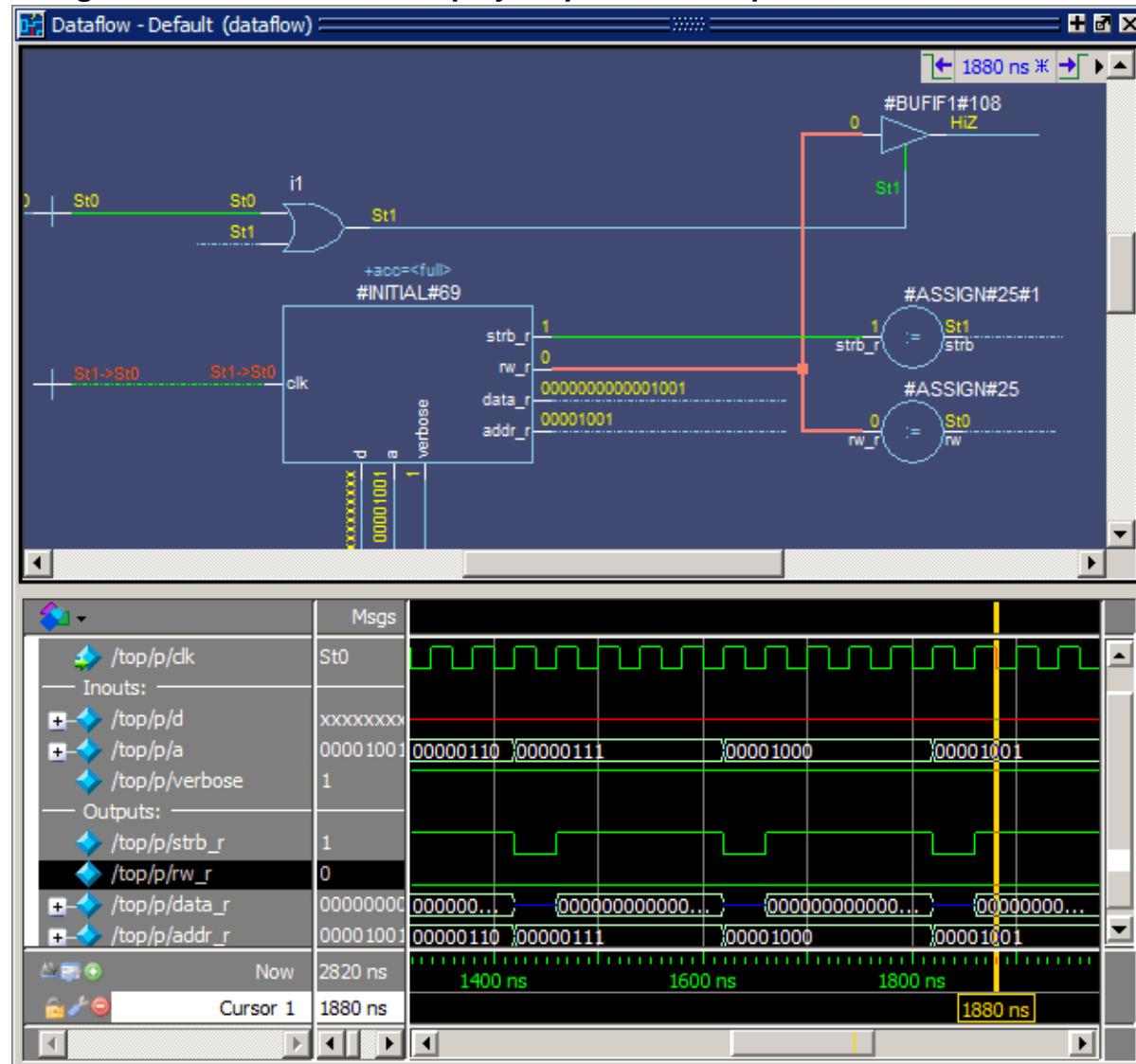
The wave viewer is opened using the **Dataflow > Show Wave** menu selection or by clicking the **Show Wave** icon. 

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 movable cursor (Cursor 1)

is automatically positioned to the location of the active cursor in the last active Wave window. The Current Time label in the upper right of the Dataflow window automatically displays the time of the currently active cursor. Refer to [Current Time Label](#) in the GUI Reference Manual for information about working with the Current 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 9-8. 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.

Related Topics

[Waveform Analysis](#)

[Tracing Events](#)

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. 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.

Procedure

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 have 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.
9. If you want to start over at the originally selected output, right-click and select **Trace Event Reset**.

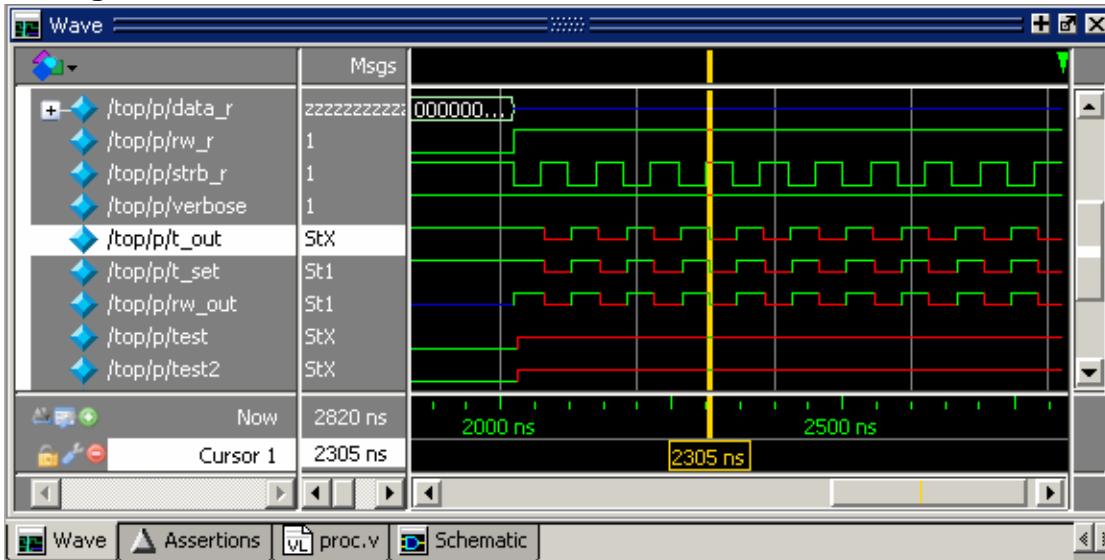
Related Topics

[Explore Designs with the Embedded Wave Viewer](#)

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 6-9) and in the wave viewer pane of the Dataflow window.

Figure 9-9. Unknown States Shown as Red Lines in Wave Window



Procedure

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 9-9](#), 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:
 - Select the signal in the Wave Window, select **Add Selected to Window** in the Standard toolbar > **Add to Dataflow**.
 - right-click the signal in the Objects window and select **Add > To Dataflow > Selected Signals** from the popup menu,
 - 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:
 - If the Dataflow window is docked, make one of the following menu selections:
 - **Tools > Trace > TraceX**
 - **Tools > Trace > TraceX Delay**
 - **Tools > Trace > ChaseX**
 - **Tools > Trace > ChaseX Delay**
 - If the Dataflow window is undocked, make one of the following menu selections:
 - **Trace > TraceX**
 - **Trace > TraceX Delay**
 - **Trace > ChaseX**
 - **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 have 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

Use one of the following procedures to trace or modify the paths between two nets:

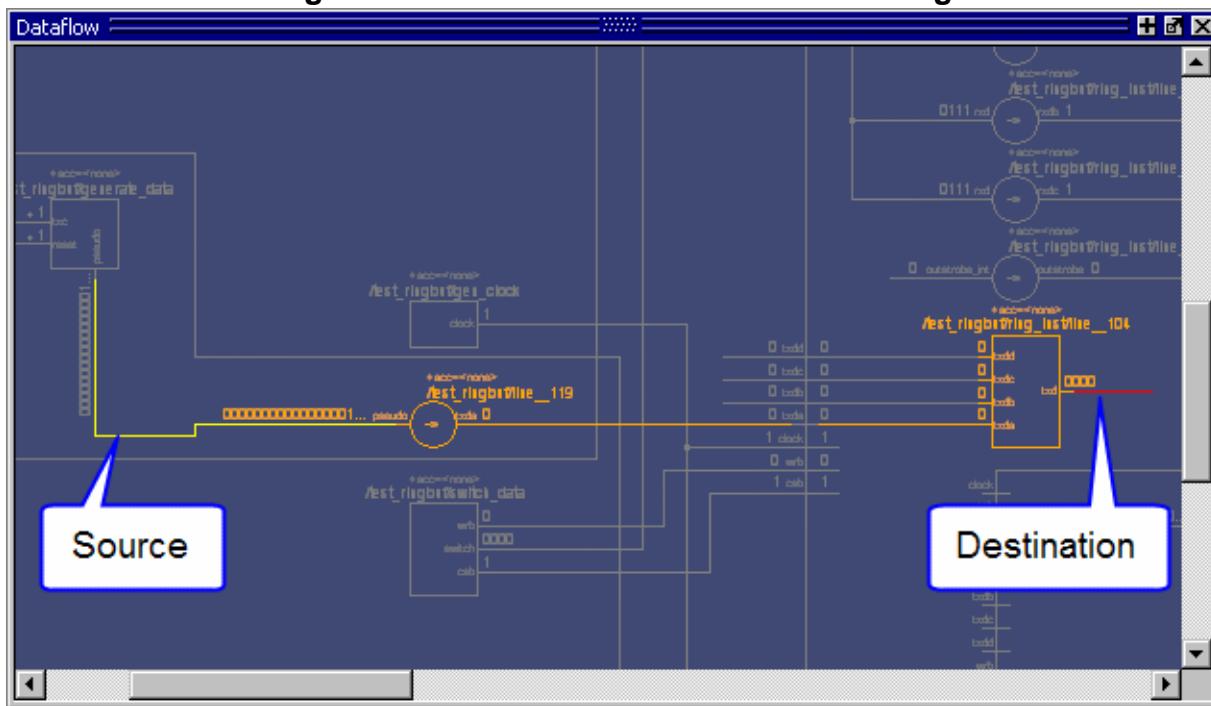
If you want to...	Do the following:
Trace a path between two nets	<ol style="list-style-type: none"> 1. Select Source — Click the net to be your source 2. Select Destination — Shift-click the net to be your destination 3. Run point-to-point tracing — Right-click the Dataflow window and select Point to Point.
Perform point-to-point tracing from the command line	<ol style="list-style-type: none"> 1. Determine the names of the nets 2. Use the add dataflow command with the -connect switch. for example: <code>add data -connect /test_ringbuf/pseudo/test_ringbuf/ring_inst/txd</code> where <i>/test_ringbuf/pseudo</i> is the source net and <i>/test_ringbuf/ring_inst/txd</i> is the destination net.
Change the limit of highlighted processes — There is a limit of 400 processes that will be highlighted	<ol style="list-style-type: none"> 1. Tools > Edit Preferences 2. By Name tab 3. Dataflow > p2plimit option
Remove the point-to-point tracing	<ol style="list-style-type: none"> 1. Right-click the Dataflow window 2. Erase Highlights

Results

After beginning the point-to-point tracing, the Dataflow window highlights your design as shown in [Figure 9-10](#):

- All objects become gray
- The source net becomes yellow
- The destination net becomes red
- All intermediate processes and nets become orange.

Figure 9-10. Dataflow: Point-to-Point Tracing



Dataflow Concepts

This section provides an introduction to the following important Dataflow concepts:

Symbol Mapping	404
User-Defined Symbols	406
Current vs. Post-Simulation Command Output.....	407

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.

Syntax

```
<bsm_line> ::= <comment> | <statement>
```

Arguments

- The following arguments are available:
 - <comment> ::= "#" <text> <EOL>
 - <statement> ::= <name_pattern> <gate>
 - <name_pattern> ::= [<library_name> "."] <du_name> ["(" <specialization> ")"] [,]<process_name>]
 - <gate> ::=
"BUF"|"BUFIFO"|"BUFIF1"|"INV"|"INVIF0"|"INVIF1"|"AND"|"NAND"
"NOR"|"OR"|"XNOR"|"XOR"|"PULLDOWN"|"PULLUP"|"NMOS"|"PMOS"|"CM
OS"|"TRAN"|"TRANIF0"|"TRANIF1"

Description

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:

Examples

- Example 1

```
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

 For primitive gate symbols, pin mapping is automatic.

User-Defined Symbols

The formal BNF format for the *dataflow.sym* file format is:

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.

Syntax

```
<sym_line> ::= <comment> | <statement>
```

Arguments

- The following arguments are available:

```
<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 is 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 driver 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.

Related Topics

[drivers and readers commands. \[ModelSim Command Reference Manual\]](#)

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?.....	408
How is the Dataflow Window Linked to Other Windows?	408
How Can I Print and Save the Display?	409
How Do I Configure Window Options?.....	411

What Can I View in the Dataflow Window?

The Dataflow window displays processes, signals, nets, and registers.

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.

How is the Dataflow Window Linked to Other Windows?

The Dataflow window is dynamically linked to other debugging windows and panes as described in the table below.

Refer to the GUI Reference Manual for more information on the windows named in the table.

Table 9-2. Dataflow Window Links to Other Windows and Panes

Window	Link
Structure 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
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?

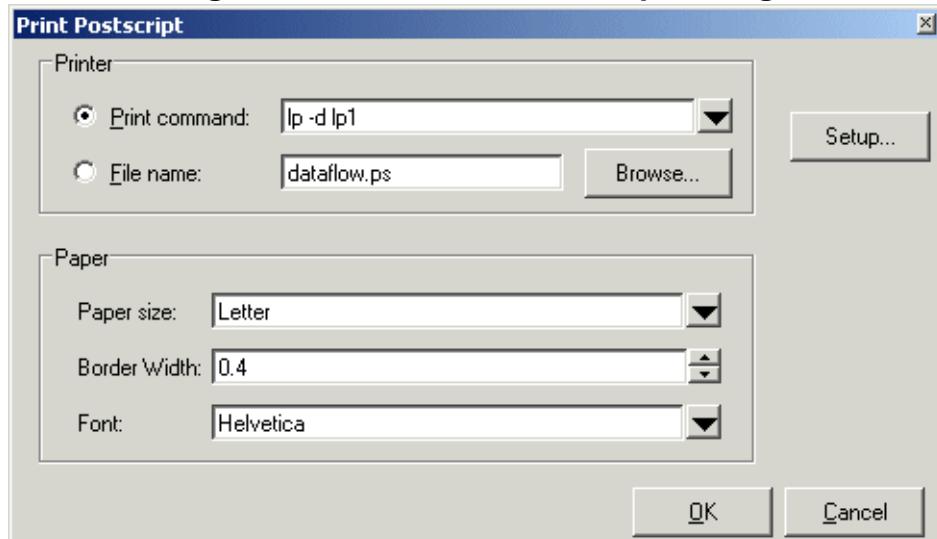
You can print the Dataflow window display from a saved *.eps* file in UNIX, or by simple menu selections in Windows. The Page Setup dialog allows you to configure the display for printing.

Save a .eps File and Printing the Dataflow Display from UNIX	409
Print from the Dataflow Display on Windows Platforms	409
Configure Page Setup	410

Save 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.

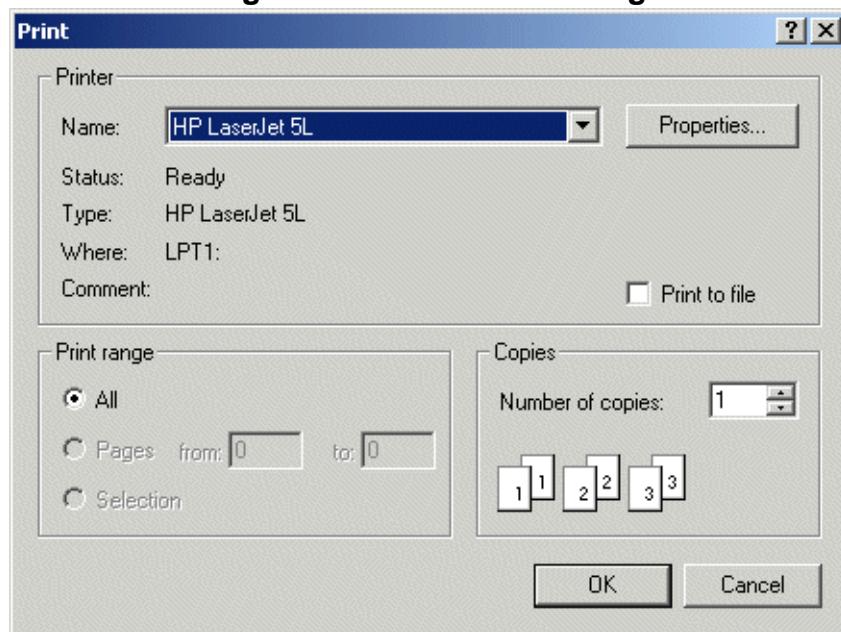
Figure 9-11. The Print Postscript Dialog



Print 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.

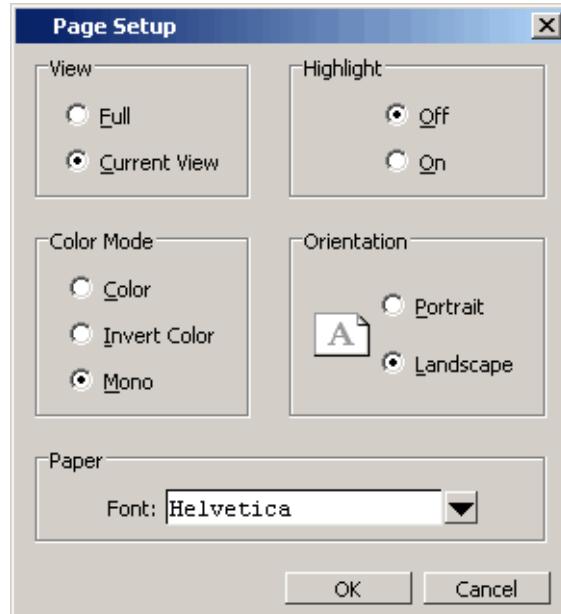
Figure 9-12. The Print Dialog



Configure Page Setup

With the Dataflow window active, select **File > Page setup** to open the Page Setup dialog. You can also open this dialog by clicking the Setup button in the Print Postscript dialog. This dialog allows you to configure page view, highlight, color mode, orientation, and paper options.

Figure 9-13. The Page Setup Dialog

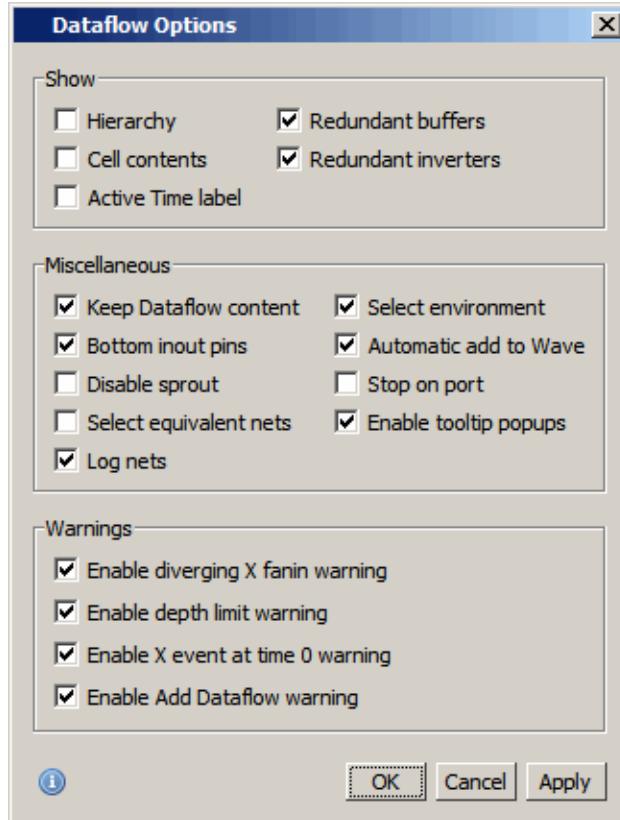


How Do I Configure Window Options?

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 9-14. Dataflow Options Dialog



Chapter 10

Source Window

This chapter discusses the uses of the Source Window for editing and debugging.

Opening Source Files	414
Changing File Permissions	414
Updates to Externally Edited Source Files	415
Navigating Through Your Design	415
Data and Objects in the Source Window	417
Object Values and Descriptions	417
Setting Simulation Time in the Source Window	418
Search for Source Code Objects	419
Debugging and Textual Connectivity	421
Hyperlinked Text	421
Highlighted Text in the Source Window	421
Drag Objects Into Other Windows	422
Breakpoints	423
Setting Individual Breakpoints in a Source File	423
Setting Breakpoints with the bp Command	424
Editing Breakpoints	425
Saving and Restoring Source Breakpoints	427
Setting Conditional Breakpoints	429
Run Until Here	431
Source Window Bookmarks	433
Setting and Removing Bookmarks	433
Source Window Preferences	433

Opening Source Files

You can open several file types in the Source window for editing and debugging.

Table 10-1. Open a Source File

To open from ...	Do the following ...
Main Menu Bar	1. Select File > Open 2. Select the file from the Open File dialog box
Other windows	Double-click objects in the Ranked, Call Tree, Design Unit, Structure, Objects, and other windows. The underlying source file for the object opens in the Source window, the indicator scrolls to the line where the object is defined, and the line is book marked.
Window context menu	Select View Source from context menus in the Message Viewer, Files, Structure, and other windows.
Command line	Enter the <code>edit <filename></code> command to open an existing file.
Create new file	1. Select File > New > Source 2. Select one of the file types from the drop down list.

Changing File Permissions **414**

Updates to Externally Edited Source Files **415**

Changing File Permissions

If a file is protected you must create a copy of the file or change file permissions in order to make changes to your source documents. Protected files can be edited in the Source window but the changes must be saved to a new file. To edit the original source document(s) you must change the read/write file permissions outside of ModelSim.

By default, files open in read-only mode even if the original source document file permissions allow you to edit the document. To change this behavior, set the **PrefSource(ReadOnly)** preference variable to 0. Refer to [Setting GUI Preferences](#) in the GUI Reference Manual for details on setting preference variables.

To change file permissions from the Source window:

Procedure

1. Right-click in the Source window
2. Select (un-check) **Read Only**.
3. Edit your file.
4. Save your file under a different name.

Updates to Externally Edited Source Files

The following preference variables control how ModelSim works with source files that have been edited outside of the simulator's Source window.

- PrefSource(CheckModifiedFiles) — Enables checking for source files for modification by an external editor.
- PrefSource(AutoReloadModifiedFiles) — Enables automatic reload of files that were modified by an external editor.

Refer to “[Setting GUI Preferences](#)” in the GUI Reference Manual for more information about changing simulator preferences.

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, similar to the functionality in most web browsers. This log allows you to easily navigate through your design hierarchy, returning to previous views and contexts for debugging purposes.

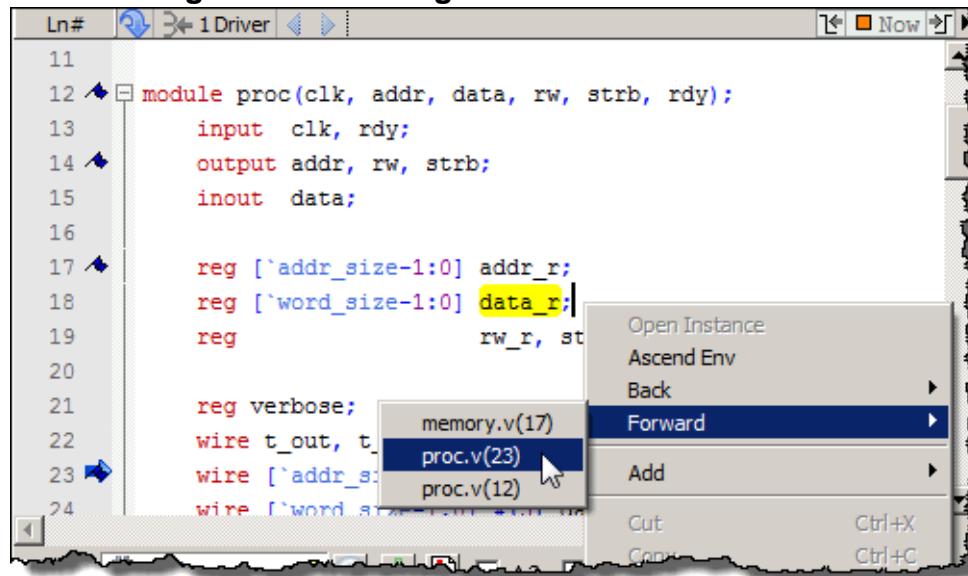
Procedure

1. Select then right-click an instance name in a source document.
2. Select one of the following options:
 - **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. Questa saves up to 50 context locations. This is not available if you have not changed your context. (Figure 10-1):

Figure 10-1. Setting Context from Source Files



Note

 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 and trace connectivity information during a simulation run.

Object Values and Descriptions	417
Setting Simulation Time in the Source Window.....	418
Search for Source Code Objects.....	419

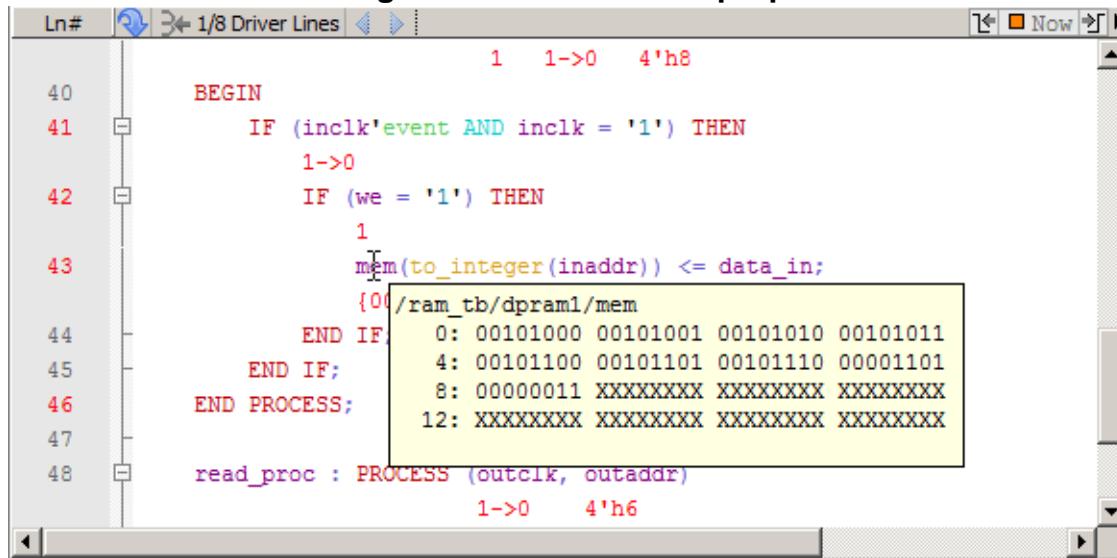
Object Values and Descriptions

You can obtain data on objects displayed in the Source window.

To determine the value and description of an object displayed in the Source window, do either of the following:

- 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 window popup. (Figure 10-2)

Figure 10-2. Examine Pop Up

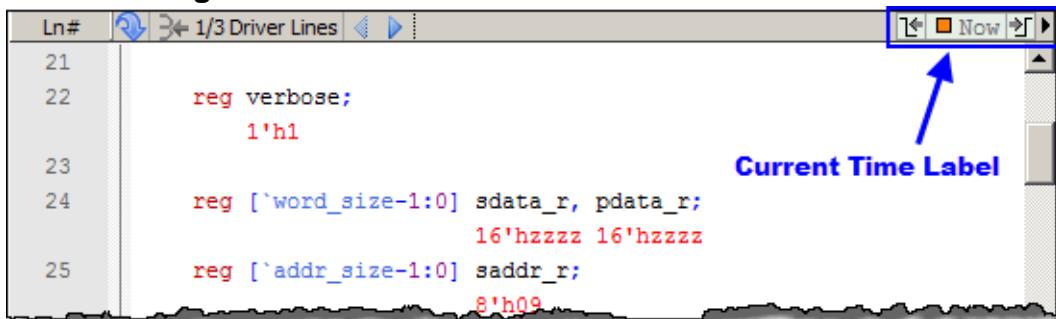


You can also invoke the **examine** and/or **describe** commands on the command line or in a DO file.

Setting Simulation Time in the Source Window

The Source window includes a time indicator in the top right corner that displays the current simulation time, the time of the active cursor in the Wave window, or a user-designated time.

Figure 10-3. Current Time Label in Source Window

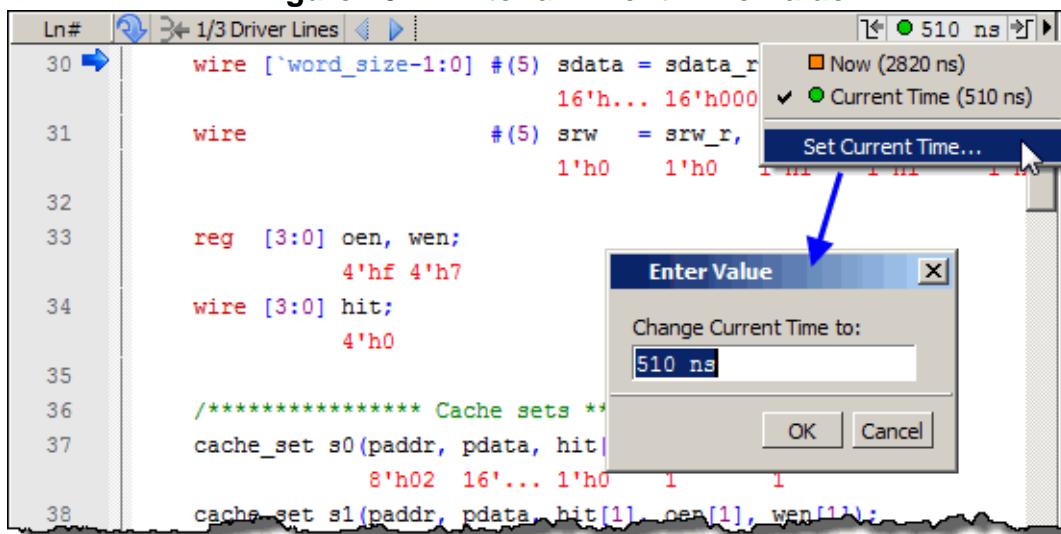


Procedure

You have several options for setting the time display in the Source window,

- Change time in the Current Time Label.
 - a. Click the time indicator to open the **Enter Value** dialog box (Figure 10-4).
 - b. Change the value to the starting time you want for the causality trace.
 - c. Click the **OK** button.

Figure 10-4. Enter an Event Time Value



Search for Source Code Objects

The Source window includes a Find function that allows you to search for specific code. You can search for one instance of a string, multiple instances, and the original declaration of a specified object.

Searching for One Instance of a String	419
Searching for All Instances of a String	419
Searching for the Original Declaration of an Object	420

Searching for One Instance of a String

You can search for one instance of a string. This search procedure starts from the current location in the open source file and finds the next instance of the specified search 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 in the current document and highlights it. Pressing the Enter key advances the search to the next instance of the string and so on through the source document.

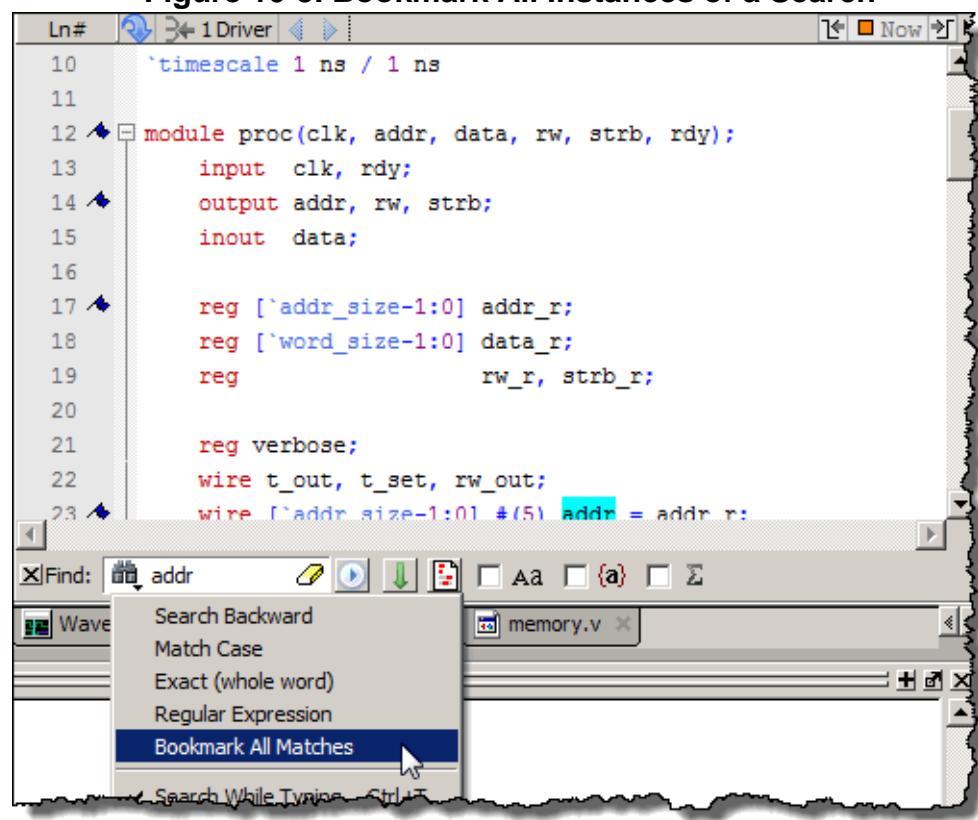
Searching for All Instances of a String

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 10-5. Bookmark All Instances of a Search



Searching for the Original Declaration of an Object

You can also search for the original declaration of an object, signal, parameter, and so on.

Procedure

1. Double click the object in many windows, including the Structure, Objects, and List windows. The Source window opens the source document containing the original declaration of the object and places a bookmark on that line of the document.
2. Double click a hyperlinked section of code in your source document. The source document is either opened or made the active Source window document and the declaration is highlighted briefly. Refer to [Hyperlinked Text](#) for more information about enabling hyperlinked text.

Debugging and Textual Connectivity

The Source window provides you with several tools for analyzing and debugging your code. You can jump to the declaration of an object with hyperlinked text from the Source and other windows. You can also determine the cause of any signal event or possible drivers or readers for a signal.

Hyperlinked Text.....	421
Highlighted Text in the Source Window	421
Drag Objects Into Other Windows	422

Hyperlinked Text

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.

Hyperlinked text is off by default. To turn hyperlinked text on or off in the Source window:

1. Make sure the Source window is the active window.
2. Select **Source > show 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).

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**

In these cases, the relevant text in the source code is shown with a persistent highlighting. To remove this highlighted display, right-click the Source window and choose **More > Clear**

Highlights. 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.

To produce a compile error that displays highlighted text in the Source window, do the following:

Procedure

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 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**.

Drag 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.

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.

Setting Individual Breakpoints in a Source File	423
Setting Breakpoints with the bp Command	424
Editing Breakpoints.....	425
Saving and Restoring Source Breakpoints	427
Setting Conditional Breakpoints	429
Run Until Here.....	431

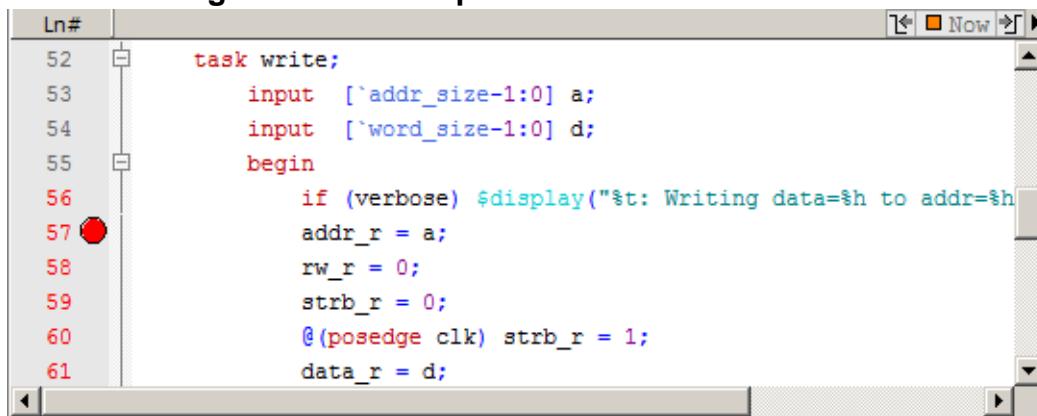
Setting Individual Breakpoints in a Source File

You can set individual file-line breakpoints in the Line number column of the Source Window.

Procedure

1. Click the line number column of the Source window next to a red line number and a red ball denoting a breakpoint will appear (Figure 10-6).
2. The breakpoint markers (red ball) are toggles. Click once to create the breakpoint; click again to disable or enable the breakpoint.

Figure 10-6. Breakpoint in the Source Window



Related Topics

[Setting GUI Preferences.](#)

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.

Procedure

1. Enter a bp command at the command line. For example, entering

bp top.vhd 147

2. sets a breakpoint in the source file *top.vhd* at line 147.

Related Topics

[bp](#)

Editing Breakpoints

There are several ways to edit a breakpoint in a source file.

- 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.

Using the Modify Breakpoints Dialog Box [425](#)

Deleting Individual Breakpoints [427](#)

Deleting Groups of Breakpoints [427](#)

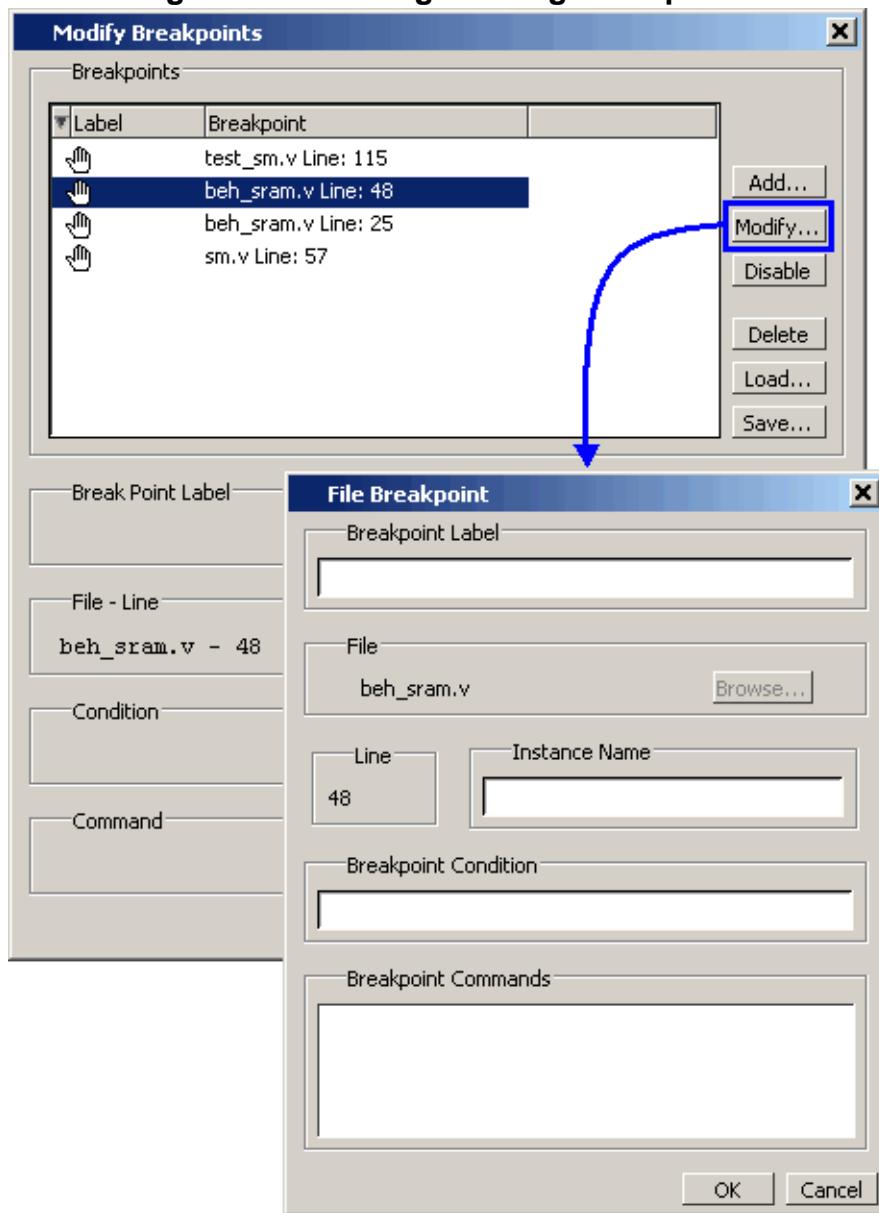
Using the Modify Breakpoints Dialog Box

The Modify Breakpoints dialog box provides a list of all breakpoints in the design organized by ID number.

Procedure

1. Select a file-line breakpoint from the list in the Breakpoints field.
2. Click **Modify**, which opens the **File Breakpoint** dialog box, [Figure 10-7](#).

Figure 10-7. Editing Existing Breakpoints



3. Fill out any of the following fields to edit the selected breakpoint:
 - **Breakpoint Label** — Designates a label for the breakpoint.
 - **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.

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 Source 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.

Procedure

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** or **add wave** 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.
-

Related Topics

[do](#)

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 10-8. 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
```

Setting a Breakpoint For a Specific Instance [430](#)

Setting a Breakpoint For a Specified Value of Any Instance [431](#)

Setting a Breakpoint For a Specific Instance

You can set a breakpoint for a value of specific instance from the GUI or from the command line.

Procedure

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 10-8](#)) 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

You can set a breakpoint for a specific value of any instance from the GUI or from the command line.

Procedure

From the command line enter:

```
bp simple.sv 13 -cond {this.cnt==8}
```

From the GUI:

- a. Right-click line 13 of the *simple.sv* source file.
- b. Select Edit Breakpoint 13 from the drop menu.
- c. Enter

```
this.cnt==8
```

in the **Breakpoint Condition** field of the **Modify Breakpoint** dialog box. (Refer to [Figure 10-7](#)) Note that the file name and line number are automatically entered.

Results

The simulation evaluates the expression at line 13 in the *simple.sv* source file ([Figure 10-8](#)), 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.

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.

To specify **Run Until Here**, right-click 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 **PrefSouce(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 “[Setting GUI Preferences](#)” in the GUI Reference Manual.

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..... 433

Setting and Removing Bookmarks

You can set bookmarks in the following ways.

Procedure

1. Set an individual bookmark.
 - a. Right-click in the Line number column on the line you want to bookmark then select **Add/Remove Bookmark**.
2. Set multiple bookmarks based on a search term refer to [Searching for All Instances of a String](#).
3. 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.

Source Window Preferences

You can customize a variety of settings for Source windows. You can change the appearance and behavior of the window in several ways.

Related Topics

[Customizing the Source Window and GUI Preferences.](#)

Chapter 11

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.

Note

 This version of ModelSim does not support the features in this section describing the use of SystemC.

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 Concepts	436
Signal Spy Reference	439

Signal Spy Concepts

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 [SystemVerilog System Tasks and Functions](#).

Table 11-1. 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()

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	436
Signal Spy Supported Types	437

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.

The following pathname lookup rules are used by the SignalSpy commands:

- If the name does not include a dataset name, then the current dataset is used.
- If the name does not start with a path separator, then the current context is used.

- If the name is a path separator followed by a name that is not the name of a top-level design unit, then the first top-level design unit in the design is used.
- For a relative name containing a hierarchical path, if the first object name cannot be found in the current context, then an upward search is done up to the top of the design hierarchy to look for a matching object name.
- If no objects of the specified name can be found in the specified context, then an upward search is done to look for a matching object in any visible enclosing scope up to an instance boundary. If at least one match is found within a given context, no (more) upward searching is done; therefore, some objects that may be visible from a given context will not be found when wildcards are used if they are within a higher enclosing scope.
- The wildcards '*' and '?' can be used at any level of a name except in the dataset name and inside of a slice specification.
- A wildcard character will never match a path separator. For example, /dut/* will match /dut/siga and /dut/clk. However, /dut* will not match either of those.

Related Topics

[VHDL Utilities Package \(util\)](#)

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]
```

Signal Spy Reference

The signal spy calls enumerated below include the syntax and arguments for the VHDL procedure, the Verilog task, and the SystemC function for each call.

disable_signal_spy	440
enable_signal_spy.....	442
init_signal_driver.....	444
init_signal_spy	448
signal_force.....	452
signal_release	456

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”.

Syntax

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>)
```

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.

Return Values

Nothing

Description

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.

Examples

See [init_signal_spy](#).

Related Topics

[init_signal_spy](#)

[enable_signal_spy](#)

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”.

Syntax

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>)
```

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.

Return Values

Nothing

Description

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.

Related Topics

[init_signal_spy](#)

[disable_signal_spy](#)

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.

Syntax

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>)
```

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 — transport

For 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.

Return Values

Nothing

Description

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.

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.

Examples

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
```

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;
```

Related Topics

[init_signal_spy](#)
[signal_force](#)
[signal_release](#)

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).

Syntax

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>)
```

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 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.

Return Values

Nothing

Description

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.

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.

Examples

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);
    elseif (enable_sig = '0')

        disable_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",0);
    end if;
end process spy_enable_disable;
    ...
end;
```

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
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
```

Related Topics

[init_signal_driver](#)
[signal_force](#)
[signal_release](#)
[enable_signal_spy](#)
[disable_signal_spy](#)

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/register bit/net, or SystemC signal (called the dest_object). This allows you to force signals, registers, bits of 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).

Syntax

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>)
```

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/bit of a 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 canceled.

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 canceled.

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 canceled.

- **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.

Return Values

Nothing

Description

A signal_force works the same as the [force](#) command with the exceptions that you cannot issue a repeating force. The force will remain on the signal until a signal_release, a force or noforce 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.

Limitations

- Verilog memories (arrays of registers) are not supported.

Examples

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
```

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/bkl1/reset", "1", 0 ns, freeze, open, 1);
        signal_force("/testbench/uut/bkl1/reset", "0", 40 ns, freeze, 2 ms,
1);
        wait;
    end process force_process;

    ...

end;
```

Related Topics

[init_signal_driver](#)
[init_signal_spy](#)
[signal_release](#)

signal_release

This reference section describes the following:

- **VHDL Procedure** — signal_release()
- **Verilog Task** — \$signal_release()
- **SystemC Function** — signal_release()

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.

The signal_release() call releases any force that was applied to an existing VHDL signal, SystemVerilog or Verilog register/register bit/net, or SystemC signal (called the dest_object). This allows you to release signals, registers, bits of 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).

Syntax

VHDL Syntax

```
signal_release(<dest_object>, <verbose>)
```

Verilog Syntax

```
$signal_release(<dest_object>, <verbose>)
```

SystemC Syntax

```
signal_release(<dest_object>, <verbose>)
```

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.

Return Values

Nothing

Examples

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;
```

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
```

Related Topics

[init_signal_driver](#)
[init_signal_spy](#)
[signal_force](#)

Chapter 12

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.

Common tasks you can perform with the Waveform Editor:

- Create waveforms using four predefined patterns: clock, random, repeater, and counter. Refer to [Accessing the Create Pattern Wizard](#).
- Edit waveforms with numerous functions including inserting, deleting, and stretching edges; mirroring, inverting, and copying waveform sections; and changing waveform values on-the-fly. Refer to [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. Refer to [Exporting Waveforms to a Stimulus File](#)

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	461
Using Waveform Editor Prior to Loading a Design	461
Using Waveform Editor After Loading a Design	462
Accessing the Create Pattern Wizard	463
Creating Waveforms with Wave Create Command	464
Editing Waveforms	464
Selecting Parts of the Waveform	466
Selection and Zoom Percentage	467
Auto Snapping of the Cursor	468
Stretching and Moving Edges.....	468
Simulating Directly from Waveform Editor	468
Exporting Waveforms to a Stimulus File	469
Driving Simulation with the Saved Stimulus File	470

Signal Mapping and Importing EVCD Files	470
Saving the Waveform Editor Commands	471

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 **461**

Using Waveform Editor After Loading a Design **462**

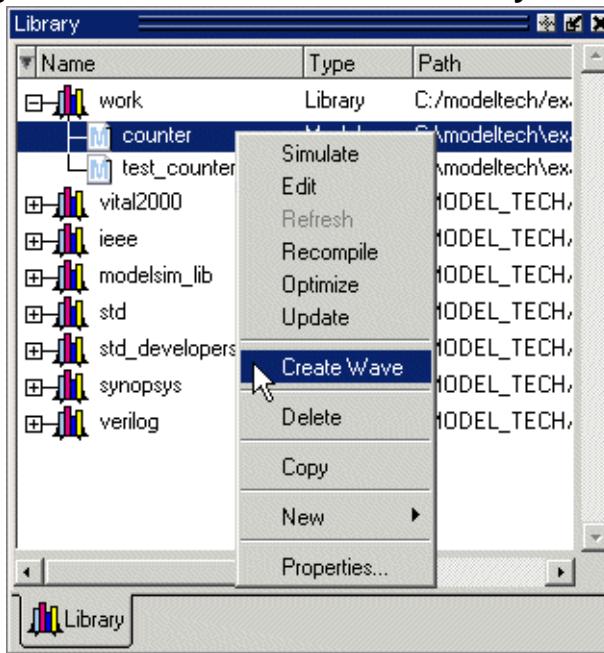
Using Waveform Editor Prior to Loading a Design

Here are the basic steps for using waveform editor prior to loading a design.

Procedure

1. Right-click a design unit on the Library Window and select Create Wave.

Figure 12-1. Waveform Editor: Library Window

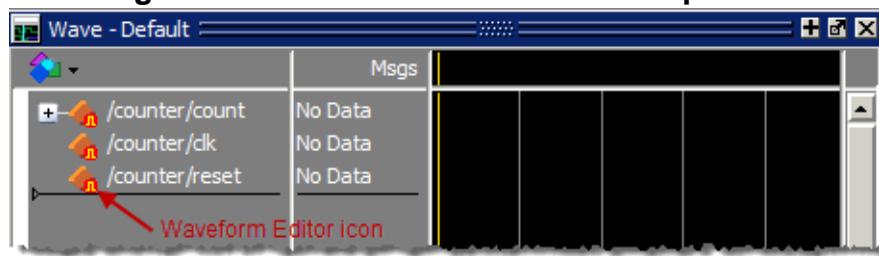


2. Edit the waveforms in the Wave window. See [Editing Waveforms](#) for more details.
3. Run the simulation (see [Simulating Directly from Waveform Editor](#)) or save the created waveforms to a stimulus file (see [Exporting Waveforms to a Stimulus File](#)).

Results

After the first step, a Wave window opens and displays signal names with the orange Waveform Editor icon ([Figure 12-2](#)).

Figure 12-2. Results of Create Wave Operation



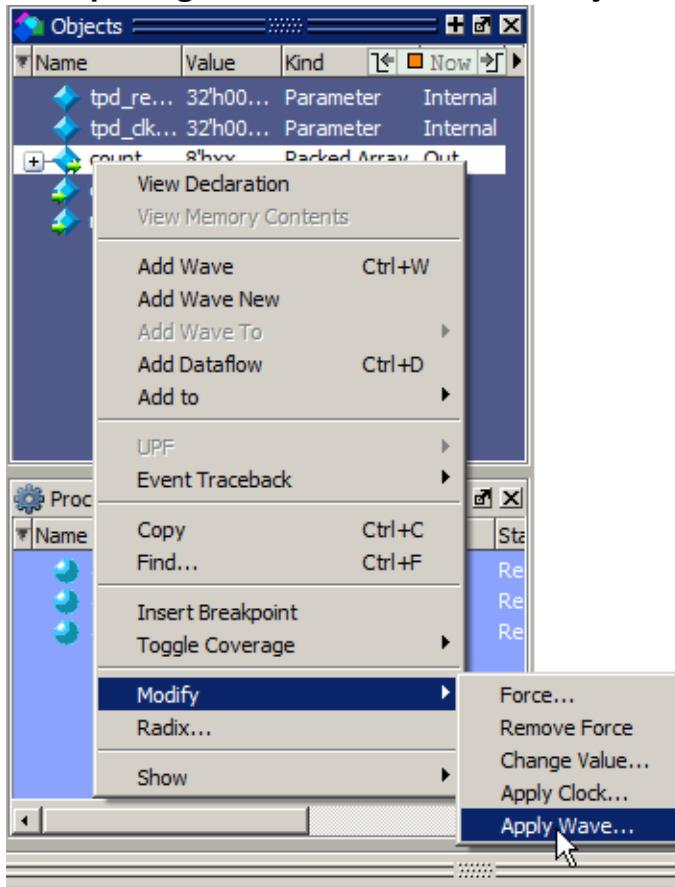
Using Waveform Editor After Loading a Design

Here are the basic steps for using waveform editor after loading a design.

Procedure

1. Right-click an object in the Objects window and select **Modify > Apply Wave**.

Figure 12-3. Opening Waveform Editor from Objects Windows



2. Use the Create Pattern wizard to create the waveforms (see “[Accessing the Create Pattern Wizard](#)”).
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](#)”).

Accessing the Create Pattern Wizard

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

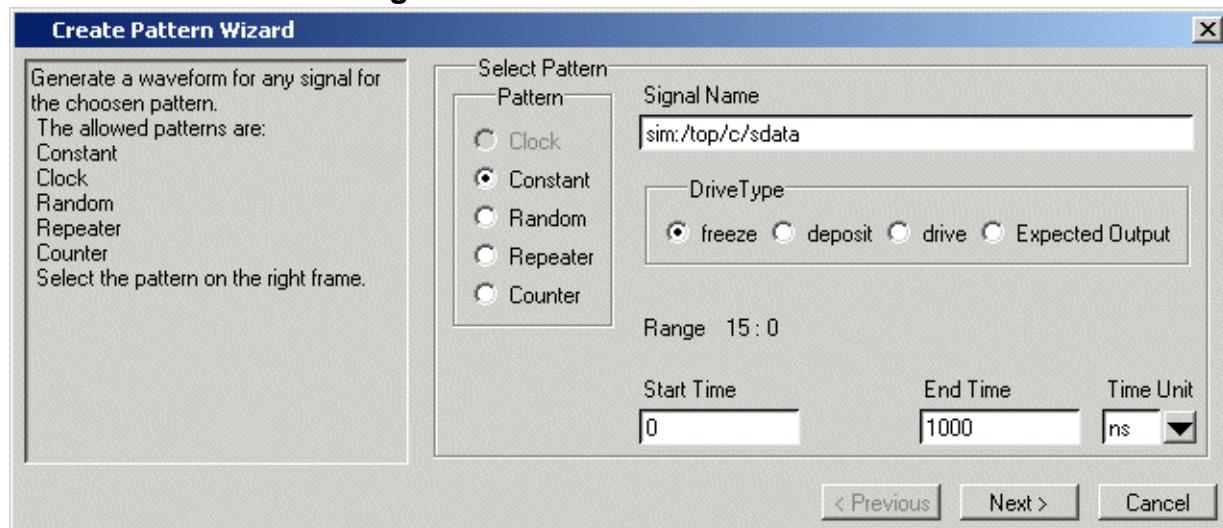
Procedure

1. Right-click an object in the Objects pane to open a popup menu.
2. Select **Modify > Apply Wave** from the popup menu.

Results

The Create Pattern Wizard opens to the initial dialog box shown in [Figure 12-4](#). Note that the Drive Type field is not present for input and output signals.

Figure 12-4. 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 12-1. 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.

Table 12-1. Signal Attributes in Create Pattern Wizard (cont.)

Pattern	Description
Random	Generates different patterns depending upon the seed value. Specify the type (normal or uniform), an initial value, and a seed value. If you do not 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.

Related Topics

[wave create \[ModelSim Command Reference Manual\]](#)

Editing Waveforms

You can edit waveforms interactively with menu commands, mouse actions, or by using the wave edit command.

Procedure

1. Create an editable pattern as described under [Accessing the Create Pattern Wizard](#).
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.

Figure 12-5. 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 the waveform and select a command from the **Wave** context menu.
5. The table below summarizes the editing commands that are available.

Table 12-2. 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
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

6. These commands can also be accessed via toolbar buttons.

Related Topics

[wave edit command and the Wave Edit Toolbar. \[ModelSim GUI Reference Manual\]](#)

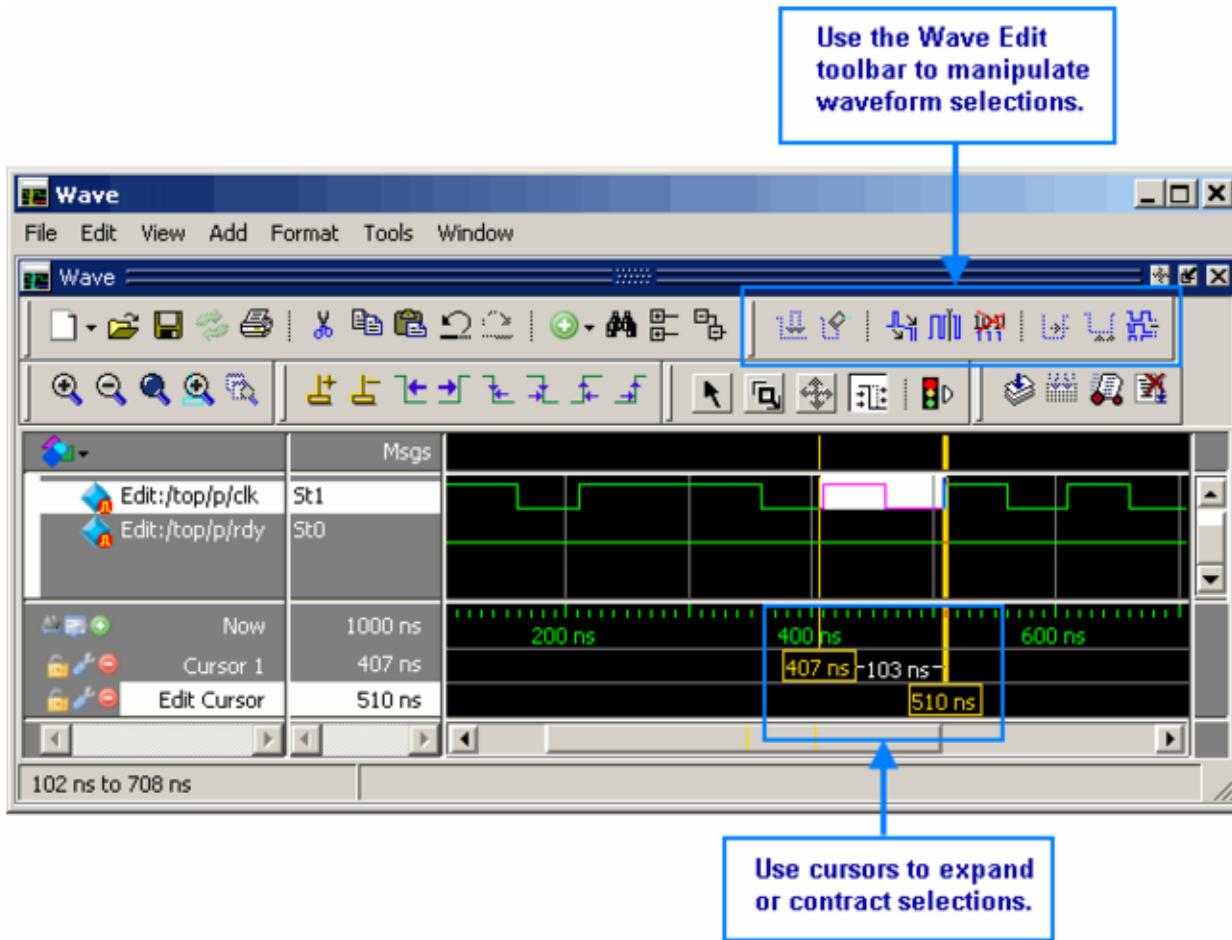
Selecting Parts of the Waveform

There are several methods for selecting edges or sections of a waveform. The table and graphic below describe the various options.

Table 12-3. Selecting Parts of the Waveform

Action	Method
Select a waveform edge	Click the waveform edge, or just to the right of the 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 12-6. Manipulating Waveforms with the Wave Edit Toolbar and Cursors



Selection and Zoom Percentage	467
Auto Snapping of the Cursor	468
Stretching and Moving Edges.....	468

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 and you should be able to select the range you want.

Related Topics

[Zooming the Wave Window Display](#)

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 12-4. 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.

Related Topics

[vsim \[ModelSim Command Reference Manual\]](#)

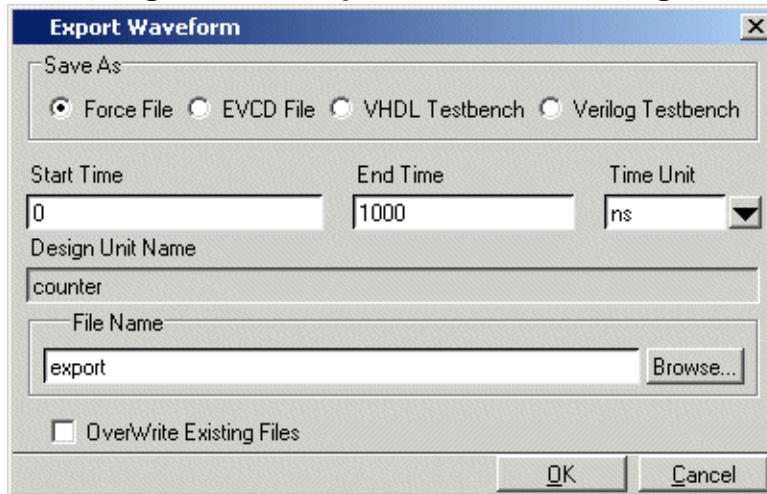
Exporting Waveforms to a Stimulus File

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.

Procedure

To save the waveform data, select **File > Export > Waveform** or use the wave export command.

Figure 12-7. Export Waveform Dialog



You can save the waveforms in four different formats:

Table 12-5. 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

Related Topics

[wave export \[ModelSim Command Reference Manual\]](#)

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 12-6. Examples for Loading a Stimulus File

Format	Loading example
Force format	vsim top -do mywaves.do
Extended VCD format ¹	vsim top -vcdstim mywaves.vcd
VHDL Testbench	vcom mywaves.vhd vsim mywaves
Verilog Testbench	vlog mywaves.v vsim mywaves

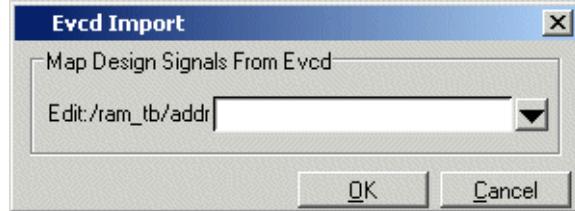
1. You can also use the **Import > EVCD** command from the Wave window. See below for more details on working with EVCD files.

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 12-8. 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.

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.

Procedure

Select **File > Save**.

Chapter 13

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.

Note

 SDF timing annotations can be applied only to your FPGA vendor's libraries; all other libraries will simulate without annotation.

Specifying SDF Files for Simulation	474
VHDL VITAL SDF	476
Verilog SDF	479
SDF for Mixed VHDL and Verilog Designs	489
Interconnect Delays	489
Disabling Timing Checks	489
Troubleshooting	491

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 `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	474
SDF Specification with the GUI	474
Errors and Warnings.....	475

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
```

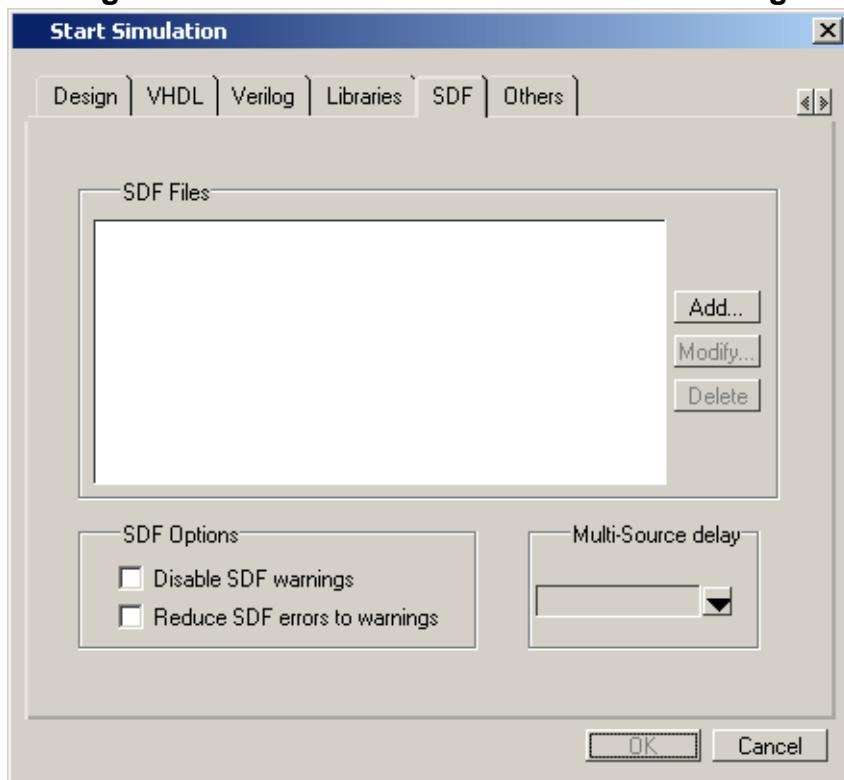
If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a test bench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this 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 13-1. 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 13-1](#)). 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.

VHDL VITAL SDF

The IEEE Std 1076.4-2000, *IEEE Standard for VITAL ASIC Modeling Specification* describes how cells must be written to support SDF annotation. The following summary reviews how SDF constructs are mapped to associated VHDL generic names, and may help you understand simulator error messages. VHDL SDF annotation works on VITAL cells only.

SDF to VHDL Generic Matching 477

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 13-1. 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
(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 **477**

Resolving Errors

If the simulator finds the cell instance but not the generic, 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 do not 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
```

Related Topics

[VITAL Usage and Compliance](#)

[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	480
SDF to Verilog Construct Matching	482

\$sdf_annotation

The \$sdf_annotation task annotates the design when it is called in the Verilog source code.

Syntax

```
$sdf_annotation
  ("<sdf_file>", [<instance>], ["<config_file>"], ["<log_file>"], ["<mtm_spec>"],
   [<scale_factor>], [<scale_type>]);
```

Arguments

- "<sdf_file>"
String that specifies the SDF file. Required.
- "<instance>"
Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the \$sdf_annotation 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_annotation("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotation("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 13-2. 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 cannot 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 13-3. 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 13-4. 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 13-5. 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 13-6. 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 13-7. 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 13-8. Matching SDF SETUPHOLD to Verilog

SDF	Verilog
(SETUPHOLD d (posedge clk) (5) (5))	\$setup(d, posedge clk, 0);
(SETUPHOLD d (posedge clk) (5) (5))	\$hold(posedge clk, d, 0);
(SETUPHOLD d (posedge clk) (5) (5))	\$setuphold(posedge clk, d, 0, 0);

- RECOVERY is matched to \$recovery:

Table 13-9. 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 13-10. 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 \$recrem:

Table 13-11. Matching SDF RECREM to Verilog

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5))	\$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5))	\$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5))	\$recrem(negedge reset, posedge clk, 0);

- SKEW is matched to \$skew:

Table 13-12. 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 13-13. Matching SDF WIDTH to Verilog

SDF	Verilog
(WIDTH (posedge clk) (5))	\$width(posedge clk, 0);

- PERIOD is matched to \$period:

Table 13-14. Matching SDF PERIOD to Verilog

SDF	Verilog
(PERIOD (posedge clk) (5))	\$period(posedge clk, 0);

- NOCHANGE is matched to \$nochange:

Table 13-15. 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, or Chapter 16 - Back Annotation Using the Standard Delay Format (SDF).

Retain Delay Behavior	485
Optional Edge Specifications	487
Optional Conditions	488
Rounded Timing Values	488

Retain Delay Behavior

The simulator processes RETAIN delays based on the definitions in an SDF file, the transition of the signal, and your use of command line arguments.

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:

- If only *rval1* is specified, *rval1* is used as the value of *rval2* and *rval3*.
- If *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 13-16 defines which delay is used depending on the transitions:

Table 13-16. 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	

Use `+vlog_retain_same2same` on the `vsim` command line to specify X insertion on outputs that do not change except when the causal inputs change. 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 13-17. 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 13-18. 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.

In other cases, the SDF file may contain more accurate data than the model can accommodate.

Table 13-19. 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 13-20. 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 13-21. 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 13-22. 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 only annotate Verilog cells.

Related Topics

[vsim](#)

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.

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.

Related Topics

[vsim](#)

Disabling Timing Checks

ModelSim offers a number of options for disabling timing checks on a global basis.

The table below provides a summary of commands and arguments to disable timing checks. You can pass some of the arguments to multiple commands, but which command you should pass them to depends on whether you intend to follow the two-step flow, or the three step flow. See the command and argument descriptions in the Reference Manual for further detail on individual commands.

You can pass the +nospecify and +notimingchecks arguments to either vlog, vopt, or vsim, but the correct choice depends on your flow choice.

- Three-step flow - pass these arguments to vopt.
- Two-step flow - pass these arguments to vsim.

When you pass +nospecify or +notimingchecks to vsim in the two-step flow, vsim will split off a vopt subprocess to handle the required actions.

- Passing to vlog - Passing these arguments on the vlog command line removes specify blocks or timing checks from the compiled Verilog cells and modules as if they were never present. As a result, running a timing simulation at a later point requires you to recompile the libraries without +nospecify or +notimingchecks. This procedure exists for backward compatibility and is not recommended usage.

Table 13-23. Disabling Timing Checks

Command and argument	Effect
vlog +notimingchecks	Removes timing checks from all Verilog cells/modules compiled into a library. The effect is as if the timing checks were never specified in those cells/modules.
vlog +nospecify	Removes specify path delays and timing checks for all instances in the specified Verilog design. The effect is as if the specify blocks were never specified in those cells/modules.
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	Removes Verilog timing checks and disables 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	Removes specify path delays and timing checks for all instances in the specified design

Troubleshooting

ModelSim provides a number of tools for troubleshooting designs that use SDF files.	
Specifying the Wrong Instance	491
Matching a Single Timing Check	492
Mistaking a Component or Module Name for an Instance Label	492
Forgetting to Specify the Instance	492
Reporting Unannotated Specify Path Objects	493
Failing to Find Matching Specify Module Path	494

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.

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.

Related Topics

[Instance Specification](#)

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.

Reporting Unannotated Specify Path Objects

ModelSim allows you to create a report about unannotated or partially-annotated specify path objects, path delays and timing checks, to better understand a design that uses SDF files.

Unannotated specify objects occur either because the SDF file did not contain any SDF statements targeting that object or (in a rather unusual situation) because all the values in the statement were null, as signified by a pair of empty parentheses "()".

The partial annotation of specify objects occurs when the SDF statements contain some null values.

Procedure

Add the `-sdfreport=<filename>` argument to your `vsim` command line.

Results

The Unannotated Specify Objects Report contains a list of objects that fit into any of the following three categories:

- Unannotated specify paths (UASP).
- Unannotated timing checks (UATC). This indicates either a single-value timing check that was not annotated or part of a \$setuphold or \$recrem that was not annotated.
- Incompletely-annotated specify path transition edges (IATE). This indicates that certain edges of a specify path, such as 0->1, 1->Z, and so on, were incompletely annotated.
- Incompletely annotated timing check (IATC)

The header of the report contains a full description of the syntax.

Failing to Find Matching Specify Module Path

The basic function of SDF annotation is to map SDF timing statements to the corresponding statements in the specify block of the module. The SDF annotation warning “Failed to find matching specify module path” indicates that there is a mismatch between the SDF data and the module timing information available to ModelSim.

The mismatch can occur because there is a difference in the Verilog module specify block description and the SDF data, or because of differences between the SDF statements and what the simulator sees for the cell timing data. The simulator's vsim/vlog command line arguments will control what vsim will have for timing information.

The [vsim/vlog](#) command arguments that affect the path delays are:

- vsim
 - +nospecify
- vlog
 - +delay_mode_unit
 - +delay_mode_zero
 - +nospecify

These are possible +define options that correspond to ifdef statements in the Verilog code. ifdef statements will affect what is actually compiled.

The +nospecify argument tells the simulator to ignore the specify block. The delay_mode_unit/ delay_mode_zero arguments force the path delay information to be a unit delay or a zero delay (again, ignoring the specify block).

Note

 You can convert the SDF annotation errors to warnings with the vsim option -sdfnoerror or +nosdferror.

To determine what timing is visible to the simulator, use the vsim command:

write timing <full_instance_pathname>

to return the timing data that vsim sees for a specific instantiation of a module.

To double check the vsim command line options from within vsim, use:

echo [StartupGetArgs]

To determine how a cell was compiled, this command returns the module name and library information for the instance:

context du <full_instance_pathname>

and this command returns the compile options for the cell:

vdir -l -lib <libname> <module_name>

If these commands do not resolve your problem, another issue could be conditional path delays. Specifically, because path delays can be very complex, ModelSim requires that path delays match identically.

For example:

((COND r1||r2 IOPATH a y (val) (val)))

does not match:

if (r2 || r1) (a *> y) = (val)

It is possible that the cell does not have a path delay declared. In that case, the vsim argument:

+sdf_iopath_to_prim_ok

forces the annotation of an SDF IOPATH statement to a primitive.

Chapter 14

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 VHDL designs. You can use these ModelSim VCD commands on Verilog and VHDL designs.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, contact your ASIC vendor.

Creating a VCD File	498
Using Extended VCD as Stimulus	500
VCD Commands and VCD Tasks	504
VCD File from Source to Output	506
VCD to WLF	509
Capturing Port Driver Data	509
Resolving Values	511

Creating a VCD File

ModelSim provides two general methods for creating a VCD file.

- [Four-State VCD File](#) — produces a four-state VCD file.
- [Extended VCD File](#) — produces an extended VCD (EVCD) file.

Both methods capture port driver changes unless you filter them out with optional command-line arguments.

Four-State VCD File	498
Extended VCD File	498
VCD Case Sensitivity.....	499

Four-State VCD File

This procedure produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information.

Procedure

1. Compile and load the design. For example:

```
cd <installDir>/examples/tutorials/verilog/basicSimulation  
vlib work  
vlog counter.v tcounter.v  
vsim test_counter
```

2. 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 as follows:

```
vcd file myvcdfile.vcd  
vcd add /test_counter/dut/*  
VSIM 3> runVSIM 4> quit -f
```

Results

Upon quitting the simulation, there will be a VCD file in the working directory.

Extended VCD File

This procedure produces an extended VCD (EVCD) file with variable changes in all states and strength information and port driver data.

Procedure

1. Compile and load the design. For example:

```
cd <installDir>/examples/tutorials/verilog/basicSimulation
vlib work
vlog counter.v tcounter.v
vsim test_counter
```

2. With the design loaded, specify the VCD file name and objects to add with the [vcd dumpports](#) command:

```
vcd dumpports -file myvcdfiile.vcd /test_counter/dut/*
run
VSIM 4> quit -f
```

Results

Upon quitting the simulation, there will be an extended VCD file called *myvcdfiile.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.

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 **500**

Replacing Instances with Output Values from a VCD File **502**

Port Order Issues..... **503**

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.

Procedure

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.

Examples

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
vsim test_counter +dumpports+nocollapse
vcd dumpports -file counter.vcd /test_counter/dut/*
run
quit -f
```

Next, rerun the counter without the test bench, using the **-vcdstim** argument:

```
vsim counter_replay -vcdstim counter.vcd
add wave /*
```

run 200

VHDL Adder

First, create the VCD file using **vcd dumpports**:

```
cd <installDir>/examples/vcd
vlib work
vcom gates.vhd adder.vhd stimulus.vhd
vsim testbench2 +dumpports+nocollapse
vcd dumpports -file addern.vcd /testbench2/uut/*
run 1000
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"
```

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
vsim top +dumpports+nocollapse
vcd dumpports -file proc.vcd /top/p/*
vcd dumpports -file cache.vcd /top/c/*
vcd dumpports -file memory.vcd /top/m/*
run 1000
quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
quit -f
vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
quit -f
vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
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.

Procedure

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.

Examples

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 -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 for 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 14-1. 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 (dumports system tasks). The table below maps the VCD dumports commands to their associated tasks.

Table 14-2. VCD Dumport Commands and System Tasks

VCD dumports commands	VCD system tasks
vcd dumports	\$dumports
vcd dumportsall	\$dumportsall
vcd dumportsflush	\$dumportsflush
vcd dumportslimit	\$dumportslimit
vcd dumportsoff	\$dumportsoff
vcd dumportson	\$dumportson

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 14-3](#) maps the VCD commands to their associated tasks. For additional details, please see the Verilog IEEE Std 1364-2005 specification.

Table 14-3. VCD Commands and System Tasks for Multiple VCD Files

VCD commands	VCD system tasks
vcd add -file <filename>	\$fdumpvars(levels, {, module_or_variable } ¹ , filename)

Table 14-3. VCD Commands and System Tasks for Multiple VCD Files (cont.)

VCD commands	VCD system tasks
vcd checkpoint <filename>	\$fdumpall(filename)
vcd files <filename>	\$fdumpfile(filename)
vcd flush <filename>	\$fdumpflush(filename)
vcd limit <filename>	\$fdumplimit(filename)
vcd off <filename>	\$fdumpoff(filename)
vcd on <filename>	\$fdumpon(filename)

1. denotes an optional, comma-separated list of 0 or more modules or variables

Compressing Files with VCD Tasks..... 505

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 code shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL Source Code **506**

VCD Simulator Commands **506**

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
11:07:43 2003                         1!
$Send                                #150
$version                             0!
    <Tool> Version                  #200
<version>                            1!
$Send                                $dumpoff
$timescale                           x!
    1ns                                x"
$Send                                x#
$scope module                        x$
shifter_mod $end                   x%
$var wire 1 ! clk                   x&
$Send                                x'
$var wire 1 " reset                x(
$Send                                x)
$var wire 1 # data_in              x*
$Send                                x+
$var wire 1 $ q [8]                 x,
$Send                                $end
$var wire 1 % q [7]                 #300
$Send                                $dumpon
$var wire 1 & q [6]                 1!
$Send                                0"
$var wire 1 ' q [5]                 1#
$Send                                0$
$var wire 1 ( q [4]                 0%
$Send                                0&
$var wire 1 ) q [3]                 0'
$Send                                0(
$var wire 1 * q [2]                 0)
$Send                                0*
$var wire 1 + q [1]                 0+
$Send                                1,
$var wire 1 , q [0]                 $end
$Send                                #350
$upscope $end                         0!
$enddefinitions $end                  #400
#0                                    1!
$dumpvars                            1+
0!                                    #450
1"                                    0!
0#                                    #500
0$                                    1!
0%                                    1*
0&                                    #550
0'                                    0!
0(                                    #600
0)                                    1!
0*                                    1)
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 14-4 shows the driver states recorded as TSSI states if the direction is known.

Table 14-4. 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 14-5. 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)
F three-state (input and output unconnected)

Table 14-5. State When Direction is Unknown (cont.)

Unknown direction
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 14-6. 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	511
When force Command is Used	511
Extended Data Type for VHDL (vl_logic).....	512
Ignoring Strength Ranges	513

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 shown in the following table to dump values to the VCD file. When the port is an inout, the direction cannot be determined.

Table 14-7. VCD Values When Force Command is Used

Value forced on net	Port Direction		
	input	output	inout
0	D	L	0
1	U	H	1

Table 14-7. VCD Values When Force Command is Used (cont.)

Value forced on net	Port Direction		
	input	output	inout
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 dumpports](#) command
- Use an optional argument to `$dumpports` (see [Extended \\$dumpports Syntax](#) below)
- Use the `+dumpports+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 14-8. 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 vcd dumpports

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 14-1. 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 14-9](#) is sample driver data.

Table 14-9. 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 15

Tcl and 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 a command you need, you can use Tcl to create your own commands.

Tcl Features	516
Tcl Command Syntax	517
Simulator State Variables.....	524
List Processing	527
Simulator Tcl Commands.....	529
Tcl Examples	531
DO Files	534

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 DO files

Tcl References **516**

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 Command Syntax

The following eleven rules define the syntax and semantics of the Tcl language.

- 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.
- 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.
- Words of a command are separated by white space (except for newlines, which are command separators).
- 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.
- 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.
- 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.
- 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 is not 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.

- 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 15-1](#) lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

Table 15-1. 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 is not in braces or quotes.
\\\	Backslash (“\”)

Table 15-1. Tcl Backslash Sequences (cont.)

Sequence	Value
\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.

1. 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.
2. 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.
3. 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	520
Command Substitution	520
Command Separator	521
Multiple-Line Commands	521
Evaluation Order	521
Tcl Relational Expression Evaluation	521
Variable Substitution	522
System Commands	522
ModelSim Replacements for Tcl Commands	522

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?
```

Arguments

None

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.

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 scripts 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 placement of the opening brace is important: If it is not there, the Tcl scanner assumes the command is complete and will try to execute what it has up to that point, which is not what you intend.

```
if { [exa sig_a] == "0011ZZ"} {
    echo "Signal value matches"
    do do_1.do
} else {
    echo "Signal value fails"
    do do_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, do not 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 gives you an error. For instance:

```
if {[exa var_2] == 001Z}...
```

gives an error.

```
if {[exa var_2] == "001Z"}...
```

does not give an error.

- 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]
```

ModelSim Replacements for Tcl Commands

For complete information on Tcl commands, select **Help > Tcl Man Pages**.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands, as shown in [Table 15-2](#).

Table 15-2. Changes to ModelSim Commands

Previous ModelSim command	Command changed to (or replaced by)
continue	run with the -continue 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.
source	vsources
wave	add wave

Related Topics

[Simulator GUI Preferences](#) [[ModelSim GUI Reference Manual](#)]

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 file scripts. The variables are referenced in commands by prefixing the name with a dollar sign (\$).

Table 15-3. Simulator State Variables

Variable	Description
architecture	Returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, Returns an empty string.
argc	Returns the total number of parameters passed to the current script.
argv	Returns the list of parameters (arguments) passed to the vsim command line.
configuration	Returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration.
delta	Returns the number of the current simulator iteration.
entity	Returns the name of the top-level VHDL entity or Verilog module currently being simulated.
library	Returns the library name for the current region.
MacroNestingLevel	Returns the current depth of script call nesting.
n	Represents a script parameter, where n can be an integer in the range 1-9.
Now	Returns the current simulation time with time units (for example, 110,000 ns). Note: the returned value contains a comma inserted between thousands.
now	Returns the current simulation time with or without time units—depending on the setting for time resolution, as follows: <ul style="list-style-type: none"> When time resolution is a unary unit (such as 1ns, 1ps, 1fs), 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), 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	Returns the current simulation time resolution.

Table 15-3. Simulator State Variables (cont.)

Variable	Description
RNGTrace	<p>Returns 1 if the RNG trace feature is enabled, and returns 0 if the feature is disabled. The value of the RNGTrace variable is associated with the vsim -rngtrace argument. For example, the following command yields the RNGTrace value “1”:</p> <pre>vsim -rngtrace</pre> <p> Note: To enable or disable the RNG trace feature, use the following command:</p> <pre>set RNGTrace <0 1></pre>
RNGTraceFile	<p>Returns the filename to which the RNG trace related output is redirected. If the value of this variable is “”(empty string), then the RNG trace output is redirected to the Transcript window. The value of the RNGTraceFile variable is associated with the vsim -rngtrace=<filename> argument. For example, the following command yields the RNGTraceFile value “myfile.txt”:</p> <pre>vsim -rngtrace myfile.txt</pre> <p> Note: To change the filename to which the RNG trace related output is redirected, use the following command:</p> <pre>set RNGTraceFile "<filename>"</pre>
RNGTraceOpt	<p>Returns the configuration option string for the RNG trace generated output. The value of the RNGTraceOpt variable is associated with the vsim -rngtraceopt argument. For example, the following command yields the RNGTraceOpt value “showciid”:</p> <pre>vsim -rngtraceopt=showciid</pre> <p> Note: To change the configuration option string for the RNG trace generated output, use the following command:</p> <pre>set RNGTraceOpt "<option_string>"</pre>
RNGTraceClassFilter	<p>Returns the RNG trace output class filter string. The value of the RNGTraceClassFilter variable is associated with the vsim -rngtraceclassfilter argument. For example, the following command yields the RNGTraceClassFilter value “TFoo”:</p> <pre>vsim -rngtraceclassfilter=TFoo</pre> <p> Note: To change the RNG trace output class filter string, use the following command:</p> <pre>set RNGTraceClassFilter "<class_filter_string>"</pre>

Table 15-3. Simulator State Variables (cont.)

Variable	Description
RNGTraceProcessFilter	Returns the RNG trace output scope filter string. The value of the RNGTraceProcessFilter variable is associated with the vsim -rngtraceprocessfilter argument. For example, the following command yields the RNGTraceProcessFilter value “/top”: <pre>vsim -rngtraceprocessfilter=/top</pre>  Note: To change the RNG trace output scope filter string, use the following command: <pre>set RNGTraceProcessFilter "<scope_filter_string>"</pre>
SolveInfo	Returns the current debug output configuration string. The value of the SolveInfo variable is associated with the vsim -solveinfo argument. For example, the following command yields the SolveInfo value, “l,gentc”. <pre>vsim -solveinfo=1,gentc</pre>  Note: To change the debug output configuration string, use the following command: <pre>set SolveInfo <option_string></pre>
SolveInfoClassFilter	Returns the current debug output class filter string. The value of the SolveInfoClassFilter variable is associated with the vsim -solveinfoclassfilter argument. For example, the following command yields the SolveInfoClassFilter value, “TBar,TBaz”. <pre>vsim -solveinfo=2,gentc -solveinfoclassfilter=TBar,TBaz</pre>  Tip: To change the debug output class filter string, use the following command: <pre>set SolveInfoClassFilter <class_filter_string></pre>

Referencing Simulator State Variables [526](#)

Special Considerations for the now Variable [527](#)

Referencing Simulator State Variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign (\$). For example, to use the now and resolution variables in an echo command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

The time is 12390 ps 10ps.

If you do not want the dollar sign to denote a simulator variable, precede it with a “\”. For example, \\$now will not be interpreted as the current simulator time.

Special Considerations for the now Variable

For the when command, 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.

Related Topics

[when \[ModelSim Command Reference Manual\]](#)

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 the following table.

Table 15-4. 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.

Related Topics

[when \[ModelSim Command Reference Manual\]](#)

Simulator Tcl Commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided in the following table.

Table 15-5. 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
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 **530**

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.

Time Conversion Tcl Commands	530
Time Relations Tcl Commands	530
Tcl Time Arithmetic Commands	531

Time Conversion Tcl Commands

The following table provides Tcl time conversion commands.

Table 15-6. 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

Time Relations Tcl Commands

The following table provides Tcl time relation commands.

Table 15-7. 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

Table 15-7. Tcl Time Relation Commands (cont.)

Command	Description
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] } {
    ...
}
```

Tcl Time Arithmetic Commands

The following table provides commands for performing arithmetic operations on time.

Table 15-8. 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

This section provides examples of Tcl command usage.

- **Tcl while Loop**

This example uses the Tcl while loop to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

```
set b [list]
set i [expr {[llength $a] - 1}]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

- **Tcl for Command**

This example uses the Tcl for command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

```
set b [list]
for {set i [expr {[llength $a] - 1}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

- **Tcl foreach Command**

This example uses the Tcl foreach command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way (the foreach command iterates over all of the elements of a list):

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

- **Tcl break Command**

This example shows a list reversal as above, this time aborting on a particular element using the Tcl break command:

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

- **Tcl continue Command**

This example is a list reversal that skips a particular element by using the Tcl continue command:

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

- **Access and Transfer System Information**

This example works in UNIX only. In a Windows environment, the Tcl exec command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

(in VHDL source):

```
signal datime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in a DO file script):

```
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [clock format [clock seconds]]
    force -deposit datime $s
    if {do_the_echo} {
        echo "New time is [examine -value datime]"
    }
}

bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date
```

- **Tcl Used to Specify Compiler Arguments**

This example specifies the compiler arguments and lets you compile any number of files.

```
set Files [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    set lappend Files $1
    shift
}
eval vcom -93 -explicit -noaccel std_logic_arith $Files
```

- **Tcl Used to Specify Compiler Arguments—Enhanced**

This example is an enhanced version of the last one. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the script assumes your VHDL files have a *.vhd* file extension.

```
set vhdfFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    if {[string match *.vhd $1]} {
        lappend vhdfFiles $1
    } else {
        lappend vFiles $1
    }
    shift
}
if {[llength $vhdfFiles] > 0} {
    eval vcom -93 -explicit -noaccel std_logic_arith $vhdfFiles
}
if {[llength $vFiles] > 0} {
    eval vlog $vFiles
}
```

DO Files

ModelSim 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	534
Using Parameters with DO Files	535
Deleting a File from a .do Script.	535
Making Script Parameters Optional	536
Breakpoint Flow Control in Nested DO files.	537
Useful Commands for Handling Breakpoints and Errors	539
Error Action in DO File Scripts	539
Using the Tcl Source Command with DO Files	540

Creating DO Files

You can create DO file scripts, like any other Tcl script, by doing one of the following.

Procedure

1. Type the required commands in any editor and save the file with the extension `.do`.
2. Save the transcript as a DO file (refer to “[Saving a Transcript File as a DO file](#)” in the GUI Reference Manual).
3. 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.
4. All “event watching” commands (for example, `onbreak`, `onerror`, and so forth) must be placed before `run` commands within the script in order to take effect.
5. The following is a simple DO file script that was saved from the transcript. It is used in the dataset exercise in the ModelSim Tutorial. This script 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 file scripts by using parameters. Parameters specify values that are passed to the corresponding parameters \$1 through \$9 in the script. For example say the DO file “*testfile*” contains the line bp \$1 \$2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do testfile design.vhd 127
```

There is no limit to the number of parameters that can be passed to DO file scripts, 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.

Procedure

1. The Tcl file command

```
file delete myfile.log
```

2. will delete the file “myfile.log.”

3. You can also use the transcript file command to perform a deletion:

```
transcript file ()
transcript file my file.log
```

4. 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 Script Parameters Optional

If you want to make DO file script parameters optional (that is, be able to specify fewer parameter values with the do command than the number of parameters referenced in the DO file script), 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.

- Specifying Files to Compile With argc DO File Scripts

This script 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. }
}
```

- Specifying Compiler Arguments With DO File Scripts

This script 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 std_logic_arith $Files
```

- Specifying Compiler Arguments With Scripts — Enhanced

This DO file script 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 script assumes your VHDL files have a *.vhd* file extension.

```
variable vhdfFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdfFilesExist 0
set vFilesExist 0
for {set x 1} {$x <= $nbrArgs} {incr x} {
    if {[string match *.vhd $1]} {
        set vhdfFiles [concat $vhdfFiles $1]
        set vhdfFilesExist 1
    } else {
        set vFiles [concat $vFiles $1]
        set vFilesExist 1
    }
    shift
}
if {$vhdfFilesExist == 1} {
    eval vcom -93 -explicit -noaccel std_logic_arith $vhdfFiles
}
if {$vFilesExist == 1} {
    eval vlog $vFiles}
```

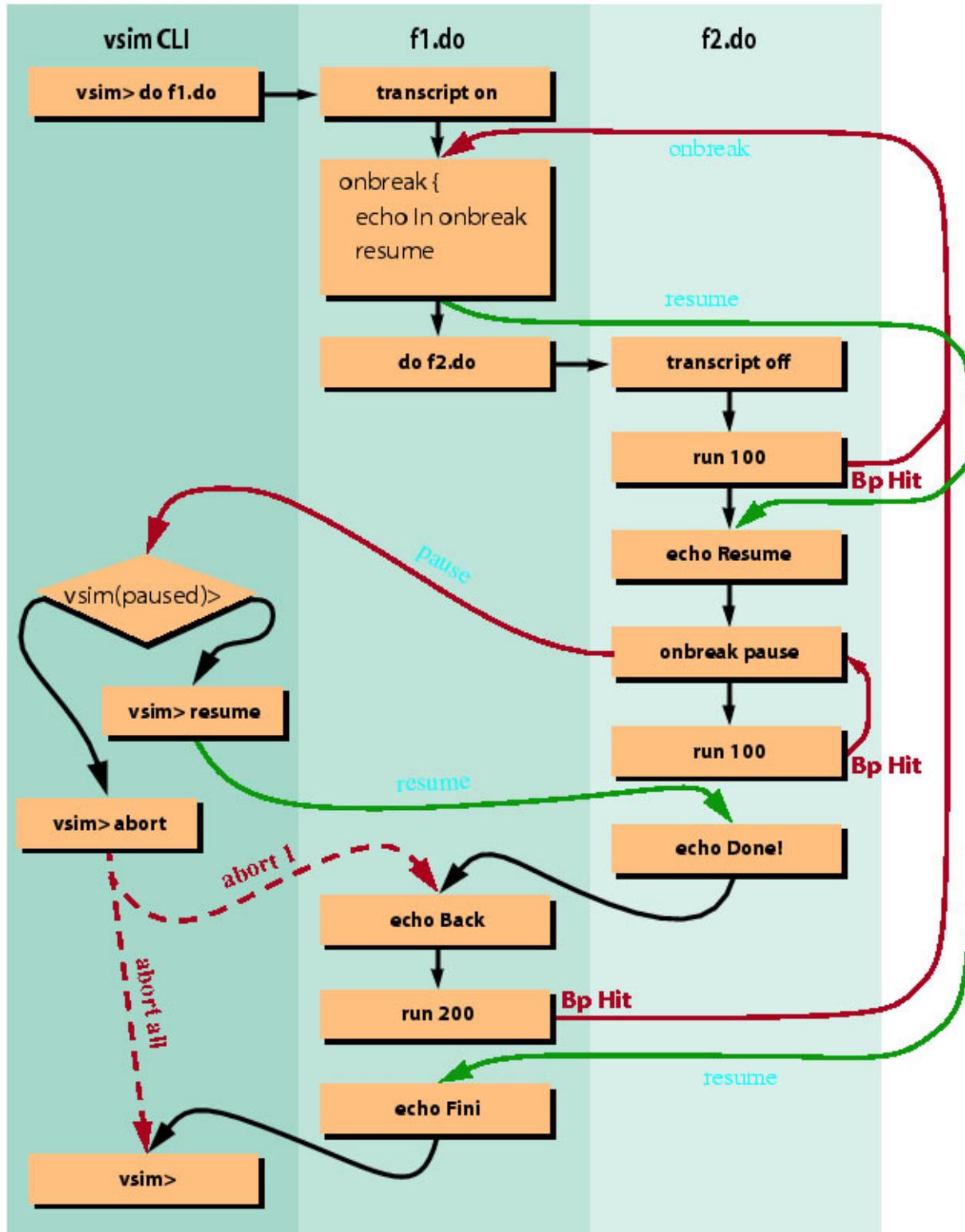
Related Topics

[Simulator State Variables](#)

Breakpoint Flow Control in Nested DO files

The following diagram shows how control flows from one DO file to another and out to the command line interface for input from the user.

Figure 15-1. Breakpoint Flow Control in Nested DO Files



Useful Commands for Handling Breakpoints and Errors

If you are executing a script when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the script and returns control to the command line. The commands in the following table may be useful for handling such events. (Any other legal command may be executed as well.)

Table 15-9. Commands for Handling Breakpoints and Errors in DO scripts

Command	Result
<code>run -continue</code>	continue as if the breakpoint had not been executed, completes the run that was interrupted
<code>onbreak</code>	specify a command to run when you hit a breakpoint within a script
<code>onElabError</code>	specify a command to run when an error is encountered during elaboration
<code>onerror</code>	specify a command to run when an error is encountered within a script
<code>status</code>	get a traceback of nested script calls when a script is interrupted
<code>abort</code>	terminate a script once the script has been interrupted or paused
<code>pause</code>	cause the script to be interrupted; the script can be resumed by entering a resume command via the command line

You can also set the `OnErrorDefaultAction` Tcl variable to determine what action ModelSim takes when an error occurs.

Error Action in DO File Scripts

If a command in a script returns an error, ModelSim does the following:

1. If an `onerror` command has been set in the 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 script 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 script running as it hits breakpoints. Add an `onbreak abort` command to the DO file if you want to exit the script and update the Source window.

Appendix A modelsim.ini Variables

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. This chapter covers the contents and modification of the *modelsim.ini* file.

Organization of the modelsim.ini File	546
Making Changes to the modelsim.ini File	546
Editing modelsim.ini Variables	547
Overriding the Default Initialization File	547
The Runtime Options Dialog	548
Variables.....	552
AccessObjDebug	558
AddPragmaPrefix	559
AllowCheckpointCpp	560
AmsStandard	561
AppendClose	562
AssertFile	563
BatchMode	564
BatchTranscriptFile	565
BindAtCompile	566
BreakOnAssertion	567
BreakOnMessage	568
CheckPlusargs	569
CheckpointCompressMode	570
CheckSynthesis	571
ClassDebug	572
CommandHistory	573
common	574
CompilerTempDir	575
ConcurrentFileLimit	576
CreateDirFor FileAccess	577
CreateLib	578
data_method	579
DatasetSeparator	580
DefaultForceKind	581
DefaultLibType	582
DefaultRadix	583
DefaultRadixFlags	584
DefaultRestartOptions	585
DelayFileOpen	586

displaymsgmode	587
DpiOutOfTheBlue	588
DumpportsCollapse	589
EnumBaseInit	590
error	591
ErrorFile	592
Explicit	593
fatal	594
FlatLibPageSize	595
FlatLibPageDeletePercentage	596
FlatLibPageDeleteThreshold	597
floatfixlib	598
ForceSigNextIter	599
ForceUnsignedIntegerToVHDLInteger	600
FsmImplicitTrans	601
FsmResetTrans	602
FsmSingle	603
FsmXAssign	604
GCThreshold	605
GCThresholdClassDebug	606
GenerateFormat	607
GenerousIdentifierParsing	608
GlobalSharedObjectList	609
Hazard	610
ieee	611
IgnoreError	612
IgnoreFailure	613
IgnoreNote	614
IgnorePragmaPrefix	615
ignoreStandardRealVector	616
IgnoreVitalErrors	617
IgnoreWarning	618
ImmediateContinuousAssign	619
IncludeRecursionDepthMax	620
InitOutCompositeParam	621
IterationLimit	622
keyring	623
LargeObjectSilent	624
LargeObjectSize	625
LibrarySearchPath	626
MessageFormat	627
MessageFormatBreak	628
MessageFormatBreakLine	629
MessageFormatError	630
MessageFormatFail	631

MessageFormatFatal	632
MessageFormatNote	633
MessageFormatWarning	634
MixedAnsiPorts	635
modelsim_lib	636
MsgLimitCount	637
msgmode	638
mtiAvm	639
mtiOvm	640
MultiFileCompilationUnit	641
NoCaseStaticError	642
NoDebug	643
NoDeferSubpgmCheck	644
NoIndexCheck	645
NoOthersStaticError	646
NoRangeCheck	647
note	648
NoVitalCheck	649
NumericStdNoWarnings	650
OldVHDLConfigurationVisibility	651
OldVhdlForGenNames	652
OnFinish	653
Optimize_1164	654
osvvm	655
PathSeparator	656
PedanticErrors	657
PreserveCase	658
PrintSimStats	659
Quiet	660
RequireConfigForAllDefaultBinding	661
Resolution	662
RunLength	663
SeparateConfigLibrary	664
Show_BadOptionWarning	665
Show_Lint	666
Show_source	667
Show_VitalChecksWarnings	668
Show_Warning1	669
Show_Warning2	670
Show_Warning3	671
Show_Warning4	672
Show_Warning5	673
ShowFunctions	674
ShutdownFile	675
SignalForceFunctionUseDefaultRadix	676

SignalSpyPathSeparator	677
SmartDbgSym.....	678
Startup.....	679
Stats.....	680
std	682
std_developerskit	683
StdArithNoWarnings	684
suppress.....	685
SuppressFileTypeReg	686
sv_std	687
SvExtensions.....	688
SVFileSuffixes	692
Svlog	693
SVPrettyPrintFlags	694
synopsys	696
SyncCompilerFiles	697
toolblock	698
TranscriptFile	699
UnbufferedOutput.....	700
UndefSyms	701
UserTimeUnit	702
UVMControl	703
verilog	704
Veriuser.....	705
VHDL93	706
VhdlSeparatePduPackage	707
VhdlVariableLogging	708
vital2000	709
vlog95compat	710
WarnConstantChange	711
warning	712
WaveSignalNameWidth	713
wholefile	714
WildcardFilter	715
WildcardSizeThreshold	716
WildcardSizeThresholdVerbose	717
WLFCacheSize	718
WLFCollapseMode	719
WLFCompress	720
WLFDeleteOnQuit	721
WLFFileLock	722
WLFFilename	723
WLFOptimize	724
WLFSaveAllRegions	725
WLFSimCacheSize.....	726

WLFSIZELIMIT	727
WLFTIMELIMIT	728
WLFUPDATEINTERVAL	729
WLFUSETHREADS	730
WRAPCOLUMN	731
WRAPMODE	732
WRAPWSCOLUMN	733
Commonly Used modelsim.ini Variables	734
Common Environment Variables	734
Hierarchical Library Mapping	735
Creating a Transcript File	735
Using a Startup File	736
Turn Off Assertion Messages	736
Turn Off Warnings from Arithmetic Packages	736
Force Command Defaults	737
Restart Command Defaults	737
VHDL Standard	737
Delay Opening VHDL Files	738

Organization of the modelsim.ini File

The *modelsim.ini* file 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 [DefineOptionset]** section allows you to define groups of commonly used command line arguments. Refer to “[Optionsets](#)” in the Reference Manual for more information.
- **The [vsim]** section contains variables that control the simulator.
- **The [msg_system]** section contains variables that control the severity of notes, warnings, and errors that come from vcom, vlog and vsim.
- **The [utils]** section contains variables that control utility functions in the tool environment.

The System Initialization chapter contains descriptions of [Environment Variables](#).

Making Changes to the modelsim.ini File	546
Editing modelsim.ini Variables	547
Overriding the Default Initialization File	547
The Runtime Options Dialog	548

Making Changes to the modelsim.ini File

When first installed, the *modelsim.ini* file is protected as a Read-only file. In order to make and save changes to the file, you must first turn off the Read-only attribute in the *modelsim.ini* Properties dialog box.

Procedure

1. Navigate to the location of the *modelsim.ini* file:
<install directory>/modelsim.ini
2. Right-click the *modelsim.ini* file and choose **Properties** from the popup menu. This displays the *modelsim.ini* Properties dialog box.
3. Uncheck the Attribute: **Read-only**.
4. Click**OK**.

5. To protect the *modelsim.ini* file after making changes, repeat the preceding steps, but at Step 3, check the **Read-only** attribute.

Editing modelsim.ini Variables

Once the Read-only attribute has been turned off, you can make changes to the values of the variables in the file.

The syntax for variables in the file is as follows:

<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.

Refer to “[Initialization Sequence](#)” for the *modelsim.ini* file search precedence.

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-1. Commands for Overriding the Default Initialization File

Simulator Commands	Compiler Commands	Utility Commands
<code>vsim</code>	<code>vcom</code>	<code>vdel</code>
	<code>vlog</code>	<code>vdir</code>
		<code>vgencomp</code>
		<code>vmake</code>

5. Refer to the `<command> -modelsimini` argument description for further information.

The Runtime Options Dialog

The **Runtime Options** dialog box writes changes to the active *modelsim.ini* file that affect the current session. To access, choose **Simulate > Runtime Options** in the Main window. The dialog contains three tabs - Defaults, Message Severity, and WLF Files.

If the read-only attribute for the *modelsim.ini* file is turned off, the changes are saved, and affect all future sessions. Refer to [Making Changes to the modelsim.ini File](#).

Figure A-1. Runtime Options Dialog: Defaults Tab

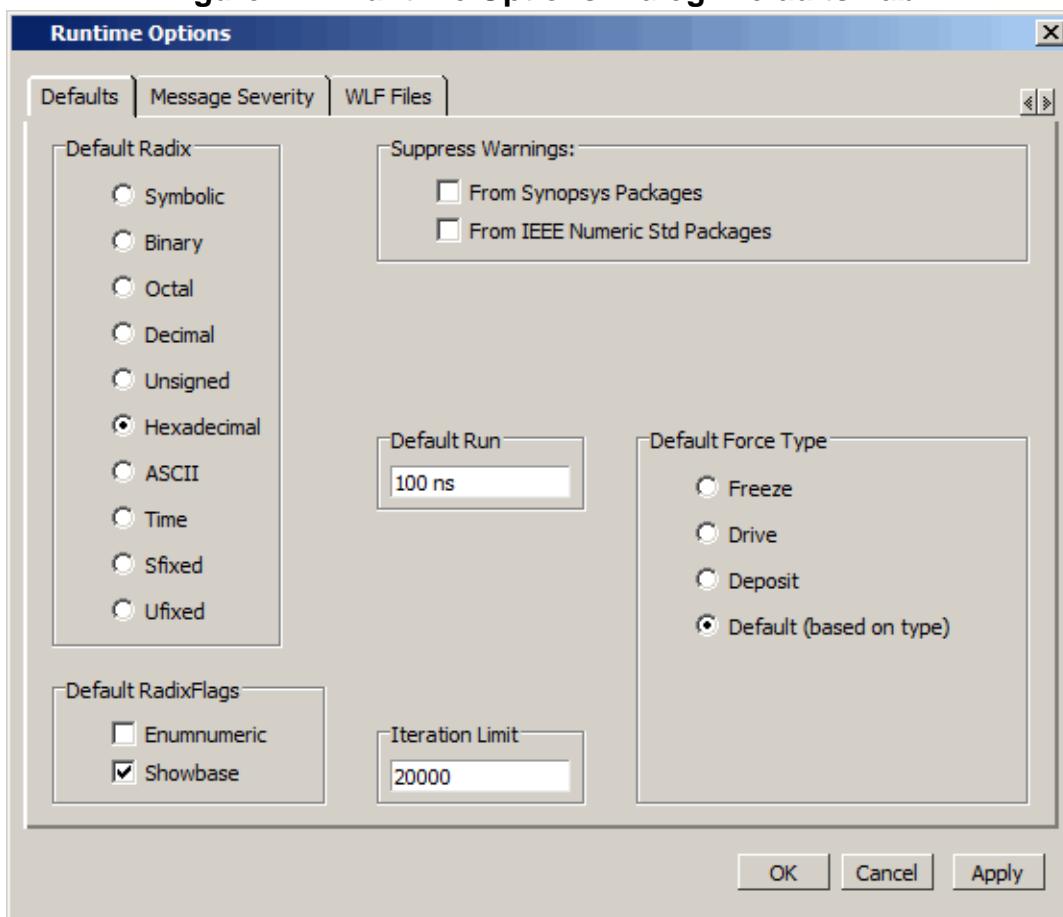


Table A-2. Runtime Option Dialog: Defaults Tab Contents

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, List, and Wave windows, as well as the Source window in the source annotation view. You can override this variable with the radix command.
Default Radix Flags	Displays SystemVerilog enums as numbers rather than strings. This option overrides the global setting of the default radix. You can override this variable with add list -radixenumsymbolic .

Table A-2. Runtime Option Dialog: Defaults Tab Contents (cont.)

Option	Description
Suppress Warnings	From Synopsys Packages suppresses warnings generated within the accelerated Synopsys std_arith packages. The corresponding <i>modelsim.ini</i> variable is StdArithNoWarnings . 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 .
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: Message Severity Tab

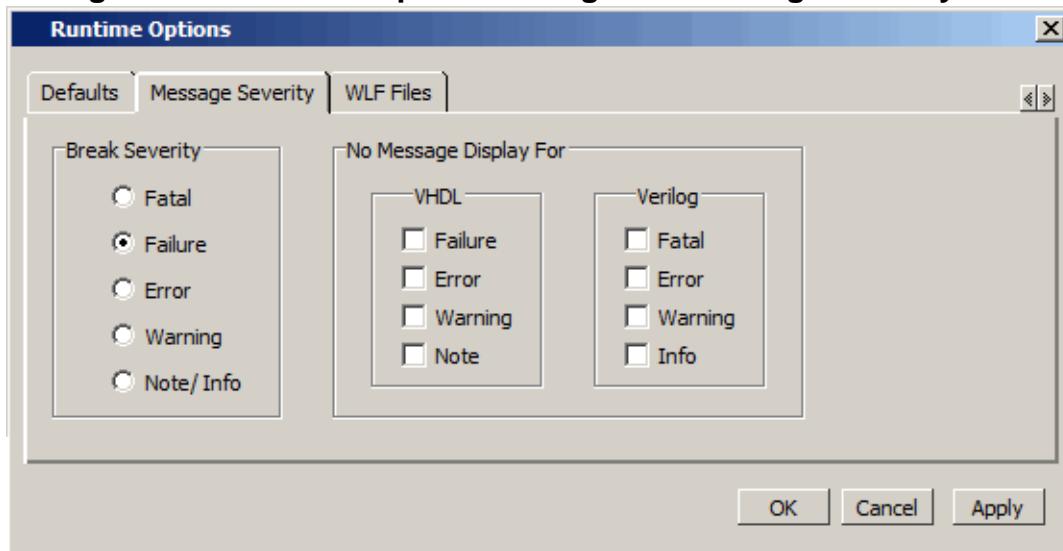


Table A-3. Runtime Option Dialog: Message Severity Tab Contents

Option	Description
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

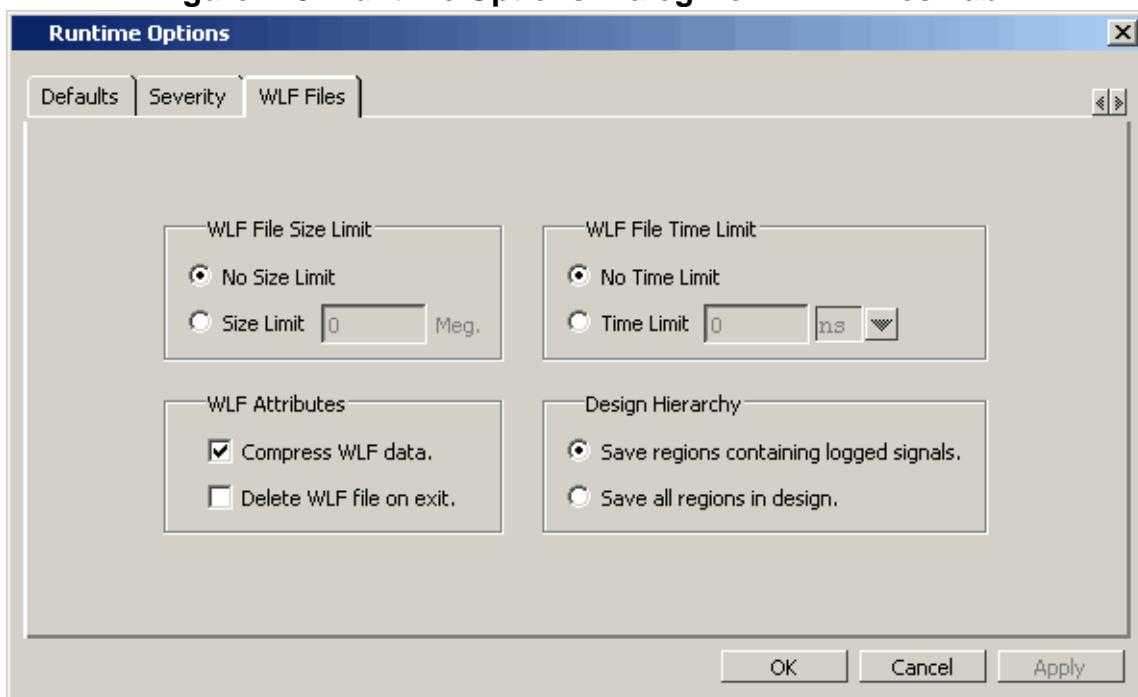


Table A-4. Runtime Option Dialog: WLF Files Tab Contents

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 .

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.

AccessObjDebug	558
AddPragmaPrefix	559
AllowCheckpointCpp	560
AmsStandard	561
AppendClose	562
AssertFile	563
BatchMode	564
BatchTranscriptFile	565
BindAtCompile	566
BreakOnAssertion	567
BreakOnMessage	568
CheckPlusargs	569
CheckpointCompressMode	570
CheckSynthesis	571
ClassDebug	572
CommandHistory	573
common	574
CompilerTempDir	575
ConcurrentFileLimit	576
CreateDirFor FileAccess	577
CreateLib	578
data_method	579
DatasetSeparator	580

DefaultForceKind	581
DefaultLibType	582
DefaultRadix	583
DefaultRadixFlags	584
DefaultRestartOptions	585
DelayFileOpen	586
displaymsgmode	587
DpiOutOfTheBlue	588
DumpportsCollapse	589
EnumBaseInit	590
error	591
ErrorFile	592
Explicit	593
fatal	594
FlatLibPageSize	595
FlatLibPageDeletePercentage	596
FlatLibPageDeleteThreshold	597
floatfixlib	598
ForceSigNextIter	599
ForceUnsignedIntegerToVHDLInteger	600
FsmImplicitTrans	601
FsmResetTrans	602
FsmSingle	603
FsmXAssign	604
GCThreshold	605
GCThresholdClassDebug	606
GenerateFormat	607
GenerousIdentifierParsing	608
GlobalSharedObjectList	609
Hazard	610
ieee	611
IgnoreError	612
IgnoreFailure	613
IgnoreNote	614
IgnorePragmaPrefix	615

ignoreStandardRealVector	616
IgnoreVitalErrors	617
IgnoreWarning	618
ImmediateContinuousAssign	619
IncludeRecursionDepthMax	620
InitOutCompositeParam	621
IterationLimit	622
keyring	623
LargeObjectSilent	624
LargeObjectSize	625
LibrarySearchPath	626
MessageFormat	627
MessageFormatBreak	628
MessageFormatBreakLine	629
MessageFormatError	630
MessageFormatFail	631
MessageFormatFatal	632
MessageFormatNote	633
MessageFormatWarning	634
MixedAnsiPorts	635
modelsim_lib	636
MsgLimitCount	637
msgmode	638
mtiAvm	639
mtiOvm	640
MultiFileCompilationUnit	641
NoCaseStaticError	642
NoDebug	643
NoDeferSubpgmCheck	644
NoIndexCheck	645
NoOthersStaticError	646
NoRangeCheck	647
note	648
NoVitalCheck	649
NumericStdNoWarnings	650

OldVHDLConfigurationVisibility	651
OldVhdlForGenNames	652
OnFinish	653
Optimize_1164	654
osvvm	655
PathSeparator	656
PedanticErrors	657
PreserveCase	658
PrintSimStats	659
Quiet	660
RequireConfigForAllDefaultBinding	661
Resolution	662
RunLength	663
SeparateConfigLibrary	664
Show_BadOptionWarning	665
Show_Lint	666
Show_source	667
Show_VitalChecksWarnings	668
Show_Warning1	669
Show_Warning2	670
Show_Warning3	671
Show_Warning4	672
Show_Warning5	673
ShowFunctions	674
ShutdownFile	675
SignalForceFunctionUseDefaultRadix	676
SignalSpyPathSeparator	677
SmartDbgSym	678
Startup	679
Stats	680
std	682
std_developerskit	683
StdArithNoWarnings	684
suppress	685
SuppressFileTypeReg	686

sv_std	687
SvExtensions	688
SVFileSuffixes	692
Svlog	693
SVPrettyPrintFlags	694
synopsys	696
SyncCompilerFiles	697
toolblock	698
TranscriptFile	699
UnbufferedOutput	700
UndefSyms	701
UserTimeUnit	702
UVMControl	703
verilog	704
Veriuser	705
VHDL93	706
VhdlSeparatePduPackage	707
VhdlVariableLogging	708
vital2000	709
vlog95compat	710
WarnConstantChange	711
warning	712
WaveSignalNameWidth	713
wholefile	714
WildcardFilter	715
WildcardSizeThreshold	716
WildcardSizeThresholdVerbose	717
WLFCacheSize	718
WLFCollapseMode	719
WLFCompress	720
WLFDeleteOnQuit	721
WLFFileLock	722
WLFFilename	723
WLFOptimize	724
WLFSaveAllRegions	725

WLFSimCacheSize	726
WLFSizeLimit	727
WLFTimeLimit	728
WLFUpdateInterval	729
WLFUseThreads	730
WrapColumn	731
WrapMode	732
WrapWSColumn	733

AccessObjDebug

Section [vsim]

This variable enables logging a VHDL access variable—both the variable value and any access object that the variable points to during the simulation.

Syntax

AccessObjDebug = {**0** | **1**}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying [vsim-accessobjdebug](#) or -noaccessobjdebug.

Description

Display-only names such as [10001] take on a different form, as follows:

- the initial character, @
- the name of the access type or subtype
- another @
- a unique integer N that represents the sequence number (starting with 1) of the objects of that designated type that were created with the VHDL allocator called new.

For example: @ptr@1

By default, this variable is turned off. This means that while access variables themselves can be logged and displayed in the various display windows, any access objects that they point to will not be logged. The value of an access variable, which is the “name” of the access object it points to, is suitable only for displaying, and cannot be used as a way for a command to reference it.

For example, for an access variable “v1” that designates some access object, the value of “v1” will show as [10001]. This name cannot be used as input to any command that expects an object name, it is for display only; but it is a unique identifier for any access object that the design may produce. This value replaces any hexadecimal address-based ‘value’ that may have been displayed in prior versions of ModelSim.

AddPragmaPrefix

Section [vcom], [vlog]

This variable enables recognition of synthesis 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 pragmas are honored.

Syntax

AddPragmaPrefix = <*prefix*>

Arguments

- The arguments are described as follows:
 - <*prefix*> — Specifies a user defined string where the default is no string, indicated by quotation marks ("").

AllowCheckpointCpp

Section [vsim]

This variable enables/disables support for checkpointing foreign C++ libraries.

Syntax

AllowCheckpointCpp 1|0

Arguments

- The arguments are described as follows:
 - **1** — Turn on the support.
 - **0** — (default) Turn off the support.

Description

This variable may be overridden with the vsim -allowcheckpointcpp command. It is not supported on Windows platforms.

AmsStandard

Section [vcom]

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.

Syntax

AmsStandard = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying [vcom {-amsstd | -noamsstd}](#).

Related Topics

[vcom \[ModelSim Command Reference Manual\]](#)

[Setting Environment Variables](#)

AppendClose

Section [vsim]

This variable immediately closes files previously opened in the APPEND mode as soon as there is either an explicit call to file_close, or when the file variable's scope is closed. You can override this variable by specifying vsim -noappendclose at the command line.

Syntax

AppendClose = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0**
Off
 - **1**
(default) On

When set to zero, the simulator will not immediately close files opened in the APPEND mode. Subsequent calls to file_open in APPEND mode will therefore not require operating system interaction, resulting in faster performance. If your designs rely on files to be closed and completely written to disk following calls to file_close, because they perform operations on the files outside the simulation, this enhancement could adversely impact those operations. In those situations, turning this variable on is not recommended.

AssertFile

Section [vsim]

This variable specifies an alternative file for storing VHDL assertion messages.

Syntax

AssertFile = <filename>

Arguments

- The arguments are described as follows:
 - <filename> — Any valid file name containing assertion messages, where the default name is *assert.log*.
You can override this variable by specifying [vsim-assertfile](#).

Description

By default, assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file. If the AssertFile variable is specified, all assertion messages will be stored in the specified file, not in the transcript.

Related Topics

[TranscriptFile](#)

[Creating a Transcript File](#)

BatchMode

Section [vsim]

This variable runs batch (non-GUI) simulations. The simulations are executed via scripted files from a Windows command prompt or UNIX terminal and do not provide for interaction with the design during simulation. The BatchMode variable will be ignored if you use the -batch, -c, -gui, or -i options to vsim. Refer to BatchMode for more information about running batch simulations.

Syntax

BatchMode = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Runs the simulator in interactive mode. Refer to vsim **-i** for more information.
 - **1** — Enables batch simulation mode.

You can also enable batch mode by specifying [vsim -batch](#).

Related Topics

[BatchTranscriptFile](#)

[TranscriptFile](#)

[vsim \[ModelSim Command Reference Manual\]](#)

BatchTranscriptFile

Section [vsim]

This variable enables automatic creation of a transcript file when the simulator runs in batch mode. All transcript data is sent to stdout when this variable is disabled and the simulator is run in batch mode (BatchMode = 1, or vsim -batch).

Syntax

BatchTranscriptFile = <filename>

Arguments

- The arguments are described as follows:
 - <filename> — Any string representing a valid filename where the default is *transcript*.
You can override this variable by specifying [vsim -logfile <filename>](#), [vsim -nolog](#).

Related Topics

[BatchMode](#)

[TranscriptFile](#)

[transcript file \[ModelSim Command Reference Manual\]](#)

[vsm \[ModelSim Command Reference Manual\]](#)

BindAtCompile

Section [vcom]

This variable instructs ModelSim to perform VHDL default binding at compile time rather than load time.

Syntax

BindAtCompile = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying [vcom {-bindAtCompile | -bindAtLoad}](#).

Related Topics

[Default Binding](#)

[RequireConfigForAllDefaultBinding](#)

BreakOnAssertion

Section [vsim]

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.

Syntax

BreakOnAssertion = {0 | 1 | 2 | 3 | 4}

Arguments

- The arguments are described as follows:
 - **0** — Note
 - **1** — Warning
 - **2** — Error
 - **3** — **(default) Failure**
 - **4** — Fatal

Related Topics

[The Runtime Options Dialog](#)

[Tcl Command Syntax](#)

BreakOnMessage

Section [vsim]

This variable stops the simulator when the severity of a tool message is equal to or higher than the value set for the variable.

Note

 The elaboration phase ignores this variable.

Syntax

BreakOnMessage = {0 | 1 | 2 | 3}

Arguments

- The arguments are described as follows:

The default behavior is to not break on any level of message.

- **0** — Note
- **1** — Warning
- **2** — Error
- **3** — Fatal

Examples

- Instruct the simulator to break on any message of level Note or higher.

```
BreakOnMessage = 0
```

- Instruct the simulator to break on any Fatal message.

```
BreakOnMessage = 3
```

CheckPlusargs

Section [vsim]

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.

Syntax

CheckPlusargs = {0 | 1 | 2}

Arguments

- The arguments are described as follows:
 - **0** — (default) Ignore
 - **1** — Issues a warning and simulates while ignoring.
 - **2** — Issues an error and exits.

CheckpointCompressMode

Section [vsim]

This variable specifies that checkpoint files are written in compressed format.

Syntax

CheckpointCompressMode = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — Off
 - **1** — (default) On

CheckSynthesis

Section [vcom]

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.

Syntax

CheckSynthesis = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying [vcom -check_synthesis](#).

ClassDebug

Section [vsim]

This variable enables visibility into and tracking of class instances.

Syntax

ClassDebug = {0 | 1}

Arguments

- The arguments are described as follows:

- **0** — (default) Off
- **1** — On

You can override this variable by specifying **vsim -classdebug**.

CommandHistory

Section [vsim]

This variable specifies the name of a file in which to store the Main window command history.

Syntax

CommandHistory = <filename>

Arguments

- The arguments are described as follows:

- <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 (;).

common

Specifies a file of common encryption directives for the vencrypt and vhencrypt commands.

Usage

common=<file_name>

Arguments

- Specifies a file containing common encryption directives used across the tools. This file is optional. If you use this variable, then its presence requires at least one "toolblock" directive within the specified file. Directives such as "author", "author_info", and "data_method" as well as the common block license specification are placed in this file.

CompilerTempDir

Section [vcom]

This variable specifies a directory for compiler temporary files instead of “work/_temp.”

Syntax

CompilerTempDir = <*directory*>

Arguments

- The arguments are described as follows:
 - <*directory*> — Any user defined directory where the default is work/_temp.

ConcurrentFileLimit

Section [vsim]

This variable controls the number of VHDL files open concurrently. This number should be less than the current limit setting for maximum file descriptors.

Syntax

ConcurrentFileLimit = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any non-negative integer where 0 is unlimited and 40 is the default.

Related Topics

[Syntax for File Declaration](#)

CreateDirFor FileAccess

Section [vsim]

This variable controls whether the Verilog system task \$fopen will create a non-existent directory when opening a file in append (a), or write (w) modes.

Syntax

CreateDirFor FileAccess = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Related Topics

[New Directory Path With \\$fopen](#)

CreateLib

Section [vcom], [vlog]

This variable enables automatic creation of missing work libraries.

Syntax

CreateLib = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Description

You can use the -nocreatelib option for the vcom or vlog commands to override this variable and stop automatic creation of missing work libraries (which reverts back to the 10.3x and earlier version behavior).

data_method

Specifies the length of the symmetric session key used by the encryption commands vencrypt and vhencrypt.

Usage

```
data_method={aes128 | aes192 | aes256}
```

Arguments

- The session key is a symmetric key that the vencrypt and vhencrypt commands randomly generate for each protected region. The data_method argument sets the length of the session key.
 - **aes128** — AES key size of 128. This is the default.
 - **aes192** — AES key size of 192.
 - **aes256** — AES key size of 256.

DatasetSeparator

Section [vsim]

This variable specifies the dataset separator for fully-rooted contexts, for example:

Syntax

DatasetSeparator = <character>

Arguments

- The arguments are described as follows:
 - <character> — Any character except special characters, such as backslash (\), brackets ({ }), and so forth, where the default is a colon (:).

Description

sim:/top

The variable for DatasetSeparator must not be the same character as the [PathSeparator](#) variable, or the [SignalSpyPathSeparator](#) variable.

DefaultForceKind

Section [vsim]

This variable defines the kind of force used when not otherwise specified.

Syntax

DefaultForceKind = {default | deposit | drive | freeze}

Arguments

- The arguments are described as follows:

- 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

[The Runtime Options Dialog](#)

DefaultLibType

Section [utils]

This variable determines the default type for a library created with the vlib command.

Syntax

DefaultLibType = {0 | 1 | 2}

Arguments

- The arguments are described as follows:
 - 0 - legacy library using subdirectories for design units
 - 1 - archive library (deprecated)
 - 2 - (default) flat library

Related Topics

[vlib \[ModelSim Command Reference Manual\]](#)

DefaultRadix

Section [vsim]

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”.

Usage

DefaultRadix = {ascii | binary | decimal | hexadecimal | octal | symbolic | unsigned}

Arguments

- The arguments are described as follows:
 - 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— (default) Display values in hexadecimal format. You can also specify 16.
 - octal — Display values in octal format. You can also specify 8.
 - symbolic — 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}, or by using the -default_radix switch with the [vsim](#) command.

Related Topics

[Changing Radix \(base\) for the Wave Window](#)

[The Runtime Options Dialog](#)

DefaultRadixFlags

Section [vsim]

This variable controls the display of numeric radices.

Syntax

DefaultRadixFlags = { " " | enumnumeric | enumsymbolic | showbase | showverbose | wreal }

Arguments

- You can specify the following arguments for this variable:
 - No argument — Format enums symbolically.
 - **enumnumeric** — Display enums in numeric format.
 - **enumsymbolic** — Display enums in symbolic format.
 - **showbase** — (**default**) Display enums showing the number of bits of the vector and the radix that was used where:

```
binary = b
decimal = d
hexadecimal = h
ASCII = a
time = t
```

For example, instead of simply displaying a vector value of “31”, a value of “16’h31” may be displayed to show that the vector is 16 bits wide, with a hexadecimal radix.

- **showverbose** — Display enums with verbose information enabled.
You can override this argument with the [radix](#) command.
- **wreal** — Display internal values of real numbers that are determined to be not-a-number (NaN) as X or Z instead of as "nan." If the internal value of NaN matches `wrealZState, then a 'Z' will be displayed; otherwise, an 'X' will be displayed.

This affects all windows and commands that display or return simulation values and is disabled by default. You can override this argument with the [radix](#) command.

DefaultRestartOptions

Section [vsim]

This variable sets the default behavior for the restart command.

Syntax

```
DefaultRestartOptions = { -force | -noassertions | -nobreakpoint | -nofcovers | -nolist | -nolog |  
-nowave }
```

Arguments

- The arguments are described as follows:
 - -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

[vsim \[ModelSim Command Reference Manual\]](#)

DelayFileOpen

Section [vsim]

This variable instructs ModelSim to open VHDL87 files on first read or write, else open files when elaborated.

Syntax

DelayFileOpen = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) On
 - 1 — Off

displaymsgmode

Section [msg_system]

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.

Syntax

displaymsgmode = {both | tran | wlf}

Arguments

- The arguments are described as follows:
 - 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 \[ModelSim GUI Reference Manual\]](#)

DpiOutOfTheBlue

Section [vsim]

This variable enables DPI out-of-the-blue Verilog function calls. The C functions must not be declared as import tasks or functions.

Syntax

DpiOutOfTheBlue = {0 | 1 | 2}

Arguments

- The arguments are described as follows:
 - **0** — (default) Support for DPI out-of-the-blue calls is disabled.
 - **1** — Support for DPI out-of-the-blue calls is enabled.
 - **2** — Support for DPI out-of-the-blue calls is enabled.

You can override this variable using vsim -dpioutoftheblue.

Related Topics

[Making Verilog Function Calls from non-DPI C Models](#)

DumpportsCollapse

Section [vsim]

This variable collapses vectors (VCD id entries) in dumpports output.

Syntax

DumpportsCollapse = {0 | 1}

Arguments

- The arguments are described as follows:

- **0** — Off
- **1** — (default) On

You can override this variable by specifying **vsim** {+dumpports+collapse | +dumpports+nocollapse}.

EnumBaselInit

Section [vsim]

This variable initializes enum variables in SystemVerilog using either the default value of the base type or the leftmost value.

Syntax

EnumBaseInit= {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Initialize to leftmost value
 - 1 — (default) Initialize to default value of base type

error

Section [msg_system]

This variable changes the severity of the listed message numbers to “error.”

Syntax

`error = <msg_number>...`

Arguments

- The arguments are described as follows:

- `<msg_number>...` — An unlimited list of message numbers, comma separated.

You can override this variable by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the -error argument.

Related Topics

[verror \[ModelSim Command Reference Manual\]](#)

[Message Severity Level](#)

[fatal](#)

[note](#)

[suppress](#)

[warning](#)

ErrorFile

Section [vsim]

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.

Syntax

ErrorFile = <filename>

Arguments

- The arguments are described as follows:
 - <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](#)

[TranscriptFile](#)

Explicit

Section [vcom]

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 Questa Sim compatible with common industry practice.

Syntax

Explicit = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this variable by specifying [vcom -explicit](#).

fatal

Section [msg_system]

This variable changes the severity of the listed message numbers to “fatal”.

Syntax

`fatal = <msg_number>...`

Arguments

- The arguments are described as follows:

- `<msg_number>...` — An unlimited list of message numbers, comma separated.

You can override this variable by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the -fatal argument.

Related Topics

[verror \[ModelSim Command Reference Manual\]](#)

[Message Severity Level](#)

[error](#)

[note](#)

[suppress](#)

[warning](#)

FlatLibPageSize

Section [utils]

This variable sets the size in bytes for flat library file pages. Very large libraries may benefit from a larger value, at the expense of disk space.

Syntax

FlatLibPageSize = <*value*>

Arguments

- The arguments are described as follows:
 - <*value*> — Specifies a library size in Mb where the default value is 8192.

FlatLibPageDeletePercentage

Section [utils]

This variable sets the percentage of total pages deleted before library cleanup can occur. This setting is applied together with FlatLibPageDeleteThreshold.

Syntax

FlatLibPageDeletePercentage = <*value*>

Arguments

- The arguments are described as follows:
 - <*value*> — Specifies a percentage where the default value is 50.

FlatLibPageDeleteThreshold

Section [utils]

Set the number of pages deleted before library cleanup can occur. This setting is applied together with FlatLibPageDeletePercentage.

Syntax

FlatLibPageDeletePercentage = <*value*>

Arguments

- The arguments are described as follows:
 - <*value*> — Specifies a percentage where the default value is 1000.

floatfixlib

Section [library]

This variable sets the path to the library containing VHDL floating and fixed point packages.

Syntax

floatfixlib = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/..../floatfixlib. May include environment variables.

ForceSigNextIter

Section [vsim]

This variable controls the iteration of events when a VHDL signal is forced to a value.

Syntax

ForceSigNextIter = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off. Update and propagate in the same iteration.
 - 1 — On. Update and propagate in the next iteration.

ForceUnsignedIntegerToVHDLInteger

Section [vlog]

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.

Syntax

ForceUnsignedIntegerToVHDLInteger = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

FsmImplicitTrans

Sections [vcom], [vlog]

This variable controls recognition of FSM Implicit Transitions.

Syntax

FsmImplicitTrans = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On. Enables recognition of implied same state transitions.

Related Topics

[vcom \[ModelSim Command Reference Manual\]](#)

[vlog \[ModelSim Command Reference Manual\]](#)

FsmResetTrans

Sections [vcom], [vlog]

This variable controls the recognition of asynchronous reset transitions in FSMs.

Syntax

`FsmResetTrans = {0 | 1}`

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Related Topics

[vcom \[ModelSim Command Reference Manual\]](#)

[vlog \[ModelSim Command Reference Manual\]](#)

FsmSingle

Section [vcom], [vlog]

This variable controls the recognition of FSMs with a single-bit current state variable.

Syntax

`FsmSingle = { 0 | 1 }`

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Related Topics

[vcom \[ModelSim Command Reference Manual\]](#)

[vlog \[ModelSim Command Reference Manual\]](#)

FsmXAssign

Section [vlog]

This variable controls the recognition of FSMs where a current-state or next-state variable has been assigned “X” in a case statement.

Syntax

`FsmXAssign = { 0 | 1 }`

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Related Topics

[vlog \[ModelSim Command Reference Manual\]](#)

GCThreshold

Section [vsim]

This variable sets the memory threshold for SystemVerilog garbage collection.

Syntax

GCThreshold = <n>

Arguments

- The arguments are described as follows:

- <n>

Any positive integer where <n> is the number of megabytes. The default is 100.

You can override this variable with the [gc configure](#) command or with vsim -threshold.

Related Topics

[Class Instance Garbage Collection](#)

[Changing the Garbage Collector Configuration](#)

[ClassDebug](#)

[Default Garbage Collector Settings](#)

GCThresholdClassDebug

Section [vsim]

This variable sets the memory threshold for SystemVerilog garbage collection when class debug mode is enabled with vsim -classdebug.

Syntax

GCThresholdClassDebug = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any positive integer where <n> is the number of megabytes. The default is 5.

You can override this variable with the [gc configure](#) command.

Related Topics

[Class Instance Garbage Collection](#)

[Changing the Garbage Collector Configuration](#)

[ClassDebug](#)

[Default Garbage Collector Settings](#)

GenerateFormat

Section [vsim]

This variable controls the format of the old-style VHDL for ... generate statement region name for each iteration.

Syntax

GenerateFormat = <*non-quoted string*>

Arguments

- The arguments are described as follows:

- <*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 white space is allowed (but discouraged) while leading and trailing white space 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](#)

[Naming Behavior of VHDL for Generate Blocks](#)

GenerousIdentifierParsing

Section [vsim]

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.

Syntax

GenerousIdentifierParsing = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

GlobalSharedObjectList

Section[vsim]

For vsim, this variable instructs ModelSim to load the specified PLI/FLI shared objects with global symbol visibility. Essentially, setting this variable exports the local data and function symbols from each shared object as global symbols so they become visible among all other shared objects. Exported symbol names must be unique across all shared objects.

Syntax

GlobalSharedObjectList = <filename>

Arguments

- The arguments are described as follows:
 - <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 -gblso with [vsim](#).

Hazard

Section [vlog]

This variable turns on Verilog hazard checking (order-dependent accessing of global variables).

Syntax

Hazard = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

ieee

Section [library]

This variable sets the path to the library containing IEEE and Synopsys arithmetic packages.

Syntax

ieee = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path, including environment variables where the default is \$MODEL_TECH/../ieee.

IgnoreError

Section [vsim]

This variable instructs ModelSim to disable runtime error messages.

Syntax

IgnoreError = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Related Topics

[The Runtime Options Dialog](#)

IgnoreFailure

Section [vsim]

This variable instructs ModelSim to disable runtime failure messages.

Syntax

IgnoreFailure = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Related Topics

[The Runtime Options Dialog](#)

IgnoreNote

Section [vsim]

This variable instructs ModelSim to disable runtime note messages.

Syntax

IgnoreNote = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Related Topics

[The Runtime Options Dialog](#)

IgnorePragmaPrefix

Section [vcom, vlog]

This variable instructs the compiler to ignore synthesis pragmas with the specified prefix name. The affected pragmas will be treated as regular comments.

Syntax

IgnorePragmaPrefix = {*<prefix>* | "" }

Arguments

- The arguments are described as follows:
 - *<prefix>* — Specifies a user defined string.
 - "" — (default) No string.

You can override this variable by specifying **vcom -ignorepragmaprefix** or **vlog -ignorepragmaprefix**.

ignoreStandardRealVector

Section [vcom]

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.

Syntax

IgnoreStandardRealVector = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this variable by specifying [vcom -ignoreStandardRealVector](#).

IgnoreVitalErrors

Section [vcom]

This variable instructs ModelSim to ignore VITAL compliance checking errors.

Syntax

IgnoreVitalErrors = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off. Allow VITAL compliance checking errors.
 - **1** — On

You can override this variable by specifying [vcom -ignorevitalerrors](#).

IgnoreWarning

Section [vsim]

This variable instructs ModelSim to disable runtime warning messages.

Syntax

IgnoreWarning = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off. Enable runtime warning messages.
 - **1** — On

Related Topics

[The Runtime Options Dialog](#)

ImmediateContinuousAssign

Section [vsim]

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.

Syntax

ImmediateContinuousAssign = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — Off
 - **1** — (default) On

You can override this variable by specifying **vsim -noimmedca**.

IncludeRecursionDepthMax

Section [vlog]

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.

Syntax

IncludeRecursionDepthMax = <n>

Arguments

- The arguments are described as follows:
 - <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

Section [vcom]

This variable controls how subprogram output parameters of array and record types are treated.

Syntax

InitOutCompositeParam = {0 | 1 | 2}

Arguments

- The arguments are described as follows:
 - 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.

You can override this variable by specifying vcom -initoutcompositeparam.

IterationLimit

Section [vlog], [vsim]

This variable specifies a limit on simulation kernel iterations allowed without advancing time.

Syntax

IterationLimit = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any positive integer where the default is 10000000.

Related Topics

[The Runtime Options Dialog](#)

keyring

Specifies the location of the common and toolblock files used by the vencrypt and vhencrypt commands.

Usage

keyring = <directory_name>

Arguments

The default location for the keyring directory is in the product installation directory keyring which is <install_dir>/keyring.

LargeObjectSilent

Section [vsim]

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.

Syntax

LargeObjectSilent = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) On
 - 1 — Off

LargeObjectSize

Section [vsim]

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.

Syntax

LargeObjectSize = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any positive integer where the default is 500000 bytes.

LibrarySearchPath

Section [vlog, vsim]

This variable specifies the location of one or more resource libraries containing a precompiled package. The behavior of this variable is identical to specifying the -L <libname> command line option with vlog or vsim.

Syntax

LibrarySearchPath = <variable> | <path/lib> ...

Arguments

- The arguments are described as follows:

- <variable>— Any library variable where the default is:

```
LibrarySearchPath = mtiAvm mtiOvm mtiUvm mtiUPF infact
```

- *path/lib* — Any valid library path. May include environment variables. Multiple library paths and variables are specified as a space separated list.

You can use the vsim -showlibsearchpath option to return all libraries specified by the LibrarySearchPath variable. You can use the vsim -ignoreinilibs to prevent vsim from using the libraries specified in LibrarySearchPath.

Related Topics

[Verilog Resource Libraries](#)

[VHDL Resource Libraries](#)

[vlog \[ModelSim Command Reference Manual\]](#)

[vsim \[ModelSim Command Reference Manual\]](#)

MessageFormat

Section [vsim]

This variable defines the format of VHDL assertion messages as well as normal error messages.

Syntax

`MessageFormat = <%value>`

Arguments

- The arguments are described as follows:
 - `<%value>` — One or more of the variables from [Table A-5](#) where the default is:
`** %S: %R\n Time: %T Iteration: %D%I\n.`

Table A-5. MessageFormat Variable: Accepted Values

Variable	Description
<code>%S</code>	severity level
<code>%R</code>	report message
<code>%T</code>	time of assertion
<code>%D</code>	delta
<code>%I</code>	instance or region pathname (if available)
<code>%i</code>	instance pathname with process
<code>%O</code>	process name
<code>%K</code>	kind of object path points to; returns Instance, Signal, Process, or Unknown
<code>%P</code>	instance or region path without leaf process
<code>%F</code>	file
<code>%L</code>	line number of assertion, or if from subprogram, line from which call is made
<code>%u</code>	Design unit name in form: library.primary. Returns <code><protected></code> if the design unit is protected.
<code>%U</code>	Design unit name in form: library.primary(secondary). Returns <code><protected></code> if the design unit is protected.
<code>%%</code>	print '%' character

MessageFormatBreak

Section [vsim]

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

Syntax

MessageFormatBreak = <%value>

Arguments

- The arguments are described as follows:

- <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n

MessageFormatBreakLine

Section [vsim]Section [vsim]

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

Syntax

MessageFormatBreakLine = <%value>

Arguments

- The arguments are described as follows:

- <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D %K: %i File: %F Line: %L\n

%L specifies the line number of the assertion or, if the breakpoint is from a subprogram, the line from which the call is made.

MessageFormatError

Section [vsim]

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.

Syntax

MessageFormatError = <%value>

Arguments

- The arguments are described as follows:
 - <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n

Related Topics

[MessageFormatBreak](#)

MessageFormatFail

Section [vsim]

This variable defines the format of messages for VHDL Fail assertions.

Syntax

MessageFormatFail = <%value>

Arguments

- The arguments are described as follows:

- <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n

Description

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used.

Related Topics

[MessageFormatBreak](#)

MessageFormatFatal

Section [vsim]

This variable defines the format of messages for VHDL Fatal assertions.

Syntax

MessageFormatFatal = <%value>

Arguments

- The arguments are described as follows:

- <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n

Description

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used.

Related Topics

[MessageFormatBreak](#)

MessageFormatNote

Section [vsim]

This variable defines the format of messages for VHDL Note assertions.

Syntax

MessageFormatNote = <%value>

Arguments

- The arguments are described as follows:

- <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D%I\n

Description

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used.

Related Topics

[MessageFormatBreak](#)

MessageFormatWarning

Section [vsim]

This variable defines the format of messages for VHDL Warning assertions.

Syntax

MessageFormatWarning = <%value>

Arguments

- The arguments are described as follows:

- <%value> — One or more of the variables from [Table A-5](#) where the default is:

** %S: %R\n Time: %T Iteration: %D%I\n

Description

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used.

Related Topics

[MessageFormatBreak](#)

MixedAnsiPorts

Section [vlog]

This variable supports mixed ANSI and non-ANSI port declarations and task/function declarations.

Syntax

MixedAnsiPorts = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying [vlog -mixedansiports](#).

modelsim_lib

Section [library]

This variable sets the path to the library containing Mentor Graphics VHDL utilities such as Signal Spy.

Syntax

modelsim_lib = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/..../modelsim_lib.
May include environment variables.

MsgLimitCount

Section [msg_system]

This variable limits the number of times warning messages will be displayed. The default limit value is five.

Syntax

MsgLimitCount = <limit_value>

Arguments

- The arguments are described as follows:
 - <limit_value> — Any positive integer where the default limit value is 5.
You can override this variable by specifying [vsim -msglimitcount](#).

Related Topics

[Message Viewer Window. \[ModelSim GUI Reference Manual\]](#)

msgmode

Section [msg_system]

This variable controls where the simulator outputs elaboration and runtime messages.

Syntax

msgmode = {tran | wlf | both}

Arguments

- The arguments are described as follows:
 - 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. \[ModelSim GUI Reference Manual\]](#)

mtiAvm

Section [library]

This variable sets the path to the location of the Advanced Verification Methodology libraries.

Syntax

mtiAvm = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/../avm
- The behavior of this variable is identical to specifying [vlog -L mtiAvm](#).

mtiOvm

Section [library]

This variable sets the path to the location of the Open Verification Methodology libraries.

Syntax

mtiOvm = <path>

Arguments

- The arguments are described as follows:

- <path> — \$MODEL_TECH/../ovm-2.1.2

The behavior of this variable is identical to specifying [vlog -L mtiOvm](#).

MultifileCompilationUnit

Section [vlog]

This variable controls whether Verilog files are compiled separately or concatenated into a single compilation unit.

Syntax

MultiFileCompilationUnit = {0 | 1}

Arguments

- The arguments are described as follows:
 - 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](#)

NoCaseStaticError

Section [vcom]

This variable changes case statement static errors to warnings.

Syntax

NoCaseStaticError = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

You can override this variable by specifying [vcom -nocasestaticerror](#).

Related Topics

[PedanticErrors](#)

NoDebug

Sections [vcom], [vlog]

This variable controls inclusion of debugging info within design units.

Syntax

NoDebug = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

NoDeferSubpgmCheck

Section [vcom]

This variable controls the reporting of range and length violations detected within subprograms as errors (instead of as warnings).

Syntax

NoDeferSubpgmCheck = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

You can override this variable by specifying [vcom -deferSubpgmCheck](#).

NoIndexCheck

Section [vcom]

This variable controls run time index checks.

Syntax

NoIndexCheck = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override NoIndexCheck = 0 by specifying [vcom -noindexcheck](#).

Related Topics

[Compilation of a VHDL Design—the vcom Command](#)

NoOthersStaticError

Section [vcom]

This variable disables errors caused by aggregates that are not locally static.

Syntax

NoOthersStaticError = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this variable by specifying [vcom -noothersstaticerror](#).

Related Topics

[Message Severity Level](#)

[PedanticErrors](#)

NoRangeCheck

Section [vcom]

This variable disables run time range checking. In some designs this results in a 2x speed increase.

Syntax

NoRangeCheck = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this NoRangeCheck = 1 by specifying [vcom -rangecheck](#).

Related Topics

[Compilation of a VHDL Design—the vcom Command](#)

note

Section [msg_system]

This variable changes the severity of the listed message numbers to “note”.

Syntax

note = <msg_number>...

Arguments

- The arguments are described as follows:
 - <msg_number>... — An unlimited list of message numbers, comma separated.
You can override this variable setting by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the -note argument.

Related Topics

[verror](#) [ModelSim Command Reference Manual]

[Message Severity Level](#)

[error](#)

[fatal](#)

[suppress](#)

[warning](#)

NoVitalCheck

Section [vcom]

This variable disables VITAL level 0 and VITAL level 1 compliance checking.

Syntax

NoVitalCheck = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

You can override this variable by specifying [vcom -novitalcheck](#).

NumericStdNoWarnings

Section [vsim]

This variable disables warnings generated within the accelerated numeric_std and numeric_bit packages.

Syntax

NumericStdNoWarnings = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 —(default) Off
 - 1 — On

Related Topics

[The Runtime Options Dialog](#)

OldVHDLConfigurationVisibility

Section [vcom]

Controls visibility of VHDL component configurations during compile.

Syntax

OldVHDLConfigurationVisibility = {0 | 1}

Arguments

- The arguments are described as follows:
 - 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 \[ModelSim Command Reference Manual\]](#)

OldVhdlForGenNames

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]

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.

Syntax

OldVhdlForGenNames = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Related Topics

[GenerateFormat](#)

[Naming Behavior of VHDL for Generate Blocks](#)

OnFinish

Section [vsim]

This variable controls the behavior of ModelSim when it encounters either an assertion failure, a \$finish in the design code.

Syntax

OnFinish = {ask | exit | final | stop}

Arguments

- The arguments are described as follows:
 - ask — (default) In batch mode, the simulation will exit; in GUI mode, the user is prompted for action to either stop or exit the simulation.
 - exit — The simulation exits without asking for any confirmation.
 - final — The simulation executes the stop command *after* executing any SystemVerilog final blocks.
You can override this variable by specifying [vsim -onfinish](#).
 - stop — The simulation executes the stop command *before* executing any SystemVerilog final blocks.

Optimize_1164

Section [vcom]

This variable disables optimization for the IEEE std_logic_1164 package.

Syntax

Optimize_1164 = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

osvvm

Section [Library]

This variable sets the path to the location of the pre-compiled Open Source VHDL Verification Methodology library.

Syntax

osvvm = <path>

Arguments

- The arguments are described as follows:

- <path> — \$MODEL_TECH/../osvvm

The source code for building this library is copied under the Perl foundation's artistic license from the Open Source VHDL Verification Methodology web site at <http://www.osvvm.org>. A copy of the source code is in the directory vhdl_src/vhdl_osvvm_packages.

PathSeparator

Section [vsim]

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.

Note

 When creating a virtual bus, you must set the PathSeparator variable to either a period (.) or a forward slash (/). For more information on creating virtual buses, refer to the section “Combining Objects into Buses”.

Syntax

PathSeparator = <n>

Arguments

- The arguments are described as follows:
 - <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](#)

[SignalSpyPathSeparator](#)

[DatasetSeparator](#)

PedanticErrors

Section [vcom]

This variable forces display of an error message (rather than a warning) on a variety of conditions. It overrides the NoCaseStaticError and NoOthersStaticError variables.

Syntax

PedanticErrors = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Related Topics

[See the vcom \[ModelSim Command Reference Manual\]](#)

[NoCaseStaticError](#)

[NoOthersStaticError](#)

[Enforcing Strict 1076 Compliance](#)

PreserveCase

Section [vcom]

This variable instructs the VHDL compiler either to preserve the case of letters in basic VHDL identifiers or to convert uppercase letters to lowercase.

Syntax

PreserveCase = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

You can override this variable by specifying [vcom -lower](#) or [vcom -preserve](#).

PrintSimStats

Section [vsim]

This variable instructs the simulator to print out simulation statistics at the end of the simulation before it exits. Statistics are printed with relevant units in separate lines. The Stats variable overrides the PrintSimStats if the two are both enabled.

Syntax

PrintSimStats = {0 | 1 | 2}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — print at end of simulation
 - 2 — print at end of each run and end of simulation

You can override this variable by specifying [vsim -printsimstats](#).

Related Topics

[simstats \[ModelSim Command Reference Manual\]](#)

[Stats](#)

Quiet

Sections [vcom], [vlog]

This variable turns off “loading...” messages.

Syntax

Quiet = {0 | 1}

Arguments

- The arguments are described as follows:

- 0 — Off
 - 1 — (default) On

You can override this variable by specifying [vlog -quiet](#) or [vcom -quiet](#).

RequireConfigForAllDefaultBinding

Section [vcom]

This variable instructs the compiler to not generate any default bindings when compiling with vcom and when elaborating with vsim. All instances are left unbound unless you specifically write a configuration specification or a component configuration that applies to the instance. You must explicitly bind all components in the design through either configuration specifications or configurations. If an explicit binding is not fully specified, defaults for the architecture, port maps, and generic maps will be used as needed.

Syntax

RequireConfigForAllDefaultBinding = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override RequireConfigForAllDefaultBinding = 1 by specifying [vcom -performdefaultbinding](#).

Related Topics

[Default Binding](#)

[BindAtCompile](#)

[vcom \[ModelSim Command Reference Manual\]](#)

Resolution

Section [vsim]

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.

Syntax

Resolution = {[n]<time_unit>}

Arguments

- The arguments are described as follows:
 - [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 ps.

Description

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

[UserTimeUnit](#)

RunLength

Section [vsim]

This variable specifies the default simulation length in units specified by the UserTimeUnit variable.

Syntax

RunLength = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any positive integer where the default is 100.
You can override this variable by specifying the [run](#) command.

Related Topics

[UserTimeUnit](#)

[The Runtime Options Dialog](#)

SeparateConfigLibrary

Section [vcom]

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.

Syntax

SeparateConfigLibrary = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this variable by specifying [vcom -separateConfigLibrary](#).

Show_BadOptionWarning

Section [vlog]

This variable instructs ModelSim to generate a warning whenever an unknown plus argument is encountered.

Syntax

Show_BadOptionWarning = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

Show_Lint

Sections [vcom], [vlog]

This variable instructs ModelSim to display lint warning messages.

Syntax

Show_Lint = {0 | 1}

Arguments

- The arguments are described as follows:

- 0 — (default) Off
 - 1 — On

You can override this variable by specifying [vlog -lint](#) or [vcom -lint](#).

Show_source

Sections [vcom], [vlog]

This variable shows source line containing error.

Syntax

Show_source = {0 | 1}

Arguments

- The arguments are described as follows:

- 0 — (default) Off
 - 1 — On

You can override this variable by specifying the [vlog](#) -source or [vcom](#) -source.

Show_VitalChecksWarnings

Section [vcom]

This variable enables VITAL compliance-check warnings.

Syntax

Show_VitalChecksWarnings = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Show_Warning1

Section [vcom]

This variable enables unbound-component warnings.

Syntax

Show_Warning1 = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Show_Warning2

Section [vcom]

This variable enables process-without-a-wait-statement warnings.

Syntax

Show_Warning2 = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Show_Warning3

Section [vcom]

This variable enables null-range warnings.

Syntax

Show_Warning3 = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Show_Warning4

Section [vcom]

This variable enables no-space-in-time-literal warnings.

Syntax

Show_Warning4 = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

Show_Warning5

Section [vcom]

This variable enables multiple-drivers-on-unresolved-signal warnings.

Syntax

Show_Warning5 = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

ShowFunctions

Section [vsim]

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.

Syntax

ShowFunctions = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — Off
 - 1 — (default) On

ShutdownFile

Section [vsim]

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.

Syntax

ShutdownFile = <filename>.do | <filename>#.do | --disable-auto-save}

Arguments

- The arguments are described as follows:
 - <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.

Related Topics

[write format restart command. \[ModelSim Command Reference Manual\]](#)

SignalForceFunctionUseDefaultRadix

Section [vsim]

Set this variable to 1 cause the signal_force VHDL and Verilog functions use the default radix when processing the force value. Prior to 10.2 signal_force used the default radix and now it always uses symbolic unless the value explicitly indicates a base radix.

Syntax

```
SignalForceFunctionUseDefaultRadix = { 0 | 1 }
```

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

SignalSpyPathSeparator

Section [vsim]

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.

Syntax

SignalSpyPathSeparator = <*character*>

Arguments

- The arguments are described as follows:
 - <*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](#)

[DatasetSeparator](#)

SmartDbgSym

This variable reduces the size of design libraries by minimizing the amount of debugging symbol files generated at compile time. Default is to generate debugging symbol database file for all design-units.

Syntax

SmartDbgSym = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this variable by specifying [vcom/vlog -smartdbgsym](#).

Startup

Section [vsim]

This variable specifies a simulation startup DO file.

Syntax

Startup = {do <DO filename>}

Arguments

- The arguments are described as follows:
 - <DO filename> — Any valid DO file where the default is to comment out the line (;).

Related Topics

[do \[ModelSim Command Reference Manual\]](#)

[Using a Startup File](#)

Stats

Section [vcom, vlog, vsim]

This variable controls the display of statistics messages in a logfile and stdout. Stats variable overrides PrintSimStats variable if both are enabled.

Syntax

Stats [=+|-]<feature>[,[+|-]<mode>]

Arguments

- The arguments are described as follows:
 - [+|-] — Controls activation of the feature or mode. You can also enable a feature or mode by specifying a feature or mode without the plus (+) character. Multiple features and modes for each instance of -stats are specified as a comma separated list.
 - <feature>
 - all — All statistics features displayed (cmd, msg, perf, time). Mutually exclusive with none option. When specified in a string with other options, +|-all is applied first.
 - cmd — (default) Echo the command line
 - msg — (default) Display error and warning summary at the end of command execution
 - none — Disable all statistics features. Mutually exclusive with all option. When specified in a string with other options, +|-none is applied first.
 - perf — Display time and memory performance statistics
 - time — (default) Display Start, End, and Elapsed times
 - <mode>

Modes can be set for a specific feature or globally for all features. To add or subtract a mode for a specific feature, specify using the plus (+) or minus (-) character with the feature, for example, Stats=cmd+verbose,perf+list. To add or subtract a mode globally for all features, specify the modes in a comma-separated list, for example, Stats=time,perf,list,-verbose. You cannot specify global and feature specific modes together.

 - kb — Prints memory statistics in Kb units with no auto-scaling
 - list — Display statistics in a Tcl list format when available
 - verbose — Display verbose statistics information when available

You can add or subtract individual elements of this variable by specifying the -stats argument with [vcom](#), [vencrypt](#), [vhencrypt](#), [vlog](#), and [vsim](#).

You can disable all default or user-specified Stats features with the -quiet argument for:

- vcom
- vencrypt
- vhencrypt
- vlog

Description

You can specify modes globally or for a specific feature.

Examples

- Use this example to enable the display of Start, End, and Elapsed time as well as a message count summary, while disabling the echoing of the command line.

vcom -stats=time,-cmd,msg

- In this example, the first -stats option is ignored. The none option disables all default settings and then enables the perf option.

vlog -stats=time,cmd,msg -stats=none,perf

Related Topics

[PrintSimStats](#)

std

Section [library]

This variable sets the path to the VHDL STD library.

Syntax

std = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/./std. May include environment variables.

std_developerskit

Section [library]

This variable sets the path to the libraries for Mentor Graphics standard developer's kit.

Syntax

std_developerskit = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/../
std_developerskit. May include environment variables.

StdArithNoWarnings

Section [vsim]

This variable suppresses warnings generated within the accelerated Synopsys std_arith packages.

Syntax

StdArithNoWarnings = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

Related Topics

[The Runtime Options Dialog](#)

suppress

Section [msg_system]

This variable suppresses the listed message numbers and/or message code strings (displayed in square brackets).

Syntax

suppress = {<msgNumber> | <msgGroup>} [,,<msg_number> | <msgGroup>] ,...]

Arguments

- <msgNumber>
A specific message number
- <msgGroup>
A value identifying a pre-defined group of messages, based on area of functionality. These groups only suppress notes or warnings. The valid arguments are:
All, GroupNote, GroupWarning, GroupFLI, GroupPLI, GroupSDF, GroupVCD,
GroupVital, GroupWLF, GroupTCHK, GroupPA, GroupLRM

Description

You can override this variable setting by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the -suppress argument.

Related Topics

[verror](#) [ModelSim Command Reference Manual]

[Message Severity Level](#)

[error](#)

[fatal](#)

[note](#)

[warning](#)

SuppressFileTypeReg

Section [vsim]

This variable suppresses a prompt from the GUI asking if ModelSim file types should be applied to the current version.

Syntax

SuppressFileTypeReg = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **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_std

Section [library]

This variable sets the path to the SystemVerilog STD library.

Syntax

`sv_std = <path>`

Arguments

- The arguments are described as follows:
 - `<path>` — Any valid path where the default is `$MODEL_TECH/./sv_std`. May include environment variables.

SvExtensions

Section [vlog], [vsim]

This variable enables SystemVerilog language extensions. The extensions enable non-LRM compliant behavior.

Syntax

SvExtensions = [+|-]<val>[,[+|-]<val>] ...

Arguments

- The arguments are described as follows:
 - [+ | -] — controls activation of the *val*.
 - + — activates the *val*.
 - - — deactivates the *val*.
 - <*val*>
 - acum — Specifies that the get(), try_get(), peek(), and try_peek() methods on an untyped mailbox will return successfully if the argument passed is assignment-compatible with the entry in the mailbox. The LRM-compliant behavior is to return successfully only if the argument and entry are of equivalent types.
 - aswe — Enables support for the symmetric wild equality operators $=?=?$ and $!?=!$.
 - atpi — Use type names as port identifiers. Disabled when compiling with $-pedanticerrors$.
 - catx — Allow an assignment of a single un-sized constant in a concat to be treated as an assignment of 'default:val'.
 - cfce — Error message will be generated if \$cast is used as a function and the casting operation fails.
 - ctlc — Casts time literals in constraints to the type: time. The LRM dictates that a time literal, such as "10ns" is to be interpreted as a "realtime" type, but this is not always adhered to, for example:

```
class Foo;
    rand time t;
    constraint c1 {
        t < 10ns; // NON-LRM compliant use of 'real' type
    }
endclass
```

- **daoa** — Allows the passing a dynamic array as the actual argument of DPI open array output port. Without this option, a runtime error, similar to the following, is generated, which is compliant with LRM requirement.

```
# ** Fatal: (vsim-2211) A dynamic array cannot be passed as an
argument to the DPI import function 'impcall' because the
formal 'o' is an unsized output.

#   Time: 0 ns  Iteration: 0  Process: /top/#INITIAL#56 File:
dynarray.sv

# Fatal error in Module dynarray_sv_unit at dynarray.sv line 2
```

- **ddup** — (Drive Default Unconnected Port) Reverts behavior to where explicit named unconnected ports are driven by the default value of the port.
- **dfsp** — (vsim only) Sets the default format specifier to %p if you do not provide one for unpacked arrays in display-related tasks.
- **extscan** — (vsim only) Enables support for values greater than 32 bits in string methods, such as atohex, atoi, atobin, and atooc. By default, these methods return a 32-bit value irrespective of the variable type. With this extension, the value returned is as per the size of variable type. For example, given the following:

```
longint d;
string s = "12341231234abcd";
d = s.atohex();
```

by default, d will be 64'h000000001234abcd, where if you specify extscan, d will be 64'h123412341234abcd

- **evis** — Supports the expansion of environment variables within curly braces ({}) within `include string literals and in `include path names. For example, if MYPATH exists in the environment then it will be expanded in the following:

```
`include "$MYPATH/inc.svh"
```

- **feci** — Treat constant expressions in a foreach loop variable index as constant.
- **fin0** — Treats \$finish() system call as \$finish(0), which results in no diagnostic information being printed.
- **idcl** — Allows passing of import DPI call locations as implicit scopes.

When there is no svSetScope() call, the DPI scope will always be implicitly set to the current DPI call location and restored after the call returns. This is different than the LRM behavior, which sets the scope to the import function declaration's scope instead of the call location.

When you call svSetScope in a C function, it stays sticky until the enclosing C function returns. The calling chain initiates inside the enclosing C function, but

after svSetScope(), will always see the same user scope setting. svSetScope() will not impact additional DPI calls in the same thread after the enclosing C function returns.

- iddp — Ignore the DPI task disable protocol check.
- jnds — (vsim only) Schedules the fork/join_none processes at the end of the active region.
- lfmt — (Legacy Format) Changes data display to show leading zeroes when displaying decimal values.
- mewq — (vlog only) Allows macro substitutions inside string literals not within a text macro.
- pae — Automatically export all symbols imported and referenced in a package.
- pae1 — Allows the export, using wildcard export only, of package symbols from a subsequent import of a package. These symbols may or may not be referenced in the exporting package.
- sccts — (default) Process string concatenations converting the result to string type.
- spsl — (default) Search for packages in source libraries specified with -y and +libext.
- stop0 — Treats \$stop and \$stop() as \$stop(0), which results in no diagnostic information being printed.
- substr1 — Allows one argument in the builtin function substr. A second argument will be treated as the end of the string.
- ubdic — (vlog and vopt only) Allows the use of a variable in a SystemVerilog class before it is defined. For example:

```
class A;
    function func();
        int y = x;           // variable 'x' is used before it is
    defined
    endfunction
    int x = 1;
endclass
```

If you do not enable this extension, you will receive unresolved reference errors.

- udm0 — Expands any undefined macro with the text “1'b0”.
- uslt — (default) Promote unused design units found in source library files specified with the -y option to top-level design units.
- voidsystf — (vlog and vopt only)

When enabled (default), specifies to evaluate any occurrences of \$void(x) within constraint contexts as (x), and issue a suppressible warning each time.

Occurrences of \$void outside of constraint contexts will behave as user-defined system function calls (PLI).

When disabled, evaluates any occurrences of \$void(x) as user-defined system function calls (PLI). Because system function calls are not generally allowed within constraint contexts, the use of \$void in a constraint generates a vlog/vopt error.

Specifying -pedanticerrors disables the voidsystf extension, even if you enable it with -svext=voidsystf.

- Multiple extensions are specified as a comma separated list. For example:

```
SvExtensions = +fec1,-uslt,pae
```

SVFileSuffixes

Section [vlog]

Defines one or more filename suffixes that identify a file as a SystemVerilog file.

Syntax

SVFileSuffixes = <suffix>...

Arguments

- <suffix> ...

A space separated list of suffixes, where the default is “sv svp svh”. 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 (\\).

Svlog

Section [vlog]

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.

Syntax

Svlog = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

SVPrettyPrintFlags

Section [vsim]

This variable controls the formatting of '%p' and '%P' conversion specifications used in \$display and similar system tasks.

Syntax

```
SVPrettyPrintFlags=[I<n><S | T>] [L<numLines>] [C<numChars>] [R{d | b | o | h}]  
[F<numFields>] [E<numElements>] [D<depth>]
```

Arguments

- The arguments are described as follows:
 - **I <n><S | T>** — Expand and indent the format for printing records, structures, and so forth by <n> spaces (S) or <n> tab stops (T).
 - <n> — (required) Any positive integer
 - **S** — (required when indenting with spaces) Indent with spaces.
 - **T** — (required when indenting with tab stops) Indent with tab stops.
 - **L<numLines>** — (optional) Limit the number of lines of output to <numLines>.
 - **R {d | b | o | h}** — (optional) Specify a radix for printing the data specified using the %p format:
 - d decimal (default)
 - b binary
 - o octal
 - h hexadecimal

For example, SVPrettyPrintFlags=Rh specifies a hexadecimal radix. Further, SVPrettyPrintFlags=R shows the output of specifier %p as per the specified radix. It changes the output in \$display and similar systasks. It does not affect formatted output functions (such as \$displayh).

- <numLines> — (required) Any positive integer.
- For example, SVPrettyPrintFlags=L10 will cause the output to be limited to 10 lines.
- **C<numChars>** — (optional) Limit the number of characters of output to <numChars>.
- <numChars> — (required) Any positive integer.

- For example, SVPrettyPrintFlags=C256 will limit the output to 256 characters.
- **F<numFields>** — (optional) Limit the number of fields of records, structures, and so forth to <numFields>.
- <numFields> — (required) Any positive integer.
- For example, SVPrettyPrintFlags=F4 will limit the output to 4 fields of a structure.
- **E<numElements>** — (optional) Limit the number of elements of arrays to <numElements>.
- <numElements> — (required) Any positive integer.
- For example, SVPrettyPrintFlags=E50 will limit the output to 50 elements of an array.
- **D<depth>** — (optional) Suppress the output of sub-elements below a specified depth to <depth>.
- <depth> — (required) Any positive integer.

For example, SVPrettyPrintFlags=D5 will suppresses the output of sub elements below a depth of 5.

Multiple options are specified as a comma separated list. For example, SVPrettyPrintFlags=I4S,L20,C256,F4,E50,D5.

synopsys

Section [vsim]

This variable sets the path to the accelerated arithmetic packages.

Syntax

`synopsys = <path>`

Arguments

- The arguments are described as follows:
 - `<path>` — Any valid path where the default is `$MODEL_TECH/./synopsys`. May include environment variables.

SyncCompilerFiles

Section [vcom]

This variable causes compilers to force data to be written to disk when files are closed.

Syntax

SyncCompilerFiles = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

toolblock

Specifies the file containing one or multiple toolblock directives to be used by the vencrypt and vhencrypt commands.

Usage

```
toolblock = {<key_file_name>[,<rights_file_name>]} ...
```

Arguments

- Specifies the root name of a file that contains directives intended for a toolblock.
 - **key file name**— The root name of a file that contains directives intended for a tool block. Generally, it is the name of a key (key_keyname) contained within the file. This file is in the "keyring" directory after appending either *.deprecated or *.active to the base name.
 - **rights file name**— Optionally, a "rights file" name may appear after the keyfile name; It contains the control directives defining user rights. You are not required to have a ".depreceated" or ".active" suffix. However, it must have a *.rights suffix.

TranscriptFile

Section [vsim]

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.

Syntax

TranscriptFile = {<filename> | transcript}

Arguments

- The arguments are described as follows:
 - <filename> — Any valid filename where transcript is the default.

Related Topics

[AssertFile](#)

[BatchMode](#)

[BatchTranscriptFile](#)

[transcript file \[ModelSim Command Reference Manual\]](#)

[vsim \[ModelSim Command Reference Manual\]](#)

UnbufferedOutput

Section [vsim]

This variable controls VHDL files open for write.

Syntax

UnbufferedOutput = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off, Buffered
 - **1** — On, Unbuffered

UndefSyms

Section [vsim]

This variable allows you to manage the undefined symbols in the shared libraries currently being loaded into the simulator.

Syntax

UndefSyms = {on | off | verbose}

Arguments

- The arguments are described as follows:
 - **on** — (default) Enables automatic generation of stub definitions for undefined symbols and permits loading of the shared libraries despite the undefined symbols.
 - **off** — Disables loading of undefined symbols. Undefined symbols trigger an immediate shared library loading failure.
 - **verbose** — Permits loading to the shared libraries despite the undefined symbols and reports the undefined symbols for each shared library.

UserTimeUnit

Section [vsim]

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.
-

Syntax

UserTimeUnit = {<time_unit> | default}

Arguments

- The arguments are described as follows:
 - <time_unit> — fs, ps, ns, us, ms, sec, or default.

Related Topics

[Resolution](#)

[RunLength](#)

[force \[ModelSim Command Reference Manual\]](#)

[run \[ModelSim Command Reference Manual\]](#)

UVMControl

Section [vsim]

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.

Syntax

UVMControl={all | certe | disable | msglog | none | struct | trlog | verbose}

Arguments

- 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

Section [library]

This variable sets the path to the library containing VHDL/Verilog type mappings.

Syntax

verilog = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/./verilog. May include environment variables.

Veriuser

Section [vsim]

This variable specifies a list of dynamically loadable objects for Verilog interface applications.

Syntax

Veriuser = <*name*>

Arguments

- The arguments are described as follows:
 - <*name*> — One or more valid shared object names where the default is to comment out the variable.

Related Topics

[Registering PLI Applications](#)

[vsim \[ModelSim Command Reference Manual\]](#)

[restart \[ModelSim Command Reference Manual\]](#)

VHDL93

Section [vcom]

This variable enables support for VHDL language version.

Syntax

`VHDL93 = {0 | 1 | 2 | 3 | 87 | 93 | 02 | 08 | 1987 | 1993 | 2002 | 2008}`

Arguments

- The arguments are described as follows:
 - **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** — (default) Support for VHDL-2002. 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}`.

VhdlSeparatePduPackage

Section [vsim]

This variable turns off sharing of a package from a library between two or more PDUs. Each PDU will have a separate copy of the package. By default PDUs calling the same package from a library share one copy of that package.

Syntax

VhdlSeparatePduPackage = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying vsim -vhdlmergepdupackage.

Related Topics

[vsim \[ModelSim Command Reference Manual\]](#)

VhdlVariableLogging

Section [vsim]

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).

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.

Syntax

VhdlVariableLogging = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying vsim -novhdlvariablelogging.

Description

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).

Related Topics

[vsim \[ModelSim Command Reference Manual\]](#)

vital2000

Section [library]

This variable sets the path to the VITAL 2000 library.

Syntax

vital2000 = <path>

Arguments

- The arguments are described as follows:
 - <path> — Any valid path where the default is \$MODEL_TECH/..vital2000. May include environment variables.

vlog95compat

Section [vlog]

This variable instructs ModelSim to disable SystemVerilog and Verilog 2001 support, making the compiler revert to IEEE Std 1364-1995 syntax.

Syntax

vlog95compat = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off
 - **1** — On

You can override this variable by specifying [vlog -vlog95compat](#).

WarnConstantChange

Section [vsim]

This variable controls whether a warning is issued when the change command changes the value of a VHDL constant or generic.

Syntax

WarnConstantChange = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — Off
 - **1** — (default) On

Related Topics

[change \[ModelSim Command Reference Manual\]](#)

warning

Section [msg_system]

This variable changes the severity of the listed message numbers to “warning”.

Syntax

warning = <msg_number>...

Arguments

- The arguments are described as follows:

- <msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the -warning argument.

Related Topics

[verror](#) [ModelSim Command Reference Manual]

[Message Severity Level](#)

[error](#)

[fatal](#)

[note](#)

[suppress](#)

WaveSignalNameWidth

Section [vsim]

This variable controls the number of visible hierarchical regions of a signal name shown in the Wave Window.

Syntax

WaveSignalNameWidth = <n>

Arguments

- The arguments are described as follows:
 - <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](#).

Related Topics

[verror](#) [ModelSim Command Reference Manual]

[Message Severity Level](#)

[Wave Window](#) [ModelSim GUI Reference Manual]

[error](#)

[fatal](#)

[note](#)

[suppress](#)

wholefile

Controls whether the encryption commands encrypt the entire file by either ignoring or using all pragmas that are present in the input.

Usage

`wholefile = { 0 | 1 }`

Arguments

- The default is 0 which directs the encryption commands to use all the protect directives embedded in the input. If you set the variable to 1, then the encryption commands will ignore all encryption directives except for "viewport" and "interface_viewport".

WildcardFilter

Section [vsim]

This variable sets the default list of object types that are excluded when performing wildcard matches with simulator commands. The default WildcardFilter variables are loaded every time you invoke the simulator.

Syntax

WildcardFilter = <object_list>

Arguments

- The arguments are described as follows:
 - <object_list> — A space separated list of objects where the default is:
 - Variable Constant Generic Parameter SpecParam Memory Assertion Cover Endpoint ScVariable CellInternal ImmediateAssert VHDLFile

You can override this variable by specifying set WildcardFilter “<object_list>” or by selecting Tools > Wildcard Filter to open the Wildcard Filter dialog. Refer to [Using the WildcardFilter Preference Variable](#) in the Command Reference Manual for more information and a list of other possible WildcardFilter object types.

Related Topics

[Using the WildcardFilter Preference Variable \[ModelSim Command Reference Manual\]](#)

WildcardSizeThreshold

Section [vsim]

This variable prevents logging of very large non-dynamic objects when performing wildcard matches with simulator commands, for example, “log -r*” and “add wave *”. Objects of size equal to or greater than the WildcardSizeThreshold setting will be filtered out of wildcard matches. The size is a simple calculation of the number of bits or items in the object.

Syntax

WildcardSizeThreshold = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any positive whole number where the default is 8192 bits (8 k). Specifying 0 disables the checking of the object size against this threshold and allows logging objects of any size.

You can override this variable by specifying **set WildcardSizeThreshold <n>** where <n> is any positive whole number.

Related Topics

[Wildcard Characters \[ModelSim Command Reference Manual\]](#)

WildcardSizeThresholdVerbose

Section [vsim]

This variable controls whether warning messages are output when objects are filtered out due to the WildcardSizeThreshold variable.

Syntax

WildcardSizeThresholdVerbose = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Off
 - 1 — On

You can override this variable by specifying **set WildcardSizeThresholdVerbose** with a 1 or a 0.

Related Topics

[Wildcard Characters \[ModelSim Command Reference Manual\]](#)

WLFCacheSize

Section [vsim]

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.

Syntax

WLFCacheSize = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any non-negative integer where the default for Windows platforms is 1000M.

You can override this variable by specifying [vsim -wlfcachesize](#).

Related Topics

[WLF File Parameter Overview](#)

WLFCollapseMode

Section [vsim]

This variable controls when the WLF file records values.

Syntax

WLFCollapseMode = {0 | 1 | 2}

Arguments

- The arguments are described as follows:
 - **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](#) -wlfcollapsesdelta.
 - **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 | -wlfcollapsesdelta | -wlfcollapsetime}.

Related Topics

[WLF File Parameter Overview](#)

WLFCompress

Section [vsim]

This variable enables WLF file compression.

Syntax

WLFCompress = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — Off
 - **1** — (default) On

You can override this variable by specifying [vsim -nowlfcompress](#).

Related Topics

[The Runtime Options Dialog](#)

[WLF File Parameter Overview](#)

WLFDeleteOnQuit

Section [vsim]

This variable specifies whether a WLF file should be deleted when the simulation ends.

Syntax

WLFDeleteOnQuit = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off. Do not delete.
 - **1** — On.

You can override this variable by specifying [vsim -nowlfdeleteonquit](#).

Related Topics

[The Runtime Options Dialog](#)

[WLF File Parameter Overview](#)

[vsim \[ModelSim Command Reference Manual\]](#)

WLFFileLock

Section [vsim]

This variable controls overwrite permission for the WLF file.

Syntax

WLFFileLock = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — Allow overwriting of the WLF file.
 - **1** — (default) Prevent overwriting of the WLF file.

You can override this variable by specifying [vsim -wllock](#) or [vsim -nowllock](#).

Related Topics

[WLF File Parameter Overview](#)

[vsim \[ModelSim Command Reference Manual\]](#)

WLFFilename

Section [vsim]

This variable specifies the default WLF file name.

Syntax

WLFFilename = {<filename> | vsim.wlf}

Arguments

- The arguments are described as follows:
 - <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](#)

WLFOptimize

Section [vsim]

This variable specifies whether the viewing of waveforms is optimized.

Syntax

WLFOptimize = {0 | 1}

Arguments

- The arguments are described as follows:

- **0** — Off
- **1** — (default) On

You can override this variable by specifying [vsim -nowlfopt](#).

Related Topics

[WLF File Parameter Overview](#)

WLFSaveAllRegions

Section [vsim]

This variable specifies the regions to save in the WLF file.

Syntax

WLFSaveAllRegions = {0 | 1}

Arguments

- The arguments are described as follows:
 - 0 — (default) Only save regions containing logged signals.
 - 1 — Save all design hierarchy.

Related Topics

[The Runtime Options Dialog](#)

WLFSimCacheSize

Section [vsim]

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.

Syntax

WLFSimCacheSize = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any non-negative integer where the default is 500.

You can override this variable by specifying [vsim -wlfsimcachesize](#).

Related Topics

[WLFCacheSize](#)

[WLF File Parameter Overview](#)

WLFSIZELIMIT

Section [vsim]

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.

Syntax

WLFSIZELIMIT = <n>

Arguments

- The arguments are described as follows:
 - <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

[WLFTIMELIMIT](#)

[Limiting the WLF File Size](#)

[WLF File Parameter Overview](#)

WLFTimeLimit

Section [vsim]

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.

Syntax

WLFTimeLimit = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any non-negative integer in units of MB where the default is 0 (unlimited).
- You can override this variable by specifying [vsim -wlftlim](#).

Related Topics

[WLF File Parameter Overview](#)

[Limiting the WLF File Size](#)

[The Runtime Options Dialog](#)

WLFUpdateInterval

Section [vsim]

This variable specifies the update interval for the WLF file. After the interval has elapsed, the live data is flushed to the *.wlf* file, providing an up to date view of the live simulation. If you specify 0, the live view of the wlf file is correct, however the file update lags behind the live simulation.

Syntax

WLFUpdateInterval = <n>

Arguments

- The arguments are described as follows:
 - <n> — Any non-negative integer in units of seconds where the default is 10 and 0 disables updating.

WLFUseThreads

Section [vsim]

This variable specifies whether the logging of information to the WLF file is performed using multithreading.

Syntax

WLFUseThreads = {0 | 1}

Arguments

- The arguments are described as follows:
 - **0** — (default) Off. 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

[Limiting the WLF File Size](#)

WrapColumn

Section [vsim]

This variable defines the column width when wrapping output lines in the transcript file.

Usage

WrapColumn = <integer>

Arguments

- <integer>

An integer that defines the width, in characters, before forcing a line break. The default value is 30000.

Description

This column is somewhat soft; the wrap will occur at the first white-space character after reaching the WrapWSColumn column or at exactly the column width if no white-space is found.

WrapMode

Section [vsim]

This variable controls wrapping of output lines in the transcript file.

Syntax

WrapMode = {0 | 1 | 2}

Arguments

- 0
(default) Disables wrapping.
- 1
Enables wrapping, based on the value of the WrapColumn variable, which defaults to 30,000 characters.
- 2
Enables wrapping and adds a continuation character (\) at the end of every wrapped line, except for the last.

WrapWSColumn

Section [vsim]

This variable defines the column width when wrapping output lines in the transcript file.

Usage

WrapWSColumn = <integer>

Arguments

- <integer>

An integer that specifies that the wrap will occur at the first white-space character after reaching the specified number of characters. If there is no white-space, the wrap will occur at the WrapColumn variable value. The default value is 27000.

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.....	734
Hierarchical Library Mapping.....	735
Creating a Transcript File	735
Using a Startup File.....	736
Turn Off Assertion Messages	736
Turn Off Warnings from Arithmetic Packages	736
Force Command Defaults	737
Restart Command Defaults.....	737
VHDL Standard.....	737
Delay Opening VHDL Files	738

Common Environment Variables

You can use environment variables in the *modelsim.ini* 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

You can use the `TranscriptFile` variable to keep a record of everything that is sent to the transcript from `stdout`: 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. You can also choose what type of data to send to the transcript with the `Stats` variable.

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

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 ""
```

Related Topics

[TranscriptFile](#)

[Stats](#)

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 DO file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

Refer to the [do](#) command for additional information on creating DO files.

Turn 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
```

Turn 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
```

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
```

Related Topics

[force \[ModelSim Command Reference Manual\]](#)

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
```

Related Topics

[restart \[ModelSim Command Reference Manual\]](#)

VHDL Standard

You can specify which version of the 1076 Std ModelSim follows by default using the VHDL93 variable.

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

Related Topics

[VHDL93](#)

Delay Opening VHDL Files

You can delay the opening of VHDL files with an entry in the *modelsim.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
```

Related Topics

[DelayFileOpen](#)

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 740

Referencing Source Files with Location Maps

ModelSim tools that reference source files from the library locate a source file in two ways.

- 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	740
Pathname Syntax	741
How Location Mapping Works	741

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

You can map your files in two steps.

Procedure

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 do not set the SRC environment variable, ModelSim will automatically set it to “/home/vhdl/src”.

Appendix C

Error and Warning Messages

This appendix describes the messages and status information that ModelSim displays in the Transcript window.

Message System	744
Suppression of Warning Messages.....	746
Exit Codes.....	747
Miscellaneous Messages	749
Enforcing Strict 1076 Compliance	752

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 window.

Accordingly, you can also access them from a saved transcript file (see [Saving the Transcript File](#) for more details).

Message Format	744
Getting More Information	745
Message Severity Level	745
Syntax Error Debug Flow	745

Message Format

The format for messages consists of several fields.

The fields for a given message appear as:

```
** <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.

- **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 consisting of four numerical digits.

You can access additional information about a message using the unique id and the [verror](#) command. For example:

```
% perror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

Message Severity Level

You can suppress or change the severity of notes, warnings, and errors that come from vcom, vlog, and vsim commands. You cannot suppress Fatal or Internal messages or change their severity.

There are three ways to modify the severity of or to suppress notes, warnings, and errors:

- Use the -error, -fatal, -note, -suppress, and -warning arguments to [vcom](#), [vlog](#), 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.

Related Topics

[Suppression of Warning Messages](#)

Syntax Error Debug Flow

ModelSim commands issue errors when you provide design files that have syntax errors due to typos or illegal code. You can work to debug these errors using this flow.

Procedure

1. Begin with the first error issued by the command.
2. Review the error message for a specific error number and information about the filename and line number.
3. Use the [verror](#) command to access more information about the error number.
4. Review the area around the line number for typos in identifiers and correct as needed.
5. Review the previous line for a malformed token or missing semicolon (;) or other ending bracket and correct as needed.

6. Review the specific line to ensure the syntax is legal based on the BNF of the language used and correct as needed.
7. Run the command again and repeat these steps for any further messages.

Suppression of Warning Messages

You can suppress the display of a specific warning message or categories of warning messages that are trivial or not relevant to operation of a given command. For example, you can suppress warning messages about unbound components that you are not interested in seeing.

Each of the following commands provides an argument you can specify to control the display of warning messages issued while that command is running:

- vcom — see [Suppress Warning Messages for the vcom Command](#).
- vlog — see [Suppress Warning Messages for the vlog Command](#).
- vsim — see [Suppress Warning Messages for the vsim Command](#).

Suppress Warning Messages for the vcom Command

Use the vcom -nowarn <category_number> argument 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
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.

SUPPRESS WARNING MESSAGES FOR THE VLOG COMMAND

Use the `vlog -nowarn <category_number>` command to suppress a specific warning message. The warning message category numbers for vlog are:

```
12 = non-LRM compliance in order to match other product behavior
```

Alternatively, you can use the `+nowarn<CODE>` argument with the vlog command to suppress a specific warning message. Warning messages that can be disabled this way contain the `<CODE>` string in square brackets, [].

For example:

```
vlog +nowarnDECAY
```

suppresses decay warning messages.

SUPPRESS WARNING MESSAGES FOR THE VSIM COMMAND

Use the `vsim +nowarn<CODE>` 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:

```
vsim +nowarnTFMPC
```

suppresses warning messages about too few port connections.

You can use `vsim -msglimit <msg_number>[,<msg_number>,...]`, or the `MsgLimitCount` variable in the `modelsim.ini` file, 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

When ModelSim exits a process, it displays a numerical exit code in the Transcript window. Each code corresponds to a status condition of the process or operation.

Table C-1 lists the exit codes used by ModelSim commands, processes, and languages.

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 (cont.)

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)
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)
93	Reserved for Verification Run Manager
99	Unexpected error in tool
100	GUI Tcl initialization failure
101	GUI Tk initialization failure
102	GUI IncrTk initialization failure
111	X11 display error
201	Hang Up (SIGHUP)

Table C-2. Exit Codes (cont.)

Exit code	Description
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 that may appear for various ModelSim commands, processes, or design languages.

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: <path/file_name>: (vlog-2605) empty port name in port list.
# ** WARNING: <path/file_name>: (vopt-2605) empty port name in port list.
```

- **Description** — ModelSim reports these warnings if you use the -lint argument with [vlog](#). It reports the warning for any NULL module ports.
- **Suggested action** — If you want to suppress this warning, do not use the -lint argument.

Lock message

```
waiting for lock by user@user. Lockfile is <library_path>/_lock
```

- **Description** — ModelSim creates a *_lock* file in a library when you begin a compilation into that library; 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.

You can turn off these messages 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 displays this message when you use the -check_synthesis argument to [vcom](#). This warning occurs 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 omit the -check_synthesis argument.

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 does not necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that does not connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

The following examples demonstrate 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. Another example is:

```
foo inst1(e, , g, h);
foo inst1(.a(e), .b(), .c(g), .d(h));
```

- **Suggested actions** —

- Check for an extra comma at the end of the port list. For example:

```
model(a,b,)
```

The extra comma is legal Verilog, but it 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 a “License Lost” error message indicates that network traffic is high, and communication with the license server times out.
- **Suggested action** — Any action you can take to improve network communication with the license server has a chance of solving or decreasing the frequency of this problem.

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 a warning message that is not suppressible.

If you compile with `vcom -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 `vcom -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 cannot 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)
- 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. In addition, you may write your own interface applications.

Implementation Information	755
GCC Compiler Support for use with C Interfaces	756
Registering PLI Applications	756
Registering DPI Applications	758
DPI Use Flow	760
Compiling and Linking C Applications for Interfaces	766
Compiling and Linking C++ Applications for Interfaces	768
Specifying Application Files to Load	770
DPI Example	771
The PLI Callback reason Argument	772
The sizetf Callback Function	774
PLI Object Handles	774
Support for VHDL Objects	775
IEEE Std 1364 ACC Routines	776
IEEE Std 1364 TF Routines	777
SystemVerilog DPI Access Routines	780
Verilog-XL Compatible Routines	780
PLI/VPI Tracing	781
Debugging Interface Application Code	782

Implementation Information

This chapter describes only the details of using the Verilog interfaces with ModelSim Verilog and SystemVerilog.

- ModelSim SystemVerilog implements DPI as defined in the IEEE Std 1800-2005.

GCC Compiler Support for use with C Interfaces

To use GCC compilers with C interfaces, you must acquire the gcc/g++ compiler for your given platform.

Related Topics

[Compiling and Linking C Applications for Interfaces](#)

[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 veriuser.h include file as follows:

```
typedef int (*p_tffn)();  
  
typedef struct t_tfcell {  
    short type;      /* USERTASK, USERFUNCTION, or USERREALFUNCTION */  
    short data;      /* passed as data argument of callback function */  
    p_tffn checktf; /* argument checking callback function */  
    p_tffn sizetf;  /* function return size callback function */  
    p_tffn calltf;  /* task or function call callback function */  
    p_tffn misctf; /* miscellaneous reason callback function */  
    char *tfname;   /* name of system task or function */  
  
    /* The following fields are ignored by ModelSim Verilog */  
    int forwref;  
    char *tfveritool;  
    char *tferrmessage;  
    int hash;  
    struct t_tfcell *left_p;  
    struct t_tfcell *right_p;  
    char *namecell_p;  
    int warning_printed;  
} s_tfcell, *p_tfcell;
```

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the calltf function, which is called when the

system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the data field (many PLI applications do not 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 a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF() for each system task or function defined. The mti_RegisterUserTF() function is declared in veriuser.h as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you do not have to provide an init_usertfs function, and the simulator automatically registers 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 PLIOBJSEnvironment 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 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>
```

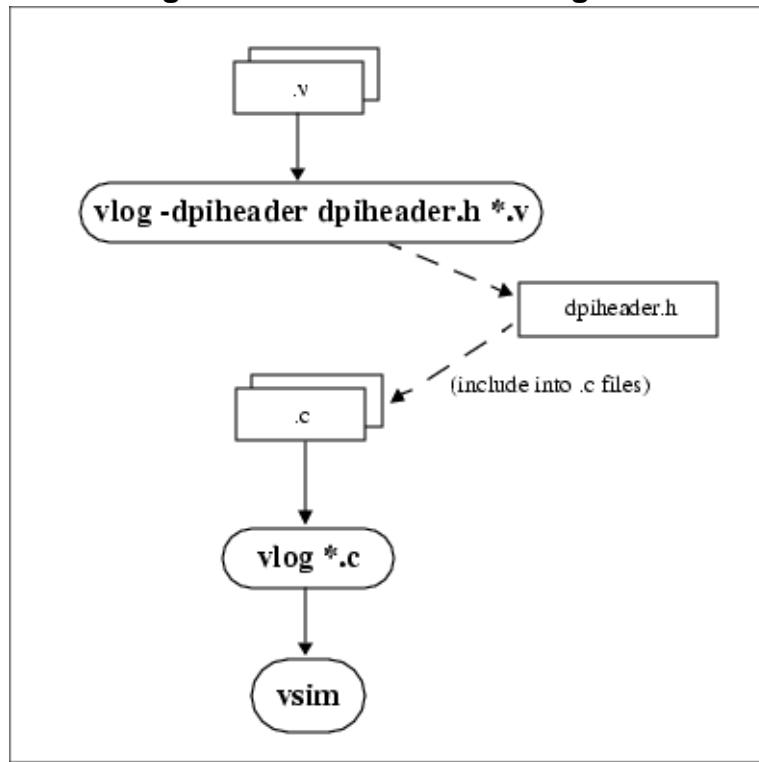
The **-sv_lib** option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see [DPI File Loading](#).

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`

[For WINDOWS platform users:] If a DPI header is not being generated or used, you need to manually attach `DPI_DLLESPEC` in front of all DPI routines. `DPI_DLLESPEC` is a standard macro defined inside `svdpi.h`.

The generated DPI header flow is recommended. Failing to do the above will incur the following warning at elab time:

```
# ** Warning: (vsim-3770) Failed to find user specified function
'foo' in DPI C/C++ source files.
```

and the fatal error at runtime:

```
# ** Fatal: (vsim-160) test.sv(11): Null foreign function pointer
encountered when calling 'foo'
```

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	761
Deprecated Legacy DPI Flows	762
When Your DPI Export Function is Not Getting Called	762
Troubleshooting a Missing DPI Import Function	762
Simplified Import of Library Functions	763
Optimizing DPI Import Call Performance	763
Making Verilog Function Calls from non-DPI C Models	764
Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code	765

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 mydpicode.c
```

This [vlog](#) 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 -ldfflags '-lcrypt'
```

This command tells vsim to pass -lcrypt to the GCC linker.

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)

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](#)”.

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 **-dpihandler** option and review the generated header file.

Troubleshooting a Missing DPI Import Function

DPI uses C function linkage and can produce incorrect C++ names if you do not use the appropriate extern “C” declaration.

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.

Simplified Import of Library Functions

In addition to the traditional method of importing HDL interface, and C library functions, a simplified method can be used: you can declare HDL interface functions as DPI-C imports. When you declare HDL interface functions as DPI-C imports, the C implementation of the import tf 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 = %ld, status = %ld", 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 a 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. And the element types can be both signed and unsigned.

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]`.

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`).

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
```

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 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` argument.
- 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.

If your shared objects contain circular references you must combine the shared object files with the `-gblso` argument to `vsim`.

Compiling and Linking C Applications for Interfaces

The following platform-specific instructions show you how to compile and link your HDL interface 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 HDL interface routines are declared in the include files located in the ModelSim <install_dir>/include directory:

- acc_user.h — declares the ACC routines
- veriuser.h — declares the TF routines
- svdpi.h — declares DPI routines

The following instructions assume that the HDL interface 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 HDL interface see [PLI and VPI File Loading](#). For DPI loading instructions, see [DPI File Loading](#).

Windows Platforms — C 766

Windows Platforms — C

Windows platforms for C are supported for Microsoft Visual Studio and MinGW.

- Microsoft Visual Studio 2013

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".

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 2013 *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
```

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.

For 64-bit:

```
gcc -c -I<install_dir>\include app.c
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win64
-lmtipli
```

SystemC is not supported on Windows 64-bit. However, the gcc-4.5.0-mingw64 compiler is shipped along with QuestaSim to enable users compile their C-interfaces source files with mingw-gcc. The C Debug feature is not supported with this compiler.

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 HDL interface functions have a standard C prototype, you must prevent the C++ compiler from mangling the HDL interface function names. This can be accomplished by using the following type of extern:

```
extern "C"  
{  
    <HDL interface application function prototypes>  
}
```

The header files *veriuser.h*, *acc_user.h*, and *vpi_user.h*, *svdpi.h*, and *dpiheader.h* already include this type of extern. You must also put the HDL interface 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 HDL interface 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 only	768
Windows Platforms — C++	769

For PLI 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++

Windows platforms for C++ are supported for Microsoft Visual Studio and MinGW.

- Microsoft Visual Studio 2013

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".

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.

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	770
DPI File Loading	770

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-1. vsim Arguments for DPI Application Using External Compilation Flows

Argument	Description
-sv_lib <name>	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.)
-sv_root <name>	specifies a new prefix for shared objects as specified by -sv_lib

Table D-1. vsim Arguments for DPI Application Using External Compilation Flows (cont.)

Argument	Description
-sv_liblist <bootstrap_file>	specifies a “bootstrap file” to use. See The format for <bootstrap_file> is as follows: #!SV_LIBRARIES <path>/<to>/<shared>/<library> <path>/<to>/<another> ... 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
```

DPI Example

The following example is a trivial but complete DPI application. For additional examples, see the <install_dir>/examples/systemverilog/dpi directory.

Given the file *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) */
}
```

and the file *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
```

You will first compile the Verilog code:

```
vlib work
vlog -sv -dpiheader dpiheader.h hello.v hello_c.c
```

then simulate and run the design:

```
vsim -c hello_top -do "run -all; quit -f"
```

which results in the following output:

```
# 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 *veriuser.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 miscf 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 will not 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 will not 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 when 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.

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-2. 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()

Table D-2. Supported VHDL Objects (cont.)

Type	Fulltype	Description
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.

IEEE Std 1364 ACC Routines

ModelSim Verilog supports the following ACC routines:

Table D-3. Supported ACC Routines

Routines
<pre> acc_append_delays acc_append_pulsere acc_closeacc_collectacc_com pare_handlesacc_configureacc c_countacc_fetch_argcacc_fetch_argvacc_fetch_attribute acc_fetch_attribute_intacc_fetch_attribute_stracc_fetc h_defnameacc_fetch_delay_mo deacc_fetch_delaysacc_fetch _directionacc_fetch_edgeacc_fetch_fullnameacc_fetch_fu lltypeacc_fetch_indexacc_fetch_locationacc_fetch_namea cc_fetch_paramtypeacc_fetch _paramvalacc_fetch_polarity acc_fetch_precisionacc_fetc h_pulsereacc_fetch_rangeacc_fetch_sizeacc_fetch_tfarga cc_fetch_itfarg acc_fetch_tfarg_intacc_fetc h_itfarg_intacc_fetch_tfarg _str </pre>

<pre> acc_fetch_itfarg_str acc_fetch_timescale_infoacc_fet ch_type acc_fetch_type_stracc_fetch_val ue acc_freeacc_handle_by_nameacc_h andle_calling_mod_macc_handle_c onditionacc_handle_connacc_hand le_hiconnacc_handle_interactive _scopeacc_handle_loconnacc_hand le_modpathacc_handle_notifierac c_handle_objectacc_handle_paren tacc_handle_pathacc_handle_path inacc_handle_pathoutacc_handle_ portacc_handle_scopeacc_handle_ simulated_netacc_handle_tchkacc _handle_tchkarg1acc_handle_tchk arg2acc_handle_terminalacc_hand le_tfargacc_handle_itfargacc_ha ndle_tfinstacc_initialize acc_nextacc_next_bitacc_next_ce llacc_next_cell_loadacc_next_ch ild </pre>	<pre> acc_next_driveracc_next_h iconnacc_next_inputacc_ne xt_loadacc_next_loconnacc _next_modpathacc_next_net acc_next_outputacc_next_p arameteracc_next_portacc_ next_portoutacc_next_prim itiveacc_next_scopeacc_ne xt_specparamacc_next_tchk acc_next_terminalacc_next _topmodacc_object_in_type listacc_object_of_typeacc _product_typeacc_product_ versionacc_release_object acc_replace_delaysacc_repa lace_pulsere acc_reset_bufferacc_set_i nteractive_scopeacc_set_p ulsereacc_set_scopeacc_se t_valueacc_vcl_addacc_vcl _deleteacc_version </pre>
--	---

`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-4. 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_iasyncoff	tf_gettimeprecision	tf_isetlongdelay
tf_asyncon	tf_gettimeunit	tf_setrealdelay
tf_iasyncon	tf_gettimeunit	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_putstrgp	tf_synchronize
tf_igetlongp	tf_iputlongp	tf_isynchronize
tf_getlongtime	tf_putstrp	tf_testpvc_flag
tf_igetlongtime	tf_iputp	tf_itestpvc_flag
tf_getnextlongtime	tf_putreapl	tf_text
tf_getp	tf_iputreapl	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_getreapl	tf_irosynchronize	tf_warning
tf_igetreapl		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.

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	781
Invoking a Trace	781

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

Context: PLI/VPI debugging

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-5. 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"

Table D-5. Values for action Argument (cont.)

Value	Operation	Result
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 miscrf routines), and Verilog VCL callbacks.

Related Topics

[vsim \[ModelSim Command Reference Manual\]](#)

[PLI/VPI Tracing](#)

Debugging Interface Application Code

The flow for debugging HDL interface application code requires that you follow specific steps.

In order to debug your HDL interface 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, do not 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>/<platform>/vsimk 1234").

3. Set an entry point using breakpoint.

Since initially the debugger recognizes only **vsim**'s HDL interface function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first HDL interface 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.

Related Topics

[vsim \[ModelSim Command Reference Manual\]](#)

[PLI/VPI Tracing](#)

Appendix E

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	785
Initialization Sequence	786
Environment Variables	789

Files Accessed During Startup

When you invoke ModelSim, it reads several files in file system and configuration environment.

Table E-1 lists the files that are read during startup. They are listed in the order in which they are accessed.

Table E-1. Files That ModelSim Accesses During Startup

File	Description
<i>modelsim.ini</i>	Contains initial tool settings; see modelsim.ini Variables for specific details on the <i>modelsim.ini</i> file and Initialization Sequence for the search precedence
location map file	Used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is <i>mgc_location_map</i>
<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><project_name>.mpf</i>	If available, loads last project file which is specified in the registry (Windows) or <i>\$(HOME)/.modelsim</i> (UNIX); see What are Projects? for details on project settings

Initialization Sequence

The numbered items listed below describe the initialization sequence for ModelSim. 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.

Names that appear 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:
 - o If the -modelsimini option is used, then the file path specified is used if it exists; else
 - o use \$MODELSIM (which specifies the directory location and name of a *modelsim.ini* file) if it exists; else
 - o use \$(MGC_WD)/*modelsim.ini*; else
 - o use ./*modelsim.ini*; else
 - o use \$(MODEL_TECH)/*modelsim.ini*; else
 - o use \$(MODEL_TECH)/./*modelsim.ini*; else
 - o use \$(MGC_HOME)/lib/*modelsim.ini*; else
 - o set path to ./*modelsim.ini* even though the file does not 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:
 - o use MGC_LOCATION_MAP if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else
 - o use mgc_location_map if it exists; else
 - o use \$(HOME)/mgc/mgc_location_map; else
 - o use \$(HOME)/mgc_location_map; else
 - o use \$(MGC_HOME)/etc/mgc_location_map; else
 - o 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](#), [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).
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*

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

- When you change the working directory within ModelSim, it reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler or simulator options dialog box or use the **vmap** command, ModelSim updates the appropriate sections of the file.
- The *pref.tcl* file references the default .ini file by using 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

When you install ModelSim, the installation process creates and reads several environment variables for the operating system of your computer. Most of these variables have default values, which you can change to customize ModelSim operation.

Expansion of Environment Variables	789
Setting Environment Variables	790
Creating Environment Variables in Windows	795
Library Mapping with Environment Variables	795
Node-Locked License File	796
Referencing Environment Variables	796
Removal of Temporary Files (VSOUT).....	797

Expansion of Environment Variables

ModelSim shell commands vcom, vlog, vsim, and vmap, do not expand environment variables in filename arguments and options. Instead, you should expand variables in the shell window in the usual manner before running these ModelSim commands. The -f switch that most of these commands support 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 **PLIOBJ\$** 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 (\$\$).

Related Topics

[Creating Environment Variables in Windows](#)

[edit \[ModelSim Command Reference Manual\]](#)

[Location Mapping](#)

[modelsim.ini Variables](#)

[vlog \[ModelSim Command Reference Manual\]](#)

[Setting Environment Variables](#)

Setting Environment Variables

Before compiling or simulating, you can specify values for a variety of environment variables to provide the functions described below.

You set the variables according the operating system of your computer, as follows:

- Windows — use the System control panel, refer to “[Creating Environment Variables in Windows](#)” for more information.

Tip

 The LM_LICENSE_FILE variable requires a value; all other variables are optional.

DISABLE_ELAB_DEBUG

The DISABLE_ELAB_DEBUG environment variable, if set, disables vsim elaboration error debugging capabilities using the find insource and typespec commands.

DOPATH

The toolset uses the DOPATH environment variable to search for DO files. 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 is not 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

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.

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 Mentor 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 HDL interface function calls. You should set this variable to any value before invoking the simulator.

MTI_FINDLOOP_STEP_LIMIT

Changes the default findloop step limit of 500 to a positive integer you supply.

MTI_LIB_DIR

Identifies the path to all Tcl libraries installed with ModelSim.

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

PLIOBJJS

The toolset uses the PLIOBJJS 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 generated file (VSOUT) containing all stdout from the simulation kernel.

TMPDIR

(UNIX environments) The TMPDIR environment variable specifies the path to a generated file (VSOUT) containing all stdout from the simulation kernel. The priority for temporary file and directory creation is as follows:

- \$TMPDIR — if defined
- /var/tmp — if available
- /tmp — if available

VISUALIZER_HOME_OVERRIDE

Specifies the directory where the Visualizer tool suite is installed. This directory will be the parent directory of the various platform directories. If set, the variable overrides any search of the PATH variable by the vsim tool. Typically, you should only set this variable if you cannot have the Visualizer installation directory placed in your PATH, or if you want to use a different version than specified in PATH. The tool validates the value of this variable by scanning for the appropriate platform directory and executables within that directory. If the value of the variable is incorrect, vsim reports a fatal error and exits. This check will only be done if the -visualizer argument is used with the vsim command.

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 you can reference by using the vmap command to add library mapping to the modelsim.ini file.

Procedure

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 you have set 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 E-2. Add Library Mappings to modelsim.ini File

Prompt Type	Command	Result added to modelsim.ini
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 with an environment variable. 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}
```

Related Topics

[Setting Environment Variables](#)

Node-Locked License File

The ModelSim node-locked (also called mobile compute) license file installation location is specified through the LM_LICENSE_FILE or MGLS_LICENSE_FILE environment variable value. The node-locked license restricts you to one instance of each product and disallows license check-out for any additional product invocations.

Attempts to invoke more than one instance of a node-locked product will result in an error message similar to this example.

Example E-1. Node-Locked License Limit Error Message

```
# ** Error: License checkout has been disallowed because
# only one session is allowed to run on an uncounted nodelocked
# license and an instance of ModelSim is already running with a
# nodelocked license on this machine.
```

Referencing Environment Variables

There are two ways you can 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 DO files using the Tcl env array mechanism. For example:

```
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**.
-

Removal of Temporary Files (VSOUT)

The temporary (temp) file named VSOUT is the communication mechanism between the simulator kernel and the Graphical User Interface.

In normal circumstances, this temp file is deleted when the simulator exits. If ModelSim crashes, however, you need to delete the temp file manually. If you specify the location of the temp file with **TMPDIR**, you can locate the file more easily for deletion.

Third-Party Information

This section provides information on third-party software that may be included in this product, including any additional license terms.

- [Third-Party Software for Questa and ModelSim Products](#)

End-User License Agreement with EDA Software Supplemental Terms

Use of software (including any updates) and/or hardware is subject to the End-User License Agreement together with the Mentor Graphics EDA Software Supplement Terms. You can view and print a copy of this agreement at:

mentor.com/eula

