cādence™

Verilog[®]-XL User Guide

Product Version 8.2 November 2008 $\ensuremath{\mathbb{C}}$ 1990-2009 Cadence Design Systems, Inc. All rights reserved. Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

- 1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
- 2. The publication may not be modified in any way.
- 3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
- 4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Patents: Cadence Product Verilog -XL, described in this document, is protected by U.S. Patents 5,095,454, 5,418,931, 5,606,698, 6,487,704, 7,039,887, 7,055,116, 5,838,949, 6,263,301, 6,163,763, 6,301,578.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1

Introducing Verilog-XL	19
The Verilog-XL Simulator	19
Major Features of Verilog-XL	19
The Design Process with Verilog-XL	20
Verilog-XL Online Documents	21
Internet News	22

<u>2</u>

Invoking Verilog-XL	23
Invoking Verilog-XL	23
Compiling Source Files	24
Using the SimVision Analysis Environment	25

<u>3</u>

Verifying Your Design	27
Overview	27
Creating a Test Fixture	27
Displaying the Simulation Time	29
Writing Simulation Data to a File	30
Displaying Signals as Graphical Waveforms	32

<u>4</u>

Debugging Your Design	35
Overview	35
Setting Event-Triggered Breakpoints	36
Setting Code-Line Breakpoints	42
Activating and Deactivating Commands	46
Examining Code and Simulation Effects	48

Displaying Expanded Macros 50
Traversing Model Hierarchy
Observing All Simulation Events
Observing a Focused Set of Simulation Events
Observing Wires and Registers Periodically 59
Observing Wires and Registers When They Change Value
Examining Wires and Registers Now
Patching a Model (Asking "What If" Questions)
Ordering Events in a Time Cycle 67
Displaying, Strobing, and Monitoring Data
Controlling the Display and Interpretation of Time
Reinitializing the Network and Simulator Clock

<u>5</u> Cont

Controlling Verilog-XL
<u>Overview</u>
Saving and Restarting a Simulation
Stopping at the Beginning of a Simulation
Stopping During a Simulation
Continuing a Stopped Simulation 80
Stepping and Tracing Through a Simulation
Ending a Simulation
Passing Values into a Module from the Command Line
Conditionally Compiling Source Code
Modifying Simulation Behavior at Run Time
Inserting a File into Another File
Generating Log Files
Reproducing Interactive Sessions Using Key Files
Providing Interactive Commands from a File
Storing Commonly Used Command Line Arguments
Specifying the Delay Type
Selecting a Delay Mode

<u>6</u>
Library Management
<u>Overview</u>
Organizing Libraries
Library Files
Library Directories
The Library.Cell:View Architecture
Reporting of Resolution Paths
Definition Renaming
Syntax Checking in Library Files
The Standard Library Management Scheme
<u>'uselib</u>
Defining Macros for the 'uselib Compiler Directive
Search Order and Efficiency
The Former Library Management Scheme
Using Library Files: The Former Scheme
Using Library Directories: The Former Scheme 106
File Extensions in Library Directories: The Former Scheme
Library Scan Precedence: The Former Scheme
Reading Library Directory Files: The Former Scheme
Use of Compiler Directives with Libraries: The Former Scheme
Efficiency Considerations of Library Usage: The Former Scheme
The Library.Cell:View Library Management Scheme
Directory Structure Example 122
Verilog-XL Notes for CAI
CAI Configurations
Specifying a CAI Simulation
Accessing Libraries

<u>7</u>

Integrating PLI and VPI Routines	129
Overview	129
The Components	130
What Cadence Provides	130

What You Provide	131
Using PLI or VPI	131
Creating a C or C++ Routine	133
Associating a C or C++ Routine with a System Task	135
Integrating Your Application with the Simulator	135
Invoking Your System Tasks	136
Error Handling	137
Debugging	137

<u>8</u>

Switch-Level Simulation 139
<u>Overview</u>
Definition of Switch-Level Networks
Major Features of the XL Algorithms
The Default Algorithm
The Switch-XL Algorithm
Choosing an Algorithm
Enabling the Algorithms
Enabling the Algorithms Globally
Enabling the Algorithms Locally
How the Default Algorithm Works
Forcing and Releasing Nets in Bidirectional Networks
Wired Logic in Bidirectional Networks
Reporting on Bidirectional Networks with \$showvars
How the Switch-XL Algorithm Works
Conversion of Channel Delay to Turn-On/Turn-Off Delay
Optimization of Switch Networks
Displaying Strength Values
Switch-XL Strength Model
Switch-XL Strength Model Example
Switch-XL Strength Model Syntax
Switch-XL Default Charge and Drive Strengths
Strength Reduction 159
Strength Mapping
Delays in Default and Switch-XL Bidirectional Networks

9	
Source Protection	165
<u>Overview</u>	165
Protecting Selected Regions in a Source Description	166
The 'protect and 'endprotect Compiler Directives	166
The +protect Command-Line Option 1	167
Protecting Multiple Files in a Single Command 1	168
Protecting All Modules and UDPs in a Source Description	168
The +autoprotect Command-Line Option 1	168
Protecting Multiple Files in a Single Command 1	170
Effect of Source Protection on Simulation 1	170
System Operations That Cannot Access Protected Data	170
System Operations That Can Access Protected Data	171
Effect of Source Protection on Library Use 1	173
Effect of Source Protection on the Display of Hierarchical Path Names	173
Error Reporting in Source-Protected Regions	176
Syntax Verification	176
Timing Checks	176
Loading Source-Protected Data into Memory1	177
The \$sreadmemh and \$sreadmemb Tasks 1	177
How \$sreadmem/h Differs from \$readmem/h1	178

<u>10</u>

Improving Performance	181
Overview	181
Displaying Memory Usage	181
Displaying Simulation Bottlenecks (Behavior Profiler)	182

<u>11</u>

Cosimulation with Verilog-XL and Quickturn	187
Overview	187
Cosimulation Software Overview	187
Setting Up the Simulator for Cosimulation	189
Accessing Quickturn Integration	189

Creating the Gate-Level Netlist
Generating a Quickturn Emulator Database and a Pin Map
Generating the Simulation Shell and Modifying the Testbench
Generating a Simulation Shell File
Example of Using the Shell Generator 194
Cadence Model Manager for Quickturn Command-Line Plus Options
Quickturn Modes for the qt_mode Option 195
Specifying Cadence Model Manager for Quickturn Options at Simulation Time 196
<u>\$omiCommand System Task</u> 197
Simulating a Model with Verilog-XL and Quickturn
Restrictions and Limitations 198

<u>A</u> Vorila

Verilog-XL Command-line Options	99
Command-Line options	99
-a (Accelerate Option)	99
<u>-c (Compile Only Option)</u> 20	00
-d (Decompile Option)	00
<u>-f (File Option)</u>	00
-i (Interactive File Option)	01
<u>-k (Key File Option)</u>	02
-I (Log File Option)	02
<u>-q (Quiet Option)</u>	02
<u>-r (Restart File Option)</u>	02
-s (Stop Option)	03
<u>-t (Trace Option)</u>	03
<u>-u (Uppercase Option)</u>	03
-v (Library File Option)	03
-version (Display Version Option)	04
-w (Warning Suppression Option)	04
-x (Vector Net Expansion Option) 20	04
-y (Library Directory Option)	04
<u>Examples</u>	05
Command-Line Plus Options	05
+accnoerr	05

+accu path delay
+alt path delays
+annotate any time
+autonaming
+autoprotect
<u>+caxl</u>
+compat twin turbo
<u>+define+</u>
+delay mode distributed
+delay mode path
+delay mode unit
+delay mode zero
+err line length
+extend tcheck data limit/ <percentage limit=""></percentage>
+extend tcheck reference limit/ <percentage limit=""></percentage>
<u>+gui</u>
<u>+incdir+</u>
<u>+libext+</u>
+libnonamehide
+liborder
<u>+librescan</u>
<u>+libverbose</u>
<u>+licq_all</u>
+licq Imchwif
<u>+licq_vxl</u>
+listcounts
<u>+loadpli1</u>
<u>+loadvpi</u>
<u>+maxdelays</u>
<u>+max err count+</u>
<u>+mindelays</u>
+multisource int delays
<u>+neg_tchk</u>
<u>+nolibcell</u>
+notimingchecks
+no cancelled e msg 214

+no_charge_decay
+no cond event error
<u>+no notifier</u>
+no_pulse_int_backanno
<u>+no pulse msg</u>
+no show cancelled e
<u>+no_speedup</u>
<u>+no_tchk_msg</u>
<u>+nowarn</u>
<u>+noxl</u>
<u>+password</u>
<u>+pathpulse</u>
<u>+ppe</u>
+pre 16a paths
<u>+profile</u>
<u>+protect</u>
+pulse e/n and +pulse r/m
+pulsestyle ondetect
+pulsestyle_onevent
+pulse int e/n and +pulse int r/m
+save twin turbo
<u>+sdf cputime</u>
+sdf error info
+sdf_file <filename></filename>
+sdf ign timing edge
+sdf nocheck celltype
+sdf no errors
+sdf_nomsrc_int220
+sdf no warnings
+sdf split two timing check
+sdf_splitvlog_suh
+sdf splitvlog recrem
<u>+sdf_verbose</u>
<u>+show_cancelled_e</u>
<u>+splitsuh</u>
<u>+switchxl</u>

<u>+sxl_keep_all</u> 221
<u>+sxl_keep_declared</u>
<u>+sxl_keep_minimum</u> 222
<u>+sxl_unidirect</u>
<u>+trace twin turbo</u>
<u>+transport int delays</u> 222
<u>+transport_path_delays</u>
<u>+turbo</u>
<u>+turbo+2</u>
<u>+turbo+3</u>
<u>+twin_turbo</u>
<u>+typdelays</u>
<u>+vra</u>
+x transport pessimism
User-Definable Command-Line Arguments 224
Testing for Plus Arguments
Lack of Command-Line Syntax Checking
Compiler Directives
<u>'accelerate and 'noaccelerate</u> 226
<u>'autoexpand_vectornets</u>
<u>'celldefine and 'endcelldefine</u> 226
<u>'default_decay_time</u> 227
<u>'default_nettype</u> 228
<u>'default_rswitch_strength</u> 228
<u>'default_switch_strength</u> 228
<u>'default_trireg_strength</u>
<u>'define</u>
<u>'delay_mode_distributed</u>
<u>'delay mode path</u>
<u>'delay mode unit</u>
<u>'delay mode zero</u>
<u>'expand_vectornets and 'noexpand_vectornets</u>
<u>'ifdef, 'else, and 'endif</u>
<u>'include</u>
<u>'pre_16a_paths and 'end_pre_16a_paths</u> 230
<u>'protect and 'endprotect</u> 231

Verilog-XL User Guide

	<u>'protected and 'unprotected</u> 231	1
	<u>'remove gatenames and 'noremove gatenames</u> 231	1
	<u>'remove netnames and 'noremove netnames</u> 232	2
	<u>'resetall</u>	2
	<u>'switch default</u>	2
	<u>'switch XL</u>	2
	<u>'timescale</u>	2
	<u>'unconnected drive and 'nounconnected drive</u>	3
	<u>'undef</u>	3
	<u>'uselib</u>	4
Co	nditional Compilation	5
	<u>Syntax</u>	5
	How 'ifdef, 'else, and, 'endif Work 235	5
	Nesting the 'ifdef, 'else, and 'endif Compiler Directives	6
	Defining Variable Names to Control Conditional Compilation	7
	The Predefined Symbol for Conditional Compilation	8
	Decompiling Source Descriptions	9
	Conditional Compilation Error Messages	9
	Conditional Compilation Source Protection	0
File	<u>e Inclusion</u>	1
	Syntax of 'include	2
	Specifying Search Directories	2
	How 'include Works in Verilog-XL	2
	Nested 'include Compiler Directives	3
	Decompiling Source Descriptions	3
	<u>'include Error Messages</u> 245	5
	Source Protection for Included Files 245	5

<u>B</u>

Interactive Control and Debugging	247
Overview	247
Getting Started	248
Interactive Recovery	250
Getting Help	251
Selecting the Foci of a Debugging Session	253

<u>\$db_setfocus</u> 254
<u>\$db_deletefocus</u>
<u>\$db_enablefocus</u> 255
<u>\$db_disablefocus</u>
<u>\$db_showfocus</u> 256
Stepping through a Simulation
Source Stepping
Stepping in Time
<u>Tracing</u>
<u>\$db_step</u>
<u>\$db_steptime</u>
<u>\$db_settrace</u>
<u>\$db_cleartrace</u>
Setting Breakpoints in a Simulation
Continuous and Non-Continuous Breakpoints
<u>\$db_breakatline</u>
<u>\$db_breakbeforetime</u>
<u>\$db_breakaftertime</u>
<u>\$db_breakwhen</u> 265
<u>\$db_breakonposedge</u> 266
<u>\$db_breakonnegedge</u> 266
<u>\$db_deletebreak</u>
<u>\$db_enablebreak</u> 268
<u>\$db_disablebreak</u> 268
<u>\$db_showbreak</u> 269
Displaying Waveforms
Simvision Waveform Viewer
<u>SHM Tasks</u>
<u>Opening a Database with \$shm_open</u>
Probing Signals with \$shm_probe
Using \$shm_suspend and \$shm_resume
Using \$recordvars and Related Tasks

<u>C</u>	
Maximizing Default Acceleration	285
Overview	285
Controlling the Application of the Default XL Algorithm	286
Items Supported by the Default XL Algorithm	287
Items Unsupported by the Default XL Algorithm	288
Reporting Non-XL Structures Using \$shownonxl	289
Differences between Default XL and Non-XL Algorithms	290
Potential Problems with Default XL Algorithm	291
Measuring and Optimizing Code	292
Estimating Model Speed	292
Establishing a Metric	293
Modeling at Different Levels	293
Reducing Memory Overhead from Switching Algorithms	294
Keeping Primitives Accelerated	295
Modeling Clock Generators	297
Using Behavioral Profiler	297
Using Different Coding Methods	298
Using UDPs	300
Using Event Controls	300
Using Aliases	301
Using Level-Sensitive Behavior	302
Hardware Upgrades	303
Reducing Executed Code	305
Simplifying the Model	305
Changing Your Debugging Style	305
Capturing Simulation Data	306
Reducing Compilation Time	307
Behavioral Performance Improvements	307

D

Stochastic Analysis	309
<u>Overview</u>	309
Queue Management	309

<u>\$q_initialize</u>	0
<u>\$q_add</u> 31	0
<u>\$q_remove</u>	1
<u>\$q_full</u> 31	1
<u>\$q_exam</u>	1
Meaning of the status parameter	2
Probabilistic Distribution Functions	2

<u>E</u>

Software Behavior and Recommendations
<u>Overview</u>
Platform- and Version-Specific Behavior
Restarting from \$save Files Created on Incompatible Hosts
System 5 UNIX C Shell Scripts Running Verilog-XL
Pulse Handling in Verilog-XL 2.0 and Earlier Versions
Use of PLI Routines
Calls to PLI Annotation and \$reset
PLI and Pulse Control
Macro Modules and Port Collapsing
Terminal and Port Lists in Macro Modules
Effect of Port Collapsing on Net Delays
Port Collapsing and 'default_nettype Specifications
Module Paths and Path Simulation
Rules for Path Destination Signals
Path Output Nets With Multiple Drivers in One Module
Path Outputs That Drive Other Path Outputs
Strength Changes That Occur on Path Inputs
Annotation of Multiple Paths with the Same Delay
Using Module Input Port Delays (MIPDs) 322
Conditional Statements
Conditional Statements in Interactive Mode
Evaluation of Expressions in Conditional Statements
Using the 'timescale Compiler Directive
Changing a Parameter During Simulation
Performing Modulo Division on \$random Outputs

325
325
326
326
326
326

<u>F</u>

Verilog-XL Turbo and Twin Turbo Options
<u>Overview</u>
<u>Turbo Option</u>
<u>Twin Turbo Option</u>
Invoking Turbo and Twin Turbo
<u>+turbo</u>
<u>+turbo+2</u>
<u>+turbo+3</u>
<u>+no_speedup</u>
Combining Non-Turbo with Turbo
Twin Turbo Restrictions
Achieving Optimal Performance

<u>G</u>

Code Examples	37
<u> </u>	37
Code Examples	37
conditional_drive.v	37
<u>counter.v</u>	8
<u>dff.v</u>	8
<u>dff_debug.v</u> 33	8
<u>dff_test.v</u>	9
<u>flipflop.v</u>	9
<u>flop.v</u>	9
<u>flop_model.v</u> 34	0
<u>flop_test.v</u>	0
<u>full_adder.v</u> 34	0

guarantee_order.v
<u>half_adder.v</u> 341
<u>harddrive.v</u>
<u>hardreg.v</u>
<u>monitor.key</u>
<u>register.v</u>
<u>register_debug.v</u> 343
register_fixed.fm
<u>register test debug.v</u>
<u>reregister_fixed.v</u>
<u>shortdrive.v</u>
<u>step.v</u>
<u>test.v</u>
<u>test_flop.v</u>
<u>tester.v</u>
<u>time_flop.v</u>
<u>two_bit_adder.v</u> 347
Sample Outputs
<u>ex_signal_values</u> 348
<u>Circuit Diagrams</u>
Graphical Output

<u>H</u>

Veriog-XL Messages 3	353
<u>Overview</u>	353
<u>Message Syntax</u>	353
Message Levels	354
Compilation Error Messages	356
Common compilation error messages	357

<u>Index</u>			
--------------	--	--	--

1

Introducing Verilog-XL

This chapter describes the following:

- <u>The Verilog-XL Simulator</u> on page 19
- <u>Major Features of Verilog-XL</u> on page 19
- <u>The Design Process with Verilog-XL</u> on page 20
- Verilog-XL Online Documents on page 21

The Verilog-XL Simulator

The Verilog®-XL simulator is a software tool that allows you to perform the following tasks in the design process without building a hardware prototype:

- determine the feasibility of new design ideas
- try more than one approach to a design problem
- verify functionality
- identify design errors

To use Verilog-XL, you develop models that describe your design and its environment in the Verilog® Hardware Description Language and then supply Verilog-XL with the file names that contain these models.

Major Features of Verilog-XL

Verilog-XL provides you with the following simulation capabilities:

setting break points during simulation that stops the simulation and allows you to enter an interactive mode to examine and debug your design

- displaying information about the current state of the design and to specifying the format of that information
- applying stimulus during simulation
- patching circuits during simulation
- tracing the execution flow of the statements in your model
- traversing the model hierarchy to various regions of your design to examine the state of the simulation in that region
- stepping through the statements of a design and executing them one at a time
- displaying the active statements in a design
- displaying and disabling the operations you entered in interactive mode
- reading data from a file and writing data to that file
- saving the current state of a simulation in a file and restoring that simulation at another time
- investigating the performance ramifications of architectural decision—stochastic modeling
- simulating with <u>SimVision</u>, the Graphical Analysis Environment for Verilog-XL.

The Design Process with Verilog-XL

The Verilog-XL digital logic simulator lets you perform the following tasks in the design process without building a hardware prototype:

- Try more than one approach to a design problem
- Verify the functional integrity of a design
- Determine the feasibility of new design ideas
- Identify design flaws

The Verilog-XL simulator processes models, which are descriptions of designs that you develop with the Verilog Hardware Description Language (the Verilog HDL).

The design process begins with a design idea and ends with a verified design as shown on the Verilog-XL Task Flow diagram below. Additional tasks support the main design flow and appear below the design flow arrow.

Verilog-XL User Guide

Introducing Verilog-XL

Verilog-XL Task Flow



Verilog-XL Online Documents

Verilog-XL online documentation set comprises the following documents:

For information about	See
New features and enhancements	Verilog-XL What's New
Verilog-XL tasks and commands or this User Guide	Verilog-XL User Guide
Implementation of Verilog HDL by Verilog-XL	Verilog-XL Reference
Modeling for Verilog-XL	Verilog-XL Modeling Style Guide
Using SimVision Graphical Analysis Environment	SimVision User Guide
How to use PLI routines with Verilog-XL	PLI 1.0 User Guide and Reference
PLI Wizard	PLI Wizard User Guide
How to use VPI routines with Verilog-XL	VPI User Guide and Reference
How to use Comparescan	Comparescan User Guide
SDF	SDF Annotator User Guide
Known Problems and their Solutions	Verilog-XL Known Problems and Solutions

Internet News

Talkverilog is a news and information source for Verilog-XL users. To receive Talkverilog, send a message to *talkv@cadence.com* with the word *subscribe* on the subject line of the email message. The subscription is *free* for all Cadence customers.

The communications program is located on the Internet at the *talkv@cadence.com* email address.

Talkverilog provides you with the following information:

- Application notes
- Suggestions for improving productivity
- Benchmark results
- Customer success stories

You are encouraged to send the following to the Verilog-XL Product Team through Talkverilog:

- General questions
- Comments
- Code examples

Invoking Verilog-XL

This chapter describes the following:

- <u>Invoking Verilog-XL</u> on page 23
- <u>Compiling Source Files</u> on page 24
- Using the SimVision Analysis Environment on page 25

Invoking Verilog-XL

To invoke Verilog-XL for compilation and simulation, you type the following at the terminal: verilog <name_of_file>

When you invoke Verilog-XL, SoftShare[™] obtains a license for you or displays a message indicating why it cannot obtain a license.

In the preceding syntax, $< name_of_file >$ is the name of the file containing the Verilog HDL source description that you want to simulate. This document assumes that standard UNIX® paths identify files; path specifications are dependent on the operating system in use.

If the source description is spread over two or more files, then all of the files are given as a space-separated list on the command line:

verilog <file1 file2 ... filen>

In this form, the Verilog-XL compiler reads and processes each of the files, starting at file1and ending at filen. Any number of files can be presented to the Verilog-XL compiler. It is important to note that the text which appears in these files appears as one long stream to Verilog-XL, as if all the text were in one file. This means that the order in which the files are specified can be important.

Verilog-XL options are either predefined or user-defined. Predefined options are invoked on the command line with a hyphen character (-) followed by an option character, or with a plus character (+) followed by one or more option characters. User-defined options are invoked with a plus character (+) followed by the user-defined string. User-defined options are often

Invoking Verilog-XL

referred to as plus options because they must be invoked with the plus character (+) and not the hyphen character (-).

The locations of predefined and user-defined options on the command line are not significant. For example, the following two command lines are equivalent:

```
verilog -x -w -t d8085a.v
verilog d8085a.v -t -w -x
```

The predefined option letters are not sensitive to case, so the following command is equivalent to the previous example:

```
verilog d8085a.v -W -X -T
```

The following is an example of a user-defined option:

```
verilog source1.v +user_string
```

A convenient way to specify a commonly used set of command options and source files is to use the -f option. This option tells Verilog-XL to read additional command-line arguments from a file. The option is described and examples are given in <u>"-f (File Option)"</u> on page 200.

Verilog-XL does not use intermediary files for linking multiple source file descriptions. *All* of the source text for the entire model is always compiled, linked, and loaded before each simulation run.

Compiling Source Files

After you type the command line, the Verilog-XL compiler parses the source text and checks for syntax and semantic errors. The compiler goes through three phases to generate the data structures needed for simulation:

Phase 1

The input source files are read and parsed. Usually most of the source description errors are detected and reported in this phase. Syntax errors are reported in this pass by displaying the line number and the line of source text that contains the error. See <u>"Compilation Error Messages" on page 356</u> for more details.

Phase 2

Modules are linked together in their hierarchical forms during this phase. If necessary, modules are copied in order to create separate data structures for module instances. (This

occurs when a module is instantiated two or more times.) During this phase, module I/O port interconnection errors are reported.

Phase 3

The data structure generated from the first two phases is scanned and further checked for consistency. Also, the hierarchical name references (with two or more identifiers joined together by the period character) are resolved. An error is reported whenever the components of a hierarchical reference name do not follow a path in the hierarchical name structure.

Using the SimVision Analysis Environment

The SimVision analysis environment is the common graphical debug environment for Cadence simulators. SimVision builds upon Cadence's interleaved native-compiled code architecture to provide a powerful design solution. The unified simulation environment optimizes performance and productivity and helps you master the tools you need to design and verify complex systems.

The SimVision environment features advanced debug and analysis tools and innovative highlevel design and visualization capabilities. These tools include:

- The SimVision Console allows you to directly interact with the simulator. You can single step, trace signals, set breakpoints, and observe signals to verify your designs. SimVision also includes other debug tools:
- The Register window lets you use a free-form graphics editor to define any number of register pages, each containing a custom view of the simulation data.
- The Design Browser lets you access the design hierarchy and the signals and variables in the design database.
- The Trace Signals sidebar lets you trace the drivers of a signal.
- The Schematic Tracer displays a Verilog design as a schematic diagram and lets you trace a signal through the design.

If you are using Verilog-XL, invoke SimVision with the +gui option.

% verilog +gui <source_filenames>

See the <u>SimVision User Guide</u> for details on using the debug environment.

Verilog-XL User Guide Invoking Verilog-XL

Verifying Your Design

This chapter describes the following:

- <u>Overview</u> on page 27
- Creating a Test Fixture on page 27
- Displaying the Simulation Time on page 29
- <u>Writing Simulation Data to a File</u> on page 30
- <u>Displaying Signals as Graphical Waveforms</u> on page 32

Overview

This chapter describes how to verify your design.

Creating a Test Fixture

Test fixtures are modules that verify the functionality of other modules. You create a test fixture by declaring a module without ports. The test fixture module instantiates the design you wish to test, applies stimulus to the design, and verifies the simulation results.

The following example shows a test fixture for a 4-bit register. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Creating a test fixture

module harddrive;	// See Step 1	
<pre>'define stim #100 data = 4'b reg clk, clr; reg [3:0] data:</pre>	// See Step 2	
<pre>wire [3:0] q; event end_first_pass; hardreg h1 (data, clk, clr, q);</pre>	// See Step 3	
initial begin		

Verilog-XL User Guide

Verifying Your Design

```
clr = 1; clk = 0;
$monitor("time=%0t,data=%b,clk=%b,clr=%b,q=%b",$time, data, clk, clr, q);
                                                                                 11
See Step 4
end
always #50 clk = ~clk;
                                           // See Step 5
                                           // See Step 5
always @(end_first_pass) clr = ~clr;
initial
begin
repeat (2)
begin
data = 4'b0000;
                                    // See Step 5
`stim 0001;
`stim 0010;
`stim 0011;
`stim 0100;
`stim 0101;
`stim 0110;
`stim 0111;
`stim 1000;
`stim 1001;
`stim 1010;
`stim 1011;
`stim 1100;
`stim 1101;
`stim 1110;
`stim 1111;
#200 ->end_first_pass;
end
$finish;
end
                                                // See Step 6
endmodule // harddrive
%verilog harddrive.v hardreg.v flop.v // See Step 7
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
time=0,data=0000,clk=0,clr=1,q=xxxx
time=50,data=0000,clk=1,clr=1,q=xxxx
time=100,data=0001,clk=0,clr=1,q=xxxx
time=139,data=0001,clk=0,clr=1,q=0000
time=150,data=0001,clk=1,clr=1,q=0000
time=3350,data=1111,clk=1,clr=0,q=0000
L40 "harddrive.v": $finish at simulation time 3400
481 simulation events + 1082 accelerated events + 936 timing check events
CPU time: 0.5 secs to compile + 0.3 secs to link + 0.2 secs in simulation
```

1. Begin a test fixture with the module statement. You must provide a module name for your test fixture, but test fixtures typically do not have port declarations.

This module is called harddrive, which is a test fixture for the hardreg module.

2. In addition to any variables you need in your test fixture, declare the port variables for the design that you are testing.

Declare the following ports of hardreg: clk, clr, data, and q.

- **3.** Instantiate the module or modules that you want to test. The module being tested by the harddrive test fixture is hardreg, a 4-bit register. The instance is called h1.
- **4.** Monitor the simulation with display commands. You can view simulation results in any number of ways. *\$monitor* displays each of the h1 port variables each time a port variable changes value.
- 5. Verify complete functionality by stimulating your design as fully as possible. The stimulus for h1 includes defining a clock oscillator, toggling clr, and applying all possible 0 and 1 combinations to the data bus. A more thorough testing would include all combinations of X and Z.
- 6. End the module definition with the endmodule statement.
- 7. Simulate your test fixture and analyze the results to verify the behavior of your design. The test fixture file is harddrive.v; hardreg.v and flop.v are the files that define the 4-bit register design.

Note: At step 7, you need to copy this module in some other file and compile this module in the place of harddrive.v.to get results that match the following

Displaying the Simulation Time

You can ask Verilog-XL for the current simulation time, which is most commonly used in formatted output statements. Three system tasks return the simulation time: \$time (64-bit integer), \$stime (32-bit integer), and \$realtime (64-bit real value). Verilog-XL returns all simulation times scaled to the timescale unit of the current module.

The following example shows each of the simulation time system tasks used in a formatted output statement. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Displaying the simulation time

```
'timescale 10 ns / 1 ps // See Step 1
module times;
time x; // See Step 2
real y;
integer z;
reg a;
parameter p = 1.55;
initial
begin
    x = $time;
    y = $realtime;
    z = $stime;
```

Verilog-XL User Guide Verifying Your Design

```
$display ("$time = %0t, $realtime = %f,
                $stime = %0d", x, y, z);
                                                  // See Step 4
    #10 $monitor ($time, $realtime, $stime,
                "a = %b", a);
                                           // See Step 5
    \#p a = 0;
    #p a = 1;
end
endmodule // times
% verilog time.v
                                        // See Step 6
Compiling source file "time.v"
Highest level modules:
times
$time = 0, $realtime = 0.000000, $stime = 0
           10 10a = x
    10
           11.55
    12
                        12a = 0
    13
                  13a = 1
           13.1
16 simulation events
CPU time: 0.7 secs to compile + 0.2 secs to link + 0.0 secs in simulation
```

- 1. Verilog-XL scales all system times to the timescale specified by `timescale.
- 2. If you need to store system times, declare time variables: time (64-bits) for \$time, real (64-bits) for \$realtime, and integer (32-bits) or time (64-bits) for \$stime.
- **3.** Assign system times to variables of the proper declared type.
- **4.** Use system-time variables anywhere the associated data type is legal. The variables containing the different forms of the system time are displayed with *\$display*.
- 5. Use system time tasks directly as arguments to formatted display tasks such as \$monitor.
- 6. Simulate the design to see the results of the different system time formats. Note that when used as direct arguments to a formatted-output system task, only *\$stime* displays its value with as small a field width as possible.

Writing Simulation Data to a File

You can save simulation data by writing to a file. All of the formatted output system tasks have a version that writes to a file (*\$fdisplay*, *\$fwrite*, *\$fstrobe*, and *\$fmonitor*). You must first open a file with *\$fopen*, and you should close the file with *\$fclose* before ending the simulation.

The example shows how you write simulation data to files other than the standard log file. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Verilog-XL User Guide

Example: Writing simulation data to a file

```
module write_files;
                         // See Step 1
integer messages,
                broadcast,
                cpu_chann,
                alu_chann;
initial
begin
                                        // See Step 2
    cpu chann = $fopen("cpu.dat");
    alu_chann = $fopen("alu.dat");
    if(cpu_chann == 0) begin
                                          // See Step 3
        $display ("Unable to open cpu.dat");
        $finish;
        end
    if(alu_chann == 0) begin
        $display ("Unable to open alu.dat");
        $finish;
        end // if
    messages = cpu_chann | alu_chann;
                                              // See Step 4
    broadcast = 1 | messages;
end
initial
begin
    $fdisplay(broadcast, "Written to all open channels,
                                          // See Step 5
including standard output");
    $fdisplay(messages, "Written to all open channels, except
standard output");
    $fdisplay(cpu_chann, "Written to cpu_chann only");
                                      // See Step 6
    $fclose(cpu_chann);
    $fdisplay(broadcast, "cpu_chann is no longer open");
end
endmodule // write_files
                                 // See Step 7
% verilog file.v
Compiling source file "file.v"
Highest level modules:
write_files
Written to all open channels, including standard output
cpu_chann is no longer open
Warning! Channel 1 not open [Verilog-CHNOP]
"file.v", 26: $fdisplay(broadcast, "cpu chann is
no longer open");
1 warning
16 simulation events
CPU time: 0.4 secs to compile + 0.1 secs to link + 0.0 secs in simulation
                                       // See Step 8
% more alu.dat
Written to all open channels, including standard output
Written to all open channels, except standard output
cpu_chann is no longer open
% more cpu.dat
Written to all open channels, including standard output
Written to all open channels, except standard output
Written to cpu_chann only
```

1. Declare integers to hold the unique file identifiers, called multi-channel descriptors (MCD), for each file or combination of files to which you will write. Each MCD is a set of

32 flags with each flag (bit position) representing a single output channel. Verilog-XL implicitly declares a channel for standard output (the terminal screen and log file) whose value is 1 (bit 0 is set). Therefore, your first call to <code>\$fopen</code> returns 2 (bit 1 is set), the next call returns 4 (bit 2 is set) and so on. The <code>write_files</code> module has four channels in addition to standard output.

- 2. Open each file with <code>\$fopen</code>, specifying the name of the file as the sole argument. The <code>\$fopen</code> system task returns a multi-channel descriptor (<code>integer</code>) for the file if Verilog-XL successfully opens the file or 0 if Verilog-XL could not open the file for writing. Verilog-XL opens both <code>cpu_chann</code> and <code>alu_chann</code> for writing.
- **3.** Verify that Verilog-XL opened your files. If Verilog-XL could not open either file, the simulation ends.
- 4. Write to multiple channels by creating new channels that are bitwise ORs of existing channels. The messages channel consists of both cpu_chann and alu_chann, and broadcast also includes standard output (MCD of 1).
- 5. Write to open files with any of the following system tasks: \$fdisplay, \$fwrite, \$fmonitor, and \$fstrobe. These system tasks are identical to their non-file counterparts except that the first argument is the MCD.
- 6. Close files with \$fclose, specifying the MCD of the file you want to close. Verilog-XL will reuse MCDs of closed files if you make subsequent calls to \$fopen. Verilog-XL automatically closes files that you have not closed when the simulation ends.
- 7. Run your simulation. Notice the warning when an attempt is made to write to a closed file.
- 8. When you run your simulation, Verilog-XL creates files in the run directory unless you specified paths when you opened the files with \$fopen. Examine these files as you would any ASCII file.

Displaying Signals as Graphical Waveforms

You can more easily analyze your input and output signals by displaying them as graphical waveforms.

The example shows how you create an SHM (Simulation History Manager) database and select signals from your simulation to write to the database for later viewing with Simvision Waveform Viewer. The example shows how you invoke Simvision Waveform Viewer from the command line and from the SimVision window. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Displaying signals as graphical waveforms

Verilog-XL User Guide

Verifying Your Design

```
// Test fixture for flop (test flop.v)
module test_flop;
reg data, clock;
flipflop f1 (clock, data, qa, qb);
initial
begin
    clock = 0; data = 0;
    #10000
               $shm_close();
                                     // Close after time 10000
    $finish;
end
initial
begin
                                                // See Step 1
    $shm_open("db1.shm");
    $shm_probe(clock);
                                                // See Step 2
    $shm_probe(data,qa,qb);
    $shm probe(f1.nt1);
end // test_flop
always #100 clock = ~clock;
always #300 data = ~data;
endmodule // test_flop
// Model RS Flip Flop (flipflop.v)
module flipflop (clock, data, qa, qb);
input clock, data;
output qa, qb;
    nand #10 ndl (a, data, clock),
    nd2 (b, ndata, clock),
    nd3 (qa, a, qb),
    nd4 (qb, b, qa);
    mynot nt1 (ndata, data);
endmodule // flipflop
module mynot (out, in);
output out;
input in;
    not(out,in);
endmodule // mynot
% verilog test flop.v flipflop.v // See Step 3
    . . .
% ls db1.shm
                                  // See Step 4
db1.trn db1.dsn
% simvision -waves db1.shm
                                                     // See Step 5
```

1. Open an SHM database with \$shm_open. If you do not specify a filename, the default is waves.shm. You can also provide a second argument to \$shm_open that names a remote SHM process that processes your requests.

The test_flop module opens the SHM file db1.shm.

2. Specify which signals Verilog-XL stores waveform information for with \$shm_probe. You can explicitly name nodes or provide the hierarchical name of a module instance that contains nodes you want to monitor. When you specify a module instance, you can optionally follow the instance name with a node-specifier string that limits which signals Verilog-XL monitors. Note that you cannot specify module instances of Verilog HDL primitives as arguments to \$shm_probe. Verilog-XL monitors <code>clock</code>, <code>data</code>, <code>qa</code>, <code>qb</code>, and all the inputs and outputs of the <code>nt1</code> instance.

- 3. Simulate.
- 4. Verify that the simulation created the appropriate SHM database.
- 5. Invoke and load your SHM database from the command line by specifying its file name on the command line with the simvision -waves command.

Alternately, you can select *Windows–New–Waveform...* from the SimVision window to display signals in the Simvision Waveform Viewer.

SimVision: Design	Browser 1	· 🗆
<u>S</u> elect E <u>×</u> plore S <u>i</u> mulation	Windows	<u>H</u> elp
× 🛱 🚳 💏 •	<u>N</u> ew ♪ Tools ↓	, D <u>e</u> sign Browser Wav <u>e</u> form
0 ▼Ins ▼Int Time: 0	<u>1</u> . Design Browser 1 <u>2</u> . Console	S <u>o</u> urce Browser S <u>c</u> hematic Tracer Signal Flow Brows
Signal/Varia	Tile <u>H</u> orizontally Tile <u>V</u> ertically <u>C</u> ascade	R <u>eg</u> ister Expression Calcula M <u>e</u> asurement
veform window	Windows	Assertion Browser

Debugging Your Design

This chapter describes the following:

- <u>Overview</u> on page 35
- <u>Setting Event-Triggered Breakpoints</u> on page 36
- <u>Setting Code-Line Breakpoints</u> on page 42
- <u>Activating and Deactivating Commands</u> on page 46
- Examining Code and Simulation Effects on page 48
- <u>Displaying Expanded Macros</u> on page 50
- <u>Traversing Model Hierarchy</u> on page 52
- Observing All Simulation Events on page 57
- Observing a Focused Set of Simulation Events on page 58
- Observing Wires and Registers Periodically on page 59
- <u>Observing Wires and Registers When They Change Value</u> on page 60
- Examining Wires and Registers Now on page 62
- Patching a Model (Asking "What If" Questions) on page 64
- Ordering Events in a Time Cycle on page 67
- <u>Displaying, Strobing, and Monitoring Data</u> on page 70
- Controlling the Display and Interpretation of Time on page 72
- Reinitializing the Network and Simulator Clock on page 73

Overview

This chapter describes how to debug your design.

November 2008

Setting Event-Triggered Breakpoints

You can interrupt simulation at a specific point to debug your model. Event-driven interrupts include:

- time-triggered breakpoints
- edge-triggered breakpoints
- level-sensitive breakpoints

This section shows you how to interactively specify time-triggered, edge-triggered, and levelsensitive breakpoints. It shows you how to specify a repetitive breakpoint, and how to cancel that breakpoint. (You can also set these breakpoints from a test fixture.)

The following example shows the command-line example. (<u>"Example: Setting event-</u> triggered breakpoints with SimVision" on page 38 shows the same example with SimVision.)

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Setting event-triggered breakpoints

```
verilog harddrive.v hardreg.v flop.v -s
                                         // See Step 1
%
C1 > $showvars;
                                                // See Step 2
Variables in the current scope:
clk (harddrive) reg = 1'hx, x
clr (harddrive) reg = 1'hx, x
data (harddrive) reg = 4'hx, x
q[3] (harddrive) wire = StX
    StX <- (harddrive.h1.f4): nand nd7(q, e, qb);</pre>
q[2] (harddrive) wire = StX
    StX <- (harddrive.h1.f3): nand nd7(q, e, qb);</pre>
q[1] (harddrive) wire = StX
    StX <- (harddrive.h1.f2): nand nd7(q, e, qb);</pre>
q[0] (harddrive) wire = StX
    StX <- (harddrive.h1.f1): nand nd7(q, e, qb);</pre>
C2 > #10 $showvars;
                                      // See Step 3
C3 > @ (posedge clk) $stop;
                                      // See Step 4
C4 > wait (q == 1) $stop;
                                      // See Step 5
                                     // See Step 6
C5 >
Variables in the current scope:
clr (harddrive) reg = 1'h1, 1
clk (harddrive) reg = 1'h0, 0
data (harddrive) reg = 4'h0, 0
q[3] (harddrive) wire = StX
StX <- (harddrive.h1.f4): nand nd7(q, e, qb);</pre>
q[2] (harddrive) wire = StX
StX <- (harddrive.h1.f3): nand nd7(q, e, qb);</pre>
q[1] (harddrive) wire = StX
StX <- (harddrive.h1.f2): nand nd7(q, e, qb);
q[0] (harddrive) wire = StX
StX <- (harddrive.h1.f1): nand nd7(q, e, qb);</pre>
```
Verilog-XL User Guide

Debugging Your Design

```
C3: $stop at simulation time 50.
C5 > .
at time 50 clr = 1 data= 0 q= x
at time 150 clr = 1 data= 1 q= 0
C4: $stop at simulation time 229
C5 > forever @ (posedge clk) $stop;
                                        // See Step 7
C6 > .
C5: $stop at simulation time 250
C6 > .
at time 250 clr = 1 data= 2 q= 1
C5: $stop at simulation time 350
C6 > $history;
                               // See Step 8
Command history:
C1
    $showvars;
C2
       #10
           $stop;
С3
      @(posedge clk)
           $stop;
C4 wait(q == 1)
           $stop;
C5* forever
         @(posedge clk)
               $stop;
C6* $history;
C7 > -5;
                                 // See Step 9
C7 > .
C7 > .
L47 "harddrive.v": $finish at simulation time 3400
1536 simulation events
CPU time: 1.2 secs to compile + 0.2 secs to link + 0.3 secs in simulation
```

- 1. Invoke Verilog-XL with the -s (stop) option to put you in interactive mode at simulation time 0.
- 2. Look at signals with \$showvars.
- 3. Show value of variables after 10 time units.
- 4. Create an edge-sensitive breakpoint on clk.
- **5.** Create a level-sensitive breakpoint when q equals 1.
- 6. Continue (.) the simulation.
- 7. Use forever for any breakpoint that you want to repeat.

Stop on every positive edge of clk.

8. To deactivate the forever command, you need the command's assigned number. If you do not know the assigned number of your forever command, show it with \$history.

Active commands are indicated with an asterisk (*).

In the example, 5 is the forever command's number.

9. Deactivate command number 5 with -5.

Example: Setting event-triggered breakpoints with SimVision

To set event-triggered breakpoints with SimVision:

1. Invoke the Verilog-XL with the <u>SimVision</u>[™] graphical environment using the following command:

verilog <u>harddrive.v</u> <u>hardreg.v</u> <u>flop.v</u> -s +gui

2. The values of the signals and/or variables can be seen in the Value column of the *Design Browser*. You need to do the following settings to see the values by default. Select Edit–Preferences. The Preferences dialog box appears.

		Preferences		
	General Opti Signal Opti VHDL Verilc Verilc Verilc Toolnet C Design Brow: Signal Lik	Startup Defaults Show Edit Buffer Show Module/Unit Names Show Values: When connected to simula	tor	
	ОК	Cancel Apply Help		

Select the Design Browser option and then select 'When connected to simulator' option from the Show Values list box.

- 1. In the Simulator window, type #10 \$showvars; at the command prompt to see the value of the variable at 10 time units
- 2. Set the breakpoint at the positive edge of the clock by selecting *Simulation–Set Breakpoint–Signal...* and typing clk in the *Signal* field of the *Set Breakpoint* dialog box. Set the *On* cyclic field from *Stop on Positive Edge* and enable the *set once and delete* button. Click *OK* to set the breakpoint.

-	SimVision: Set Breakpoint 🕢 📃					
Break based on Time Line Signal						
Sigr	Add 🔃 Browser 🔹					
te S	 ●f Stop on positive edge ▼ ※ Set once and delete ▼ 					
0	K Cancel Apply Help					

SimVision produces output from these actions in the Simulator Window of the SimVision window as follows:

C6 > \$db_breakonceonposedge(clk); Set break (2) [once] on pos edge harddrive.clk.

3. Set the next breakpoint for when object *q* equals 1 as shown in the following figure. Select *Simulation–Set Breakpoint–Signal…* and type q[3] in the *Signal* field of the Set Breakpoint dialog box. Set the *On* cyclic field to *Stop whenvalue is* and type a value of 1 in the field next to it. Enable the set once and delete button and click OK to set the breakpoint. Repeat these steps for q[2], q[1], and q[0].

Note: You can set all vectors of *q* at the same time by typing the following command at the interactive prompt:

wait (q == 1) \$stop; // setting a level-sensitive breakpoint

4. Click on the *Run Simulation* button **b** two times, which produces the following output in the Simulator Window of the SimVision window:

```
C11 > .
Variables in the current scope:
clr (harddrive) reg = 1'h1, 1
clk (harddrive) reg = 1'h0, 0
data (harddrive) reg = 4'h0, 0
q[3] (harddrive) wire = StX
StX <- (harddrive.h1.f4): nand nd7(q, e, qb);
q[2] (harddrive.h1.f3): nand nd7(q, e, qb);
q[1] (harddrive) wire = StX
StX <- (harddrive.h1.f2): nand nd7(q, e, qb);</pre>
```

```
q[0] (harddrive) wire = StX
StX <- (harddrive.hl.fl): nand nd7(q, e, qb);
Break (1) [once] occured on pos edge harddrive.clk at time 50.
Disabled break (1) [once] on pos edge harddrive.clk.
Cl1 > .
at time 50 clr =1 data= 0 q= x
at time 150 clr =1 data= 1 q= 0
Break (6) [once] occured when harddrive.q[0] = 1 at time 229.
Disabled break (6) [once] when harddrive.q[0] = 1.
Cl1 >
```

5. Issue the following command then click on the *Run Simulation* button kince. The following is displayed in the Simulator Window:

```
Cl1 > forever @ (posedge clk) $stop;
Cl2 > .
Cl1: $stop at simulation time 250
Cl2 > .
at time 250 clr =1 data= 2 q= 1
Break (5) [once] occured when harddrive.q[1] = 1 at time 329.
Disabled break (5) [once] when harddrive.q[1] = 1.
```

6. To deactivate the forever command, you need the command's assigned number. If you do not know the assigned number, select *Show–History* which produces the following output in the Simulator Window.

Active commands are indicated with an asterisk (*). In the example, 11 is the forever command's number.

```
C12 > $history;
Command history:
C1
    $display("clk, = ", clk, );
    $display("clr, = ", clr, );
$display("data, = ", data, );
C2
C3
C4 $display("q[3],q[2],q[1],q[0] = ", q[3], q[2], q[1], q[0]);
    $db_breakbeforetime($time + 10);
C5
C6
    $db_breakonceonposedge(clk);
    $db_breakoncewhen(q[3], 1);
$db_breakoncewhen(q[2], 1);
C7
C8
    $db_breakoncewhen(q[1], 1);
C9
Cl0 $db_breakoncewhen(q[0], 1);
Cl1* forever
         @(posedge clk)
              $stop;
C12* $history;
```

- 7. Deactivate command number 11 by typing -11 on the command line.
- **8.** Click the *Run Simulation* button **b** to continue the simulation.

Steps 9 and 10 produce the following output in the Simulator Window:

```
Cl3 > -11

Cl3 > .

at time 350 clr =1 data= 3 q= 2

at time 450 clr =1 data= 4 q= 3

Break (4) [once] occured when harddrive.q[2] = 1 at time 529.

Disabled break (4) [once] when harddrive.q[2] = 1.
```

```
C13 > .
at time 550 clr =1 data= 5 q= 4
at time 650 clr =1 data= 6 q= 5
at time 750 clr =1 data= 7 q= 6
at time 850 clr =1 data= 8 q= 7
Break (3) [once] occured when harddrive.q[3] = 1 at time 929.
Disabled break (3) [once] when harddrive.q[3] = 1.
C13 > \dots
```

Setting Code-Line Breakpoints

You can execute your code up to a specified line, then single-step through it.

The examples in this section show how to display the source code line numbers, set a breakpoint at one of those line numbers, and single-step through that source code.

The following example shows the command line interface example. (<u>"Example: Setting code-line breakpoints with SimVision</u>" on page 43 shows the same example with SimVision.) "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Setting code-line breakpoints

```
% verilog shortdrive.v hardreg.v flop.v -s
                                               // See Step 1
C1 > $list;
                                                       // See Step 2
1
   module shortdrive;
2
       reg [3:0]
2
            data; // = 4'hx, x
3
       req
3
            clk, // = 1'hx, x
3
            clr; // = 1'hx, x
4
       wire [3:0]
4
            q; // = 4'hx, x (scalared)
5
       hardreg
5
           h1(data, clk, clr, q);
6*
       initial
7
           begin
8
                repeat(2)
9
                    begin
10
                          data = 4'b0;
11
                          #100
11
                              data = 4'b1;
12
                          #100
12
                              data = 4'b10;
13
                     end
14
                 $finish;
             end
15
16
    endmodule
C2 > $db breakonceatline (8);
                                                       // See Step 3
Set break (1) [once] at line 8, scope shortdrive, file shortdrive.v.
                                                       // See Step 4
C3 >
Break (1) [once] occured at line 8, scope shortdrive, file shortdrive.v, time 0.
```

Verilog-XL User Guide Debugging Your Design

```
Variables in the current scope:
clk (shortdrive) wire = HiZ]
clr (shortdrive) wire = HiZ
q (shortdrive) wire = StX
   StX <- (shortdrive.h1): port 4</pre>
data (shortdrive) reg = 4'hx, x
C4 >
                                            // See Step 6
L8 "shortdrive.v": repeat(2) >>> 32'h2, 2
C4 >
L9 "shortdrive.v": begin
C4 > ,
L10 "shortdrive.v": data = 4'b0;
C4 > .
                                           // See Step 7
L14 "shortdrive.v": $finish at simulation time 400
```

- **1.** Invoke Verilog-XL with the -s option to get into interactive mode.
- 2. Display the line numbers for the model source code with *\$list* to see where you want to set a breakpoint. The asterisk (*) indicates an active command.
- 3. Set a breakpoint at line 8, the repeat statement, with \$db_breakonceatline.
- **4.** Continue (.) the simulation.
- 5. Display the values of signals with \$showvars.
- 6. Single-step (,) through the source code.
- 7. Continue (.) the simulation.

Example: Setting code-line breakpoints with SimVision

The following example shows how to set code-line breakpoints with SimVision.

1. Invoke the Verilog-XL with the SimVision Graphical Analysis environment using the following command:

```
verilog <u>shortdrive.v</u> <u>hardreg.v</u> <u>flop.v</u> -s +gui
```

2. To set a line breakpoint on line 8, select *Simulation–SetBreakpoint–Line* and fill in the SetBreakpoint dialog box. When a line breakpoint is set, a small break icon (⁺) appears next to the line number in the Source Browser.

- SimVision: Set Breakpoint						
Break based on						
Time Line Signal		_				
n Add 🦉 Browser 🔹 📄 Sourc	e .	•				
Line: File:						
OK Cancel Apply	Ηe	elp 				

1. Select the Scope Shortdrive by clicking the left mouse button

Note: You can also choose *Select–Signals* to accomplish the same task, but viewing all the signals by scope is a more **affer**ient method.

Then click on the Register Window President button. A new Register Window appears.

2. Rename the *Register Window* to *shortdrive*. The *Register Window* appears as shown below.

— SimVision: shortdrive		
<u>F</u> ile <u>E</u> dit <u>V</u> iew E <u>x</u> plore For <u>m</u> at <u>P</u> age S <u>i</u> mu	ulatio	on
<u>W</u> indows	H	elp
Page-1 📑 💣 🛛 🐰 🛍 K 🖡	n (r× :
TimeA ➡ = 0 ➡ ns ➡ /ित्ते ◄ (
🚺 🔣 🕻 👬 📅 í 🍏 í Simulation Time: O		
v∰v clk =x v∰v clr =x v∰v. data= 'hx		\$ \
Vin. q = ² hx		•
0 objects s	elec	ted

Verilog-XL User Guide

Debugging Your Design

3. Click the *Run Simulation* button and the simulation will stop at the line 8 breakpoint you had set.

Click on the Single Step button three times, advancing to line 11. The Register Window (below) shows the data signal with a value of 0000.



4. Click on the *Run Simulation* button **I** to complete the simulation.

Activating and Deactivating Commands

You can periodically activate and deactivate debugging commands, such as breakpoint commands. The following example shows how to display the number assigned to a command and how to then use that number to activate and deactivate a command. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Activating and deactivating commands

```
% verilog <u>harddrive.v</u> <u>hardreg.v</u> flop.v -s // See Step 1
```

Verilog-XL User Guide

Debugging Your Design

```
// See Step 2
C1 > $showvars (clk);
clk (harddrive) reg = 1'hx, x
                                                      // See Step 3
C2 > forever @ (posedge clk) $stop;
C3 > $history;
                                                      // See Step 4
Command history:
C1 $showvars(clk);
C2* forever
        @(posedge clk)
            $stop;
C3* $history;
                                                      // See Step 5
C4 >
C2: $stop at simulation time 50
                                                  // See Step 6
C4 > -2;
C4 > $history;
                                                  // See Step 7
Command history:
C1
   $showvars(clk);
C2 forever
        @(posedge clk)
            $stop;
C3 $history;
C4* $history;
C5 > 2;
                                                 // See Step 8
                                                 // See Step 9
C5 > $history;
Command history:
C1 $showvars(clk);
C2* forever
        @(posedge clk)
            $stop;
C3 $history;
C4 $history;
C5* $history;
C6 > .
at time 50 clr = 1 data= 0 q= x
C2: $stop at simulation time 150
```

- 1. Invoke Verilog-XL with -s to get into interactive mode. To invoke Verilog-XL with SimVision, add the +gui plus option to the command line.
- 2. Look at the clk signal with \$showvars.
- 3. Set a repetitive breakpoint with forever.
- **4.** With *shistory*, verify that the *forever* command is now active. If you are running SimVision, you can select *Show-History*.

The asterisk (*) indicates an active command. Remember that in this session, the forever command is number 2.

- 5. Continue (.) the simulation.
- 6. Deactivate command number 2 (forever).
- 7. With \$history, note that forever is deactivated (it has no asterisk).
- 8. Activate the forever command by typing its number (2).

9. With \$history, note that forever is activated (the command number is followed by an asterisk).

Examining Code and Simulation Effects

You can stop and look at your source code, see the values of signals, and see what commands are active and not active.

This section shows fragments of a decompiled test fixture. It shows that listing the source code during simulation also displays the changes in values caused by simulation.

The following example shows the command-line interface example. (<u>"Example: Examining</u> <u>Code and Simulation Effects with SimVision</u>" on page 49 shows the same example with SimVision.) "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Examining Code and Simulation Effects

<pre>% verilog <u>harddrive.v</u> <u>hardreg.v</u> <u>flop.v</u> -s //</pre>	See Step 1
C1 > \$list;	// See Step 2
<pre>// harddrive.v 1 module harddrive; 2 reg 2 clk, 2 clr;</pre>	// = 1'hx, x // = 1'hx, x
 16* always 16 #50 16 clk = ~clk;	
C2 > #200 \$stop; C3 > .	// See Step 3 // See Step 4
actual : at time 50 clr =1 data= 0 q= x at time 150 clr =1 data= 1 q= 0 C2: \$stop at simulation time 200	
C3 > \$list; // harddrive.v 1 module harddrive; 2 reg 2 clk, // = 1'h1, 1 2 clr; // = 1'h1, 1	// See Step 5
16 always 16* #50 16 clk = ~clk;	

1. Invoke Verilog-XL with – s to get into interactive mode.

2. Show decompiled code with \$list.

\$list shows current values of declared signals. At time 0, clk and clr are undefined.
Active commands are indicated with an asterisk. The always command is now active.
Numbers represent source line location in the source file.

- **3.** Simulate for 200 time units, then stop.
- **4.** Continue (.) the simulation.
- 5. Show decompiled code with \$list. Current values of declared signals are shown. At time 200, clk and clr are both 1. Active commands are indicated with an asterisk. The delay (#) is now active.

Example: Examining Code and Simulation Effects with SimVision

The following example shows how to examine code and simulation effects with SimVision.

1. Invoke the Verilog-XL with the SimVision graphical environment using the following command:

verilog <u>harddrive.v</u> <u>hardreg.v</u> <u>flop.v</u> -s +gui

2. Select signals clk and clr and select *Describe* by right clicking.

The Simulator Window shows the following output:

```
C1 > $showvars(clk,clr);
clk (harddrive) reg = 1'hx, x
clr (harddrive) reg = 1'hx, x
```

3. Type the following command in the Simulator window:

#200 \$stop;

4. Click the *Run Simulation* button **b** to continue the simulation.

The Simulator Window shows the following output for steps 3 and 4:

```
C2 > #200 $stop;
C3 > .
at time 50 clr =1 data= 0 q= x
at time 150 clr =1 data= 1 q= 0
C2: $stop at simulation time 200
```

5. Repeat step 2 to show the value of clk and clr. The Simulator Window shows the following values for clk and clr:

```
C3 > $showvars(clk,clr);
clk (harddrive) reg = 1'h1, 1
clr (harddrive) reg = 1'h1, 1
```

Note: If you have a Register window open with clk and clr, the values are

automatically displayed, as shown.



Displaying Expanded Macros

To be certain that Verilog-XL expands your macros the way you planned, you can display a listing of expanded (decompiled) source code.

The following example contrasts unexpanded and expanded versions of macros defined with 'define, 'ifdef, and 'else. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Displaying expanded macros

module harddrive;	// See Step 1
reg clk, clr;	-
reg [3:0] data;	
wire [3:0] q;	
`define stim #100 data = 4'b	// See Step 2
hardreg h1 (data, clk, clr, q);	

Verilog-XL User Guide

Debugging Your Design

```
initial
begin
    clr = 1;
    clk = 0;
end
                                     // See Step 3
`ifdef fast
 always #50 clk = ~clk;
`else
  always #100 clk = ~clk;
`endif
initial
begin
    data = 4'b0000;
    `stim 0001;
                                       // See Step 4
    `stim 0010;
  $finish;
end
endmodule // harddrive
% verilog conditional drive.v hardreg.v flop.v -s +define+fast
// See Step 5
                                       // See Step 6
C1 > $list;
// conditional_drive.v
2 module conddrive;
3
       req
3
            clk; // = 1'hx, x
4
       req
4
            clr; // = 1'hx, x
5
       reg [3:0]
5
            data; // = 4'hx, x
       wire [3:0]
6
6
            q; // = 4'hx, x (scalared)
7
       hardreg
7
           h1(data, clk, clr, q);
8*
       initial
9
            begin
10
                 clr = 1;
                 clk = 0;
11
12
             end
13*
        always
14
             #50
14
                 clk = ~clk;
15*
        initial
16
             begin
17
                 data = 4'b0;
                 #100
18
18
                     data = 4'b1;
19
                 #100
19
                     data = 4'b10;
20
                 $finish;
21
             end
22
    endmodule
```

- 1. The harddrive test fixture contains two macros: one with `define, another with `ifdef, `else, and `endif.
- 2. The compiler substitutes the text string, "#100 data = 4'b" where `stim appears in the model.

- **3.** If you specify +define+fast on the Verilog-XL command line, the compiler uses the `ifdef clk timing (#50). If you do not, the compiler uses the `else timing (#100) for the signal clk.
- **4.** The compiler replaces `stim with the text string, "#100 data = 4'b".
- 5. Invoke Verilog-XL with -s to go into interactive mode and with +define+ to pass a value to the compiler. Because the argument to +define+ is *fast*, Verilog-XL uses the faster clock in the *'ifdef* clause.
- 6. Look at your decompiled code with \$list.
- 7. Note that the compiler uses the fast (#50) timing for signal clk.
- 8. Note that the `stim macro has been replaced with the text string "#100 data = 4'b".

Traversing Model Hierarchy

Like a UNIX[®] directory tree, Verilog HDL models have hierarchy. Just as you can view a lowlevel UNIX directory by either specifying a full path or by moving to that directory with the cd command and listing the contents, you can view low-level Verilog HDL models and their components by specifying a full path or by moving to that level (or scope) and then viewing the contents.

This section shows you how to use scoping to walk you down the hierarchy of a two-bit adder. The following example shows the command line interface example. (<u>"Example:</u> <u>Traversing model hierarchy with SimVision</u>" on page 54 shows the same example with SimVision.) "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Traversing model hierarchy

```
% verilog half_adder.v full_adder.v two_bit_adder.v tester.v -s
// See Step 1
C1 > $showscopes;
                                  // See Step 2
Directory of scopes at current scope level:
module (two_bit_adder), instance (under_test)
Current scope is (tester)
Highest level modules:
tester
                                       // See Step 3
C2 > $scope (under_test);
C3 > $showscopes;
                                               // See Step 4
Directory of scopes at current scope level:
    module (full_adder), instance (p0)
    module (full_adder), instance (p1)
```

Verilog-XL User Guide Debugging Your Design

```
Current scope is (tester.under_test)
Highest level modules:
tester
C4 > $scope (p1);
                                               // See Step 5
C5 > $showscopes;
                                               // See Step 6
Directory of scopes at current scope level:
    module (half_adder), instance (m1)
    module (half_adder), instance (m2)
Current scope is (tester.under_test.pl)
Highest level modules:
tester
                                                // See Step 7
C6 > $scope (m2);
C7 > $showscopes;
                                               // See Step 8
Directory of scopes at current scope level:
Current scope is (tester.under test.pl.m2)
Highest level modules:
tester
C8 > $showvars;
                                               // See Step 9
Variables in the current scope:
a (tester.under_test.pl.m2) wire = StX
   StX <- (tester.under_test.pl.ml): xor nl(sum, a, b);</pre>
b (tester.under_test.pl.m2) wire = StX
   StX <- (tester.under_test.p0): or m3(c_out, ci2, ci1);</pre>
sum (tester.under test.pl.m2) wire = StX
   StX <- (tester.under_test.pl.m2): xor nl(sum, a, b);</pre>
c_out (tester.under_test.pl.m2) wire = StX
   StX <- (tester.under_test.pl.m2): and n2(c_out, a, b);</pre>
C9 > $scope (tester.under_test);
                                               // See Step 10
C10 > $showvars (tester.under_test.p0.si);
si (tester.under_test.p0) wire = StX
   StX <- (tester.under_test.p0.ml): xor n1(sum, a, b);</pre>
```

- 1. Load four models into Verilog-XL: tester, which instantiates two_bit_adder, which instantiates full_adder, which instantiates half_adder.
- 2. With \$showscopes, display the module names and the instance names. For a test fixture, they are the same.
- 3. With \$scope, move down the hierarchy into under_test.
- 4. With \$showscopes, look at the modules and instances of those modules. In this case, there are two instances of the full_adder module.
- 5. With <code>\$scope</code>, move down the hierarchy into instance <code>pl of the module full_adder</code>.
- 6. With \$showscopes, notice that the full_adder module consists of two instances of module half_adder.
- 7. With \$scope, move down the hierarchy into instance m2 of module half_adder.
- 8. With \$showscopes, notice that there are no modules instantiated in *m*2. This is the bottom of the hierarchy.

- 9. With \$showvars, display the variables at this level of scope.
- **10.** Move directly to a particular scope by using a hierarchical name. Look at a wire using a full hierarchical name.

```
In the example, the scope is set to tester.under_test, and the wire tester.under_test.p0.si is examined.
```

Example: Traversing model hierarchy with SimVision

The following example shows how to traverse a model hierarchy with SimVision.

1. Invoke the Verilog-XL with the <u>SimVision</u> graphical environment using the following command:

```
verilog <u>half adder.v</u> <u>full adder.v</u> <u>two bit adder.v</u> <u>tester.v</u> -s +gui
```

2. Type \$showscopes in the simulator window to display the module and instance names. The Simulator Window displays the following:

```
C1 > $showscopes;
Directory of scopes at current scope level:
module (two_bit_adder), instance (under_test)
Current scope is (tester)
Highest level modules:
tester
```

3. Open the source browser by clicking Windows-New-Source Browser

4. Click on the down arrow on the dropdown list of the *Scopes* field to display the subscope under_test. Clicking on the arrow closes the *Subscopes* list.

- SimVision: S	ource Browser 1 [/DOC_ 🕝 🔲
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>3</u>	<u>elect Format Si</u> mulation <u>W</u> indows
	<u>H</u> elp
💣 🌘 😼 🛔	🔁 🐝 🗸 Send To: 🚳 🚟 🗏 🖉 🚱 🖬 📓
x2 TimeA ▼ = 0	🔹 🔊 🖬 🕞 💏 ד 🔤 🥨 🖉 Search
	📫 🧯 🗍 Simulation Time: 0
🏹 - 🚺 - Scope:	tester 🚽 🔝 //hm/1 🛒 🔍
↑ 1 module tes 2 3 4 5 5 7	tester under_test ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;

5. Select under_test. The Simulator Window displays the following:

C2 > \$scope(tester.under_test);

6. Type \$showscopes again to display the module and instance names of under_test. The Simulator Window displays the following:

```
C3> $showscopes;
Directory of scopes at current scope level:
module (full_adder), instance (p0)
module (full_adder), instance (p1)
Current scope is (tester.under_test)
Highest level modules:
tester
```

- 7. Click on the down arrow |V|, on the dropdown list of the *Scopes* field and click on p1.
- 8. Type \$showscopes again to display the module and instance names of p1. The Simulator Window displays the following:

```
C5> $showscopes;
Directory of scopes at current scope level:
module (half_adder), instance (m1)
module (half_adder), instance (m2)
Current scope is (tester.under_test.p1)
Highest level modules:
tester
```

- **10.** Type \$showscopes again to display the module and instance names of m2. The Simulator Window displays the following:

```
C7> $showscopes;
Directory of scopes at current scope level:
Current scope is (tester.under_test.pl.m2)
Highest level modules:
tester
```

- 11. Choose Select–Signals to select all the signals in the m2 module.
- **12.** Select Describe option by clicking the right mouse button.

The Simulator Window displays the following:

```
C8 > $showvars(tester.under_test.pl.m2.a,tester.under_test.pl.m2.sum,
tester.under_test.pl.m2.b,tester.under_test.pl.m2.c_out);
a (tester.under_test.pl.m2) wire = StX
StX <- (tester.under_test.pl.m1): xor nl(sum, a, b);
sum (tester.under_test.pl.m2) wire = StX
StX <- (tester.under_test.pl.m2): xor nl(sum, a, b);
b (tester.under_test.pl.m2) wire = StX
StX <- (tester.under_test.pl.m2): or m3(c_out, ci2, ci1);
c_out (tester.under_test.pl.m2) wire = StX
StX <- (tester.under_test.pl.m2) wire = StX</pre>
```

- **13.** Display the hierarchy of modules by clicking on the down arrow **v** on the dropdown list of the *Scopes* field and click on under_test.
- 15. Click on the si signal in the Source Browser to select the signal.
- **16.** Upon selecting 'Describe' option by clicking right mouse button, the Simulator Window displays the following:

```
C11> $showvars(tester.under_test.p0.si);
si (tester.under_test.p0) wire = StX
    StX <- (tester.under_test.p0.ml): xor nl(sum, a, b);</pre>
```

Observing All Simulation Events

When you first start debugging, you can view all simulation events as they occur. However, you may want to limit the focus of these simulation events by narrowing in on an area of your design to debug. For more information, see <u>"Observing a Focused Set of Simulation Events"</u> on page 58.

The following example shows how to single-step through your source code, observing each line of code as it executes. The example also shows how to send information about code execution to the screen during simulation. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Observing all simulation events

```
% verilog <u>half_adder.v</u> full adder.v two_bit_adder.v tester.v -s
         // See Step 1
                                   // See Step 2
C1 >
L10 "tester": always'
C1 >
L10 "tester": @(sum or c out)
C1 >
L14 "tester": initial
                                                // See Step 3
C1 > $db_settrace;
                                    // See Step 4
C2 >
L15 "tester": begin
L16 "tester": $display("bus_a + bus_b (+ c_in) = result (c_out and sum)");
bus_a + bus_b (+ c_in) = result (c_out and sum)
L17 "tester": c in = 1'b0;
L18 "tester": bus a = 2'b0;
L19 "tester": bus_b = 2'b1;
L20 "tester": #2000
L3 "2bit_adder": wire c_in >>> NET = St0
L6 "2bit_adder": wire a[0] >>> NET = St0
L6 "2bit_adder": wire a[1] >>> NET = St0
L6 "2bit_adder": wire b[0] >>> NET = St1
L6 "2bit_adder": wire b[1] >>> NET = St0
SIMULATION TIME IS 10
L6 "half adder.v" (tester.under test.p0.m2): and n2 >>> XL GATE = St0
L6 "half_adder.v" (tester.under_test.p0.m1): and n2 >>> XL GATE = St0
L5 "half_adder.v" (tester.under_test.p0.ml): xor nl >>> XL GATE = St1
L5 "half_adder.v" (tester.under_test.pl.ml): xor n1 >>> XL GATE = St0
L6 "half_adder.v" (tester.under_test.pl.ml): and n2 >>> XL GATE = St0
L7 "full_adder.v" (tester.under_test.p0): or m3 >>> XL GATE = St0
SIMULATION TIME IS 20
L5 "half_adder.v" (tester.under_test.pl.m2): xor n1 >>> XL GATE = St0
L6 "half_adder.v" (tester.under_test.p1.m2): and n2 >>> XL GATE = St0
L5 "half_adder.v" (tester.under_test.p0.m2): xor n1 >>> XL GATE = St1
L7 "full_adder.v" (tester.under_test.pl): or m3 >>> XL GATE = St0
L5 "tester.v": wire sum[1] >>> FROMXL NET = St0
L5 "tester.v": wire sum[0] >>> FROMXL NET = St1
L6 "tester.v": wire c_out >>> FROMXL NET = St0
L10 "tester.v": @(sum or c_out) >>> CONTINUE
L11 "tester.v": #100
SIMULATION TIME IS 120
    . . .
```

- 1. Invoke Verilog-XL with -s to go into interactive mode.
- 2. Step and trace (,) the simulation.
- 3. Trace all simulation activity with \$db_settrace.
- **4.** Continue (.) the simulation. Verilog performs the entire simulation and displays all simulation events.

Observing a Focused Set of Simulation Events

It is possible to display all simulation events as they occur. This method is helpful when you start to debug. However, as soon as you know what area of your design you are interested in, you will want to restrict or focus the amount of data being displayed.

The following example shows how to view data about a selected instance of a module. The example narrows the focus, sets a trace, and steps through the source code. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Observing a focused set of simulation events

```
% verilog <u>half_adder.v</u> <u>full_adder.v</u> <u>two_bit_adder.v</u> <u>tester.v</u> -s
       // See Step 1
                                                          // See Step 2
C1 > $db_setfocus (tester.under_test.p0);
Set focus (1) on scope tester.under_test.p0.
                                 // See Step 3
C2 > $db_settrace;
                                  // See Step 4
C3 > $db_step;
                   Time bus_a + bus_b (+ c_in) = result (c_out and sum)
SIMULATION TIME IS 10
L7 "full_adder.v" (tester.under_test.p0): or m3 >>> XL GATE = St0
L7 "full_adder.v" (tester.under_test.p0): or m3 >>> FROMXL GATE = St0
Stepped to line 7, scope tester.under_test.p0, file full_adder.v, time 10.
                                        // See Step 5
C4 > 3
                                       (+ 0)
                        00
                            + 01
                 120
                                                      001
SIMULATION TIME IS 2010
L7 "full_adder" (tester.under_test.p0): or m3 >>> XL GATE = St1
L7 "full_adder" (tester.under_test.p0): or m3 >>> FROMXL GATE = St1
Stepped to line 7, scope tester.under_test.p0, file full_adder, time 2010.
C4 > 3
                 2120
                          01
                             +
                                  01
                                        (+ 0)
                                                  =
                                                       010
                             + 01
                 3110
                                        (+ 1)
                         01
                                                       011
                                                  =
SIMULATION TIME IS 5010
L7 "full_adder" (tester.under_test.p0): or m3 >>> XL GATE = St0
L7 "full adder" (tester.under test.p0): or m3 >>> FROMXL GATE = St0
Stepped to line 7, scope tester.under_test.p0, file full_adder, time 5010.
                                        // See Step 6
C4 > .
SIMULATION TIME IS 5020
L7 "full_adder" (tester.under_test.p0): or m3 >>> XL GATE = St1
L7 "full_adder" (tester.under_test.p0): or m3 >>> FROMXL GATE = St1
                 5120
                         10
                             + 01
                                       (+ 1)
                                                       100
                                                  =
SIMULATION TIME IS 6010
L7 "full_adder" (tester.under_test.p0): or m3 >>> XL GATE = St0
```

L7 "full_adder" (tester.under_test.p0): or m3 >>> FROMXL GATE = St0 6110 10 + 01 (+ 0) = 011 0 simulation events (use +profile or +listcounts option to count) + 49 accelerated events

CPU time: 1.4 secs to compile + 0.1 secs to link + 0.1 secs in simulation

- **1.** Invoke Verilog-XL with -s to go into interactive mode.
- 2. Restrict focus to instance p0 of module full_adder.
- **3.** Turn on tracing. The trace acts only on the focus set in the previous step.
- **4.** Step only through the area of focus. Ignore activity elsewhere.
- 5. Step again using command-line recall (command number 3 from step 4).
- 6. Continue (.) the simulation.

Observing Wires and Registers Periodically

Monitoring signals to observe them as they change allows you to see sequences and transitory states. However, if you want to look at certain registers or outputs with the inputs driving them, you need to look during a certain window in time—after the registers or outputs have stabilized and before the inputs change.

The following example shows how to look at signals periodically starting with an offset. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Observing wires and register periodically

```
% verilog half adder.v full adder.v two bit adder.v test.v -s -i
                        // See Step 1
 strobe.key
                                              // See Step 2
C1 > $showvars;
Variables in the current scope:
bus_a (tester) reg = 2'hx, x
bus b (tester) req = 2'hx, x
c_in (tester) reg = 1'hx, x
sum[1] (tester) wire = StX
        StX <- (tester.under_test.pl.m2): xor nl(sum, a, b);</pre>
sum[0] (tester) wire = StX
        StX <- (tester.under_test.p0.m2): xor n1(sum, a, b);</pre>
c_out (tester) wire = StX
        StX <- (tester.under_test.p1): or m3(c_out, ci2, ci1);</pre>
C2 > begin
                                        // See Step 3
> #100 $stop;
> $strobe ($time,, "a=%b, b=%b, c_in=%b, answer=%b%b%b",
          bus_a, bus_b, c_in, c_out, sum[1], sum[0]);
>
> while ($time < 6000)
                                                     // See Step 4
          #1000 $strobe ($time,, "a=%b, b=%b, c_in=%b, answer=%b%b%b",
>
                   bus_a, bus_b, c_in, c_out, sum[1], sum[0]);
>
> end;
```

C3 > . // See Step 5 C2: \$stop at simulation time 100 C3 > . 100 a=00, b=00, c_in=0, answer=000 1100 a=01, b=00, c_in=0, answer=001 2100 a=01, b=01, c_in=0, answer=010 3100 a=11, b=11, c_in=1, answer=110 4100 a=10, b=11, c_in=1, answer=101 6100 a=10, b=10, c_in=1, answer=101

- 1. Use the -i option to read in a key file containing debugging commands. Use -s; without it, Verilog-XL ignores the key file until you issue an interactive command. To invoke Verilog-XL with SimVision, add the +gui plus option to the command line.
- 2. Observe the signals in the current scope using \$showvars.

Note: To observe signals with SimVision you need to select the signals and then choose 'Describe' option by right clicking the mouse button.

- 3. The longest path through the adder is less than 100. So, advance the clock 100 before strobing so as to catch stable outputs with the inputs that produced them. Enclose the delay of 100 and the *strobe* commands in begin end blocks to force sequential (rather than concurrent) execution.
- 4. Strobe every 1000 as long as simulation time is less than 6000.
- 5. Continue (.) the simulation.

Observing Wires and Registers When They Change Value

If you want information whenever certain signals change value, use monitoring. Monitoring catches intermediate and transitory values. If you prefer to let values propagate until the device is stable, see<u>"Observing Wires and Registers Periodically</u>" on page 59.

The following example shows how to use monitoring to observe the values of signals whenever they change. (<u>"Example: Observing wires and registers with SimVision when they change value</u>" on page 61 shows the same example with SimVision.) "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Observing wires and registers when they change value

```
c_in (tester) reg = 1'hx, x
sum[1] (tester) wire = StX
          StX <- (tester.under_test.pl.m2): xor nl(sum, a, b);</pre>
sum[0] (tester) wire = StX
          StX <- (tester.under_test.p0.m2): xor n1(sum, a, b);</pre>
c_out (tester) wire = StX
          StX <- (tester.under test.pl): or m3(c out, ci2, ci1);</pre>
C2 > $monitor ($time,, "a=%b, b=%b, c_in=%b, answer=%b%b%b"
          bus_a, bus_b, c_in, c_out, sum[1], sum[0]);// See Step 3
                                            // See Step 4
C_{3} > .
                     0 a=00, b=01, c_in=0, answer=xxx
20 a=00, b=01, c_in=0, answer=001
2000 a=01, b=01, c_in=0, answer=001
                     2020 a=01, b=01, c_in=0, answer=010
                     3000 a=01, b=01, c_in=1, answer=010
                     3010 a=01, b=01, c_in=1, answer=011
                     5000 a=10, b=01, c_in=1, answer=011
                     5020 a=10, b=01, c_in=1, answer=010
                     5030 a=10, b=01, c_in=1, answer=100
6000 a=10, b=01, c_in=0, answer=100
6010 a=10, b=01, c_in=0, answer=101
6020 a=10, b=01, c_in=0, answer=011
```

• • •

- 1. Use the -i option to read in a key file containing debugging commands. Use -s; without it, Verilog-XL ignores the key file until you issue an interactive command.
- 2. Look at the wires and buses.
- 3. Display the values of all these signals when any of them change using \$monitor.
- 4. Continue (.) the simulation.

Example: Observing wires and registers with SimVision when they change value

The following example shows how to observe wires and registers with SimVision when they change value.

1. Invoke the Verilog-XL with the SimVision graphical environment using the following command:

```
% verilog <u>half adder.v</u> <u>full adder.v</u> <u>two bit adder.v</u> <u>test.v</u> -s -i
monitor.key</u> +gui
```

2. Choose Select–Signals to highlight all signals, then select 'Describe' option by right clicking the mouse button to display the variables in the Simulator Window as shown:

```
C4 > $showvars(test.sum,test.c_out,test.bus_a,test.bus_b,test.c_in);
sum[1] (test) wire = StX
   StX <- (test.under_test.pl.m2): xor nl(sum, a, b);
sum[0] (test) wire = StX
   StX <- (test.under_test.p0.m2): xor nl(sum, a, b);
c_out (test) wire = StX
   StX <- (test.under_test.pl): or m3(c_out, ci2, ci1);
bus_a (test) reg = 2'hx, x
bus_b (test) reg = 2'hx, x
```

Verilog-XL User Guide

Debugging Your Design

3. Click the *Run Simulation* button **b** to continue the simulation.

The following is displayed in the Simulator Window:

C3	>						
			0	a=00,	b=01,	c_in=0,	answer=xxx
			20	a=00,	b=01,	c_in=0,	answer=001
			2000	a=01,	b=01,	c_in=0,	answer=001
			2020	a=01,	b=01,	c_in=0,	answer=010
			3000	a=01,	b=01,	c_in=1,	answer=010
			3010	a=01,	b=01,	c_in=1,	answer=011
			5000	a=10,	b=01,	c_in=1,	answer=011
			5020	a=10,	b=01,	c_in=1,	answer=010
			5030	a=10,	b=01,	c_in=1,	answer=100
			6000	a=10,	b=01,	c_in=0,	answer=100
			6010	a=10,	b=01,	c_in=0,	answer=101
			6020	a=10,	b=01,	c_in=0,	answer=011

Examining Wires and Registers Now

When you stop your simulation, you can examine the values of wires or registers and how they are interconnected.

The following example shows how to examine the value of a wire or signal by using a full path name or by moving to a level in the model hierarchy containing the wire or signal. (<u>"Example:</u> <u>Examining wires and registers with SimVision now</u>" on page 63 shows how the example works with SimVision.)

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Examining wires and registers now

```
% verilog <u>half adder.v</u> <u>full adder.v</u> <u>two bit adder.v</u> <u>tester.v</u> -s
// See Step 1
                                                // See Step 2
C1 > $showvars;
Variables in the current scope:
bus_a (tester) reg = 2'hx, x
bus_b (tester) reg = 2'hx, x
c_in (tester) reg = 1'hx, x
sum[1] (tester) wire = StX
   StX <- (tester.under_test.pl.m2): xor nl(sum, a, b);</pre>
sum[0] (tester) wire = StX
   StX <- (tester.under test.p0.m2): xor n1(sum, a, b);</pre>
c out (tester) wire = StX
   StX <- (tester.under_test.pl): or m3(c_out, ci2, ci1);</pre>
                                               // See Step 3
C2 > $showvars (bus_a);
bus_a (tester) reg = 2'hx, x
C3 > $showvars (under_test.p0.ml.sum);
                                                     // See Step 4
sum (tester.under_test.p0.ml) wire = StX
   StX <- (tester.under_test.p0.ml): xor nl(sum, a, b);</pre>
```

```
C4 > $scope (under_test.p0.ml); // See Step 5
C5 > $showvars; // See Step 6
Variables in the current scope:
a (tester.under_test.p0.ml) wire = StX
StX <- (tester.under_test): port 1
b (tester.under_test.p0.ml) wire = StX
StX <- (tester.under_test): port 2
sum (tester.under_test.p0.ml) wire = StX
StX <- (tester.under_test.p0.ml): xor nl(sum, a, b);
c_out (tester.under_test.p0.ml) wire = StX
StX <- (tester.under_test.p0.ml): and n2(c_out, a, b);</pre>
```

- 1. Bring up Verilog-XL in interactive mode with -s. To invoke Verilog-XL with SimVision, add the +gui plus option to the command line.
- **2.** Look at all the wires and registers in the current scope.

To observe signals with SimVision you need to select the signals and then choose 'Describe' option by right clicking the mouse button.

- **3.** Look at a specific register (bus_a).
- 4. Look at the wire called sum in instance m1. This command uses the full path name of the sum wire.
- 5. Move to instance m1 in instance p0 in instance under_test.
- 6. Look at all wires and registers in the current scope.

Example: Examining wires and registers with SimVision now

The following example shows how to observe wires and registers with SimVision now.

1. Invoke the Verilog-XL with the SimVision graphical environment using the following command:

verilog half adder.v full adder.v two bit adder.v tester.v -s +gui

- 2. To display the values of the signals, choose *Select–Signals*. This highlights all the signals in the Source Browser.
- **3.** Choose 'Describe' option by right clicking the mouse button.

The following output appears in the Simulator Window:

```
bus_b (tester) reg = 2'hx, x c_in (tester) reg = 1'hx. x
```

4. Click on the down arrow on the dropdown list of the *Scopes* field to display the subscope under_test. The arrow changes to an up arrow as shown. Clicking on under_test changes the scope.

SimVision: Source Browser 1 [/	′DOC
<u>F</u> ile <u>E</u> dit <u>V</u> iew <u>S</u> elect <u>F</u> ormat S <u>i</u> mulation	<u>W</u> indows
	<u>H</u> elp
💣 🌔 😼 💏 🐝 - Send To: 🙋 🚟 [= 2 % <mark>7</mark> (i
x2 TimeA ▼ = 0 ▼ ns ▼ , 🔂 ▼ . 🤇	🛍 🦚 🌔 Search
D 📖 🔣 í 📆 📅 🧇 í 🍏 í Simulation Time	:: O
🏹 - 🜔 - Scope: tester	🚮 /hm/1🛨 🔍
➡ 1 module te tester	<u>Z</u>
2 under_test	;
4	
5	
1	objects selected
	objects selected

- 5. Repeat step 4 to change scopes to p0, then to m1.
- 6. Click on the down arrow not be dropdown list of the *Scopes* field to display the scopes to which you can change. To display the values of signals in any of these scopes, click on the scope, then repeat steps 2 and 3.

Patching a Model (Asking "What If" Questions)

You can ask "What if" questions about your model by interactively forcing nodes to desired values and seeing if that fixes the problem. If it does, you can then make a change in the model itself.

The following example shows the model 3buf, which contains three buffers in sequence.(<u>"Example: Patching a model with SimVision</u>" on page 66 shows how the example works with SimVision.) The examples force the output of the first buffer to be the not of the input, thereby converting the buffer into an inverter. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Patching a model

```
% verilog 3buf buf_tester -s
                                         // See Step 1
                                // See Step 2
C1 > $showvars;
Variables in the current scope:
test_in (buf_tester) reg = 1'hx, x
test_out (buf_tester) wire = StX
        StX <- (buf_tester.simp_test): buf n3(out, out2);</pre>
                                    // See Step 3
C2 > $scope (simp_test);
C3 > $showvars;
Variables in the current scope:
out1 (buf_tester.simp_test) wire = StX
        StX <- (buf_tester.simp_test): buf n1(out1, in);</pre>
out2 (buf_tester.simp_test) wire = StX
        StX <- (buf_tester.simp_test): buf n2(out2, out1);</pre>
out (buf_tester.simp_test) wire = StX
    StX <- (buf_tester.simp_test): buf n3(out, out2);</pre>
C4 > $monitor ($time, " in $b, out1 $b, out2 $b, out $b", in
out1, out2, out); // See Ste
                                                     // See Step 4
C5 > force out1 = ! in;
                                              // See Step 5
C6 > #150 $stop;
                                               // See Step 6
C7 > .
                 0 in 0, out1 1, out2 x, out x
10 in 0, out1 1, out2 1, out x
20 in 0, out1 1, out2 1, out 1
                                                         // See Step 7
                 100 in 1, out1 0, out2 1, out 1
                 110 in 1, out1 0, out2 0, out 1
                 120 in 1, out1 0, out2 0, out 0
C6: $stop at simulation time 150
C7 > $list_forces;
                                                 // See Step 8
force buf_tester.simp_test.out1 = !buf_tester.simp_test.in;
C8 > release out1;
                                                 // See Step 9
C9 > .
                 150 in 1, out1 1, out2 0, out 0
                 160 in 1, out1 1, out2 1, out 0
                 170 in 1, out1 1, out2 1, out 1
                 200 in 0, out1 1, out2 1, out 1
                 210 in 0, out1 0, out2 1, out 1
                 220 in 0, out1 0, out2 0, out 1
                 230 in 0, out1 0, out2 0, out 0
```

- **1.** Bring up Verilog-XL interactive mode with -s.
- 2. Look at the signals with \$showvars.
- 3. Move into instance simp_test with \$scope.
- 4. Have Verilog-XL display all changes to the buffer chain with \$monitor.

- 5. Force signal out1 to be the inverse of signal in.
- 6. Stop simulation after 150 time units.
- 7. Continue (.) the simulation.
- 8. Show currently active force statements.
- 9. Free the forced signal out1 with the release statement.

Example: Patching a model with SimVision

The following example shows how to patch a model with SimVision.

1. Invoke the Verilog-XL with the SimVision graphical environment using the following command:

% verilog <u>half adder.v</u> <u>full adder.v</u> <u>two bit adder.v</u> <u>tester.v</u> -s +gui

2. Select Simulation–CreateForce...and fill in the CreateForce dialog box as shown:

-	SimVision: Create Force				
🎭 Add 🔯 Browser					
0	oject: (tester	c_out			
Value: Ic_in					
,					
	OK	Cancel	Apply	Hel	p

When you click on OK, you force the c_out signal to be in the inverse of the c_in signal. The following output is displayed in the Simulator Window:

C1 > force tester.c_out = ! c_in;

3. To show the forces that are set, select *Simulation–Show–Forces...* and the following dialog box appears:

	— Sii	mVision: Prope	rties	•	
	<u>F</u> ile <u>W</u> indows				
	F P	simulator:Forces		F)	
		×		0	
			Forced Value		
		tester.c_out	!c_in		
ĺ	[

You can set other forces by clicking on the *Set...* button (see step 2) or delete existing forces by highlighting a force and clicking on the *Delete* button.

Ordering Events in a Time Cycle

In concurrent simulation, many events occur in the same time cycle. Nevertheless, Verilog-XL executes those events sequentially, code line by code line, in that time cycle. To display the sequential execution order, use tracing. If you have an always block that must be executed last, use a delay of 0 to schedule it near the end of the time slot.

Verilog-XL considers any transition to logic 1 to be a positive edge. However, you might want to trigger an action only when clk changes from 0 to 1. Thus, on every positive edge, you compare the present clk value with the old clk value to check that it's a true rising edge (from 0 to 1). Then, you save the present value in oldvalue for the next comparison.

The following example shows how to order events in a time cycle. (<u>"Example: Ordering</u> events in a time cycle with SimVision" on page 69 shows how the example works with SimVision.) "See Step n" comments in the example correspond to descriptive steps that follow the example.

Verilog-XL User Guide

Example: Ordering events in a time cycle

```
module guarantee_order;
reg clk, oldvalue;
always @ clk
                                                      // See Step 1
    #0 oldvalue = clk;
                                                      // See Step 2
always @ (posedge clk)
  if ((oldvalue == 0) & (clk == 1))
    $display ("true rising edge at %d", $time);
initial
                                                      // See Step 3
    begin
        #1 clk = 'bz;
        #1 clk = 1;
        #1 clk = 'bx;
        #1 clk = 1;
        #1 \ clk = 0;
        #1 clk = 1;
    end
endmodule // guarantee_order
                                                      // See Step 4
% verilog guarantee_order -s
                                                      // See Step 5
C1 > #6 $stop;
C2 > .
                                                      // See Step 6
C1: $stop at simulation time 6
                                                      // See Step 7
C2 > $db_settrace;
C3 > .
SIMULATION TIME IS 6
L19 "guarantee_order": #1 >>> CONTINUE
L19 "guarantee_order": clk = 1;
L20 "guarantee_order": end
L5 "guarantee_order": @clk >>> CONTINUE
L6 "guarantee order": #0
                                                       // See Step 8
L8 "guarantee_order": @(posedge clk) >>> CONTINUE
L9 "guarantee_order": if((oldvalue == 0) & (clk == 1)) >>> TRUE
L10 "guarantee_order": $display("true rising edge at %d", $time);
true rising edge at L10 "guarantee_order": $time
L8 "guarantee_order": always
L8 "guarantee_order": @(posedge clk)
    L6 "guarantee_order": #0 >>> CONTINUE
L6 "guarantee_order": oldvalue = clk; >>> oldvalue = 1'h1, 1;
L5 "guarantee order": always
L5 "guarantee_order": @clk
```

- 1. The zero delay (#0) causes this evaluation and assignment to occur at the end of the time slot.
- 2. This always block executes before the zero delay always block.
- 3. For our test case, create three positive edges: Z to 1, X to 1, and, at time 6, 0 to 1.
- 4. Bring up Verilog-XL interactive mode with -s.
- 5. Stop after six time units to verify that the execution order is correct.
- 6. Continue (.) the simulation.
- 7. With \$settrace, display lines of code as they are executed.

8. See that the zero delay assignment is performed near the end of the time cycle.

Example: Ordering events in a time cycle with SimVision

The following example shows how to order events in a time cycle with SimVision.

1. Invoke the Verilog-XL with the SimVision graphical environment using the following command:

```
verilog guarantee order.v -s +gui
```

2. Set a breakpoint at time 6 by selecting *Simulation–Set Breakpoint–Time...* and typing 6 in the *Time* field of the Set Break dialog box. Click on *OK* to set the breakpoint.

SimVision: Set Breakpoint		
Break based on Time Line Signal		1
Stop at relative time: 🕶 🕞		
When: just BEFORE time 🔫		
Current Time 6		
OK Cancel Apply	Help	

The Simulator Window shows the following:

Cl > \$db_breakbeforetime(\$time + 6); Set break (1) before time 6.

3. Click the *Run Simulation* button **b** to continue the simulation.

The Simulator Window shows the following:

```
C2 > .
Break (1) occured before time 6.
```

Verilog-XL User Guide

Debugging Your Design

4. Type the following command to trace the simulation:

C2 > \$db_settrace;

5. Click the *Run Simulation* button to continue the simulation.

The Simulator Window shows the following:

```
C3 >
SIMULATION TIME IS 6
L19 "guarantee order.v": #1 >>> CONTINUE
L19 "guarantee_order.v": clk = 1;
L20 "guarantee_order.v": end
L5 "guarantee_order.v": @clk >>> CONTINUE
L6 "guarantee_order.v": #0
L8 "guarantee_order.v": @(posedge clk) >>> CONTINUE
L9 "quarantee order.v": if((oldvalue == 0) & (clk == 1)) >>> TRUE
L10 "guarantee_order.v": $display("true rising edge at %d", $time);
true rising edge at L10 "guarantee_order.v": $time
L8 "guarantee_order.v": always
L8 "quarantee order.v": @(posedge clk)
L6 "guarantee_order.v": #0 >>> CONTINUE
L6 "guarantee_order.v": oldvalue = clk; >>> oldvalue = 1'h1, 1;
L5 "guarantee_order.v": always
L5 "guarantee order.v": @clk
0 simulation events (use +profile or +listcounts option to count)
CPU time: 3.5 secs to compile + 0.1 secs to link + 1.9 secs in simulation
```

Displaying, Strobing, and Monitoring Data

To display data, there are three main techniques: displaying, strobing, and monitoring. The display of data (the \$display and \$write system tasks) executes immediately. Strobing is like displaying except it occurs at the end of the time cycle. Monitoring occurs when any of its variables change value. Verilog-XL also supports equivalent commands that write to files, such as \$fdisplay.

The following example demonstrates comparable display, strobe, and monitor commands. (<u>"Example: Displaying, strobing, and monitoring data with SimVision</u>" on page 71 shows how the example works with SimVision.) "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Displaying, strobing, and monitoring data

```
% verilog half adder.v full adder.v two bit adder.v test.v -s
Cl > $strobe ("at time %0d strobe bus_a is %b", $time, bus_a); // See Step 1
C2 > $monitor ("at time %0d monitor bus_a is %b", $time, bus_a); // See Step 2
C3 > $display ("at time %0d display bus_a is %b", $time, bus_a); // See Step 3
at time 0 display bus_a is xx
C4 > $write ("at time %0d write bus_a is %b", $time, bus_a); // See Step 4
at time 0 write bus_a is xxC5 > #4001 $stop;
```

```
C6 > .
                                                                   // See Step 5
at time 0 monitor bus a is 00
at time 0 strobe bus_a is 00
at time 2000 monitor bus_a is 01
C5: $stop at simulation time 4001
C6 > -2i
                                                                   // See Step 6
C6 > $display ("bus_a is %o, bus_a is %h", bus_a, bus_a);
                                                                 // See Step 7
bus_a is 1, bus_a is 1
C7 >
                                                                 // See Step 8
bus a is 01, bus a is 1
C8 > $display ("sum[0] has a strength of %v", sum[0]);
                                                               // See Step 9
sum[0] has a strength of St1
C9 > $display ("current scope is %m");
                                                                // See Step 10
current scope is test
```

- 1. \$strobe displays at the end of the time cycle. The 0 (zero) in the %0d syntax causes the time to be displayed in the minimum space.
- 2. \$monitor displays the signals when they change.
- 3. \$display displays the signals now.
- 4. \$write displays the signals now. The \$write command does not insert a carriage return; instead, it runs text into whatever follows (in this case, the interactive prompt and the #4001 \$stop command).
- 5. Continue (.) the simulation.
- 6. Deactivate the \$monitor (command number two) task.
- 7. Display the signals in octal (o) and hex (h).
- 8. Display the signals in binary (b) and decimal (d).
- 9. Display the signal strength with &v.
- **10.** Display the current scope with %m.

Example: Displaying, strobing, and monitoring data with SimVision

The following example shows how to order events in a time cycle with SimVision.

1. Invoke the Verilog-XL with the SimVision graphical environment using the following command:

verilog <u>half adder.v</u> <u>full adder.v</u> <u>two bit adder.v</u> <u>test.v</u> -s +gui

2. The values of the signals can be seen in the Design Browser by selecting the corresponding scope.

3. To monitor a signal, select the signals you want to monitor (for example, c_out, c_in, and bus_a) in the Design Browser, then select *Simulation–CreateMonitor...* and click on OK in the Create Monitor dialog box that appears.

C2 > \$monitor(test.c_out,test.c_in,test.bus_a);

Controlling the Display and Interpretation of Time

You can control how Verilog-XL displays and interprets times specified interactively. This does not affect the timing of models. It only affects how Verilog-XL displays and interprets the times you type interactively.

The following example changes the time units to nanoseconds, and then to picoseconds.

Example: Controlling the display and interpretation of time

<pre>% verilog buftick buf_tester -s C1 > #1 \$stop; C2 > .</pre>	 	See See See	Step Step Step	1 2 3
<pre>C1: \$stop at simulation time 1000 C2 > \$timeformat (-9, 2, "ns"); C3 > #1 \$stop;</pre>	//	See	Step	4
C4 > . C3: \$stop at simulation time 2.00ns C4 > \$timeformat (-9, 3, "ns"); C5 > #1 \$stop; C6 >	//	See	Step	5
C5: \$stop at simulation time 3.000ns C6 > \$timeformat (-12, 2, "ps"); C7 > #1 \$stop; C8 >		See	Step	6
C7: \$stop at simulation time 3001.00ps C8 > \$finish;	//	See	Step	7

- 1. Enter interactive mode with -s. The model contains the compiler directive `timescale 1 ns/1 ps; This statement sets the simulation time unit to 1 nanosecond and the time precision to 1 picosecond.
- 2. Stop after 1 time unit.
- **3.** Continue (.) the simulation.
- 4. Change the time units to nanoseconds with a precision of 2.
- 5. Change the time units to nanoseconds with a precision of 3.
- 6. Change the time units to picoseconds with a precision of 2.
- 7. Stop the simulation.
Reinitializing the Network and Simulator Clock

When you identify a bug and a possible solution to that bug, you often want to quickly patch the model and resimulate before taking the time to alter your source files, recompile, and resimulate. You can reinitialize the network and the simulator clock, and then quickly patch a model.

The following example shows how to reinitialize the network. It suggests that you can follow up by resetting your log files before patching and running. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Reinitializing the network and simulator clock

```
C1 >
                                            // See Step 1
100 a=00, b=00, c_in=0, answer=000
1100 a=01, b=00, c_in=0, answer=111
C2: $stop at simulation time 2100
C3 > $reset;
                                             // See Step 2
C3: $reset at simulation time 2100
C4 > $showvars;
                                             // See Step 3
Variables in the current scope:
bus a (tester) req = 2'hx, x
bus_b (tester) reg = 2'hx, x
c_in (tester) reg = 1'hx, x
sum[1] (tester) wire = StX
    StX <- (tester.under_test.pl.m2): xor nl(sum, a, b);</pre>
sum[0] (tester) wire = StX
    StX <- (tester.under_test.p0.m2): xor n1(sum, a, b);</pre>
c_out (tester) wire = StX
    StX <- (tester.under_test.pl): or m3(c_out, ci2, ci1);</pre>
C5 > $log ("my second log");
                                             // See Step 4
```

- 1. The original adder model works incorrectly— it adds 1 (a) and 0 (b) and gets 7 for an answer.
- 2. Reinitialize the network and the simulator clock with *\$reset*. To perform this with SimVision, select *Simulation-Reset to Start* from the SimVision window.
- 3. With \$showvars, notice that the network has been reinitialized.
- 4. Open a new log file (\$reset closes the previous one). Execute other commands.

Verilog-XL User Guide Debugging Your Design

Controlling Verilog-XL

This chapter describes the following:

- <u>Overview</u> on page 75
- <u>Saving and Restarting a Simulation</u> on page 76
- <u>Stopping at the Beginning of a Simulation</u> on page 78
- <u>Stopping During a Simulation</u> on page 79
- <u>Continuing a Stopped Simulation</u> on page 80
- Stepping and Tracing Through a Simulation on page 81
- Ending a Simulation on page 83
- Passing Values into a Module from the Command Line on page 84
- <u>Conditionally Compiling Source Code</u> on page 85
- <u>Modifying Simulation Behavior at Run Time</u> on page 87
- Inserting a File into Another File on page 88
- <u>Generating Log Files</u> on page 89
- <u>Reproducing Interactive Sessions Using Key Files</u> on page 91
- Providing Interactive Commands from a File on page 92
- <u>Storing Commonly Used Command Line Arguments</u> on page 94
- <u>Specifying the Delay Type</u> on page 95
- <u>Selecting a Delay Mode</u> on page 96

Overview

This chapter describes how to control the Verilog-XL[®] simulator.

November 2008

Saving and Restarting a Simulation

You can create simulation checkpoints for large simulations or perform quick "try and see" tests by saving your simulation (a snapshot of all simulation data structures) and restarting the simulation at a later time. With text-based Verilog-XL, you can perform full and incremental saves and restart from either type of save.

The following example shows how you restart a simulation from the command-line using an incremental save file created during a previous run of the simulation.

<u>"Example: Saving and Restarting a Simulation in SimVision</u>" on page 77 uses the same example to show how to save and restart a simulation with SimVision graphical environment.

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Saving and restarting a simulation

```
module harddrive;
initial
begin
                                                 // See Step 1
    #100 $save("save.dat");
    #1000 $incsave("incl.dat");
                                                 // See Step 2
    #1000 $incsave("inc2.dat");
    #1000 $incsave("inc3.dat");
end
endmodule // harddrive
% verilog harddrive.v hardreg.v flop.v
                                          // See Step 3
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
L55 "harddrive.v": $finish at simulation time 3400
460 simulation events + 1082 accelerated events + 936 timing check events
CPU time: 1.7 secs to compile + 0.3 secs to link + 0.3 secs in simulation
% verilog -r inc2.dat
                                                  // See Step 4
Restarting from data file "inc2.dat"
Using full save file "save.dat" for incremental restart
Data was compiled Aug 2, 1993 12:49:00
Highest level modules:
harddrive
L55 "harddrive.v": $finish at simulation time 3400
459 simulation events + 1082 accelerated events + 252 timing check events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.2 secs in simulation
```

1. Perform a full save with \$save soon after the simulation begins. By not saving the simulation at time 0, Verilog-XL will have completed more of the initial simulation setup, which makes subsequent incremental saves smaller in size.

The save file is called save.dat.

2. Perform periodic incremental saves (\$incsave) to catch all simulation changes since the last full save.

Each incremental save has a different file name so that the simulation can be restarted from different places in the simulation.

3. Invoke Verilog-XL to perform the simulation.

Verilog-XL creates the primary save file and the three incremental save files.

4. Restart a simulation by invoking Verilog-XL with the -r command-line option (or use the \$restart system task).

This simulation is restarted at the second incremental save. Note that both the incremental save file (inc2.dat) and the original full save file (save.dat) must be present for Verilog-XL to restart the simulation.

Example: Saving and Restarting a Simulation in SimVision

To save the complete simulation data structure into a permanent file that can be reloaded at a later time, select *Simulation–Save Checkpoint…* from the SimVision window. The Save Simulation dialog box appears. Fill it out and click *OK* to save the simulation.

—	Save Verilog–XL Session					
Dire	ectory:	arif/junk/temp/DOC_Examples/chp4/Bre	98.ingd 主			
N						
	File	<u>n</u> ame:	Save			
	Files o	of <u>t</u> ype: Save Files (*.save)	<u>C</u> ancel			

To restart the simulation using a previously saved data structure file select *Simulation– Restart From Checkpoint...* from the SimVision window. The Restart Simulation dialog box appears. Fill it out and click *OK* to run the simulation.

		Restart Verilog	–XL Session			
<u>D</u> ir	Directory: arif/junk/temp/DOC_Examples/chp4/Bread 主					
şma,	File <u>r</u>	ame:				
	Files of	type: Save Files (*.sa	ave)			

Stopping at the Beginning of a Simulation

You can stop a simulation at its beginning and enter interactive mode by specifying the -s option on the command line for either the text-based environment or the SimVision graphical environment.

The following example shows how to enter interactive mode after compiling your design and before simulating your design. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Stopping at the beginning of a simulation

```
% verilog harddrive.v hardreg.v flop.v -s // See Step 1
...
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
Type ? for help
C1 > $db_breakwhen (q[0], 1); // See Step 2
Set break (1) when harddrive.q[0] = 1.
C2 > . // See Step 3
at time 50 clr = 1 data= 0 q= x
```

at time 150 clr = 1 data= 1 q= 0 Break (1) occured when harddrive.q[0]=1 at time 229.

- Invoke Verilog-XL with the -s command-line option. (Use the +gui plus option to invoke Verilog-XL with SimVision.) Your simulation enters interactive mode after Verilog-XL compiles your design.
- **2.** From the interactive prompt, execute any Verilog-XL commands or Verilog HDL statements you want. A breakpoint is defined.
- 3. Continue your simulation with the continue (.) command.

Because of the defined breakpoint, the simulation stops again, letting you enter more interactive commands. Use the continue command again to continue the simulation or \$finish to end the simulation.

Stopping During a Simulation

You can interrupt a simulation by using the *\$stop* system task in your design in either the text-based environment or the SimVision graphical environment.

The following example shows how to stop the simulation and enter interactive mode.

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Stopping during a simulation

```
module harddrive;
reg clk, clr;
reg [3:0] data;
wire [3:0] q;
event end_first_pass;
always @(end_first_pass)
begin
    clr = ~clr;
                                           // See Step 1
    $stop;
end
endmodule // harddrive
% verilog <u>harddrive.v</u> <u>hardreg.v</u> <u>flop.v</u>
                                          // See Step 2
    . . .
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
L20 "harddrive.v": $stop at simulation time 1700
Type ? for help
```

Controlling Verilog-XL

```
C1 > $showvars(clr); // See Step 3

clr (harddrive) reg = 1'h0, 0

C2 > . // See Step 4

L49 "harddrive.v": $finish at simulation time 3400

449 simulation events + 1082 accelerated events + 936 timing check events

CPU time: 1.5 secs to compile + 0.3 secs to link + 0.2 secs in simulation
```

- 1. Place a <code>\$stop</code> system task in your source code. When the <code>end_first_pass</code> event flag gets set, Verilog-XL toggles the <code>clr</code> signal and stops the simulation.
- 2. Invoke Verilog-XL. (Use the +gui plus option to invoke Verilog-XL with SimVision.)
- **3.** From the interactive prompt, execute any Verilog-XL commands or Verilog HDL statements you want.

\$showvars enables you to examine the value of clr.

4. Continue yc simulation with the continue (.) command or by pressing the *Run Simulation* button in the SimVision window.

Continuing a Stopped Simulation

You can continue a simulation that you have stopped. You can also continue the simulation on a statement-by-statement basis using stepping.

The following example steps through a portion of a simulation and then continues the entire simulation. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Continuing a stopped simulation

```
module step;
initial
begin
    $display ("First Statement");
    $display ("Second Statement");
    $display ("Third Statement");
    $display ("Fourth Statement");
    $display ("Last Statement");
end
endmodule // step
% verilog <u>step.v</u> -s
                          // See Step 1
    Highest level modules:
test
Type ? for help
                       // See Step 2
C1 > ,
L3 "step.v": initial
```

Verilog-XL User Guide

Controlling Verilog-XL

```
Cl > ,
L4 "step.v": begin
C1 > ,
L5 "step.v": $display("First Statement");
First Statement
C1 > i
                      // See Step 3
Second Statement
C1 > ;
Third Statement
                      // See Step 4
C1 > .
Fourth Statement
Last Statement
8 simulation events
CPU time: 0.3 secs to compile + 0.2 secs to link + 0.0 secs in simulation
```

- 1. You can stop a simulation with the -s command-line option, the \$stop system task, or an asynchronous interrupt (Control-c). To invoke Verilog-XL with SimVision, specify the +*gui* plus option on the command line.
- 2. Step through each statement in your simulation with the step-trace (,) command. Unlike step (;), Verilog-XL displays the statement that is currently being executed.

Verilog-XL step-traces the first three statements in module test (initial, begin, and the first \$display).

You can step through a simulation in using the *Single Step* button in the SimVision window.

3. You can step through a simulation with the step (;) command. Unlike the step-trace (,) command, Verilog-XL does not display the statement currently being executed.

Verilog-XL steps through the next two statements. Note that Verilog-XL does not display the \$display statements, although Verilog-XL does display the output from these statements.

4. Continue a simulation with the continue (.) command. Verilog-XL executes the remaining statements in the module (two \$display statements) and ends the simulation. If you are running the SimVision graphical environment, you can continue the remaining statements by clicking on the *Run Simulation* button.

Stepping and Tracing Through a Simulation

You can examine the order in which Verilog-XL executes the statements in your model by step-tracing your simulation. Step through your simulation line by line and display trace information for the current line by using the step-trace (,) interactive command while in interactive mode.

The following example shows how you step and trace your simulation after stopping the simulation with the *\$stop* system task. When you are done examining the simulation with step-trace, you can continue the simulation with the continue (.) interactive command. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Stepping and tracing through a simulation

```
module harddrive;
reg clk, clr;
req [3:0] data;
wire [3:0] q;
'define stim #100 data = 4'b
event end_first_pass;
hardreg h1 (data, clk, clr, q);
initial
begin
    clr = 1; clk = 0;
end
always #50 clk = ~clk;
always @(end_first_pass)clr = ~clr;
initial
begin
    repeat (2)
    begin
    data = 4'b0000;
    `stim 0001;
                        // See Step 1
$stop;
     `stim 0010;
    `stim 0011;
    `stim 0100;
    `stim 0101;
     `stim 0110;
     `stim 0111;
     `stim 1000;
     `stim 1001;
    `stim 1010;
    `stim 1011;
    `stim 1100;
    `stim 1101;
    `stim 1110;
     `stim 1111;
    #200 ->end_first_pass;
end // repeat loop
    $finish;
end
endmodule // harddrive
% verilogg <u>harddrive.v</u> <u>hardreg.v</u> <u>flop.v</u>
                                            // See Step 2
    . . .
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
L22 "harddrive.v": $stop at simulation time 100
Type ? for help
```

Controlling Verilog-XL

```
// See Step 3
Cl > ,
SIMULATION TIME IS 100
L23 "harddrive.v": #100
C1 >
L13 "harddrive.v": #50 >>> CONTINUE
C1 >
L13 "harddrive.v": clk = \sim clk; >>> clk = 1'h0, 0;
L13 "harddrive.v": always
C1 > ,
L13 "harddrive.v": #50
C1 >
L3 "hardreq.v": wire d[0] >>> NET = St1
                      // See Step 4
C1 > .
L22 "harddrive.v": $stop at simulation time 1800
C1 > .
L39 "harddrive.v": $finish at simulation time 3400
448 simulation events + 1082 accelerated events + 936 timing check events
CPU time: 1.4 secs to compile + 0.3 secs to link + 0.2 secs in simulation
```

1. Place a *\$stop* system task in your module where you want to begin step-tracing.

Verilog-XL enters interactive mode after the first stimulus has been applied to the register.

- 2. Invoke Verilog-XL. Verilog-XL simulates until the *\$stop* system task, at which time Verilog-XL enters interactive mode. To invoke Verilog-XL with SimVision, use the +gui plus option on the command line.
- **3.** Step-trace through the simulation with the step-trace (,) interactive command and execute any other commands you want. You can step through a simulation in the SimVision graphical environment using the *Single Step* button.

In this example, the first five statements after the \$stop are step-traced.

4. Continue a simulation with the continue (.) command. In this example, Verilog-XL encounters the \$stop system task a second time, so the continue command is issued a second time to complete the simulation. If you are running the SimVision graphical environment, you can continue the simulation to execute the remaining statements by clicking on the *Run Simulation* button.

Ending a Simulation

You can end your simulation at any time by placing a \$finish system task at the appropriate place in your source code. Verilog-XL ends the simulation and passes control to the post processing environment (ppe) mode of the GUI, if the GUI is connected. While in interactive mode, you can either enter \$finish at the prompt, use the finish command, or use your operating system's end-of-file character (for example, control-d).

Controlling Verilog-XL

The following example shows how the *\$finish* system task ends a simulation.

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Ending a simulation

```
module harddrive;
...
initial // See Step 1
#500 $finish;
...
endmodule // harddrive
% verilog harddrive.v hardreg.v flop.v // See Step 2
...
Highest level modules:
harddrive
L21 "harddrive.v": $finish at simulation time 500
79 simulation events + 195 accelerated events + 182 timing check events
CPU time: 1.6 secs to compile + 0.4 secs to link + 0.0 secs in simulation
```

- 1. Place a *\$finish* command in your model. At 500 time units, the simulation ends regardless of what other simulation activity is scheduled after that time.
- 2. Invoke Verilog-XL. The simulation ends at time 500.

Passing Values into a Module from the Command Line

You can use values in your module that you pass from the command line, called macros, by using the +define+ command-line option. Any macro passed from the command line overrides macros with the same name that you define in the module with the `define compiler directive. Macro definitions exist for all subsequent modules unless you specify the `undef compiler directive.

The following example shows how you override a macro value as defined in a module with the `define compiler directive by specifying a +define+ command-line option when you invoke Verilog-XL. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Passing values from the command line

Verilog-XL User Guide

Controlling Verilog-XL

```
hardreg h1 (data, clk, clr, q);
                                                    // See Step 2
    always #('cycle/2) clk = ~clk;
endmodule // harddrive
       See Step 3
11
% verilog +define+cycle=50 harddrive.v hardreg.v flop.v
Highest level modules:
harddrive
L49 "harddrive.v": $finish at simulation time 1900
478 simulation events + 1170 accelerated events + 1032 timing check events
CPU time: 1.6 secs to compile + 0.3 secs to link + 0.1 secs in simulation
% verilog harddrive.v hardreg.v flop.v //
                                        See Step 4
Highest level modules:
harddrive
L49 "harddrive.v": $finish at simulation time 3400
446 simulation events + 1082 accelerated events + 936 timing check events
CPU time: 1.4 secs to compile + 0.3 secs to link + 0.1 secs in simulation
```

1. Optionally define macros in your model with `define. If you use macros in your design that are not defined with `define, you must specify the macro with +define+ at every invocation or Verilog-XL flags an error during compilation.

The cycle macro defines the clock cycle for the design and is also used as the delay for assigning values to the data line.

2. Use the macros in your model.

The clock cycle is defined using the cycle macro.

3. To pass a macro value into a module from the command line, invoke Verilog-XL with the +define+ command-line option.

The clock cycle is 50, which overrides the default (as defined by `define) of 100. The simulation ends at time 1900.

4. To use the default macro values (as defined in your model with `define), invoke Verilog-XL without the +define+ command-line option.

The clock cycle is 100 (the default). The simulation ends at time 3400.

Conditionally Compiling Source Code

You can conditionally include lines in your source description during compilation using the `ifdef compiler directive, which checks for the definition of a specified macro. Conditional compilations are useful when you want to choose different timing or structural information, different stimulus for a given Verilog-XL run, or different representations of a design.

The following example shows how you specify a macro from the command line to choose between a structural or behavioral description of a design. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Conditionally compiling source code

```
module myand (a,b,c);
output a;
input b, c;
                                      // See Step 1
'ifdef behavioral
                                    // See Step 2
    wire a = b \& c;
    initial
        $display ("Behavioral description");
`else
                         // See Step 3
    and (a,b,c);
    initial
        $display ("Gate-level description");
`endif
                          // See Step 4
initial
    $display("This line is always displayed.");
endmodule // myand
% verilog conditional compile.v
                                                // See Step 5
    . . .
Highest level modules:
mvand
Gate-level description
This line is always displayed.
5 simulation events
CPU time: 1.7 secs to compile + 0.1 secs to link + 0.0 secs in simulation
% verilog conditional_compile.v +define+behavioral // See Step 6
Highest level modules:
myand
Behavioral description
This line is always displayed.
5 simulation events
CPU time: 1.4 secs to compile + 0.1 secs to link + 0.0 secs in simulation
```

1. Introduce conditionally-compiled source code with <code>`ifdef</code> followed by the name of the macro that determines the behavior of the conditional statement.

Verilog-XL checks for the definition of the behavioral macro to determine what code to compile.

2. Verilog-XL compiles the code after `ifdef but before `else or `endif when the specified macro is defined.

If behavioral is defined, Verilog-XL compiles the behavioral description of an AND gate and displays an informational message.

3. Optionally include an `else directive to specify code that Verilog-XL compiles when the specified macro is not defined.

If behavioral is not defined, Verilog-XL compiles the gate-level description of an AND gate and displays an informational message.

- 4. End conditionally-compiled code with `endif.
- 5. Invoke Verilog-XL without defining the `ifdef macro when you do not want to include the source code following `ifdef (but include the `else code if you specified any).

Verilog-XL compiles the gate-level description because behavioral was not defined.

6. Invoke Verilog-XL with +define+ to define the `ifdef macro when you want to include the source code following `ifdef. Note that you can also define the macro in your source code with `define.

Verilog-XL compiles the behavioral-level description because behavioral was defined.

Modifying Simulation Behavior at Run Time

You can modify the behavior of your simulation at invocation time by testing for the presence of plus arguments on the invoking command line with the *stestsplusargs* system task. You can only check for plus options (+), not minus options (-). You can create your own plus arguments by specifying any string preceded by a plus sign on the command line.

The example shows how you specify user-defined plus arguments and modify your simulation based on the presence of the plus argument. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Modifying simulation behavior at run time

```
module test;
reg reset;
initial
    if ($test$plusargs("reset")) // See Step 1
    begin
        reset = 1;
        #100 reset = 0;
    end
    else
        reset = 0;
    ...
endmodule // test
% verilog test.v +reset // See Step 2
    ...
```

- Test for the presence of a plus argument on the command line that invokes the simulation with \$test\$plusargs. This system task takes one argument—a string that specifies the name of the plus argument, and returns true (1) if the plus argument is present and false (0) if not present. Verilog-XL assigns different values to the reset register based on the presence of the reset plus argument on the command line.
- 2. Define plus arguments by specifying any string preceded by a plus sign (+) on the command line. This invocation defines a reset plus argument, so the \$test\$plusargs system task in the test module returns true (1).

Inserting a File into Another File

You can insert the entire contents of a source file into another file during Verilog-XL compilation with the `include compiler directive. File inclusion is useful when you have global or commonly used definitions and tasks and do not want to repeat the code in many files. You can specify the search path that Verilog-XL uses to locate included files with +incdir+.

The following example shows how you insert one file into another and how you can specify a search path for included files from the command line. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Inserting a file into another file

```
module top;
initial $display("Start module top");
`include "one.v"
                                 // See Step 1
initial $display("End module top");
endmodule // top
// ***The file one.v***
initial $display(" First included file");
`include "two.v"
                                // See Step 2
// ***The file two.v***
initial $display("
                     Second included file");
% verilog top.v +incdir+test1+test2
                                                // See Step 3
Compiling source file "top.v"
                                                   // See Step 4
Compiling included source file "one.v"
Compiling included source file "test1/two.v"
Continuing compilation of source file "one.v"
Continuing compilation of source file "top.v"
Highest level modules:
top
Start module top
 First included file
   Second included file
End Module Top
9 simulation events
```

CPU time: 1.4 secs to compile + 0.0 secs to link + 0.1 secs in simulation

- 1. Insert another file anywhere in your source code by specifying the `include compiler directive followed by the name of the file. The file name must be surrounded by double quotes, and can include absolute or relative path names. The one module includes the file two.v.
- 2. You can nest 'include directives up to eight levels.

The file two.v, which is included in module top, itself includes the three.v file.

3. Optionally specify a search path with +incdir+. Verilog-XL searches the specified directories for included files.

Verilog-XL searches the directories test1 and test2 for any files that are included by module top.

4. Verilog-XL displays informational messages about the file currently being compiled.

Note that Verilog-XL successfully finds the file two.v because test1 is specified in the +incdir+ search path.

Generating Log Files

You can keep a record of all simulation output, interactive commands, and the command line invocation by generating log files. By default, Verilog-XL generates a log file called verilog.log. You can specify different log file names and select which simulation output you want Verilog-XL to write to the file by using the -1 command-line option and the log and log system tasks.

The following example shows how you specify a log file other than verilog.log from the command line with the -1 command-line option, and how you control output from within a module with the \$log and \$nolog system tasks. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Generating log files

```
// File name is test.v
module test;
initial
begin
   $display ("Write this to the log file"); // See Step 1
   $nolog; // See Step 2
   $display ("But don't write this");
   $log("test1.log"); // See Step 3
   $display ("First line of new log file!");
   $finish;
```

Verilog-XL User Guide

Controlling Verilog-XL

end endmodule // test % verilog test.v -l test.log // See Step 4 . . . Compiling source file "test.v" Highest level modules: test Write this to the log file But don't write this This should be the first line in the new log file L10 "test.v": \$finish at simulation time 0 9 simulation events CPU time: 1.2 secs to compile + 0.1 secs to link + 0.0 secs in simulation // See Step 5 % more test.log Host command: ~/release/software/sun/verilog/tools. sun4/vtools/vlog/exe/verilog.exe Command arguments: -f ~/release/pwf test.v -l test.log Compiling source file "test.v" Highest level modules: test Write this to the log file % more test1.log // See Step 5 This should be the first line in this new log file L9 "test.v": \$finish at simulation time 0 8 simulation events CPU time: 1.6 secs to compile + 0.2 secs to link + 0.0 secs in simulation

- 1. Verilog-XL writes all output to the log file specified from the command line by the -1 option. If you do not specify -1, the default log file is verilog.log.
- 2. To disable output to the log file, specify \$nolog.
- **3.** Reenable log file output with \$log. You can either resume output to the existing log file by not providing an argument or open a new log file by specifying a new file name, in which case Verilog-XL closes the old log file.

The log file is now test1.log.

4. Invoke Verilog-XL with or without the -1 command-line option.

Because the -1 option was specified, Verilog-XL writes startup messages to the file test.log.

5. View log files as you would any ASCII file.

This simulation produced two log files—test.log and test1.log.

Reproducing Interactive Sessions Using Key Files

You can reproduce all the commands that you enter during interactive mode by generating a key file and using the key file as a command input file to another simulation. By default, Verilog-XL generates the file verilog.key every time you enter interactive mode, but you can specify different file names and enable or disable writing to the key file with the -k command-line option and the key and nokey system tasks.

The following example shows how you specify a key file other than verilog.key during an interactive session with the key system task. You can enable and disable writing to the key file with key and nokey and use the key file as input to another simulation with sinput.

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Using key files

```
% verilog harddrive.v hardreg.v flop.v -s
                                          // See Step 1
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
Type ? for help
    C1 > $key("set_break.key");
                                                  // See Step 2
C2 > $showvars;
Variables in the current scope:
clk (harddrive) reg = 1'hx, x
clr (harddrive) reg = 1'hx, x
data (harddrive) req = 4'hx, x
q[3] (harddrive) wire = StX
    StX <- (harddrive.h1.f4): nand nd7(q, e, qb);
q[2] (harddrive) wire = StX
    StX <- (harddrive.h1.f3): nand nd7(q, e, qb);</pre>
q[1] (harddrive) wire = StX
    StX <- (harddrive.h1.f2): nand nd7(q, e, qb);</pre>
q[0] (harddrive) wire = StX
    StX <- (harddrive.h1.f1): nand nd7(q, e, qb);</pre>
C3 > $db setbreakonceonposedge(q[1]);
          Task or function ($db setbreakonceonposedge)
Error!
not defined [Verilog-TOFD] Command 3:
C3 > $db_breakonceonposedge(q[1]);
Set break (1) [once] on pos edge harddrive.q[1].
                              // See Step 3
C4 > $nokey;
C5 > $display ("Anything entered now is not written to the key file");
Anything I enter now is not written to the key file
C6 > $key;
                            // See Step 4
C7 > .
at time 50 clr = 1 data= 0 q= x
at time 150 clr = 1 data= 1 q= 0
at time 250 clr = 1 data= 2 q= 1
```

```
Break (1) [once] occured on pos edge harddrive.q[1] at time 329.
Disabled break (1) [once] on pos edge harddrive.q[1].
C7 > $showvars(q[1]);
q[1] (harddrive) wire = St1
    St1 <- (harddrive.h1.f2): nand nd7(q, e, qb);</pre>
C8 > $finish;
C8: $finish at simulation time 329
82 simulation events + 138 accelerated events + 128 timing check events
CPU time: 1.4 secs to compile + 0.2 secs to link + 0.1 secs in simulation
% emacs set break.key // See Step 5
% verilog <u>harddrive.v</u> <u>hardreg.v</u> flop.v -s // See Step 6
                                               // See Step 7
C1 > $input("set_break.key");
C2 > $showvars;
Variables in the current scope:
clk (harddrive) reg = 1'hx, x
clr (harddrive) reg = 1'hx, x
    . . .
```

- 1. Enter interactive mode using the -s command-line option (or \$stop system task or Control-c asynchronous interrupt). By default, Verilog-XL writes all interactive commands to verilog.key.
- 2. Optionally specify a key file other than the default of verilog.key with the \$key system task. Verilog-XL records all subsequent interactive commands to the new key file. Verilog-XL writes all interactive commands to set_break.key.
- **3.** Disable the key file with *\$nokey* before entering commands that you do not want Verilog-XL to write to the key file.

Verilog-XL does not write the \$display system task to the key file.

- 4. Re-enable the key file or open a new key file with the \$key system task. Because there is no argument to \$key, Verilog-XL continues writing to the current key file set_break.key.
- 5. Optionally edit the key file as you would any ASCII file to make changes to the commands stored in the key file. The set_break.key file is edited to remove the mistyped command (\$db_setbreakonceonposedge).
- 6. Rerun your simulation again entering interactive mode.
- 7. Specify the key file as an input file using the \$input system task. Verilog-XL executes
 the commands in the set_break.key key file.

Providing Interactive Commands from a File

You can provide interactive commands to Verilog-XL by specifying an input file instead of issuing commands from the terminal interactively. Use either the *sinput* system task or the

-i command-line option to specify an input file. You can use key files as input files. When Verilog-XL has processed all commands in the input file, or if you interrupt processing with Control-c, input switches back to the terminal.

The following example shows how you create a text file containing interactive commands that Verilog-XL executes when you specify the -i command-line option.

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Providing interactive commands from a file

```
// the command input file set break.inp -- See Step 1
11
$db_breakonceonposedge(q[1]);
$finish
% verilog harddrive.v hardreg.v flop.v -s -i set break.inp
Compiling source file "harddrive.v"
Compiling source file "hardreg.v"
Compiling source file "flop.v"
Highest level modules:
harddrive
Type ? for help
C1 > $db_breakonceonposedge(q[1]);
Set break (1) [once] on pos edge harddrive.q[1].
C2 > .
at time 50 clr = 1 data= 0 q= x
at time 150 clr = 1 data= 1 q= 0
at time 250 clr = 1 data= 2 q= 1
Break (1) [once] occured on pos edge harddrive.q[1] at time 329.
Disabled break (1) [once] on pos edge harddrive.q[1].
C_{2} > .
SIMULATION TIME IS 330
L10 "flop.v" (harddrive.h1.f1): nand nd8 >>> XL GATE = St1
SIMULATION TIME IS 339
L12 "flop.v" (harddrive.h1.f1): nand nd7 >>> XL GATE = St0
L10 "flop.v" (harddrive.h1.f2): nand nd8 >>> XL GATE = St0
SIMULATION TIME IS 350
L18 "harddrive.v": #50 >>> CONTINUE
C2 >
L18 "harddrive.v": clk = ~clk; >>> clk = 1'h1, 1;
L18 "harddrive.v": always
C2 > $finish;
C2: $finish at simulation time 350
79 simulation events + 142 accelerated events + 128 timing check events
CPU time: 1.7 secs to compile + 0.4 secs to link + 0.1 secs in simulation
```

1. Create your input file using any editor or by using a Verilog-XL key file.

The file set_break.inp sets a breakpoint, continues the simulation until the breakpoint triggers, then step-traces twice and finishes the simulation.

2. Invoke Verilog-XL with the -i command-line option (or use the \$input system task). You must also cause Verilog-XL to enter interactive mode with either -s, \$stop, or Control-c.

The input file is called $set_break.inp$. Verilog-XL enters interactive mode immediately after compilation because of the -s command-line option and then executes the commands in the file as though you had entered them interactively.

Storing Commonly Used Command Line Arguments

You can eliminate excessive typing for frequently used or lengthy command lines by storing command line arguments (options and model names) in a text file. Verilog-XL reads your command argument file when you invoke Verilog-XL with the -f command-line option followed by the name of your command line argument file.

The following example shows command line argument files used to run Verilog-XL with a standard set of command options.

Example: Storing commonly used command line arguments

```
/* project1.vc: this file contains the conventions for describing
     project1 hardware -- See Step 1 */
    keep all
                                 keeps channel-connected nets
+sxl
                           11
+incdir+</net/switches> // Verilog-XL searches this directory
+delay_mode_unit //use unit delay mode
/**** end of file: project1.vc ****/
/* user.vc: this file contains my usual options */
              //accelerate the simulation
-a
-l design.log // name the log file
-k design.key // name the key file
-i run1000.vi // input file to run at first
// $stop or with -s option
-f cpu.vc // simulate the cpu design
    /**** end of file: user.vc ****/
    /* cpu.vc: this file specifies the cpu models */
cpu_netlist.v // cpu description
array_lib_version2.v // gate array cell library
joe_alu.v // use Joe's alu description
    /**** end of file: user.vc ****/
% verilog -f project1.vc -f user.vc // See Step 2
```

1. Create your command line argument files with any text editor.

The file project1.vc is a central file of conventions for a particular project.

The file user.vc contains options specific to the user. Note that this file itself accesses an argument file called cpu.vc.

The file cpu.vc contains the models that Verilog-XL compiles after invocation.

2. Invoke Verilog-XL with the -f command-line option followed by the name of the argument file.

This invocation has two -f command-line options. The files project1.vc and user.vc provide the command line arguments for this invocation.

Specifying the Delay Type

Specify whether you want to use minimum, maximum, or typical delay values (as defined in your model with *min:typ:max* delay expressions) when you invoke Verilog-XL. Specify one of the following options in your command line: +mindelays, +typdelays, +maxdelays. If you do not specify one of these options, Verilog-XL uses typical delays.

The following example shows a model containing min:typ:max delay expressions that is simulated twice, once explicitly with minimum delays and again with the default behavior using typical delays.

"See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Specifying the delay type

```
/* This is a simple example showing -- See Step 1 */
/* how you select delay values.
                                      * /
module select delays;
reg a,b,c,i;
initial
    begin
    #0 a=1; b=0; c=0;
    #1 b <= #(1:2:3) a; // min:typ:max specification</pre>
       c <= #2 a; // single delay specification</pre>
end
initial
for (i=0; i<5; i=i+1)</pre>
    #1 $display("time=%0d, a=%0d, b=%0d, c=%0d",
                         $stime, a, b, c);
endmodule // select_delays
% verilog +mindelays delays.v
Compiling source file "delays.v"
Highest level modules:
select_timing
time=1, a=1, b=0, c=0
time=2, a=1, b=0, c=0
```

```
time=3, a=1, b=1, c=0
time=4, a=1, b=1, c=1
time=5, a=1, b=1, c=1
31 simulation events
% verilog delays.v
...
Compiling source file "delays.v"
Highest level modules:
select_timing
time=1, a=1, b=0, c=0
time=2, a=1, b=0, c=0
time=3, a=1, b=0, c=0
time=4, a=1, b=1, c=1
time=5, a=1, b=1, c=1
31 simulation events
```

- 1. Write your models with min: typ: max delay specifications. This module has two delay specifications. The first has the standard min: typ: max format. The second has only a single value, which Verilog-XL uses regardless of which command-line option for selecting the delay type you choose. Note that you cannot specify two delays in a single delay specification; you must specify either a single delay or the entire min: typ: max format.
- 2. Invoke Verilog-XL with a command-line option selecting minimum (+mindelays), typical (+typdelays), or maximum (+maxdelays) delays.

This invocation selects minimum delays. The second invocation does not explicitly specify which delays to use, so Verilog-XL uses its default of typical values.

3. When you invoke Verilog-XL without specifying +mindelays, +typdelays, or +maxdelays Verilog-XL uses its default of typical values (+typdelays).

Selecting a Delay Mode

Modify the delay behavior (not procedural delays) in your design by selecting one of Verilog-XL's delay modes with compiler directives (for individual modules) and command-line options (for the entire simulation). Changing modes lets you make trade-offs between improved simulation time and timing correctness. The five delay modes are unit, zero, distributed, path, and default.

The following example shows a simulation with two modules, each with different delay modes specified by compiler directives. The delay command-line option overrides the delay modes for the modules. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Selecting a delay mode

Verilog-XL User Guide

Controlling Verilog-XL

```
// Flop model (flop_model.v)
11
'delay_mode_distributed
                                            // See Step 1
module flop (data, clock, clear, q, qb);
input data, clock, clear;
output q, qb;
nand #10 nd1 (a, data, clock, clear),
                                                 // See Step 2
          nd2 (b, ndata, clock),
          nd4 (d, c, b, clear),
          nd5 (e, c, nclock),
nd6 (f, d, nclock),
          nd8 (qb, q, f, clear);
nand #9 nd3 (c, a, d),
          nd7 (q, e, qb);
not #10 iv1 (ndata, data),
          iv2 (nclock, clock);
endmodule // flop
// Flop Text Fixture (flop_test.v)
11
`resetall
                            // See Step 3
module flop test;
reg clk, clr;
reg data;
wire q;
flop f1 (data, clk, clr, q,);
initial
begin
    clr = 1;
                  clk = 0;
    $monitor("time = %0t, data = %b, q = %b", $stime, data, q);
end
                                           // See Step 4
always #50 clk = ~clk;
initial
begin
    data = 0;
    #100 data = 1;
    #100 \text{ data} = 0;
    #100 \ clr = 0;
    #100 data = 1;
    #100 \text{ data} = 0;
    #100 $finish;
end
endmodule // flop test
% verilog flop test.v flop model.v // See Step 5
time = 0, data = 0, q = x
time = 100, data = 1, q = x
time = 139, data = 1, q = 0
time = 200, data = 0, q = 0
time = 229, data = 0, q = 1
time = 319, data = 0, q = 0
time = 400, data = 1, q = 0
time = 500, data = 0, q = 0
L24 "flop test.v": $finish at simulation time 600
83 simulation events + 71 accelerated events + 46 timing check events
CPU time: 0.4 secs to compile + 0.2 secs to link + 0.1 secs in
simulation
```

```
% verilog flop test.v flop model.v +delay_mode_zero // See Step 6
...
time = 0, data = 0, q = x
time = 100, data = 1, q = 0
time = 200, data = 0, q = 1
time = 300, data = 0, q = 0
time = 400, data = 1, q = 0
time = 500, data = 0, q = 0
L24 "flop_test.v": $finish at simulation time 600
83 simulation events + 71 accelerated events
CPU time: 0.3 secs to compile + 0.2 secs to link + 0.1 secs in simulation
```

- 1. Specify the delay mode for all subsequent modules with one of the following compiler directives: `delay_mode_zero, `delay_mode_unit, `delay_mode_path, `delay_mode_distributed. The flop module uses distributed delay mode Verilog-XL uses distributed (gate) delays and ignores path delays (if any).
- 2. Specify delays for your model, whether they are distributed or path delays. The flop module uses distributed delays.
- 3. Override active compiler directives by specifying another delay mode directive or the `resetall directive. The `resetall directive resets the delay mode for subsequent modules back to the default delay mode. (Verilog-XL honors both distributed and path delays.)
- 4. Delay modes do not affect procedural delays.

Changing the delay modes does not affect the behavioral description of a periodic clock waveform and the delays given to the input assignments.

- 5. To use the delay modes specified by compiler directives in your source code, do not specify a delay mode command-line option. Because of the distributed delays in flop.v, the output q follows data by 9 time units.
- 6. To override the delay modes specified by compiler directives in your source code, use one of the following command-line options: +delay_mode_zero, +delay_mode_unit, +delay_mode_path, +delay_mode_distributed.

Because the command line sets the delay mode to zero, the output ${\tt q}$ now follows ${\tt data}$ with zero delay.

Library Management

This chapter describes the following:

- <u>Overview</u> on page 99
- Organizing Libraries on page 100
- <u>The Standard Library Management Scheme</u> on page 102
- The Former Library Management Scheme on page 106
- <u>The Library.Cell:View Library Management Scheme</u> on page 121
- <u>Accessing Libraries</u> on page 127

Overview

Normally, Verilog-XL compiles all the modules that are defined in a source text file; those that are not instantiated become top-level modules. Creating libraries avoids this unnecessary origination of top-level modules, saving compile time and memory. When Verilog-XL cannot find a module or UDP definition in the design description to match a module or UDP instantiation, it searches the libraries associated with the design description for the definition.

Cadence supports three library management schemes for Verilog-XL.

- The standard library management scheme
- The former library management scheme
- The Library.Cell:View library management scheme

The standard and former schemes can coexist and use the same libraries, but explicit application of the standard scheme take precedence over the former scheme. The Library.Cell:View management scheme cannot be used with the other schemes; one or the other(s) must be used.

Organizing Libraries

This section describes the three methods of organizing libraries: library files. library directories, and the Library.Cell:View architecture.

Library Files

Library files are Verilog-XL source text files that contain one or more module or UDP definitions. The module definitions can have their own hierarchical structures that instantiate modules within themselves.

The names of the definitions in library files are always one of the following:

- The names of the instantiations in the design description that have no matching definitions in the design description. In this case, Verilog-XL completes the design description by referring to those names.
- The names of the definitions instantiated in the library file itself.

Note: If a module definition in a library file has a hierarchical structure, the definition's top-level module should precede its lower-level definitions. This rule applies to all other modules defined in this library file as well.

Library Directories

Library directories are operating system directories that contain Verilog source text files. Each library directory file can contain one module or UDP definition, or a hierarchical design description. If it contains a hierarchical design description, the hierarchy's top-level module definition should precede its lower level definitions. The names of the UDPs or top-level modules in library directory files are the names of the instantiations in the design description that do not have matching definitions in the design description itself.

You can specify extensions of library directory filenames to differentiate the versions of the definitions for which Verilog-XL searches. Verilog-XL concatenates these extensions to the ends of names of instantiations that lack matching definitions and searches for the files with the resulting names.

Each library directory file has either the same name as the UDP or top-level module that it contains, or it has that name plus an extension. Deviating from this rule results in a warning message.

The Library.Cell:View Architecture

Compiled objects are stored in a library according to the following Library.Cell:View (L.C:V) architecture. For more information about this library management scheme, see <u>"The Library.Cell:View Library Management Scheme"</u> on page 121.

Reporting of Resolution Paths

You can query Verilog-XL in the following ways regarding the files in which the definitions of modules and UDPs were found during library scanning:

- The \$showallinstances system task displays the name of the file that contains the definition of the modules and UDPs in the circuit description. This works for all modules and UDPs, not just those in libraries. However, this system task can only be used if compilation completes without error.
- The +libverbose command-option tells Verilog-XL to display information about opening files and the resolution of module and UDP definitions during the library scanning. Use this option to debug any problems related to the resolution of instances during library scanning.

Definition Renaming

To differentiate between module or primitive definitions that have the same name, Verilog-XL adds suffixes that make the names unique. This feature applies to the standard and former library management schemes.

Renaming the primitive definitions has no effect on simulation. You can see these unique names when you invoke the <code>\$showallinstances</code> or the <code>\$list</code> task.

- To rename a definition accessed with the former scheme, Verilog-XL adds a suffix such as \$lib4 to the definition's name. The \$lib portion of the suffix is present in all names that Verilog-XL generates for definitions accessed with the former scheme.
- To rename a definition accessed with the standard scheme, Verilog-XL adds a suffix such as \$inst4_1\$9 to a primitive definition's name or to a module definition's name. The number following the second dollar sign in the name that Verilog-XL generates for a module definition is the number of characters in the original name. The number following the first dollar sign in the name that Verilog-XL generates for a module definition is not significant.

Syntax Checking in Library Files

In library files, Verilog-XL checks the syntax of only those definitions that actually resolve instances; other module and UDP definitions within these files are not checked. You can use the -c command-line option to direct Verilog-XL to compile all of the definitions within the library files and stop the Verilog-XL process before simulation begins.

The Standard Library Management Scheme

The standard library management scheme has the following advantages over the former scheme:

- Avoids compiling all of the module and UDP definitions in a source file.
- Duplicate names for modules and UDPs in different libraries can exist without causing any problems.
- The commands that configure the library search do not have interdependencies, and their command-line order is not significant. Order dependency that exists in the compiler directives that implement the scheme is a source of efficiency.
- You can control where the library search takes place, removing or including library files for each device whose instantiation needs a definition.

'uselib

The `uselib compiler directive specifies where Verilog-XL searches for the definitions of modules and UDPs that are instantiated in a design description that does not include their definitions. Each `uselib directive explicitly defines the library search that resolves the instances that follow it until the compiler encounters another `uselib directive, which completely redefines the search. If the design description includes no `uselib compiler directives, the compiler searches the command line for options that are part of the former scheme to configure the library search. The +librescan and +liborder options have no effect on `uselib compiler directives. An empty `uselib directive makes the preceding `uselib directives ineffective.

The following example shows the `uselib directive syntax.

```
<uselib_compiler_directive>

::= `uselib <library_reference>*

<library_reference>

::= file= <LIBRARY_FILE_NAME>

||= dir= <LIBRARY_DIRECTORY_NAME> libext= <FILE_EXTENSION>

||= <empty>
```

Defining Macros for the 'uselib Compiler Directive

It is efficient to use the +define+ command-line option and the `define compiler directive to define the paths with which the `uselib compiler directive configures the search. This practice enables you to use brief macro names in the `uselib compiler directives anywhere in a design description and to localize the search paths for easy alteration and portability.

The following example shows this technique for defining paths:

```
`define ASIC1 dir=/net/library/asic1/source libext=.v
    `define ASIC1_UDP file=/net/library/asic1/udp.v
    `uselib `ASIC1 `ASIC1_UDP
module asic (in,out,bidir);
    input [56:0] in;
    output [21:0] out;
    inout [63:0] bidir;
    ...
    endmodule
    `uselib
```

In the previous example, one `uselib compiler directive configures all of the librarysearching for the model. The first `define directive configures a search of all the files in the directory /net/library/asic1/source that have names that are the concatenations of the names of unresolved instantiations and that have the extension .v. If such a file includes a module or UDP that has the name of an unresolved instantiation in the source text, then that module or UDP supplies the definition for the module or UDP in the source text.

The second `define directive configures a search of the file /net/library/asic1/ udp.v for modules or UDPs that have the names of unresolved instantiations.

The empty `uselib compiler directive at the end of the module makes the compiler use the options on the command line to configure library searches until the compiler encounters a non-empty `uselib compiler directive.

Note: If you define the same macro with both a compiler directive and a command-line option, the command-line option takes precedence and Verilog-XL displays a warning message.

The following example shows a board-level simulation model that uses the 'uselib compiler directive with nested macro definitions:

103

Verilog-XL User Guide Library Management

In the previous example, the first `define directive defines a macro that is incorporated in the others. The first `uselib directive specifies that the search for the definitions of the U1 and U2 devices.

The second `uselib directive adds a path to the first one to specify the search for the definition of U3. Adding this path requires a new `define directive, because each `define directive completely redefines the search. The second `define directive can specify the search for devices U1, U2, and U3. The last `uselib directive defines a search path for an instantiation that has just the same name as the first instantiation, but the different path enables Verilog-XL to find the correct definition.

The final `resetall compiler directive has the same effect as an empty `uselib directive; it makes the compiler use the command line to configure the library search. In addition, the `resetall compiler directive makes any other compiler directives that precede it ineffective.

The following example shows how macro definitions are expanded to increase portability:

```
`define ASIC1 dir=/net/library/asic1/source libext=.v
`define ASIC1_UDP file=/net/library/asic1/udp.v
/* The two lines above expand to the single directive below: */
`define ASIC_LIB dir=/net/lib/asic1/source file=/net/lib/asic1/udp.v libext=.v
`define LIB_ROOT /net/lib
`define TTL_LIB dir=`LIB_ROOT/TTL_LIB/source libext=.v
`define TTL_UDP file=`LIB_ROOT/TTL_LIB/udp.v
`define FAST_LIB dir=`LIB_ROOT/FAST/source libext=.v
/* The four lines above expand to the two directives below: */
`define TTL_LIB dir=/net/lib/TTL_LIB/source file=/net/lib/TTL_LIB/udp.v libext=.v
```

The `uselib directive can also make a multiple-line specification when you use the backslash character (\), as the following lines show:

It is possible to include names in specifications that contain characters that have special significance to the compiler by preceding such characters with the backslash character. The backslash character deprives the character following it of any special significance to the compiler, which then reads the character only for identification purposes. This is helpful when

you work with libraries whose naming you cannot control. Characters that require such treatment include the accent grave (`), the dollar sign (\$), the white space, and the backslash (\) itself.

Search Order and Efficiency

Library scanning occurs in the left-to-right order in which you specify the paths to the 'uselib compiler directive. Each time the compiler encounters an unresolved instantiation in the design description, it starts a search that begins at the beginning of the path specified in the applicable 'uselib directive. The compiler also scans the paths specified in the 'define compiler directives and in the +define+ command-line options in left-to-right order.

Consequently, the compiler obeying the `uselib directive in the following example scans the macros in the following order: ASIC1, ASIC2, ASIC_UDP1, ASIC_UDP2, ASIC_UDP3.

```
`define ASIC1 dir=/net/library/asic1/udp.v libext=.v
`define ASIC2 dir=/net/library/asic2/udp.v libext=.v
`define ASIC_UDP1 file=/net/library/asic1/udp.v
`define ASIC_UDP2 file=/net/library/asic2/udp.v
`define ASIC_UDP3 file=/net/library/asic3/udp.v
`define ASICS `ASIC_UDP1 `ASIC_UDP2 `ASIC_UDP3
```

'uselib 'ASIC1 'ASIC2 'ASICS

If you want the compiler to scan the ASIC_UDP libraries in the reverse order, you could insert the following directive at the top of the file identified by file=/net/library/asic1/udp.v or in the design description:

`define ASICS `ASIC_UDP3 `ASIC_UDP2 `ASIC_UDP1

This `define directive makes the compiler scan the macros in the following order: ASIC1, ASIC2, ASIC_UDP3, ASIC_UDP2, ASIC_UDP1.

With one exception, any instantiation inside a library must have its definition within the library path; searching a path specified by a `uselib directive must not lead to searching outside that path. The exception is that a `uselib specification within a library can specify a search outside the library path. A `uselib specification in a library can include the source file, so that the source file can supply the definitions for the instantiations in a library, but this is not a recommended practice. If there is an unresolved instantiation inside a library, the search to resolve it begins at the beginning of the path that controls the search when the compiler initially encounters the instantiation, unless the library contains a `uselib directive.

If it is not possible to resolve an instantiation by searching the paths specified in the applicable `uselib directive, a message similar to the following provides information about the problem. The path following File: shows the area of the search. The path in quotes shows

the file that includes the unresolvable instantiation. The last item is the unresolvable primitive itself.

```
Error! Instance specific item not found in `uselib
  path:
    File : /net/machine/home/wally/1.6c/co/lib1/multparts [Verilog-LISRE]
    ``/net/machine/home/wally/1.6c/co/mult1", 5: not1 (cnot, c);
```

The following practices make library searches more efficient:

- Give each `uselib directive the least input possible
- Configure the search specifications so that the definitions the compiler seeks most frequently precede other definitions in the maximum number of searches.

The Former Library Management Scheme

The following sections describe the former library management scheme.

Using Library Files: The Former Scheme

To use a library file, you specify the -v command-line option with the name of the library file you want to use. Verilog-XL scans this file for module and UDP definitions that cannot be resolved in the normal source text files specified.

The following example shows how to use the -v command-line option:

verilog sourcel.v -v libfile.v

Using Library Directories: The Former Scheme

To use a library directory, you specify the -y command-line option with the path to the library directory. Verilog-XL scans this directory for files containing definitions of modules or primitives that are unresolved in the specified source file. The following example shows how to use the -v command-line option:

verilog source1.v -y /usr/me/proj/lib/cmos

Files in library directories may contain just one module or UDP definition, or they may be complete hierarchies. If they are hierarchical, then the top level of the hierarchy must be the first module declared in the file.

Library directory files are not scanned unless they have the same name as a module or UDP that has been instantiated within the normal source text but has not been resolved.

File Extensions in Library Directories: The Former Scheme

You can specify the files in a library directory that you want Verilog-XL to use to resolve module and UDP definitions by specifying these files' extensions. If you choose not to specify any extensions, each filename in the library directory that you specify must be identical to the name of the module or UDP they contain. You specify library directory file extensions using the +libext+ command-line option.

Enter +libext on the command line followed immediately (no spaces) by the strings of characters that make up each extension. You must surround each extension string with two + signs. Since the + signs separate one extension string from another, the final + sign in the argument is optional. For example, the following two command lines are equivalent:

```
verilog sourcel.v -y /usr/me/lib/cmos +libext+.v
verilog sourcel.v -y /usr/me/lib/cmos +libext+.v+
```

This example specifies the files in the library directory named <module_or_UDP_name>.v.

You can specify multiple library directory file extensions. If a file that has the first extension is not found, Verilog-XL tries each extension that follows it on the command line until the file is found or until Verilog-XL has tried all the listed extensions. All specified extensions must follow a single +libext option on the command line, as in the following example:

verilog source1.v -y /usr/me/lib/cmos +libext+.v+.v2+

The extension string need not contain a period character. Verilog-XL concatenates each string specified to the ends of the names of the modules and UDPs that need to be resolved during library directory searching. This means that all the extensions shown below are valid:

```
verilog sourcel.v -y /usr/me/proj/lib/cmos \
+libext+.v+_version_3+64+
```

Suppose the modules NAND2, MUX, and ADDER cannot be resolved in source1.v in the command line shown above. Verilog-XL scans the following files if they reside in the library directory /usr/me/proj/lib/cmos:

NAND2.v MUX_version_3 ADDER64

Note: Only one +libext command-line option can be specified for any invocation of Verilog.

In some situations, you may want Verilog-XL to scan the library directory files that have no extensions—or *null extensions*—along with the files that do have extensions. You can specify a null extension as two adjacent + signs, like this: ++. Here is an example:

verilog source1.v -y /usr/proj/lib/cmos +libext++.v+

This command line directs Verilog-XL to look first for files with no extension, and then for files with the extension .v.



There is currently no syntax checking of plus command options. Be very careful in specifying them to avoid confusing results. If you misspell libext, the option gets ignored.

Library Scan Precedence: The Former Scheme

When Verilog-XL finds an instance of a module or user-defined primitive (UDP) that cannot be resolved in the source description files, it scans for a definition in the library files or directories that you specify on the command line. Once Verilog-XL finds a definition, it resolves the reference and ignores all subsequent definitions of the module or UDP that it encounters in library files or directories.

If your library files or directories contain multiple modules or UDPs with the same name, then the scan precedence Verilog-XL uses to find definitions becomes important.

Verilog-XL can use three possible scan precedences:

- the default scan precedence
- the +liborder scan precedence
- the +librescan scan precedence

Default Scan Precedence

The default scan precedence that Verilog-XL uses to search for definitions in library files or directories is as follows:

- If the unresolved instance is in a source file, Verilog-XL scans library files or directories in the order in which they are entered on the command line. It begins with the left-most library file or directory, no matter where the source file appears on the command line. After Verilog-XL scans the left-most library, it scans the others in the order in which they appear on the command line.
- If the unresolved instance is in a library file or directory, Verilog-XL scans in the following order:
 - a. It checks the library file or directory that contains the unresolved instance.
 - **b.** If the instance remains unresolved, it then scans the remaining libraries, beginning with the one that follows the library containing the unresolved instance. It continues
in a circular manner—that is, it scans the libraries as they follow on the command line and then "wraps around" to the left-most library until it visits each one.

When you use the default scan precedence, you should enter library files and directories on the command line in the order in which you want them scanned. Consider the following command-line example:

verilog src1.v -y /usr/lib/NMOS \
src2.v -v usr/lib/TTL/ttl.v -y /usr/lib/CMOS

In this example, if source file src1.v or src1.v or src2.v, Verilog-XL scans for a definition first in /usr/lib/NMOS, then in /usr/lib/TTL/ttl.v, and finally in /usr/lib/CMOS.

However, if the module is neither instantiated nor defined in either source file, but is instead instantiated in the library file /usr/lib/TTL/ttl.v then Verilog-XL looks for a definition first in /usr/lib/TTL/ttl.v, then in /usr/lib/CMOS, and finally in /usr/lib/NMOS.

If you have multiple modules or UDPs with the same name in your libraries, you can control how Verilog-XL resolves undefined instances with the default scan precedence in the following ways:

Enter library files and directories on the command line in the order in which you want them scanned for module and UDP definitions. For example:

```
verilog src1.v src2.v -y /usr/lib/new -\
y /usr/lib/old
```

In the previous example, Verilog-XL first scans for the definitions of all undefined modules and UDPs from the source files first in /usr/lib/new. If any instances remain unresolved, Verilog-XL scans /usr/lib/old. This method may not work if a library module contains an unresolved instance of another module or UDP.

If your libraries contain multiple modules with the same name that have unresolved instances, you should use the +librescan option.

Use library directory filename extensions and the +libext+ command-line option. As described in <u>"File Extensions in Library Directories: The Former Scheme" on page 107</u>, Verilog-XL searches for the extensions listed after the +libext+ command-line option from left to right in the order in which you specify them on the command line.

To make one set of files within a library directory take precedence over another set, use different extensions on the two sets, and list the extension of the dominant set immediately after the +libext option. Consider this example: A library named /usr/lib contains multiple modules with the same name. The newer versions of these modules have a .v2 extension; the older versions have a .v1 extension. To give the modules with the .v2 extension higher precedence, enter the following command line:

verilog src1.v src2.v -y /usr/lib +libext+.v2+.v1+

The module and UDP definition library files with the $.v_2$ extension are used to resolve undefined instantiations first. because the $.v_2$ extension appears first after the plus option.

+liborder

The +liborder plus option allows you to order a library search according to where the first instance of an unresolved module is detected. The compiler can find instances of unresolved modules in source descriptions or in library files. When the origin is a source description, +liborder directs the compiler to start searching in the library file or directory immediately following that source file. However, if the module instance is detected in a library file, +liborder initiates the search in that library.

In either case, if the module remains unresolved, +liborder directs Verilog-XL to scan the remaining library files and directories in a *circular order*—that is, to scan libraries as they follow on the command line and then "wrap around" to preceding libraries that it has not yet visited.

For example, suppose that you add the +liborder option to the command line, as in the following example:

verilog source1.v -v lib1.v source2.v -v lib2.v +liborder

Now suppose that the compiler detects an instance of the unresolved module dff in the source description <code>source2.v</code>. To resolve the module, Verilog-XL first searches for a description of dff in <code>lib2.v</code>. If the module remains unresolved, the search continues in <code>lib1.v</code>.

For an instance of dff that appears in sourcel.v, the compiler searches for the module definition first in lib1.v, and then in lib2.v.

Now consider the following command line:

```
verilog src1.v -y /usr/lib/NMOS \
src2.v -v /usr/lib/TTL/ttl.v \
-y /usr/lib/CMOS +liborder +libext+.a+.b++
```

Suppose that the compiler finds an unresolved module ttl in library

/usr/lib/TTL/ttl.v. The +liborder option directs the search for a description of ttl in the following manner:

- **1.** Scan library file /usr/lib/TTL/ttl.v.
- 2. If ttl remains unresolved, scan library files ttl.a, ttl.b, and ttl in the order listed in the directory /usr/lib/CMOS.

3. If ttl is still unresolved, scan library files ttl.a, ttl.b, and ttl in the order listed in the directory /usr/lib/NMOS.

Suppose that the description of ttl is detected in one of the library files in the directory / usr/lib/CMOS. However, in the process of resolving ttl, Verilog-XL detects an instance of a new unresolved module ttl_buf in the description of ttl. The +liborder option directs the compiler to search for a description of ttl_buf in the following manner:

- 1. Scan library files ttl_buf.a, ttl_buf.b, and ttl_buf in the order listed in the directory /usr/lib/CMOS.
- 2. If ttl_buf remains unresolved, scan library files ttl_buf.a, ttl_buf.b, and ttl_buf in the order listed in the directory usr/lib/NMOS.
- 3. If ttl_buf is still unresolved, scan the library file /usr/lib/TTL/ttl.v.

The option +liborder is especially useful for resolving multiple descriptions of modules or primitives that have the same name. For example, consider this situation: Suppose you want to compare the timing performance of two modules called ttl_fast, each from a different library.

To make this comparison, you create two source descriptions—test_ttl_1.v and test_ttl_2.v—that each contain instantiations of ttl_fast. It is critical that the compiler resolve all instances of ttl_fast in test_ttl_1.v from one library and all instances of ttl_fast in test_ttl_2.v from the other library. The following command line will accomplish this task:

verilog test_ttl_1.v -v lib1.v test_ttl_2 \ -v lib2.v +liborder

In this type of situation, Verilog-XL internally appends a unique string to the names of modules or primitives that would otherwise be identical. The string is created by concatenating the prefix \$lib with a number denoting the position of the resolving library file or directory on the command line.

Since the goal is to preserve uniqueness, the first definition of a module or primitive retains its original name, and only *subsequent* definitions of the same name receive the *slib* string.

Therefore, in the previous example, the first definition of the module detected in $test_ttl_1.v$ and resolved in lib1.v retains the name ttl_fast . However, when an identically named module is encountered in $test_ttl_2.v$, it is resolved in lib2.v (the second library on the command line) and the name of this second definition becomes ttl_fast ?

Verilog-XL User Guide Library Management

Though primarily for internal use, these unique identifiers show up whenever you directly or indirectly request information about module or primitive instances—for example, by invoking \$showallinstances or specifying +libverbose (see <u>"Reporting of Resolution Paths" on page 101</u>).

Note: You cannot use the +liborder plus option with the +librescan plus option.

+librescan

The +librescan plus option allows you to specify one order in which Verilog-XL scans library files and directories to resolve all undefined module and UDP instances from both source files and libraries. The behavior of +librescan depends on the location of the undefined instance—that is, it depends on whether the instance is located in a source file, a library file, or a file within a library directory.

When the undefined instance is located in a source file, +librescan acts the same as the default scan precedence—scanning begins with the left-most library on the command line and continues through the remaining libraries from left to right.

When the undefined instance is located in a library file, and +librescan is in effect, Verilog-XL does not continue to search the library file for the matching definition; instead, it begins to scan through the left-most library on the command line, and then scans the remaining libraries in the order in which they appear.

Finally, when the undefined instance is located in a library directory file, and +librescan is in effect, Verilog-XL scans the library directory file first to try to resolve the instance. If the instance remains unresolved, it begins to scan the left-most library on the command line, followed by the remaining libraries in the order that they appear.

The following example includes three library files: lib.orig.v, lib.revised.v, and lib.latest.v. Library lib.orig.v contains the original versions of all the modules. Library lib.revised.v contains revised versions of many of the modules from lib.orig.v.

The library lib.latest.v contains the latest revisions of just a few of the modules. To resolve all undefined instances with the most up-to-date modules, use the following command line:

```
verilog source.v -v lib.latest.v \
    -v lib.revised.v -v lib.orig.v +librescan
```

In this example, if lib.orig.v instantiates a module that is defined in that library, Verilog-XL looks for a definition of the module instance first in lib.latest.v, then in lib.revised.v, and finally in lib.orig.v.

Note: You cannot use the +librescan plus option with the +liborder plus option.

Summary of Library Scan Precedence

The order that Verilog-XL uses to search libraries for definitions of modules and UDPs depends on the location of the unresolved instance and the type of scan precedence used. The differences between the different types of scan precedences are shown in <u>Table 6-1</u> on page 113.

Table 6-1

Location	Scan Precedence	Library Search Order
Source file	default and +librescan	1. the left most library file or directory on the command line
		2. the remaining libraries in the order in which they appear on the command line
Source file	+liborder	1. the library file or directory that follows the source file on the command line
		2. the remaining libraries in a circular order, scanning libraries as they follow on the command line, and then "wrapping around" to the left-most library and any following libraries as yet unvisited
Library file	+librescan	1. the left-most library file or directory on the command line
		2. the remaining libraries in the order in which they appear on the command line
Library file	default and +liborder	1. the library file that contains the instance
		2. the remaining libraries in a circular order, scanning libraries as they follow on the command line, and then "wrapping around" to the left-most library and any following libraries as yet unvisited
File in a Library Directory	+librescan	1. the file in a library directory that contains the instance
		2. the left-most library file or directory on the command line
		3. the remaining libraries in the order in which they appear on the command line

Verilog-XL User Guide

Library Management

Location	Scan Precedence	Library Search Order
File in a Library Directory	default and +liborder	1. the file in the library directory that contains the instance
		2. the other files in that library directory
		3. the library file or directory that follows what on the command line
		4. the remaining libraries in a circular order, scanning libraries as they follow on the command line, and then "wrapping around" to the left-most library and any following libraries as yet unvisited.

Reading Library Directory Files: The Former Scheme

By default, when Verilog-XL reads in the contents of a library directory file, it assumes that the file contains a complete hierarchy. A library directory file contains a complete hierarchy if the first module or UDP definition in the file is the only definition in the file that resolves an instance outside the file. Any other definitions that follow it in the file are used only to resolve instances within the file itself. The default method of reading library directory files ensures that these "internal" definitions are used to resolve instances in other library directory files.

The Default Method

When Verilog-XL opens a library directory file, it reads in only the module and UDP definitions that it needs to resolve undefined instances. Starting at the beginning of the file, the compiler continues to read the definitions it needs until it reads in the definition whose name matches the library directory filename.

After it reads in the definition with the matching name. The compiler reads in all of the remaining module or UDP definitions, whether or not they resolve any undefined instances in the source files.

While reading the remaining module or UDP definitions, Verilog-XL appends a special character string to each definition name. For any instance resolved by an "internal" definition, the compiler appends the same character string to the module or UDP type that is named in the instantiation. Any "internal" module definitions that are not instantiated become top-level modules. The same character string is appended to each definition name in the file. It begins with a dollar sign (\$) followed by the name of the module or UDP that matches the filename.

The following figure shows two definitions in a library directory file. In the default method, Verilog-XL appends a character string to one of these definition names.



The previous figure shows the contents of a library directory file named $mux4_1.v$. The first definition in this file is a definition of a module for a four-into-one multiplexer named $mux4_1$. (Its name is the same as the filename.) Module $mux4_1$ contains three instances of module type $mux2_1$.

The definition of module $mux2_1$ follows the definition of module $mux4_1$. Module $mux2_1$ describes a two-into-one multiplexer.

When Verilog-XL opens this library directory file, it performs the following functions:

1. Verilog-XL reads in the definition of the module named mux4_1. Since the definition name matches the filename, Verilog-XL does *not* append a character string to this definition name, but makes mux4_1 part of the character string that it appends to all the definitions that follow.

2. Verilog-XL reads in the definition of the module named mux2_1. It resolves the instances in module mux4_1 with this definition. Verilog-XL also resolves all instances of mux2_1 in subsequent library directory files with this definition.

The *\$list* system task displays module and UDP names with the character strings that were appended by the default method.

Guidelines for Using the Default Method

Use the default method if your library directory files contain complete hierarchies. A library directory file contains a complete hierarchy if the first module or UDP definition in the file is the only definition in the file that resolves an instance outside the file.

Using the default method, Verilog-XL can use only the following definitions in library directory files to resolve instances in subsequent library directory files:

- the definitions that precede the definition whose name matches the filename
- the definition whose name matches the filename

The following example illustrates when to use the default method. Suppose you use library directories from two different vendors. Both vendors wrote definitions of UDPs for D flip-flops.

These definitions have the same name, but they define different kinds of D flip-flops. The following figure shows the contents of the library directory files that contain these definitions:



The previous figure shows the contents of:

- A file named dffl.v and the contents of the library file in directory vendor1, and
- A file named dff32.v and the contents of the library file in directory vendor2.

File dffl.v contains a module definition named dff1 that contains an instance of a UDP named dflop, while the library file in directory vendor1 contains the definition of dflop that defines a *rising-edge* D flip-flop.

File dff32.v contains a module definition named dff32 that also contains an instance of a UDP named dflop, while the library file in directory vendor2 contains the definition of dflop that defines a *falling-edge* D flip-flop.

If the source.v file contains instances of the modules named dff1 and dff32, Verilog-XL opens both files when you enter the following command line:

verilog source.v -y vendor1 -y vendor2 +libext+.v

Verilog-XL opens file dffl.v first because library vendor1 appears first on the command line. It resolves the instance of dflop in module dffl with the definition in that file of a UDP for a *rising-edge* D flip-flop. In the default method, Verilog-XL appends \$dffl to the definition name dflop.

When Verilog-XL opens file dff32.v and encounters the instance of dflop in module dff32, it also resolves this instance with the definition in file dff1.v.

However, if you want to ensure that Verilog-XL resolves the instance of dflop with the definition in file from Library vendor 2 of a UDP for a falling-edge D flip-flop, you should use the 'uselib compiler directive before the begining of a module instantiation definition in your design description file. Let's consider the following example where the dff32.v has been rewritten to include the 'uselib compiler directive:

```
module dff32(clk,d,q);
   .
   .
   .uselib file = vendor2/dflop.v
   dflop d1 (clk,d,q);
   .
   .
   endmodule
```

+libnonamehide

You can override the default method with the +libnonamehide plus option. This option directs Verilog-XL not to append character strings to any of the definition names in library directory files.

If you include the +libnonamehide plus option, Verilog-XL reads in only the module and UDP definitions that it needs to resolve instances. It reads in the module and UDP definition names as they are written in the file without appending character strings.

Guidelines for Using +libnonamehide

Use the +libnonamehide plus option if your library directory files contain more than one definition that Verilog-XL could use to resolve instances outside the file. With +libnonamehide, Verilog-XL can use every module and UDP definition that it reads in from a library directory file to resolve instances in subsequent library directory files.

Combining Library Management Plus Options

All predefined plus options are compatible with both methods of reading in the contents of library directory files. You can use the default method or +libnonamehide with +librescan, +libext, or +liborder.

- If you include both the +libnonamehide and the +librescan options on the command line then when Verilog-XL finds an unresolved instance in a library directory file, it does not look first for a definition in the remainder of the library directory file. Instead, it begins to scan through the left-most library on the command line, and then scans the remaining libraries in the order that they appear.
- The +liborder plus option appends a second character string to a definition name if the character string appended by the default method creates a name already used by another module or UDP. This second character string begins with \$lib, followed by the position number of the file's library directory among the source files, library files, and library directories on the command line.

The following figure shows the contents of two directory files that have the same name but reside in different library directories. These files have the same definition names, so +liborder adds a character string to a definition name.



In the previous figure, libraries vendor1 and vendor2 both contain files named top.v, and both versions of top.v have a module named top followed by a module named bottom.

Consider the following command line:

verilog source1.v -y vendor1 source2.v -y vendor2 +liborder +libext+.v

Verilog-XL records the definition of module bottom in library vendor1 as bottom\$top, and it reads in the definition name of module bottom in library vendor2 as bottom\$top\$lib4, because vendor2 is in the fourth position out of all the source files, libraries, and library directories on the command line.

Use of Compiler Directives with Libraries: The Former Scheme

Because the effects of Verilog-XL compiler directives span source text files, you need to consider their use in conjunction with libraries.

'resetall

The `resetall compiler directive should be placed at the beginning of each file followed by any compiler directives that you want to be active for the contents of that file. If you omit this directive, any compiler directives that are active for the previous source file are also active for the next source file, and Verilog-XL may apply unwanted directives to the contents of your source files.

'protected and 'unprotected

The boundaries of the protected region determine whether Verilog-XL can find a module in a library to resolve an undefined instance. If the `protected directive appears after the module name, and the `unprotected directive appears before the endmodule keyword, Verilog-XL can find the module definition. If these directives appear before the module name and after the endmodule keyword, Verilog-XL cannot find the module definition.

If you compile your library module with the +autoprotect option, Verilog-XL automatically enters the `protected directive after the module name, and the `unprotected directive before the endmodule keyword. This is the recommended format for library scanning.

Efficiency Considerations of Library Usage: The Former Scheme

There are two ways to maximize the efficiency of the library management feature:

- Do not specify libraries that are not needed.
- Construct library files so that all instances referenced within the library are forward references (that is, with the reference before the definition).

The Library.Cell:View Library Management Scheme

To use the Library.Cell:View scheme, your entire design must be organized in the Cadence Application Infrastructure (CAI). The components of this architecture are described as follows:

The library is a collection of related cells that describe components of a single design (a design library) or common components used in many designs (a reference library). Each library is referenced by a logical name and has a unique physical directory associated with it. You define library names and map them to directories in the cds.lib file. The library used for your current design work is called the working or work library.

- A cell is an object with a unique name stored in a library. Each module, macromodule, or UDP is a unique cell. Each cell within a library is a separate file system directory. Every unique element of a design is its own cell and, therefore, has its own cell directory.
- Views can be used to delineate between representations (schematic, VHDL, Verilog), abstraction levels (behavior, RTL, postsynthesis), status (experimental, released, golden), and so on. Each view within a cell is a separate file system directory in which Verilog-XL locates all of the files pertaining to a particular representation of a given design element. For example, one view directory might contain the RTL representation of a particular module, while the behavioral representation is stored in another view directory.
- A configuration is a set of search rules used to determine a binding of an instance to a module. For more information, see <u>"CAI Configurations"</u> on page 125

Directory Structure Example

In this example, a module mychip instantiates two other modules, m1 and m2.



A single Verilog description of mychip is contained in the file mychip.v, but you have generated multiple descriptions of m1 and m2, as follows:

■ For m1, you generate a behavioral description (m1.vb) and an RTL description (m1.vr).

■ For m2, you generate RTL description (m2.vr) and a synthesized gate-level representation (m2.vg).

Cell	View Type	Source Files
mychip	Structural	mychip.v
m1	Behavioral	m1.vb
	RTL	m1.vr
m2	RTL	m2.vr
	Gates	m2.vg

All of these source files reside in the src/subdirectory, from which all Verilog-XL tools are invoked. You create a subdirectory (worklib/) at the same level as src/, which you want to use as the work library.



The following CAI directory structure was previously created by a CAI-compliant tool:

The cds.lib file is located in the current directory (src/), and includes the following statement, which defines a library called worklib:

DEFINE worklib ../worklib

Verilog-XL Notes for CAI

Note the following:

If you code the instances as in Example 1, you will not have separate bindings for inst1 and inst2. Example 2 shows how to code the description to have different bindings for inst1 and inst2.

```
Example 1: Example 2:

module test1;

adder inst1 (a, b,c)

inst2 (a,b,c)

endmodule Example 2:

module test2;

adder inst1 (a, b,c)

endmodule endmodule
```

If another module of the same name already exists, the name of a module is changed by prepending the library name, and appending the module type and a unique identifier. For example, LIB1\$adder\$RTL_inst1_1.

All elements of an array of instances are bound to the same cellview. You cannot bind a separate element of an array.

CAI Configurations

A CAI configuration specifies design units by defining a set of search rules which are used to bind an instance to a module. The default configuration filename is expand.cfg. You can create a configuration file with the Hierarchy Editor. For more information about the Hierarchy Editor, see the *Hierarchy Editor User Guide*. The syntax for a configuration is as follows:

```
<config declaration>
            config <config_id>;
     ::=
           design <library.cell:view>;
                  {source <library.cell:view> { "<string>"};}
                  {const <constname> {<viewlist>};}
                  {<config rule statement>}
             endconfig
     <config rule statement>
         ||= stoplist <viewlist>;
         ||= lib <libname> <viewlist> {<expansion clause>};
         ||= cell <library.cell> {<expansion clause>};
         ||= inst (<cellname>) {<expansion_clause>};
     <expansion clause>
         ::=
                liblist <liblist>
                viewlist <viewlist>
         | | =
         | | =
                binding <library.cell:view>
     <viewlist>
         ::= <viewname>{,<viewname>+}
     <liblist>
         ::= <libname>{,<libname>+}
     <cellname>
         ::= (<library.cell:view>).<instname>{[<integer>{:<integer>}]}
```

The following configuration file shows how to define search rules and bind instances to modules. The comments in the code refer to the numbered list following the example.

```
config MUX-Gamma;
                                                     // 1
   design WORKLIB.MUX:rtl
                                                    // 2
    source apple.modulator:config "VHDL"
                                                  // 3
    const MyLibs my1, my2, my3;
                                                 // 4
                                                // 5
    liblist L1, L2;
   viewlist $MYLibs, module, gate;
                                               //6
    stoplist state, gates;
                                              // 7
    lib view1, view4, view2;
                                             // 8
    cell L2.path2:MUX-Gamma viewlist MUX1, MUX2;
                                                         // 9
                                                        // 10
    inst (WORKLIB.adder:hdl) U1 viewlist schematic;
                                                         // 11
    inst (mpeg).i2 binding syslib.alu:rtl;
endconfig; # MUX-Gamma
```

1. Identifies the configuration with the name MUX-Gamma.

- 2. Names the library, cell, and view of the top-level module in the design hierarchy.
- 3. Identifies the location of an automatically created configuration.
- 4. Defines a variable called MyLibs with a search precedence of three libraries.
- **5.** Defines the global library search precedence (L1, then L2) of a configuration when it is automatically created from another form.
- 6. Defines the global view search precedence (my1, my2, my3, module, gate). (Note the use of the const variable MyLibs.)
- 7. Defines a list of views that prevent further expansion when a view is selected.
- 8. Defines a specific view list precedence.
- 9. Names the cell to which the view list applies.
- **10.** Limits the search for the specified instance to the U1 library and the schematic view.
- **11.** Binds an instance to a specific library.cell:view.

Specifying a CAI Simulation

To simulate a design that is in the CAI architecture, use the following plus options on the command line:

<pre>+config+<lib.cell:view></lib.cell:view></pre>	Specifies the top-level CAI configuration. You can specify multiple +config+ plus options.
+cdslib+ <filepath></filepath>	Specifies the cds.lib file. By default, Verilog-XL looks for a cds.lib file in the current working directory.
+cellview+ <lib.cell:view></lib.cell:view>	Specifies a CAI cell view as a supporting top-level module for the actual top-level module that is specified in the CAI configuration. You can specify more than one +cellview+ plus option.

Note: You cannot specify the -v or -y command-line options, or use the `uselib system task, when simulating with the CAI architecture.

On the command line, you can specify verilog files that have top modules:

- if the files do not have any instances in them.
- if the files have instances of top-level design specified in the CAI configuration.

For more information about the Cadence Application Infrastructure, see the *Cadence Application Infrastructure User Guide*.

Accessing Libraries

You can avoid compiling all the modules in a source text file when you need only specific modules by accessing the modules as libraries. Using libraries reduces compilation time and memory usage. Specify the `uselib compiler option to access libraries from your design. The following example shows you how the `uselib compiler option lets you access libraries from your design. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Managing libraries

```
// See Step 1
'define LIB_ROOT /net/lib
`define TTL_LIB dir=`LIB_ROOT/TTL_LIB/source libext=.v
'define TTL UDP file='LIB ROOT/TTL LIB/udp.v
'define FAST_LIB dir='LIB_ROOT/FAST/source libext=.v
module board (in,out,bus);
    input [142:0] in;
    output [39:0] out;
    inout [127:0] bus;
'uselib 'TTL_LIB
                                               // See Step 2
    SN7400 U1 (clear,in[0],int_clear);
                                              // See Step 3
    SN7404 U2 (clear bar,clear);
'uselib 'TTL_LIB 'TTL_UDP
    SN7474 U3 (slave,,data0,set_bar,
                    clear_bar);
'uselib 'FAST_LIB
    SN7400 U4 (out[3],data0,
                    data1);
`uselib
                                             // See Step 4
    endmodule // board
```

- 1. To eliminate excessive typing and to make your source file more readable, define macros for your libraries. This module accesses three libraries: two library directories (dir) and a file that defines a UDP (file). Specify the extension of library files with the libert keyword.
- 2. Before referencing a library module, specify the library path with the `uselib compiler directive. You can specify any number of `uselib compiler directives in your module. You can combine all three `uselib directives into a single `uselib that specifies all the library paths, but Verilog-XL's search efficiency increases when you are precise about where to locate libraries.

- **3.** Reference library components as though they are defined in your module. Verilog-XL searches the active library search path, specified by the TTL_LIB macro, to resolve the SN7400 and SN7404 definitions.
- 4. To force Verilog-XL to look at the command line to configure a library search, or to keep subsequent modules from using previous library paths, specify an empty `uselib directive.

Integrating PLI and VPI Routines

This chapter describes the following:

- <u>Overview</u> on page 129
- <u>Using PLI or VPI</u> on page 131
- Error Handling on page 137
- <u>Debugging</u> on page 137

Overview

PLI and VPI are application program interfaces (APIs) to your simulator environment. PLI and VPI routines let you write applications that create new simulator system tasks and manipulate instantiated simulation objects contained in a Verilog HDL design. You integrate your PLI or VPI application with the simulator to create a customized version of the simulator that contains the new simulator system tasks.

PLI and VPI consist of a set of access and utility routines that you call from standard C programming language functions. The PLI or VPI routines interact with instantiated simulation objects created in Verilog HDL designs. Instantiated simulation objects are individually accessible instances of low-level modules that are contained in higher-level modules.

For example, if a module named flipflop contains wire q and is instantiated in module shifter as ff1 and ff2, then shifter.ff1.q and shifter.ff2.q are two distinct objects, each with its own set of accessible values and properties. Using the PLI or VPI access routines, you can get information from a design about each of these separate instances.

Some applications of PLI include:

- Customized debugging routines
- Delay calculator routines
- Annotation routines

The PLI or VPI mechanism reads and writes to the internal data structures of Verilog-XL.

PLI and VPI routines operate only on instantiated internal simulation objects; they do not directly access the Verilog source description netlist.

This chapter aims to provide an overview of PLI and VPI mechanisms. Please refer the following documents for details:

- PLI 1.0 User Guide and Reference
- <u>VPI User Guide and Reference</u>

The Components

What Cadence Provides

Cadence Design Systems provides the following components that you use to create PLI or VPI applications:

A set of access routines

Access routines are called from your programs to read and write information about Verilog HDL objects by accessing internal Veritool data structures. Access routines do not interact directly with the HDL netlist, but access internal data structures only.

A set of utility routines

These routines are called from your programs to pass data to and from the simulation environment and to provide timing synchronization between system tasks or functions.

■ The veriuser.c file

A source file named veriuser.c that contains the task registration array. This file is used to associate user-defined routines as new system tasks and functions. This file is required for:

- □ Static linking of VPI applications
- Linking of applications that use both VPI and PLI 1.0 routines
- PLI specific files

Four header files named veriuser.h, acc_user.h, vxl_veriuser.h, and vxl_acc_user.h. The files veriuser.h and acc_user.h contain the IEEEcompliant definitions. The files vxl_veriuser.h and vxl_acc_user.h contain the Cadence-specific definitions. ■ VPI specific files

A file named vpi_user.c that contains an empty startup array that you must edit to add your VPI function registration calls. You compile and link this file with your application.

Two header files named vpi_user.h and vpi_user_cds.h. The file vpi_user.h contains the function declarations as well as the constant and data structure definitions for all of the IEEE 1364 VPI routines.

The file $vpi_user_cds.h$ contains the macros that Cadence provides in addition to the IEEE 1364 specification.

The Verilog-XL simulator includes the vconfig utility, the Verilog-XL configuration utility, which you use to compile and link the Verilog-XL object modules and your PLI or VPI application.

What You Provide

You must provide the following components to create a PLI or VPI application to work with Verilog-XL simulator:

- A compiler and a linker for the C or C++ programming language
- A C or C++ language application that calls the PLI or VPI routines
- Code that registers your system tasks by defining the association between your new system tasks and your C or C++ functions.
- Code that initializes your system tasks/functions
- Your Verilog HDL design, which invokes your new system tasks by calling them from a procedural block.

Note: You can also call your system tasks from the command line during simulation.

Using PLI or VPI

To use PLI or VPI, you create an application in C or C++, which calls the PLI or VPI routines that manipulate your simulation objects. You associate each application function with a new system task that you register with the simulator. Then, you call your application functions by calling the corresponding system tasks from either your Verilog HDL design or the simulator interactive prompt.

This process is outlined in the following steps:

- **1.** Create C or C++ functions that contain the calls to PLI or VPI routines.
- 2. Associate each C or C++ function with a simulator system task.
- **3.** Register your system tasks so that the simulator recognizes them.
- 4. Integrate your C or C++ application either dynamically or statically with the simulator.
- 5. Call any of your C or C++ functions by calling the corresponding system task from either an initial block of your Verilog HDL design, or the simulator interactive prompt.

Figure 7-1 on page 133 shows the flow of the steps you perform to use PLI or VPI.

Figure 7-1 Using PLI or VPI with the Verilog-XL Task flow



Creating a C or C++ Routine

You use C or C++ language to write your PLI or VPI application. Your C or C++ functions call the PLI or VPI routines, which are implemented in C. You can place your functions in one or more source files.

When you write your C or C++ application, make sure you include all the standard include files, such as <stdio.h>, that your application needs at the top of all your source files.

Integrating C++ Routines

You can write your PLI application in either C or C++. However, because Verilog-XL is written in C, you must address some integration issues before you can integrate your C++ application with the simulator.

When you write a C++ application, the main program must be a C++ compiled routine to ensure correct initialization of memory management (constructors/destructors), file I/O, and so on.

However, the Verilog-XL main() routine that controls execution is a C routine. It calls the function $vlog_main()$ and passes it the argc and argv arguments that were passed to main(). The $vlog_main()$ routine is defined in the libvoids.a library as follows:

```
extern void vlog_main(int, char **);
main (argc, argv)
int argc;
char *argv[];
{
 vlog_main(argc,argv);
}
```

To integrate your C++ application with Verilog-XL, which is controlled by a C main() routine, you must override the C main() routine with a C++ main() routine. To do so:

1. In your application, include a C++ main() routine, defined as follows:

```
extern "C" void vlog_main(int, char **);
main (argc, argv)
int argc;
char *argv[];
{
vlog_main(argc,argv);
}
```

- 2. Run the vconfig utility to create the cr_vlog shell script.
- **3.** Edit the cr_vlog shell script before you run it.

Change the default C compiler to your C++ compiler.

4. Run the cr_vlog script to compile and link the input files into a new Verilog-XL executable.

The new C++ main() allows C++ to initialize properly when the executables are started and then calls the Verilog-XL main program $vlog_main()$. You now can use all C++ features in your application.

Associating a C or C++ Routine with a System Task

When writing your C or C++ source files, you call PLI or VPI routines from C language functions. To use your new C or C++ routines, you must associate them with corresponding HDL system task names.

To associate your C or C++ routine name with a system task name, you edit the definition of the veriusertfs array, provided by Cadence in the veriuser.c file. Part of the information you enter in the array is the new system task name and the name of your C or C++ routine that maps to the system task. Because of this routine-to-system task mapping, you can implicitly call your PLI or VPI application routine by calling the corresponding system task from an HDL source description.

Integrating Your Application with the Simulator

After you write your PLI application, you have to integrate it with the simulator. Your application code must be compiled and linked so that it is recognizable to the simulator. The integration enables the simulator to recognize and execute your application functions.

You can integrate your application in one of two ways:

- Dynamic linking
 - You compile and link your application into a shared library that the simulator accesses at run time to find your application functions.
 - □ The simulator executables are provided. When you make changes to your application, you only rebuild the library.

Dynamic linking is useful when you want your application to be configurable and flexible. You can use your dynamic shared library with Verilog-XL. In addition, you can change your path variable to point to different versions of a shared library, so you can run your simulation with different configurations. Dynamic linking builds faster than static linking.

- Static linking
 - You compile and link your application with simulator object modules to create a new set of executables that include your application functions.
 - Every time you change your application, you must rebuild the executables.

Static linking is useful when you want your application to run simply. Because all the information is built into your custom executables, you do not need to set any environment variables to run your application. A statically linked VPI application runs faster than one linked dynamically.

Invoking Your System Tasks

- You can call your new system tasks in any of the following ways during the simulation of your Verilog HDL design:
 - From an initial block in your Verilog HDL design.

For example, the following module calls the new system task θ when the simulator simulates module top:

endmodule

□ From the simulator interactive prompt.

For example, if your new system task is \$hello from the preceding example, you can run the system task by typing the task name, followed by a semicolon:

c4> \$hello;

For more information on the Verilog-XL simulator interactive prompt, see <u>Appendix B, "Interactive Control and Debugging"</u>.

If your PLI or VPI application is dynamically linked to the simulator, you run the simulator in the normal way, by typing:

verilog hdl.v

If your PLI or VPI application is statically linked to the simulator, you call your new customized executable with your source description file. For example, if you have built a new Verilog-XL executable called my_vlog, you would run it with the hdl.v source file from the preceding example by typing:

```
% my_vlog hdl.v
```

Calling a System Task from the Verilog-XL Interactive Prompt

Because your task is built into the simulator, you can call it from the Verilog-XL command prompt, as follows:

- **1.** Invoke the SimVision user interface:
 - > my_vlog +gui -s hdl.v

You can enter commands at the interactive prompt in the Simulator window.

2. Invoke the \$hello system task by entering \$hello; at the interactive prompt in the Simulator window.

Error Handling

When you simulate your Verilog HDL design with a simulator that has been integrated with your PLI or VPI application, errors can occur during the execution of any PLI or VPI access or utility routine. For example, if the $vpi_printf()$ utility routine attempts to store information in a log file but the system's disk space is exhausted, the simulator generates an error.

When a PLI access routine detects an error, it performs these functions:

- Sets the global error flag acc_error_flag to non-zero
- Displays an error message at run time to standard output
- Returns an exception value

VPI routines do not return values that indicate success or failure. Moreover, when you call a VPI routine, Verilog-XL clears the error information from the previous PLI or VPI routine call. Therefore, to catch all errors, you must trap them after they occur by checking the error status after each VPI routine call. You can do this by:

- Calling the vpi_chk_error() routine after each VPI routine call to check whether the VPI routine has caused an error and to get information (optional) about that error.
- Creating an error-handling wrapper routine for each VPI routine. This wrapper routine contains the direct VPI routine call and the error-checking code after the call. Every time you want to call a VPI routine, you call its wrapper routine instead.
- Forcing VPI to display messages about all errors that occur during the execution of VPI routines. To do this, you run the Verilog-XL executable with the +accwarn flag.

Debugging

You can use the same tools that you use for other C or C++ language applications to write and debug your PLI or VPI applications. When you encounter errors in your PLI or VPI applications, you can use your normal debugger to assist you in resolving problems. The dbx or gdb source-level debuggers are examples of tools that you can use to debug your applications. These debuggers assist you in:

- Program execution and tracing
- Setting and clearing breakpoints
- Accessing and displaying data
- Accessing and displaying source files

Switch-Level Simulation

This chapter describes the following:

- <u>Overview</u> on page 139
- Definition of Switch-Level Networks on page 139
- <u>Major Features of the XL Algorithms</u> on page 140
- <u>Choosing an Algorithm</u> on page 142
- Enabling the Algorithms on page 143
- <u>How the Default Algorithm Works</u> on page 144
- <u>How the Switch-XL Algorithm Works</u> on page 148
- <u>Switch-XL Strength Model</u> on page 157
- Delays in Default and Switch-XL Bidirectional Networks on page 162

Overview

For both cmos and nmos circuits, modeling at the switch level better represents the hardware implementation of a design than modeling at the gate level. This chapter discusses switch-level models that represent the behavior of field effect transistors. Verilog-XL lets you simulate switch-level networks with a default algorithm or with the Switch-XL algorithm.

Definition of Switch-Level Networks

This section describes the switch-level networks that the Verilog-XL and Switch-XL algorithms simulate.

There are two types of Verilog HDL switches: unidirectional and bidirectional.

The following primitives are bidirectional switches:

tran tranif1 tranif0 rtran rtranif1 tranif0

The following primitives are unidirectional switches:

nmos pmos cmos rnmos rpmos rcmos pullup pulldown

Switch-level networks are composed of bidirectional switches, unidirectional switches, or both. The default and Switch-XL algorithms simulate regions of channel-connected switches, which are defined as one of the following:

- a single switch and the nets that connect to its source and drain terminals
- a group of switches connected together through their source and drain terminals, and the nets connected to the switches' source and drain terminals

The following figure shows three channel-connected switch networks, each delineated by a dashed rectangle:



Major Features of the XL Algorithms

This section introduces and compares the features of the default and Switch-XL algorithms. The following section contains a detailed discussion of the default algorithm. <u>"How the</u>

<u>Switch-XL Algorithm Works</u>" on page 148 then begins the detailed discussion of the Switch-XL algorithm.

The following table summarizes the features of the two XL simulation algorithms. This chapter refers to the system of signal strengths available in the default Verilog HDL gate and switch declarations as the standard-strength model.

Feature	Default Algorithm	Switch-XL Algorithm
Strength Model	Standard Strength	Standard-strength with exceptions Also 256 strength model
Switch Strength	None - Maintain or drop by one	Integer strength
Net Strength	None, small, medium and large	None, small, medium and large Also integer strength
Delay Calculation	User-specified	User-specified
Spikes	Not handled	Not handled

The Default Algorithm

The default algorithm simulates only bidirectional switches. When the default algorithm is in use, both the XL algorithm and the non-XL algorithm simulate unidirectional switches.

The default algorithm shows normal Verilog-XL behaviors in the following areas: timing models of switches, strength reduction by switches, accessibility of strength information, and force and release statements.

In the default algorithm, the large charge strength exceeds the weak drive strength.

The Switch-XL Algorithm

Verilog-XL allows you to optionally use the Switch-XL algorithm, which provides you with the following features:

- High-speed simulation of bidirectional switches
- Two strength models
 - A modification of the standard-strength model

An integer strength model that allows you to specify a wide range of trireg capacitances and switch conductances

The XL algorithm accelerates the simulation of gates and unidirectional switches. By default, the Switch-XL algorithm simulates every channel-connected network that includes at least one bidirectional switch. A command-line option can make the Switch-XL algorithm also simulate networks composed solely of unidirectionals.

In simulating a network composed solely of unidirectionals, the Switch-XL algorithm follows the unidirectionals' normal channel-delay timing model. In simulating a network that includes a bidirectional, the Switch-XL algorithm models the channel delays of the unidirectionals in the network as turn-on/turn-off delays.

Simulating with the Switch-XL algorithm does not preserve strength information for access with system tasks. Therefore simulation with the Switch-XL algorithm that does not employ the Switch-XL integer strength algorithm has a standard-strength model that is modified in the following ways:

- Drive strengths are always greater than charge strengths, so a weak drive strength is greater than a large charge strength.
- If a signal passes through a channel-connected series of bidirectional resistive devices that all reduce strength in the standard-strength model, then the signal's strength is reduced only once, by the highest resistance switch.

Simulation with the Switch-XL algorithm prevents you from changing net values with the force and release procedural continuous assignments. The Switch-XL algorithm treats a wand or a wor as a wire.

Choosing an Algorithm

If you are interested in performing a functional simulation specifying discrete delays, you must choose between the default and Switch-XL algorithms. If the standard-strength model cannot simulate your circuit, Switch-XL is your only option since it has an alternate integer-strength model.

The difficult decision is whether to use the default algorithm or the Switch-XL algorithm when the standard-strength model can simulate the circuit. The choice is dependent upon the topology of the circuit. If the circuit contains many gates implemented with bidirectionals, Switch-XL outperforms the default algorithm by a factor of five to fifteen. If the circuit is a densely connected pass transistor circuit such as a RAM or a barrel shifter, then Switch-XL requires more memory and is slower than the default algorithm by a factor as great as three to five.

Enabling the Algorithms

You can enable the algorithms either globally for the whole design or locally for individual channel-connected regions.

Enabling the Algorithms Globally

To enable the default algorithm globally, do not invoke Switch-XL using either the plus options discussed in this section or the compiler directives discussed in the next section. To enable the Switch-XL algorithm, enter the +switchxl plus option on the command line as shown:

verilog source.v +switchxl

Invoking the Switch-XL algorithm globally overrides all conflicting `switch compiler directives in your source files. The next section discusses the `switch compiler directive.

Enabling the Algorithms Locally

The `switch compiler directive allows you to choose a switch-level algorithm for a subset of switches.

The `switch compiler directive has the following syntax:

```
<switch_directive>

::= `switch <algorithm_specification>

<algorithm_specification>

::= default

||= XL

<technology_name>

<IDENTIFIER>
```

The <algorithm_specification> argument specifies the algorithm that simulates the switch primitives that follow it until the compiler encounters another `switch compiler directive that calls for a different algorithm. The effects of the `switch compiler directives cross module boundaries.

Note: When you enable algorithms locally, only one algorithm must model all the switches in a channel-connected network. Attempting to model different switches in a channel-connected network with different algorithms forces the software to choose an algorithm to model all the switches in the network based on the algorithms you attempt to apply. In this case, Verilog-XL tries to use the Switch-XL algorithm before using the default algorithm.

A channel-connected network consisting of a mix of unidirectional and bidirectional components can create conflicts due to different requirements of each algorithm. Switch-XL

does not model networks consisting solely of unidirectionals unless you enter a commandline option. The default algorithm models only bidirectional switches. Verilog-XL attempts to reconcile these requirements, and issues a warning if it perceives a conflict.

How the Default Algorithm Works

This section discusses the factors that make bidirectional network simulations that use the default algorithm different from other Verilog-XL simulations.

Forcing and Releasing Nets in Bidirectional Networks

The strength of the value on a forced net is strong. The forced value on a net takes precedence over any value that the default algorithm calculates for that net. Forcing a value on each path from one part of a channel-connected region to another part of that channel-connected region effectively divides the channel-connected region into two pieces. The default algorithm simulates each of these two pieces separately, as the following figure shows:



No nets are forced in the channel connected switches above, and all the switches are on. Forcing net o2 to a strong 0 value creates the results below.



Initially, no nets are forced in the channel-connected switches in the previous figure, and all the switches are on. vcc has a strong 1 value, and all the other nets take on the value of vcc.
Forcing net $\circ 2$ to a strong 0 value divides the network into two networks. One network involves the path from $\circ 2$ to $\circ 1$ to vcc. The other network involves the path from $\circ 2$ to $\circ 3$ to $\circ 4$. Net $\circ 1$ in the first network has a strong x signal as a result of the combination of two strong signals with values 0 and 1. Nets $\circ 3$ and $\circ 4$ in the second network take the value forced onto $\circ 2$; they are unaffected by the nets in the first network.

With the exception of triregs in bidirectional networks, releasing nets from force statements has the following effect: it allows the signals on the nets to regain the values and the strengths that they had before the force statement, unless stronger signals prevail.

When a trireg connected to a data terminal of a bidirectional switch is released from a force, the signal on the trireg becomes the capacitive equivalent of the forced state, unless a stronger signal prevails. The declaration of the trireg thus determines the strength of its signal.

Wired Logic in Bidirectional Networks

The use of wired logic (wand, wor, triand, trior) in bidirectional networks is discouraged because it does not model any type of hardware, and because it is prone to hidden race conditions.

Under limited conditions, the default algorithm gives correct logic results within a channel-connected bidirectional switch network.

The following conditions must exist for the simulator to model wired logic:

- The network does not contain a mix of net types.
- No gate or force statement injects a signal with a value of x into the network.

A warning appears if you specify wired logic in bidirectional networks.

Verilog-XL User Guide

Switch-Level Simulation

The following example demonstrates a simulation and its results, including the warnings that locate the cases of wired logic in the bidirectional networks according to the file and the name of the net.



The results of the preceding simulation follow:

```
Warning! Wired logic on node top.pl in bidirectional network may lead to
inconsistent simulation results[Verilog-WLBN]
        "temp.v" 2: pl
Warning! Wired logic on node top.p2 in bidirectional network may lead to
inconsistent simulation results [Verilog-WLBN]
        "temp.v", 3: p2
            0 ol=x p1=0 o2=x p2=0
```

2 Warnings

The values on the nets in bidirectional networks that are simulated in the presence of wired logic can demonstrate an anomaly. The simulator normally determines the signal on a net by referring to the elements to which the net connects. In determining the signal on a net in a channel-connected bidirectional network, the default algorithm identifies all paths to the net originating at voltage sources and resolves the signals propagated through those paths.

The following figure demonstrates the results of wired logic in channel-connected bidirectional networks:



The wire nets in both cases in this figure have a signal with a value of x, because their signals are the results of combining equal strength signals with values of 1 and 0. The wand nets in both cases have values of 0 because their signals are the result of ANDing the values of 1 and 0 that originate in the supply1 vcc and the supply0 gnd.

Reporting on Bidirectional Networks with \$showvars

The \$showvars system task produces status information for register and net variables in different ways, depending on whether they are inside or outside of bidirectional networks. The \$showvars task typically presents the following information:

- the value for the register or net at the previous time unit
- the drivers of the register or net in the previous time unit
- the currently scheduled value of the register or net
- the future values propagated from the drivers for the net or register, if scheduled

The \$showvars task shows when Verilog-XL schedules the values of nets and their drivers in bidirectional networks, without giving any information about the currently scheduled values of the nets or the drivers.

The following example demonstrates how \$showvars reports on nets in bidirectional networks:

```
module top;
    req a;
    supply1 vcc;
    tranif1 (trandata, vcc, a);
    nmos (nmosdata, vcc, a);
    initial
        begin
            #10 a = 1; $showvars(trandata, nmosdata);
        end
    endmodule
// Simulating the model above produces the following results:
// $showvars does not report the scheduled value of nets in
// bidirectional networks.
trandata (top) wire = StH
    StH <-> (top): tranif1 (trandata, vcc, a);
        scheduled
// $showvars reports the scheduled value of nets outside
// bidirectional networks
nmosdata (top) wire = StH
    StH <- (top): nmos (nmosdata, vcc, a);</pre>
        scheduled = St1
```

The <code>\$showvars</code> task reports that the unidirectional nmos is scheduled to have a value of <code>St1</code>, but <code>\$showvars</code> reports only that the bidirectional <code>tranif1</code> is scheduled, without any mention of a value.

How the Switch-XL Algorithm Works

The Switch-XL algorithm performs the following steps:

- 1. Searches your source description for channel-connected switch networks.
- 2. Converts the timing model of the unidirectional switches in channel-connected switch networks from channel delay to turn-on/turn-off delay.
- 3. Optimizes the contents of channel-connected switch networks.
- 4. Compiles channel-connected switch networks into equations that the XL algorithm can simulate.
- 5. Enables the XL algorithm to perform the simulation.

When Switch-XL compiles channel-connected switch networks into equations, you can no longer display accurate strength information about their nets.

The following sections describe the conversion of the timing model, the optimization of your source description, and the invisibility of strength values in channel-connected switch networks when using the Switch-XL algorithm.

Conversion of Channel Delay to Turn-On/Turn-Off Delay

Verilog-XL uses the following two delay-timing models for switches:

- Channel delay
- Turn-on/turn-off

The channel-delay timing model

Verilog-XL uses the channel-delay timing model for unidirectional switches that are not accelerated by the Switch-XL algorithm. A channel delay specifies an interval of simulation time for a value change on the source terminal to propagate to the drain terminal. A transition on the gate terminal enables or disables the propagation of a value from the source terminal to the drain terminal.

The turn-on/turn-off delay timing model

A turn-on/turn-off delay specifies an interval of simulation time between a transition on the gate terminal and the enabling or disabling of the propagation of values between the source and the drain terminal. In switches that use only the turn-on/turn-off delay timing model, no simulation time elapses in the propagation of values between the source and the drain terminals.

How Verilog-XL uses delay timing models for switches

Verilog-XL employs the turn-on/turn-off delay timing model for bidirectional switches. You can specify one or two delays for bidirectional switches. If you specify one delay, that delay applies to both the turn-on and turn-off times. If you specify two delays, the first is the turn-on time, the second is the turn-off time. The following example shows the declaration of a bidirectional switch with both a turn-on and a turn-off delay time. 5 is the turn-on delay; 6 is the turn-off delay.

tranif1 #(5,6) tr1_1 (d,s,g);

Verilog-XL employs no timing model for the pullup and pulldown unidirectional switches; you cannot specify delays for these switches.

Verilog-XL employs a combination of both timing models for the MOS switches (nmos, pmos, cmos, rnmos, rpmos, and rcmos). In MOS switches, you can specify up to three delays. If you specify three delays, the first is the rise delay, the second is the fall delay, and the third is the turn-off delay. You cannot specify a turn-on delay for a MOS switch.

The following example shows the declaration of a MOS switch with three delays. 5 is the rise delay, 6 is the fall delay, and 7 is the turn-off delay. 5 and 6 are the channel delays.

nmos #(5,6,7) nm_1 (d,s,g);

If you specify only two delays for a MOS switch, Verilog-XL interprets them as rise and fall delays and uses the smaller of the two delays as the turn-off delay. If you specify one delay, then the rise, fall, and turn-off delays all have the same value.

The timing model conversion of unidirectional switches

If Switch-XL finds a bidirectional switch in a channel-connected switch network, it forces all of the unidirectional switches in a channel-connected switch network that use both channel delay and turn-on/turn-off delay timing models to use turn-on/turn-off delay timing model exclusively so that all switches in the channel-connected network have the same delay timing model. These unidirectional switches remain unidirectional, but their delay-timing models change.

The following figure shows the unidirectional switches whose delay-timing models Switch-XL converts to use the turn-on/turn-off delay-timing model exclusively:



If you specify a rise, fall, and turn-off delay for a MOS switch, Switch-XL ignores the larger of the rise and fall delays and uses the smaller delay as the turn-on delay.

The following example shows a MOS switch with three delays whose delay-timing model the Switch-XL converts to use the turn-on/turn-off delay-timing model exclusively:

nmos #(5,6,7) nm_1 (d,s,g);

In this example

- Switch-XL converts the smaller channel delay (5) to the turn-on delay.
- Switch-XL ignores the larger channel delay (6).
- 7 is the turn-off delay.

If you specify only a rise and fall delay for a MOS switch when Switch-XL converts its delaytiming model to use the turn-on/turn-off delay-timing model exclusively, then the rise and fall delays become the turn-on and turn-off delays. The following example shows a MOS switch with only a rise and a fall delay whose delay-timing model Switch-XL converts to use the turn-on/turn-off delay model exclusively:

nmos #(5,6) nm_1 (d,s,g);

In this example

- Switch-XL converts the rise delay (5) to the turn-on delay.
- Switch-XL converts the fall delay (6) to the turn-off delay.

Using +sxl_unidirect

Switch-XL provides the +sxl_unidirect plus option to convert all the unidirectional switches in your source description to the turn-on/turn-off delay-timing model. Use this plus option when you want to enable Switch-XL and do not want unidirectional switches that use two kinds of delay-timing models.

Optimization of Switch Networks

During the compilation of channel-connected switch networks into equations, Switch-XL optimizes the channel-connected switch networks in your source description. Optimization saves memory and increases performance. During optimization, Switch-XL looks for opportunities to remove nets from equations when those nets do not help Switch-XL to describe the function of the network. You can control this optimization with command-line plus options.

This section describes the following aspects of Switch-XL optimization:

- removing nets
- the effect of removing a net on interactive commands
- using plus options to control optimization

Removing nets

The Switch-XL algorithm preserves a net under the following conditions:

- The net has a fanout.
- The net is an argument for a system task.
- Another part of the design description refers to the net.

■ The net is in a network that includes no nets that are subject to the three preceding conditions. This permits you to probe the nets with interactive monitoring tasks.

Otherwise, the Switch-XL algorithm removes the net.

The effect of removing a net on interactive commands

The removal of a net affects the results of interactively entered system tasks that display its value. When Switch-XL removes a net from an equation, Verilog-XL displays a warning and reports a strong strength and an x value for that net throughout the simulation. The following example shows a NAND gate network and its source description:



```
tranif0 left (vss,out,in1),
    right (vss,out,in2);
tranif1 sup (out,int,in1), // <-- Net named int
    inf (int,vdd,in2); // <-- Net named int
endmodule
```

Net int is in a channel-connected network. int does not meet the criteria for preservation, and its value is not monitored by a system task in the source description.

The Value Change Link and other PLI routines always return a strong strength and an x value for a removed net.

In the previous example, the net named int is between two bidirectional switches in a channel-connected network. This net is implicitly declared in the terminal connection lists for the tranif1 switches named sup and inf.

The following is a log file from a simulation *without Switch-XL* of the source description in the previous example. This simulation includes interactive entries of the *\$scope* and *\$showvars* system task.

```
Compiling source file "cmos_nand.v"
Highest level modules:
top
out=1
L11 "cmos_nand.v": $stop at simulation time 5
Type ? for help
C1 > $scope(cm1);
C2 > $showvars(int);
int (top.cml) wire = HiZ
   HiZ <-> (top.cm1): tranif1 inf(int, vdd, in2);
   HiZ <-> (top.cml): tranif1 sup(out, int, in1);
C3 > .
out=0
L14 "cmos_nand.v": $stop at simulation time 15
C3 > $showvars(int);
int (top.cm1) wire = St0
    St0 <-> (top.cm1): tranif1 inf(int, vdd, in2);
   St0 <-> (top.cml): tranif1 sup(out, int, in1);
C4 > .
L15 "cmos_nand.v": $finish at simulation time 15
43 simulation events
CPU time: 0 secs to compile + 0 secs to link + 0 secs in simulation
```

In this log file, the simulation includes the interactive entry of system tasks to report the value of net int. Verilog-XL reports first a value of z and then a value of 0 for net int.

The following is the log file from a simulation with Switch-XL of the same source description:

```
Compiling source file "cmos_nand.v"
Highest level modules:
top
out=1
L11 "cmos_nand.v": $stop at simulation time 5
```

Verilog-XL User Guide

Switch-Level Simulation

Type ? for help C1 > \$scope(cm1);C2 > \$showvars(int); Warning! Following node has been optimized by switchXL, value might not be accurate [Verilog SXLOPT] "cmos_nand.v", 26: int int (top.cm1) wire = StX StX <-> (top.cml): tranif1 inf(int, vdd, in2); StX <-> (top.cml): tranif1 sup(out, int, inl); C3 > out=0 L14 "cmos_nand.v": \$stop at simulation time 15 C3 > \$showvars(int); Warning! Following node has been optimized by switchXL, value might not be accurate [Verilog SXLOPT] "cmos_nand.v", 26: int int (top.cm1) wire = StX StX <-> (top.cml): tranif1 inf(int, vdd, in2); StX <-> (top.cml): tranif1 sup(out, int, in1); C4 > . L15 "cmos_nand.v": \$finish at simulation time 15 2 warnings 25 simulation events + 32 accelerated events CPU time: 0 secs to compile + 0 secs to link + 0 secs in simulation

Switch-XL removes int from the equations, so in this log file, Verilog-XL always reports a value of Strong X for net int.

Using Plus options to control optimization

You can control the extent to which Switch-XL optimizes your source description with plus options. These plus options, and their effect on how Switch-XL optimizes, are in the following table:

Plus Option	Affect on Optimization
+sxl_keep_all	Switch-XL performs no optimization. No nets are removed from the equations.
+sxl_keep_declared	Switch-XL does not remove explicitly declared nets. It removes implicitly declared nets if Verilog-XL does not need them for another purpose.
+sxl_keep_minimum	Switch-XL removes both the explicitly and the implicitly declared nets if Verilog-XL does not need them for another purpose.
	This plus option is an explicit method for specifying the default case.

The following figure shows the schematic and source description for a NAND gate with nets between bidirectional switches in a channel-connected network. The figure also shows the nets that Switch-XL does not remove when you enter above plus options.



Displaying Strength Values

When Switch-XL compiles channel-connected switch networks into equations, Verilog-XL can still report accurate logic values, but strength information is only partially available. Verilog-XL reports accurate charge strength, but it no longer reports accurate drive strength. You cannot display drive strength information about the nets in these networks with the \$showvars system task or with the %v format specification in system tasks such as \$monitor and \$strobe. If you enter any of these system tasks, you always see either a strong strength or the capacitive strength of a trireg.

Switch-XL Strength Model

In the Switch-XL algorithm, you can specify the capacitance of a trireg with a keyword for capacitance (small, medium, or large), and you can specify the conductance by selecting a resistive or non-resistive switch. You can also use a strength model syntax that applies only to Switch-XL simulations to specify wide ranges of relative capacitances or conductances with integers.

This section describes the Switch-XL strength model and illustrates the type of problem that this model solves.

Switch-XL Strength Model Example

The following figure shows a simulation problem that Switch-XL's wider range of drive strengths solves:



In this figure, switch a must have a lower conductance than switches b and c, but it must have a higher conductance than switch h. With Switch-XL, integers can specify more than two conductances for switches; without the Switch-XL integer specifications, there are two conductances, one for resistive switches and one for non-resistive switches.

Switch-XL Strength Model Syntax

The Switch-XL strength model allows you to specify both charge strengths and drive strengths with the strength keyword followed by an expression in parentheses. This expression specifies either a relative capacitance of a trireg or a relative conductance of a transistor as compared to other triregs or transistors in a channel-connected network.

Verilog-XL must be able to evaluate the expression to a constant value. If the expression evaluates to a real number, Verilog-XL truncates it to the nearest integer.

The following example shows the declaration statements for four triregs that use the Switch-XL strength model:

```
trireg strength(5) a;
trireg strength(3) b;
trireg strength(2) c;
trireg strength(1) d;
```

In this example, trireg a has the largest capacitance; trireg d has the smallest capacitance. These declarations *do not* specify that trireg a has five times the capacitance of trireg d; they only specify the relative capacitance of the triregs in the source description.

In trireg declarations, the expression must evaluate to a number from 0 to 250. A value of 0 specifies that the trireg stores no charge, which makes the declaration equivalent to a wire or tri.

The following example shows the declaration statements for three switches that use the Switch-XL strength model:

```
tranif1 strength(3) tla(s0,d0,g0);
tranif0 strength(2) t0a(s1,d1,g1);
rtran strength(1) rt1(s2,d2);
```

In this example, tranif1t1a has the largest conductance; rtran rt1 has the smallest conductance. These declarations *do not* specify that tranif1t1a has three times the conductance of rtran rt1; they only specify the relative conductance of the switches in the source description.

In switch declarations, the expression must evaluate to a number from 1 to 250. Unlike trireg declarations, the expression cannot evaluate to 0.

Note: The sum of the charge strengths plus the sum of the drive strengths in a channel-connected network cannot exceed 254.

Switch-XL Default Charge and Drive Strengths

The following table shows the default relative strengths that the Switch-XL strength model assigns when you omit the strengths from trireg and switch declarations.

Strength	Primitive
2	trireg
3	tran tranif1 tranif0 nmos pmos cmos
2	rtran rtran ifl rtranif0 pullup pulldown rnmos rpmos rcmos tril tri0

You can change the default strength of one of these primitives by entering the compiler directive that applies to the primitive followed by an integer that specifies the new default strength. These compiler directives, the integers that follow them and the primitives to which they apply are in the following table:

Compiler Directive	Valid Integers	Primitive
`default_trireg_strength	0 to 250	trireg
<pre>`default_switch_strength</pre>	1 to 250	tran tranifl tranif0 nmos pmos cmos
'default_rswitch_strength	1 to 250	rtran rtranifl rtranif0 pullup pulldown rnmos rpmos cmos tril tri0

The following example shows an entry of one of the above compiler directives:

`default_trireg_strength 1

In this example, the triregs in the modules that follow this compiler directive have a relative capacitance of 1.

Strength Reduction

To determine the path strength from an input to a net, Switch-XL uses the smallest conductance in the path. Without Switch-XL, Verilog-XL reduces the path strength by either one level or two levels for each resistive switch along the path.

This difference in strength reduction methods can produce different test results when you enable Switch-XL. The design and source description that follows results in different strengths and values with and without Switch-XL:



In this design, if you use Switch-XL, all three <code>rtranif1</code> switches have default conductance of 2. This is because Switch-XL resolves the conductance from d to f to be the same as the conductance from g to f. The display system task thus displays the logic value of wire f as x.

If you do not use Switch-XL, Verilog-XL reduces the strength from d to e from supply1 to pull1, and it reduces the strength from e to f from pull1 to weak1. Verilog-XL also reduces the strength from g to f from supply0 to pull0. The sdisplay system task thus displays the logic value of wire f as 0.

Strength Mapping

If your design is a mixture of switches and gates or triregs, it simulates using both a modified standard-strength model and the Switch-XL strength model. The following table shows how Switch-XL maps standard Verilog-XL strengths to Switch-XL integer strengths for each channel-connected network:

Supply	The greater of the following for the channel-connected network:
	(strongest conductance + strongest charge strength + 1) (strongest charge strength +4)
Strong	The greater of the following for the channel-connected network:
	(strongest conductance + strongest charge strength) (strongest charge strength +3)

Non-Switch-XL Sv	vitch-XL
------------------	----------

Verilog-XL User Guide Switch-Level Simulation

Non-Switch-XL	Switch-XL
Pull	Switch-XL calculates this value for the middle of the range of drive strengths:
	pull= ((strongest conductance + 1) / 2) + strongest charge
	If this value is not a whole number, Switch-XL truncates it to a whole number. If the result is less than:
	(strongest charge in the channel - connected network + 2)
	Then:
	pull=(strongest charge in channel - connected network + 2)
Weak	Strongest charge in the channel-connected network + 1.
Large	This is the strongest charge in the channel-connected network; if this value is less than 3, Switch-XL replaces it with 3.
medium	The algorithm calculates this value for the middle of the range of charge strengths in a channel-connected network:
	<pre>medium = (strongest charge + 1) / 2</pre>
	If this quotient is not a whole number, Switch-XL truncates it to a whole number. If the result is less than 2, Switch-XL replaces it with 2.
small	A charge strength value of 1
wire	A charge strength of 0

The following example describes how Switch-XL maps the non-Switch-XL strengths to Switch-XL strengths:



endmodule ..

In the previous example, the pull0 driving strength from buffer buf1 is mapped to a Switch-XL driving strength of 3 (dividing by 2 the sum of 5 plus 1). Turning on the tranif1 switches thus produces the following results:

- When you turn on tranif1 tr1, the value of net f is 0 because Switch-XL maps pull0 driving strength of buf1 to a conductance higher than that of tr1.
- When you turn on tranif1 tr2, the value of net f is x because Switch-XL maps pull0 driving strength of buf1 to a conductance equal to that of tr2.
- When you turn on tranif1 tr3, the value of net f is 1 because Switch-XL maps pull0 driving strength of buf1 to a conductance less than that of tr3.

Delays in Default and Switch-XL Bidirectional Networks

The Verilog-XL and Switch-XL algorithms simulate a channel-connected bidirectional network with net delays in a manner that can misrepresent the behavior of the circuit. This is because both algorithms schedule the appearance of value changes on nets by adding the delay on each net to the time at which the value change enters the network.

The following figure demonstrates how net delays work in channel-connected bidirectional networks:



time	ctl	n1	n2	n3
10	0->1			
12		z->1		
13				z->1
14			z->1	

The following figure is a schematic of Verilog-XL's model:



In the previous figure, at time 10, a value change from z to 1 enters the network from the left as switch ctl turns on the nmos. The value change appears on net n1 at time 12 because 10 plus n1's net delay of 2 equals 12. The value change appears on net n2 at time 14 because 10 plus n2's net delay of 4 equals 14. The value change appears on net n3 at time 13 because 10 plus n3's net delay of 3 equals 13.

Port collapsing can make these net delays unpredictable. This is because both algorithms use the net delay of the collapsed net.

Verilog-XL User Guide Switch-Level Simulation

Source Protection

This chapter describes the following:

- <u>Overview</u> on page 165
- Protecting Selected Regions in a Source Description on page 166
- <u>Protecting All Modules and UDPs in a Source Description</u> on page 168
- Effect of Source Protection on Simulation on page 170
- Effect of Source Protection on Library Use on page 173
- Effect of Source Protection on the Display of Hierarchical Path Names on page 173
- Error Reporting in Source-Protected Regions on page 176
- Loading Source-Protected Data into Memory on page 177

Overview

This chapter describes how to protect proprietary Verilog-XL source descriptions from being accessed or modified. The chapter also discusses how source protection affects the system tasks and interactive commands used during simulation.

Source protection can protect Verilog HDL descriptions simulated with the Verilog-XL simulator. Examples of proprietary designs that you may want to protect include ASIC cells and standard VLSI parts such as microprocessors.

There are two ways to protect your Verilog-XL source description:

- Protect selected modules or regions within modules.
- Automatically protect all modules.

Protecting Selected Regions in a Source Description

To protect the source description of selected modules or regions, follow these steps:

- 1. Place two compiler directives in the source description to define the protected region: `protect marks the beginning of the protected region; `endprotect marks the end of the protected region.
- 2. Compile the Verilog-XL source description file with the command-line option +protect.

Compilation creates a new source file in which the regions marked for protection become unreadable. This protected source file does *not* overwrite the original, unprotected source file.

Note: Whether you protect the source of an entire module or only selected regions inside that module, Verilog-XL considers the entire the module to be protected.

The 'protect and 'endprotect Compiler Directives

The compiler directives `protect and `endprotect can appear inside or outside a module or user-defined primitive (UDP). You must always pair each `protect directive with an `endprotect directive in a single source file. If a `protect appears without a corresponding `endprotect, the compilation appears successful. However, when you recompile the generated protected source file, an error occurs.

Multiple sets of the 'protect and 'endprotect compiler directives may appear within modules or primitives. However, you cannot nest blocks of source code bounded by 'protect and 'endprotect inside one another.

The following two examples show how to use the `protect and `endprotect compiler directives in a source file. In the first example, a region within module top_design is marked for protection:

```
module top_design (a, b, c)
    bottom inst ();
    'protect
        initial
        $display ("Inside module top_design");
        'endprotect
endmodule
```

In the second example, the entire module bottom is marked for protection, including the module name:

```
`protect
   module bottom ();
        initial
        begin
        $display ("Inside module bottom");
        end
```

endmodule **`endprotect**

The +protect Command-Line Option

Although the compiler directives `protect and `endprotect mark the regions Verilog-XL protects in your source description, the protection actually occurs only *after* you compile the source file with the +protect command-line option.

Compiling a Verilog-XL source description with the +protect command-line option protects only the regions marked with `protect and `endprotect compiler directives. After compilation, a new source file is created that differs from the original file in the following ways:

- The directives `protect and `endprotect become `protected and `endprotected respectively.
- The regions marked for protection in the original source description become unreadable.

/Important

While Verilog-XL compiles a source file for protection with +protect, it performs no syntax checking, simulation, or library resolution, and it ignores any command-line options designated for these functions.

Once the files in the previous two examples are compiled for protection, they take the forms shown in the following example:

```
module top_design (a, b, c);
    bottom inst ();
    'protected
    a*lejodi)dlj@lsfj4gRekv*9l#sIjnd<;pXywUHvow%emhiITvne(@mengTVpe
    prK58s53<gf:dneURtnd&8ejsWqpsu*ehtsY=wkxOrkp$
        'endprotected
endmodule
'protected
    fkeop*456gjkl@%^&^&s85Kfmv(:wjvdwLSchrmx*2uPQjsu=:wucgwigIWsuxnt
    pr"W84&@(shxjMvn02:wjd8%&!0s$
'endprotected
```

The new, protected source file does not overwrite the original, unprotected source file. When compiling the original source file with +protect, you can specify an optional file extension that is automatically appended to the name of the protected source file. If you do not specify an extension, Verilog-XL automatically appends a p to the protected file's name.

The following command line that directs Verilog-XL to protect the file src.v. Since no extension is specified, Verilog-XL produces a protected file called src.vp.

verilog src.v +protect

The following command line below specifies an extension .myext to be appended to the file design.v. As a result, Verilog-XL generates a protected source file called design.v.myext.

verilog design.v +protect.myext

Note: If the name of the protected file conflicts with the name of an existing file, Verilog-XL does not create the protected file; instead, it issues a message that alerts you to the filename conflict.

Protecting Multiple Files in a Single Command

You can use +protect to protect multiple Verilog-XL source files in a single command line, as shown in this example:

verilog src4.v src5.v src6.v +protect

Here, three files are created—src4.vp, src5.vp, and src6.vp—that present all regions marked by `protect and `endprotect from src4.v, src5.v, and src6.v in protected format.

Note: If you protect a design in one version of Verilog-XL, you can simulate the protected design in another version of Verilog-XL. For example, a design protected in Verilog-XL 3.0, can be simulated in Verilog-XL 3.3.

Protecting All Modules and UDPs in a Source Description

To protect modules and UDPs in a design automatically, compile the Verilog-XL source description file with the command-line option +autoprotect.

The +autoprotect Command-Line Option

The plus option +autoprotect protects all modules and UDPs in the specified source file automatically. This option is particularly useful for protecting libraries that contain a large number of files with many modules and UDP descriptions. Compiling your source file with +autoprotect creates a new source file that differs from the original source file in the following ways:

- The directive `protected is inserted after all module and UDP names, immediately before their port and terminal lists.
- The source descriptions inside all modules and UDPs become unreadable.

■ The `endprotected directive is inserted just before the endmodule keyword in modules and just before the endprimitive keyword in UDPs.

To see how compiling with +autoprotect alters a source file, consider the following source code:

```
module top_design (a, b, c);
    input a,b;
    output c;
    initial
      $display ("Inside module top_design");
endmodule
```

Compiling the previous module example with +autoprotect creates the protected file shown in the following example:

```
module top_design 'protected a%jioDT:3e(prlXCWN67suwOpw%3(j&ls)?l
j8wPQhsmchALsxy23XM#&0):3Wbv
9DwoPs,x>s:2yTJfSlsBx,>?uri839tkd%whfx8$
'endprotected endmodule
```

Notice that the module name and the keywords module and endmodule remain *outside* the protected region. Anything following the module name—typically from the port list onward—becomes protected and therefore cannot be read from the source file. Protection ends prior to the keyword endmodule to make it easier for Verilog-XL to scan files during library searches.

The new, protected source file does not overwrite the original, unprotected source file. The option +autoprotect takes an argument for an optional extension to append to the name of the protected file, as in this sample command line:

verilog design.v +autoprotect.protall

Here, Verilog-XL produces a file called design.v.protall, in which all modules and UDPs are protected.

If no extension is specified after +autoprotect on the command line, Verilog-XL appends a p to the protected filename.

Note: If the name of the protected file will conflict with the name of an existing file, Verilog-XL does not create the protected file; instead, it issues a message that alerts you to the filename conflict.

You do not need to insert the `protect and `endprotect compiler directives in any source description that you plan to compile with +autoprotect. However, if these directives *already exist* in your source file, you can still compile it with +autoprotect. In this situation, +autoprotect creates a protected source file by performing the following actions:

■ It removes any existing `protect and `endprotect directives.

- It inserts `protected after all module and UDP names, immediately before their port and terminal lists.
- It changes the source descriptions inside all modules and UDPs so that they become unreadable.
- It inserts `endprotected just before endmodule in all modules and just before endprimitive in all UDPs.

Protecting Multiple Files in a Single Command

You can use +autoprotect to protect multiple Verilog-XL source files in a single command line. The following example shows how to protect multiple files with +autoprotect:

verilog src1.v src2.v src3.v +autoprotect

Here, three files are created—src1.vp, src2.vp and src3.vp—that present all modules and UDPs from src1.v, src2.v and src3.v in protected format.

Effect of Source Protection on Simulation

When you protect regions in your source description, you limit access to the information inside these regions during simulation, and therefore you limit the performance of Verilog-XL system tasks and interactive commands.

System Operations That Cannot Access Protected Data

Protecting regions in a source description limits access to information about objects in those regions and, therefore, affects the system operations shown in the following table:

Source Protection	Effect of Source Protection
\$list	Unable to list source within protected module
\$settrace	Unable to trace activity within protected modules
\$showvars	Unable to display the structural information of elements within protected modules
<pre>\$showexpandednets</pre>	Unable to reveal information for protected nets
<pre>\$showportsnotcollapsed</pre>	Unable to reveal information for protected ports
-d command-line option	Unable to decompile protected modules

Verilog-XL User Guide

Source Protection

Source Protection	Effect of Source Protection
step and trace throughprotected text interactively	Unable to single-step through protected modules and display active statement

Note: For the operations in the previous table, protecting any portion of a module has the same effect as protecting the entire module.

System Operations That Can Access Protected Data

The following system tasks access information about objects in protected regions, as long as you pass the full path names of the object instances, relative to scope:

\$display
\$write
\$monitor
\$strobe
\$dumpvars
\$scope

The following example shows how \$monitor accesses information about two output nets within a protected region whose full path names are passed as arguments. The source description—a model of D flip-flop DFF—contains a protected region, marked by the `protect and `endprotect compiler directives. The following example shows the source description before protection.

```
module top_design;
// declarations of ports
reg dCLK,dD;
DFF1 d1 (dCLK, dD, dQ, dQB); // instance of protected module DFF1
//stimulus
    initial
    begin
        dCLK = 0;
        #1000 dD = 0;
        \#1000 \text{ dD} = 1;
        #1000 dD = 0;
        #1000 $finish;
    end
    always
        #500 dCLK = ~dCLK;
//monitor outputs
    initial
    begin
        $display ("Monitor internal connections");
    // dl.m4 and dl.m5 connections declared in protected region
        $monitor ($time,,"DFF1 m4=%b DFF1 m5=%b",d1.m4,d1.m5);
    end
endmodule
```

Verilog-XL User Guide

Source Protection

```
//D flip-flop
module DFF1 (CLK, D, Q, QB);
protect
  input CLK, D;
 output Q, QB;
//netlist
    not g1(CKB,CLK);
    not g2(CK,CKB);
    cmos #1 g3(m1,D,CKB,CK);
    not g4(m2,m1);
    not q5(m3,m2);
    cmos #1 g6(m1,m3,CK,CKB);
    cmos #1 g7(m4,m2,CK,CKB);
    not g8(m5,m4);
    not q9(m6,m5);
    cmos #1 g10(m4,m6,CKB,CK);
    // m4 and m5 connections declared in protected region
    not #(150,200) g11(Q,m4);
    not #(250,300) g12(QB,m5);
'endprotect
endmodule
```

The next example shows the same source description as in the previous example after protection. The full path names of the internal connections dl.m4 and dl.m5 are passed to the \$monitor system task. Notice the `protected and `endprotected compiler directives that surround the protected region.

```
module top design;
// declarations of ports
reg dCLK,dD;
DFF1 d1 (dCLK, dD, dQ, dQB);
//stimulus
    initial
    begin
        dCLK = 0;
        #1000 dD = 0;
        #1000 dD = 1;
        #1000 dD = 0;
        #1000 $finish;
    end
    always
    #500 \text{ dCLK} = ~ \text{dCLK};
//monitor outputs
    initial
    begin
        $display ("Monitor internal connections");
        $monitor ($time,,"DFF1_m4=%x DFF1_m5=%x",d1.m4,d1.m5);
    end
endmodule
// D flip-flop
module DFF1 (CLK, D, Q, QB);
'protected
^JR5JF6Ek;`\9K2eN4NVT9;3:40Y\f0WVQ<2]ACk 3foE2kF\bq B=041DQ:`I!1</pre>
a9\eTScXG8606o3]K0QlXN66OnHT[a8CheUiP^ADekm8cL>mR?KLC?pI[Sf5qyru
06;Yom[36]OmYI4F`QZ9?HSPd:9Jfn]dh4SXK7>SUV^L0G^Pe<SfnpU<f_k[K@ow
```

;cHDnnN<K0Q[:Hg3:oEQfHfWI[VhjU9g6JV=Eh;e6MY19>g`C2ocEOUBq?H>d9o> Whjkl3WXKM^5qAdJ]hEW2_k6U2Y7khPU^K[FOCoZc;8qBLXQdG>:PINO5wh28ao; FAT[9EJG4\Qhgq1Y7=eTZYkI=cRL`acmjgEU5gb:=1BKpUbfQkHY7YDgbd\dz,/2 B:HeDcS\Z=c<7A0p@OPk3N>5RoebEdd<XPGaX4S7jBM7F3dC51aC`;R_bMsd8*ou]\=JaEL\A\3Fmb[kVLJHQTq62BlTgAdm61_c4L1SoGh7T=UfXHn8H]KF>I\$ 'endprotected endmodule

The following example shows the Verilog-XL output during the simulation of the protected source description. Notice that *\$monitor* displays the values for d1.m4 and d1.m5, even though they are defined in a protected region.

Compiling source file "example.vp"
Highest level modules:
top_design
Monitor internal connections
 0 DFF1_m4=x DFF1_m5=x
1501 DFF1_m4=1 DFF1_m5=0
2501 DFF1_m4=0 DFF1_m5=1
3501 DFF1_m4=1 Dff1_m5=0
L15 "example.vp": \$finish at simulation time 4000
135 simulation events
CPU time: 1 secs to compile + 0 secs to link + 0 secs in simulation

Effect of Source Protection on Library Use

You can use the Verilog-XL library management capability on protected modules whose names appear outside the protected regions. The +autoprotect option protects modules and UDPs in just this way, so that they can be used with the library feature.

You *cannot* successfully perform library searches by using the -v and -y command-line options while protecting source descriptions with +protect or +autoprotect.

Note: Verilog-XL does not support the use of the `uselib compiler directive within a protected region.

See <u>Appendix A, "Verilog-XL Command-line Options,"</u> for more information about command-line options. See <u>Chapter 6, "Library Management,"</u> for more information about library commands and options.

Effect of Source Protection on the Display of Hierarchical Path Names

Verilog-XL does not display path names that contain components from protected regions. The example that follows the figure shows how source protection affects two Verilog-XL system tasks that display hierarchical names: \$display and \$showvars. There are three modules in this example:

- an unprotected top-level module, one
- a protected module, two, that is instantiated as inst2 within the unprotected top-level module, one
- an unprotected module, three, that is instantiated as inst3 within the protected module, two

The following figure shows the register port connections among the three modules:



The source description for the circuit design in this figure is in the following example:

```
// File: src.v
module one ();
                           //top level module
reg [7:0] aa,bb,cc;
two inst2 (aa,bb,cc);
    initial
   begin
        aa = 0;
        bb = 1;
        $display("--> Inside top level module 'one'");
        // shows signals on unprotected module
        $showvars(aa,bb);
        // shows ports to protected module
        $showvars(inst2.a,inst2.b);
        // shows ports to unprotected module, which goes through
        // unprotected module (does not show full hierarchical
```

Verilog-XL User Guide

Source Protection

```
// path name)
        $showvars(inst2.inst3.a,inst2.inst3.b);
        #10;
    end
endmodule // one
module two (a,b,c);
`protect
input [7:0] a,b,c;
three inst3 (a,b,c);
    initial
    begin
        $display("--> Inside protected module 'two'");
        $showvars(a,b);
    end
`endprotect
endmodule // two
module three (a,b,c);
input [7:0] a,b,c;
    initial
    begin
        $display("--> Inside unprotected module 'three'");
        $showvars(a,b);
    end
endmodule // three
```

The following example shows the actual Verilog-XL output before and after compiling the sample source description for protection. The boxes in the first column highlight the elements that are affected by source protection; the corresponding boxes in the second column show how the output changes after protection.

Output Before Protection	Output After Protection
Inside unprotected module three	Inside unprotected module three
<pre>a (one.inst2.inst3) wire = 8'hx, x 8'hx, x <- (one.inst2) : port 1</pre>	a () wire = 8'hx, x 8'hx, x <- () : port 1
<pre>b (one.inst2.inst3) wire = 8'hx, x 8'hx, x <- (one.inst2) : port 2</pre>	b () wire = 8'hx, x 8'hx, x <- () : port 2
Inside protected module two	Inside protected module two
a (one.inst2) wire = 8'hx, x 8'hx, x <- (one.inst2): port 1`	Unable to display information for protected elements
<pre>b (one.inst2) wire = 8'hx, x 8'hx, x <- (one.inst2): port 2</pre>	Unable to display information for protected elements

Verilog-XL User Guide

Source Protection

Output Before Protection	Output After Protection
Inside top-level module one	Inside top-level module one
aa (one) reg = 8'h0, 0 bb (one) reg = 8'h1, 1	aa (one) reg = 8'h0, 0 bb (one) reg = 8'h1, 1
a (one.inst2) wire = 8'hx, x schedule = 8'h0, 0 8'h0, 0 <- (one.inst2): port 1	Unable to display information for protected elements
b (one.inst2.inst3) wire = 8'hx, x 8'h1, 1 <- (one.inst2) : port 2	Unable to display information for protected elements
a (one.inst2.inst3) wire = 8'hx, x 8'h0, 0<- (one.inst2) : port 1	a() wire = 8'hx, x 8'h0, 0 <- () : port 1
b (one.inst2.inst3) wire = 8'hx, x 8'h1, 1<- (one.inst2) : port 2	b() wire = 8'hx, x 8'h1, 1 <- () : port 2

Notice that the information contained within the protected module, two, instance inst2, is not displayed. Information such as the strengths in unprotected module three, instance inst3, is displayed, but the full path name does not appear, since inst3 is instantiated through protected module two.

Error Reporting in Source-Protected Regions

When error messages originate from protected regions, they identify the source of the error. Yet they do not relay proprietary structural information. Messages about errors that occur within protected regions do not make explicit references to lines of code in the source description. Here is an example:

```
Compiling source file "test.vp"
"test.vp": syntax error in protected region
"test.vp": error in protected region
"test.vp": error in protected region
3 compilation errors
```

Syntax Verification

Verilog-XL does *not* verify syntax when compiling a file for protection. Therefore, it is important that you verify the syntax of a source description by compiling with the -c option before protecting the file.

Timing Checks

Timing checks placed within protected modules *do* report timing violations with full signal names. Therefore, if you do not want to reveal signal names in protected modules, remove all

timing checks prior to protection. For more information on timing checks, see <u>Chapter 13,</u> <u>"Timing Checks"</u> of *Verilog-XL Reference*.

Loading Source-Protected Data into Memory

Two system tasks—<code>\$sreadmemb</code> and <code>\$sreadmemh</code>—load data into memory from a Verilog-XL source character string, thus supporting the protection of that data.

The \$sreadmemh and \$sreadmemb Tasks

The \$sreadmemh and \$sreadmemb system tasks take memory data values and addresses as string arguments. These strings take the same format as the strings that appear in the input files passed as arguments to \$readmemh and \$readmemb. For more information about these tasks, see <u>"How \$sreadmem/h Differs from \$readmem/h"</u> on page 178 and <u>"Loading</u> <u>Memories from Text Files"</u> of the Verilog-XL Reference.

The syntax for <code>\$sreadmemh</code> and <code>\$sreadmemb</code> is as follows:

```
$sreadmemb(<mem_name>,<start_addr>,<finish_addr>,<string1>,<string2>,,,);
$sreadmemh(<mem_name>,<start_addr>,<finish_addr>,<string1>,<string2>,,,);
```

The following table describes the \$readmemb and \$readmemb variables:

<mem_name></mem_name>	Name of the memory structure
<start_addr></start_addr>	Memory start address
<finish_addr></finish_addr>	Memory end address
<string></string>	The string containing either the actual data to be placed into memory, or a set of specific addresses and their corresponding data values

In the following example, *\$sreadmemh* loads the memory memx with 16 hexadecimal values—0 through F—starting at address 0 and ending at address 15:

```
'protect
module readmx ();
reg [3:0] memx [0:15];
    initial
    begin
        $sreadmemh (memx,0,15,"0 1 2 3 4 5 6 7","8 9 A B C D E F");
    end
endmodule // readmx
'endprotect
```

After this source file is compiled for protection with +protect, the contents of memory memx become unreadable in the protected source file and cannot be modified during simulation. Note that in this example, the string argument to \$sreadmemh contains data values but no addresses.

The next example uses \$sreadmemb to load the memory memy with six binary values—0, 1, 1, 0, 1, and 0—at noncontiguous memory addresses @0, @1, @7, @8, @9, and @15:

```
`protect
module readmy ();
    reg [7:0] memy [0:99];
    initial
    begin
        $sreadmemb (memy,0,15,"@0 0 @1 1 @7 1","@8 0 @9 1 @F 0");
    end
endmodule // readmy
`endprotect
```

After this source file is compiled for protection with +protect, the contents of memory memy become unreadable in the protected source file and cannot be modified during simulation. Note that in this example, the string argument to \$sreadmemh contains both data values and addresses. The addressess are specified with the prefix @.

How \$sreadmem/h Differs from \$readmem/h

The system tasks <code>\$sreadmemh</code> and <code>\$sreadmemb</code> load data into memory from a Verilog-XL source character string, thus supporting protection of that data. By contrast, <code>\$readmemh</code> and <code>\$readmemb</code> load data into memory from external files that are not protected. Therefore, when you call <code>\$readmemh</code> and <code>\$readmemb</code> from within protected regions, the data loaded into memory will not be protected.

To illustrate this difference, the next example uses <code>\$sreadmemh</code> to load the memory mema, and <code>\$readmemh</code> to load the memory memb. The <code>\$sreadmemh</code> system task reads memory addresses and data values from a string argument, while <code>\$readmemh</code> reads memory addresses and data values from a separate file, <code>data1</code>.

```
'protect
module readm ();
    reg [7:0] mema [0:99];
    reg [7:0] memb [0:99];
    initial
    begin
        $sreadmemh (mema,0,5,"@0 B @1 A","@2 9 @3 2 @4 1 @5 0 ");
        $readmemh ("data1",memb,6,26);
    end
endmodule
'endprotect
```

After this source file is compiled for protection with +protect, the arguments to the \$sreadmemh system task are protected and you can not be read them, but you can read the contents of the external data file data1.

Verilog-XL User Guide Source Protection
Improving Performance

This chapter describes the following:

- Overview on page 181
- Displaying Memory Usage on page 181
- <u>Displaying Simulation Bottlenecks (Behavior Profiler)</u> on page 182

Overview

This chapter describes how you can improve your simulation performance.

Displaying Memory Usage

You can display the size of the data structure that describes your simulation by providing the parameter 2 to either *\$stop* or *\$finish*. When you specify this optional parameter, both system tasks display the following information: simulation time, location, and statistics about memory usage and CPU time.

The following example shows you how to get simulation information, including memory usage, from the *\$finish* system task. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Displaying memory usage

```
module DFF_test;
reg clk, clr, d;
wire q, qb;
DFF dff1 (d, clk, clr, q, qb);
initial
begin
  clr=0; d = 0; clk = 0;
  $monitor("time = %0t, q = %b", $stime, q);
end
```

Improving Performance

```
initial
begin
 #80 d = 1;
 #100 \ clr = 1;
\#10 \ d = 0;
 \#100 d = 1;
#100 $finish(2);
                                           // See Step 1
end
always #50 clk = ~clk;
endmodule // DFF_test
% verilog dff test.v dff.v
    . . .
Compiling source file "dff_test.v"
Compiling source file "dff.v"
Highest level modules:
DFF_test
time = 0, q = x
time = 4, q = 0
time = 158, q = 1
L21 "dff test.v": $finish at simulation time 390
                                                  // See Step 2
Data structure takes 8800 bytes of memory
110 simulation events
CPU time: 0.4 secs to compile + 0.1 secs to link + 0.1 secs in simulation
```

- 1. To display simulation information, including memory usage, specify the parameter 2 to the \$finish (or \$stop) system task.
- 2. When Verilog-XL executes \$finish(2) or \$stop(2), Verilog-XL displays simulation information, including memory usage.

Displaying Simulation Bottlenecks (Behavior Profiler)

You can find out what statements in your source code use the most CPU time during simulation by using the Behavior Profiler. By identifying simulation bottlenecks, you can concentrate performance improvement efforts on the most CPU-intensive parts of your simulation. Because the Profiler gathers statistics by continually interrupting your simulation, expect longer simulation times with the Profiler enabled.

The following example shows you how to collect and interpret simulation performance information from the Behavior Profiler. It shows two implementations of a D flip-flop and shows how modeling style can greatly influence simulation efficiency. "See Step n" comments in the example correspond to descriptive steps that follow the example.

Example: Displaying simulation bottlenecks

```
module behavior (q1,qb1,q2,qb2);
output q1,qb1,q2,qb2;
reg clk1,clk2,d1,d2,clr1,clr2;
```

Verilog-XL User Guide

```
inefficient ineff (clk1,d1,clr1,q1,qb1);
efficient eff (clk2,d2,clr2,q2,qb2);
initial
fork
    begin
        #100 $startprofile;
                                                      // See Step 1
        repeat(2000) @(posedge clk1);
        $listcounts;
                                              // See Step 2
        $listcounts(behavior.ineff);
        $listcounts(behavior.eff);
        $stopprofile;
                                               // See Step 3
        $finish;
    end
    begin
         clk1=0; clk2=0;
         forever begin
         #30 clk1=~clk1; clk2=~clk2;
     end
end
    begin
        d1=0; d2=0;
        #15 forever
        begin
            #120 d1=~d1; d2=~d2;
        end
    end
    begin
        clr1=0; clr2=0;
        #15 forever
        begin
            #240 clr1=~clr1; clr2=~clr2;
        end
    end
 join
endmodule // behavior
% verilog profiler.v profiler_eff.v profiler_ineff.v
    . . .
Highest level modules:
behavior
     // profiler.v
                                             See Step 4
          1 module behavior(q1, qb1, q2, qb2);
          2
                  output
          2
                      q1, // = St0
                      qb1, // = St1
          2
          2
                      q2, // = St0
          2
                      gb2; // = St0
          3
                  req
                      clk1, // = 1'h1, 1
          3
          3
                      clk2, // = 1'h1, 1
                      d1, // = 1'h0, 0
          3
          3
                      d2, // = 1'h0, 0
          3
                      clr1, // = 1'h0, 0
          3
                      clr2; // = 1'h0, 0
          5
                  inefficient
          5
                      ineff(clk1, d1, clr1, q1, qb1);
          6
                  efficient
                      eff(clk2, d2, clr2, q2, qb2);
          6
```

Verilog-XL User Guide

Improving Performance

8 1: initial 1: 9 fork 1: 10 begin 1: 11 clk1 = 0;1: clk2 = 0;11 4004: 12 forever+ 4004: 12 begin 4004: 13* #30 4003: 14 clk1 = ~clk1; 4003: 15 clk2 = ~clk2;16 end 17 end 1: 19 begin d1 = 0;1: 20 1001: 50 endmodule // behavior // profiler noteff.v 1 module inefficient(clk1, d1, clr1, q1, qb1); 2 input 2 clk1, // = St0 2 d1, // = St0 2 clr1; // = St0 3 output 3 q1, // = 1'h0, 0qb1; // = 1'h1, 1 3 4 reg 4 q1, // = 1'h0, 04 qb1; // = 1'h1, 1 always 1: 6 2002: б @(posedge clk1) 22 endmodule // inefficient . . . // profiler_eff.v 1 module efficient(clk2, d2, clr2, q2, qb2); 2 input 2 clk2, // = St02 d2, // = St0clr2; // = St0 2 3 output q2, // = 1'h0, 03 qb2; // = 1'h0, 03 4 reg 4 q2, // = 1'h0, 0qb2; // = 1'h0, 04 1: 6 always 752: б wait(clr2 === 1'b1) 250: 7 begin :clock_trigger 1250: 8 forever 1250: 8 @(posedge clk2) 1000: 9 begin . . . 22 endmodule // efficient L46 "profiler.v": \$finish at simulation time 120090 82850 simulation events CPU time: 1.5 secs to compile + 0.1 secs to link + 19.4 secs in simulation Report limit: 100 // See Step 5 Profile ranking by statement: Self% Cum.% Samples Statement ____ ____ _____ ------

Verilog-XL User Guide Improving Performance

9.1%	9.1%	172	profiler.v,	L14, behavior
8.9%	18.0%	168	profiler.v,	L15, behavior
5.0%	23.0%	95	profiler_noteff.v,	L9, behavior.ineff
4.9%	27.9%	92	profiler_noteff.v,	L13, behavior
3.9%	31.8%	74	profiler_noteff.v,	L2, behavior.ineff
3.8%	35.6%	72	profiler_noteff.v,	L2, behavior.eff
3.5%	39.1%	67	profiler_noteff.v,	L8, behavior.ineff
3.3%	42.4%	62	profiler.v,	L42, behavior
3.1%	45.5%	59	profiler.v,	L15, behavior
0.1% 1	00.0%	2	profiler_eff.v,	L2, behavior.eff
Total sa	mples in	n report:	1892	
Profile	ranking	by module	e instance: // See S	Step 6
Self%	Cum.%	Samples	(Self + submodules)	Instance
53.2%	53.2%	1007	(100.0% 1892)	behavior
27.0%	80.2%	511	(27.0% 511)	behavior.ineff
19.8% 1	00.0%	374	(19.8% 374)	behavior.eff
Profile	by state	ement type	e:// See Step 7	
Self%	Cum.%	Samples	Statement type	
33.0%	33.0%	624	assign_stat	
17.2%	50.2%	325	norm_node	
16.6%	66.8%	315	assign_delay_stat	
7.5%	74.3%	141	delay_stat	
6.0%	80.3%	114	event_stat	
5.5%	85.8%	104	seq_block	
4.1%	89.9%	77	null_stat	
3.7%	93.6%	70	forever_stat	

4.1% 89.9% 77 null_stat
3.7% 93.6% 70 forever_stat
3.1% 96.7% 59 wait_stat
2.9% 99.5% 54 contassign_stat
0.5% 100.0% 9 deassign_stat
1. Invoke the Behavior Profiler with \$startprofile. Verilog-XL automatically displays the Profiler's results (the ranking by statement, ranking by statement type, and ranking

- the Profiler's results (the ranking by statement, ranking by statement type, and ranking by module instance tables) at the end of the simulation. You can force output before the end of the simulation with the *sreportprofile* sytem task.
- 2. You can display an execution count listing with \$listcounts. This listing is the same as \$list with an additional field for the number of times Verilog-XL executed each statement. By default, Verilog-XL displays the listing for the current scope, but you can optionally specify a hierarchical name as an argument to \$listcounts. The execution count listing is displayed for the top-level module (behavior), and the two instantiated modules ineff and eff.
- 3. Stop Behavior Profiler sampling with \$stopprofile. If you do not specify
 \$stopprofile, the Profiler samples from the time it is invoked (\$startprofile) until the end of the simulation.

- 4. The output from *\$listcounts* consists of the following: the first column (nonexistent for declaration statements) contains the number of times Verilog-XL executes the statement, the second is the line number, and the third is the source description.
- 5. The "Profile ranking by statement" table lists the statements in order of CPU usage. The first column is the percentage of total simulation CPU time used by the statement. The second is the cumulative percentage of CPU time for the current and previously displayed statements. The third is the number of Behavior Profiler samples. The final column contains the source file, line number, and hierarchical name of the module of the source line.
- 6. The "Profile ranking by module instance" table lists the CPU usage for each module instance. Note that the efficient model of the D flip-flop takes almost one-third less CPU time than ineff.
- 7. The "Profile ranking by statement type" table lists the CPU usage for each type of Verilog HDL statement.

Cosimulation with Verilog-XL and Quickturn

This chapter describes the following:

- <u>Overview</u> on page 187
- Cosimulation Software Overview on page 187
- Setting Up the Simulator for Cosimulation on page 189
- <u>Generating a Simulation Shell File</u> on page 190
- Simulating a Model with Verilog-XL and Quickturn on page 198
- Restrictions and Limitations on page 198

Overview

Verilog HDL designs can include models that simulate in conjunction with Quickturn Emulation Systems. You can use Verilog-XL as a front-end and testbench simulator and use Quickturn's emulators to perform hardware emulation.

Cosimulation Software Overview

The software elements for cosimulation are:

- Verilog-XL simulator
- Quickturn Design Systems Q/Bridge emulation environment, which allows Quickturn emulators to work with the Verilog-XL simulator.
- Cadence Model Manager for Quickturn[™], which manages the interaction between the Quickturn emulation system and the Cadence Verilog-XL simulator. With the Cadence Model Manager for Quickturn, you can simulate in an environment where, for example, the testbench in the simulator provides the stimulus to the design in a hardware emulator.

Other environments simulate with both the design and the testbench in the same domain.

You also can have part of the design on the simulator and some instantiated part on the emulator. However, for performance reasons, Cadence recommends that you download the entire design to the emulator and leave only the testbench on the simulator.

Shell Generator, which loads the Cadence Model Manager for Quickturn using the shellgen command and creates the shell file using a pin map.

Overview of Co-simulation with Quickturn



Setting Up the Simulator for Cosimulation

Before you can cosimulate, you must complete the preparation phases that are described in the following sections:

- <u>"Accessing Quickturn Integration</u>" on page 189
- <u>"Creating the Gate-Level Netlist"</u> on page 189
- <u>"Generating a Quickturn Emulator Database and a Pin Map"</u> on page 189
- <u>"Generating the Simulation Shell and Modifying the Testbench"</u> on page 190

Accessing Quickturn Integration

To provide Verilog-XL with access to Quickturn integration, perform the following steps:

- **1.** Install Product 20600 using Cadence SoftLoad, which supplies the Quickturn Model Manager necessary for cosimulation.
- 2. Add the following paths to the Quickturn software and emulators in your .cshrc file:

3. Install the Model Manager license using Cadence SoftLoad.

Creating the Gate-Level Netlist

The Quickturn database requires gate-level input in either Verilog or Electronic Data Interchange Format (EDIF). Before you can import either a Verilog or VHDL design into the Quickturn database, you must synthesize the design into a Verilog gate-level or EDIF netlist.

Note: If you plan to synthesize the design, first verify that library primitives are supported by Quickturn. Then make sure that you have set up a common library between the synthesis netlist and questCompile.

Generating a Quickturn Emulator Database and a Pin Map

To generate an emulator database and a pin map file, refer to the "Compiling IC Designs" chapter of the *Q/Bridge User's Guide* from Quickturn Design Systems.

Generating the Simulation Shell and Modifying the Testbench

At the workstation that contains the simulator software and testbench design, perform these steps:

- 1. Generate the simulation shell using the shellgen command. For information on using this command, see <u>"Generating a Simulation Shell File"</u> on page 190.
- 2. Modify your original testbench by instantiating the component from the shell into the testbench.

Note: These modifications pertain to generating a shell for the Cadence Model Manager for Quickturn's pseudo event mode option. For more information, see <u>"Event Mode"</u> on page 195 and <u>"Clocked Mode"</u> on page 196.

Generating a Simulation Shell File

A simulation shell file contains the following information:

- A Verilog module
- Name, ports, and signal declarations
- Attributes that identify the following:
 - a shell as an OMI 2.0- and 4.0-compliant model
 - □ the model manager
 - □ information for finding and loading the model manager

To generate the simulation shell, specify the shellgen command, which packages the model and generates the simulation shell. When you invoke the Shell Generator, it:

- Loads the model manager for the specified simulator in the bootstrap file
- Queries the model manager for the presence of the model(s) you specified with the model_name option of shellgen
- Checks model boundary information (parameters, ports, and viewports)
- Creates the shell file

The shellgen command has the following syntax:

```
shellgen
[-vhdl | -verilog]
[-o output_file]
```

```
-b bootstrap_file
[-l library_name] [-m model_name]
[-r] [-f file_name] [-unresolved]
[-version] [-h] [qt_mmoptions]
```

The arguments to the shellgen command are as follows:

Argument	Description
-verilog	Creates a Verilog shell for Verilog-XL. If you do not specify – verilog or -vhdl, the Shell Generator creates a Verilog shell file.
-vhdl	Creates a VHDL shell file for Leapfrog. For more information, see the <i>Leapfrog VHDL Simulation Reference</i> .
−o output_file	That the shell will be written to the specified output file. If the file name does not include an absolute path, then the Shell Generator interprets the file name as relative to the current directory.
	If you specify more than one model using multiple $-m$ options, or specify all models of a library by skipping $-m$ option, or specify all models known to the model manager by skipping both -1 and $-m$ options, then it creates a single output file by concatenating the shells.
	If you also specify the $-r$ option, the Shell Generator overwrites the existing output file.
	Note: If the output file exist and you do not specify the $-r$ option, the Shell Generator exits with an error message.
	Note: This argument is optional. If you omit it, the Shell Generator writes the shell to <i>model_name.v</i> . A Note message is also generated to indicate the name of the output shell file.
-b bootstrap_file	A required parameter, specifies the OMI bootstrap file, which locates and loads the model manager of the simulator. For information about the format of this file, see the OMI Specification.
	For OMI 4.0 compliant Quickturn models, use the <code>qtbootstrap</code> file located in <pre>cyour_install_dir>/tools/lib.</pre>
	For OMI 2.0 compliant Quickturn models, use the oldqtbootstrap file located in <your_install_dir>/ tools/lib along with +omi2_0 plus option.</your_install_dir>
-llibrary_name	Specifies the library from which the model(s) are to be loaded.

Verilog-XL User Guide

Cosimulation with Verilog-XL and Quickturn

Argument	Description
-m model_name	The model(s) for which the shell is to be generated.
	You can use this option multiple times. For example, you can specify
	-m ml -m m2 -m m3
	However only one output shell file is generated where the shell outputs for each model is concatenated.
	The Shell Generator uses the model names without modification in its calls to the model manager.
-r	Overwrites the existing file specified with the $-\circ$ option.
	Note: If the output file(s) exist and you do not specify this option, the Shell Generator exits with an error message.
-f file_name	Specifies a file in which each shellgen option is included on a separate line using the following format:
	option[=option_value]
-unresolved	Uses the unresolved type <pre>std_ulogic</pre> and <pre>std_ulogic_vector ports and viewports of any logic type in the <pre>design, instead of the default resolved types <pre>std_logic</pre> and <pre>std_logic_vector.</pre></pre></pre>
-version	Displays the version of the shell generator on the screen.
-h	Displays the options for the shellgen command on the screen.
qt_mmoptions	Specifies the plus options. Two of the options, +qt_model and +qt_pin_map, are required for shell generation. See the table in <u>"Cadence Model Manager for Quickturn Command-Line Plus</u> <u>Options"</u> on page 194 for more information.

Note: The order of I/O pins in the Simulation Shell File is determined by the order of the I/O pins in the pin map file. If the order of the I/O pins in the Simulation Shell File and its instantiation in the testbench do not match, modify the instantiation or the pin map file. (If the pin map file is modified a new emulation database must be compiled). **Do not modify the order of the I/O pins in the Simulation Shell File, or signals will not be correctly connected between the simulator and emulator!**

Note: You can use the -1 and -m options together in many ways to generate the shell files for desired models.

When -I option is	And -m option is	Then shellgen generates a shell for
used	used once or multiple times	Specified model(s) in the specified library.
used	not used	Every model in the specified library.
not used	used once or multiple times	Specified model(s) in all libraries known to the Model Manager
not used	not used	Every model in all libraries known to the Model Manager

Table 11-1	Using -I	and -m	options	together
------------	----------	--------	---------	----------

Note: You can use the -o and -m options together in many ways to generate the desired output shell file.

Table 11-2	Using -o	o and -m	options	together
------------	----------	----------	---------	----------

When -o option is	And -m option is	Then the output file name is
used	used once	output_file
used	used multiple times	output_file
used	not used	output_file
not used	used once	<pre>model_name.v where model_name is same as that of the model.</pre>
not used	multiple times	<pre>model_name.v where model_name is same as that of the first model specified.</pre>
not used	not used	model_name.v where model_name is same as that of the first model found.

Example of Using the Shell Generator

The following command-line example generates a shell file for the Verilog-XL simulator and gives the shell file the name $shell_vxl.v$. The example uses information in the bootstrap file <code>qtbootstrap</code>, the model <code>testmodel</code>, and the pin map file <code>top.map</code> to generate the shell file.

```
shellgen -verilog -o shell_vxl.v -b \

            <gour_install_dir>/tools/lib/qtbootstrap \
        +qt_model=testmodel +qt_pin_map=top.map +qt_mode=event \
        +qt_emulator=qtr +qt_qbridge=QUICKTURN
```

The following example shows a generated shell file:

```
module testmodel (memAddr , memDataIn , read , clk , memDataOut , nint , reset ,
test_mode , scan_enable , scan_in , scan_out )
                          = "<full_path_name>";
    (* integer mm_path
        integer mm_object = "qtmanager.so";
        integer mm_bootstrap = "qt_manager";
                            = "testmodel";
        integer model
    *);
    output [12:0] memAddr ;
    output [7:0] memDataIn ;
    output read ;
    input clk ;
    input [7:0] memDataOut ;
    input nint ;
    input reset ;
    input test_mode ;
    input scan_enable ;
    input scan in ;
    output scan_out ;
   parameter qt_mode = "event";
   parameter qt_pin_map = "top.map";
   parameter qt_emulator = "qtr";
   parameter qt_qbridge = "QUICKTURN";
   parameter qt_strobedelay = 15;
   parameter qt_outputdelay = 1; `
   parameter qt_lib = "libqteapi";
   reg (* integer omi_viewport = 1; *) \u1.foo ;
endmodule
```

Cadence Model Manager for Quickturn Command-Line Plus Options

If you are using the Cadence Model Manager for Quickturn with the Shell Generator, you can also specify the following plus options on the shellgen command line.

Plus option	Description
+qt_model =modelname	Specifies the name of the model for which the shell is being generated. The -m option of the shellgen command overrides this plus option.

Verilog-XL User Guide Cosimulation with Verilog-XL and Quickturn

Plus option	Description	
+qt_pin_map =filename	Specifies the path to the pin map file in the Q/Bridge database directory. The Quickturn model manager uses this file to generate a shell with the Shell Generator.	
+qt_mode= <i>mode</i>	Specifies how the model manager relays stimulus from the simulation to the emulator. The mode parameter can be one of the following:	
	event clockedDelayed clocked clockedPosDelayed clockedPos clockedNegDelayed clockedNeg	
	For more information about these modes, see <u>"Quickturn Modes</u> for the qt_mode Option" on page 195.	
+qt_lib=path	Specifies the path to the Quickturn integration Library. You can specify this option with the shellgen or the verilog command. The default is libgteapi.	
+qt_emulator = <i>address</i>	Specifies the host name or address of the Quickturn emulator. The default is emulator.	
+qt_qbridge = <i>address</i>	Specifies the host name or address of the NT server of Q/Bridge. The default is <code>qbridge</code> . If your Q/Bridge system does not use an NT Server, specify the host name or address of the Quickturn emulator.	
+qt_strobedelay =[115]	Specifies a number from 1 to 15 that the model manager uses to control how long the NT server waits before it strobes the output of the emulator. The default is 15.	
+qt_outputdelay =value	Specifies the delay that the model manager uses to control the update of the models output. You must specify an integer that is in the simulator's precision. The minimum value is 1. The default is 1.	

Quickturn Modes for the qt_mode Option

The model manager operates in event mode or clocked mode.

Event Mode

In event mode, the model manager collects any input changes within a specified time range and downloads them at the end of the cycle. After a specified simulation delay (using the +qt_outputdelay plus option), the model manager pushes only the changed output signals into simulation.

When the model manager updates the status of the outputs from the emulator for the first time, the output ports transition from an 'X' value to a known state. The model manager subsequently updates only the outputs that change since the previous update. The same value is never driven into an output port.

Clocked Mode

In clocked mode, the shell generator adds an input pin to the shell, called SYSTEMCLOCK, at the end of the port list. The test fixture drives SYSTEMCLOCK, which in turn controls the inputs and outputs from the Quickturn emulator. The emulator drives the output values into the simulation at the opposite edge of the SYSTEMCLOCK signal.

The clocked mode options are as follows:

qt_mode option	Description
event	This is the default. Specifies the current edge of the SYSTEMCLOCK signal that the Model Manager uses to download the new input values to the emulator. The emulator drives the output values into the simulation at the opposite edge of the SYSTEMCLOCK signal.
clocked Or clockedPos	Downloads the new input values to the emulator when the SYSTEMCLOCK signal makes a 0 to 1 transition.
clockedNeg	Downloads the new input values to the emulator when the SYSTEMCLOCK signal makes a 1 to 0 transition.
clockedDelayed or clockedPosDelayed	After waiting for a delay specified in the +qt_ouputdelay plus option, downloads the new input values to the emulator when the SYSTEMCLOCK signal makes a 0 to 1 transition.
clockedNegDelayed	After waiting for a delay specified in the +qt_ouputdelay plus option, downloads the new input values to the emulator when the SYSTEMCLOCK signal makes a 1 to 0 transition.

Specifying Cadence Model Manager for Quickturn Options at Simulation Time

You can pass the following Cadence Model Manager for Quickturn options at simulation time, instead of specifying them to the shell generator:

qt_lib
qt_emulator
qt_qbridge
qt_strobedelay
qt_outputdelay

To pass Cadence Model Manager for Quickturn options at simulation time, use the option with the verilog command. For example:

verilog +qt_lib=/mylib/quickturnlib

Note: A command-line option supersedes an option of the same function that is specified to the shell generator.

\$omiCommand System Task

You can access the emulator memory with the <code>\$omiCommand</code>, which has the following syntax:

```
$omiCommand(<instance>, "<argument list>");
<instance> := name of the emulator shell
<argument list> :=
    "listmem {CMM | LMM}"
    ||= "readmem <QTmemory> <filename> <radix> <start> <end>"
    ||= "writemem <QTmemory> <filename> <radix>"
<QTmemory> := the memory object in the source file
<filename> :=
    name of the file that is loaded into memory (readmem)
    || name of the file into which memory is loaded (writemem)
<radix> := 2 (binary) | 8 (octal) | 10 (decimal) | 16 (hexadecimal)
<start> := starting memory address
<end> := ending memory address
```

For information about the format of the memory file, see the *Q/Bridge User Guide*. The following examples show how to use the \$omiCommand system task.

\$omiCommand(testFix.DUT, "listmem CMM");

This command display a list of different memory contents in the CMM design type.

\$omiCommand(testFix.DUT, "readmem u1.mem file.mem 16 0 100");

This example loads the contents of the file into memory from address 0 to address 100, hexdecimal.

\$omiCommand(testFix.DUT, ""writemem u1.mem file.mem 16");

This example loads the contents of memory into the specified file in hexadecimal format.

Simulating a Model with Verilog-XL and Quickturn

When you invoke Verilog-XL to simulate a model with Quickturn, Verilog-XL recognizes the model as a Quickturn model and invokes the Cadence Quickturn Model Manager to initialize and control the Quickturn emulator.

To simulate a model, you must have these files:

- Testbench
- Simulation shell file
- Quickturn database

Restrictions and Limitations

The following items for Verilog-XL and Quickturn cosimulation are restrictions and limitations:

- □ You can use all Verilog-XL commands on ports.
- □ You can use all Verilog-XL commands on viewports except assignment commands.
- □ Verilog-XL does not support the save/restore simulation command when the Quickturn model is integrated into the design.

Verilog-XL Command-line Options

This appendix describes the following:

- <u>Command-Line options</u> on page 199
- <u>Command-Line Plus Options</u> on page 205
- <u>User-Definable Command-Line Arguments</u> on page 224
- <u>Compiler Directives</u> on page 226
- <u>Conditional Compilation</u> on page 235
- <u>File Inclusion</u> on page 241

The command-line options that compose the Verilog-XL Turbo option are discussed in <u>Appendix F, "Verilog-XL Turbo and Twin Turbo Options."</u>

Command-Line options

This section describes the Verilog-XL command-line options. These options consist of a single character preceded by a hyphen. Verilog-XL command-line plus options are described separately in <u>"Command-Line Plus Options" on page 205</u>.

-a (Accelerate Option)

The -a option overrules the +noxl option and causes the XL algorithm (which is normally the default) to accelerate all possible elements of the description.

See <u>Appendix C, "Maximizing Default Acceleration,"</u> for more information on default behavioral acceleration by the default XL algorithm.

-c (Compile Only Option)

The -c option compiles the source text only and then exits. You can use this option to check a Verilog HDL source description for errors without simulating. If you use the -r (restart) option with this option, then the data file is loaded but no simulation is performed.

-d (Decompile Option)

The –d option prints out a decompilation of the generated data structure after source text has been compiled and linked or after a saved data file has been loaded. Use this option for system testing purposes. You can also use it to decompile a saved data file to find out what it contains, or to find out where the original source text has been lost.

-f (File Option)

Syntax:

-f <filename>

The -f option reads the text file that is specified following the option. The text file can contain source text filenames and Verilog-XL command options, including other -f options. You can nest up to 1024 -f options in this manner.

Illegal -f Option Specifications

The filename must follow the option on the command line. The following command is illegal because Verilog-XL attempts to use -1 as the filename for the -f option:

verilog -f -l options.vc

The -f option must be alone in a command line argument. For example, the following command is illegal:

verilog -cf options.vc

Example: -f File Option

The project1.vc and user.vc files in the following example are used to run Verilog-XL with a standard set of command options and with a particular design and library:

```
verilog -f project1.vc -f user.vc
```

The project1.vc file is a central library file containing the conventions for a particular project. It contains the following text:

/*conventions for describing project1 hardware*/
/* Hi-Tech Widget Corp., Feb. 1996 */
+sxl_keep_all //keeps channel-connected nets
+incdir+/net/switches //Verilog-XL will search this directory
+delay_mode_unit //use unit delay mode

The user.vc file is the user's local options file. It contains the following text:

The cpu.vc file, specified in the user.vc file, contains the following text:

cpu_netlist.v -f lib.vc

The lib.vc file, specified in the cpu.vc file, contains the following text:

array_lib_version2.v // gate array cell library
joe_alu.v // use Joe's alu description

-i (Interactive File Option)

Syntax:

-i <filename>

The -i option reads commands from a file, and then opens a command line for you to enter interactive commands. The -i option reads only one file; therefore, you can use the -i option only once on the command line.

You must include the -s option with the -i option on the command line, or include the stop system task in the source description. The -s option and the stop system task cause Verilog-XL to enter interactive mode. After Verilog-XL enters interactive mode, it executes the -i option.

You can also use the -i option to interactively recover a previously generated key file by performing an exact replay of a simulation run. See <u>"Interactive Recovery"</u> on page 250 for more information about interactive recovery.

Note: Do not specify the default key filename, verilog.key, when you enter a command with the -s and -i options. Enter the -k option (see<u>"-k (Key File Option)" on page 202</u>) to create a new key file with name other than verilog.key. If you specify the -s and -i options with the

default key file, Verilog-XL will create a new key file with the same name and never close the existing key file.

-k (Key File Option)

Syntax:

-k <filename>

The -k option changes the default key filename (verilog.key) to the name you specify following the -k option. A key file contains standard input (usually typed in from the terminal). See <u>"Key File"</u> of Verilog-XL Reference for more information on the -k option.

-I (Log File Option)

Syntax:

-l <filename>

The -1 option changes the default log filename (verilog.log) to the name you specify following the -1 option. The log file contains a copy of all the text that is printed to the standard output, and also includes, at the beginning of the file the host command that was used to run Verilog-XL. See "Log File" of Verilog-XL Reference for more information on the -1 log file option.

-q (Quiet Option)

The -q option suppresses the printing of messages during the major steps in compilation and simulation.

-r (Restart File Option)

Syntax:

-r <filename>

The -r option restarts the simulator from a data file that was previously saved using the save system task. Do not specify source text files on the command line with this option. See <u>"Command-Line Restart"</u> of *Verilog-XL Reference* for more information on the -r option.

Note: When you restart a simulation with the -r option using a previously saved Verilog save file (instead of using the *srestart* system task), you must reprobe the signals you want to see.

-s (Stop Option)

The -s option initiates entry into interactive mode immediately after compilation. To read an interactive input file, the -s option must be issued on the same command line as the -i input file option. You can also use the stop system task in the source description instead of the -s option.

If the simulation source files are structural descriptions that generate no procedural events, then the -s option has no effect.

-t (Trace Option)

The -t option performs a full trace from the start of simulation (same trace as given by the \$settrace system task).

-u (Uppercase Option)

The -u option converts all lowercase letters to uppercase, (except for text within strings), so that a source description becomes case-insensitive. This option also translates interactive command input text.

-v (Library File Option)

Syntax:

-v <filename>

The -v option reads a library file containing unresolved module and UDP definitions from the text files that you have specified on the command line. The following command line includes the libfile.v library file:

verilog source1.v -v libfile.v

See <u>Chapter 6, "Library Management,"</u> for more information about library management.

-version (Display Version Option)

Syntax:

-version

The -version option displays the version number of Verilog-XL. The following example shows how to use the -version option.

verilog -version

-w (Warning Suppression Option)

The -w option suppresses messages that report inconsistencies in module port connections, such as when vector sizes are mismatched or when not enough port connections have been specified.

-x (Vector Net Expansion Option)

The -x option expands all vector nets, except those that are specifically overridden by the compiler directives and keywords that control expansion.

Note: Verilog-XL cancels the -x option when it encounters a subsequent -x option.

-y (Library Directory Option)

Syntax:

-y <directoryname>

The -y option specifies the name of a directory that contains either a single module or UDP definition file, or a set of complete Verilog HDL hierarchies. The following example shows how to specify the -y option:

verilog source1.v -y /usr/me/proj/lib/cmos

You can also specify more that one library directory on the command line with multiple -y specifications, as follows:

verilog source1.v -y /usr/me/proj/lib/cmos -y /usr/you/proj/lib/cmos

If the files are hierarchical, then the top level of the hierarchy should be the first module declared in the file, and the other modules and UDPs should follow. All entries must conform

to the Verilog-XL library directory format. For more information about library management, see <u>Chapter 6, "Library Management."</u>

Examples

The following command line stops Verilog-XL from processing after reading the srcl.v and src2.v files.

verilog src1.v src2.v -s

The following command restarts a simulation that was previously saved with the *\$save* system task in the *save.dat* data file:

verilog -r save.dat

The following command lets you restart from a previously saved simulation (in save.dat) but Verilog-XL still read interactive commands from the commands.i file and stops to allow you to enter interactive commands:

verilog -r save.dat -i commands.i

The following command restarts from the save.dat file, decompiles the data structure to the standard output, and then exits. The quiet option specifies that only the decompilation text goes to the output log file.

verilog -dqcr save.dat

After you have performed these steps, you can copy the log file to another file (called dsrc.v) and re-route it back into Verilog-XL, as in the following example:

verilog dsrc.v

Command-Line Plus Options

This section describes the plus options that are supplied with Verilog-XL.

+accnoerr

The +accnoerr plus option suppresses error message reporting from PLI access routines.

+accu_path_delay

The +accu_path_delay plus option specifies an alternative path delay algorithm for any module output for which there is a path delay specification. The default algorithm is optimized for performance in many cases. However, some hardware models may require an alternative

algorithm to make a path delay choice that describes hardware more accurately. See <u>Enhancing Path Delay Accuracy</u> for details on the limitations of the default algorithm, and for examples of cases in which the alternative delay selection algorithm can make superior choices.

+alt_path_delays

The <code>+alt_path_delays</code> plus option calculates each path delay schedule time based on the transition from the current output value rather than from the pending scheduled transition value. See <u>Understanding Path Delays</u> for more information and an example.

+annotate_any_time

The <code>+annotate_any_time</code> plus option allows SDF annotation to occur at times other than time 0.

+autonaming

The +autonaming plus option generates names for those instances of Verilog-XL standard and user-defined primitives that you did not name. See <u>Automatic Naming</u> for more information about automatic naming.

+autoprotect

The +autoprotect plus option protects all modules and UDPs in a source description. See <u>"Protecting All Modules and UDPs in a Source Description"</u> on page 168 for more information on the +autoprotect plus option.

+caxl

The +cax1 plus option accelerates the continuous assignments in your design. See <u>Controlling the Acceleration of Continuous Assignments</u> for more information on the +cax1 plus option.

+compat_twin_turbo

The +compat_twin_turbo plus option ensures the compatibility of results between twin turbo levels. Twin Turbo levels can result in event ordering different from that shown in the

corresponding Turbo mode results. For example, results may differ between the second-level Turbo mode (+turbo+2) and the corresponding Twin Turbo level (+turbo+2 with +twin_turbo).

+define+

The +define+ plus option defines variable names as empty text macros throughout the Verilog-XL compilation process, or it defines macro names as strings. This section discusses defining macro names as strings;

"Defining Variable Names to Control Conditional Compilation" on page 237 discusses the +define+ plus option as a part of conditional compilation. The `uselib compiler directive controls library searches with specifications that can include macros defined by the +define+ plus option. The +define+ plus option has the following syntax when it defines a macro name as a string:

```
+define+<macro_name>="<macro_string>"
```

For example, including the following in a command line defines the macro name gate as the string or:

+define+gate="or"

To avoid parsing problems, you can define only one macro with each +define+ on the command line, but the number of +define+ plus options on the command line is unlimited.

Note: If you define the same macro name differently in a command-line +define+ plus option and a `define compiler directive, the command-line plus option overrides the compiler directive.

For example, if you simulate the following code with a command line that does not include a +define+ plus option, the value of c fluctuates because the compiler directive defines macro gate as an and gate.

```
module test;
'define gate and
req a,b;
`gate (c,a,b);
    initial
    begin
        #1;
        a=0;
        b=1;
        $monitor ($time,,c);
        #5;
        a=1;
        #5;
        $finish;
    end
endmodule
```

However, if you include the following +define+ plus option in the command line, the value of c remains constant because the plus option defines gate as an or gate:

```
+define+gate="or"
```

When a command-line +define+ plus option overrides a `define macro definition, Verilog-XL displays a warning similar to the following warning. This warning indicates that the plus option redefines the macro, regardless of the fact that the text in the warning is inconclusive.

You can define a macro name to be a string of any length with the +define+ plus option. There is no limit to the number of +define+ macros that you can define. The +define+ plus option cannot define macros of more than one line. Verilog-XL does not require quotation marks at the ends of a macro string if the string does not contain white spaces.

+delay_mode_distributed

The +delay_mode_distributed plus option specifies the distributed delay mode for your simulation. See <u>Command-Line Plus Options</u> for more information on the +delay_mode_distributed plus option.

+delay_mode_path

The $+delay_mode_path$ plus option specifies the path delay mode for your simulation. See <u>Command-Line Plus Options</u> for more information on the $+delay_mode_path$ plus option.

+delay_mode_unit

The +delay_mode_unit plus option specifies the unit delay mode for your simulation. See <u>Command-Line Plus Options</u> for more information on the +delay_mode_unit plus option.

+delay_mode_zero

The <code>+delay_mode_zero</code> plus option specifies the zero delay mode for your simulation. See <u>Command-Line Plus Options</u> for more information on the <code>+delay_mode_zero</code> plus option.

+err_line_ length

The <code>+err_line_length+</code> plus option specifies the number of characters Verilog-XL displays in an error message. You must supply a value with this plus option. If you specify a value of less than the minimum 20 characters, a warning message is issued and the value is set to 80 characters. In the following example, the line length is set to 60 characters in length:

verilog example.v +err_line_length+60

See <u>Appendix H, "Veriog-XL Messages"</u> for more information on Verilog-XL messages.

+extend_tcheck_data_limit/<percentage_limit>

The +extend_tcheck_data_limit/<percentage_limit> plus option automatically changes the hold or recovery limit in timing checks to extend the violation regions by a specified percentage so that they overlap.

If a decimal value is specified as the limit in the +extend_tcheck_data_limit/ <percentage_limit> option, the Verilog-XL simulator automatically truncates the value of the specified limit. For example, both 10.2 and 10.9 are considered as 10, by the Verilog-XL simulator.

+extend_tcheck_reference_limit/<percentage_limit>

The +extend_tcheck_reference_limit/<percentage_limit> plus option automatically changes the setup or removal limit in timing checks to extend the violation regions by a specified percentage so that they overlap.

If a decimal value is specified as the limit in the +extend_tcheck_reference_limit/ <percentage_limit> option, the Verilog-XL simulator automatically truncates the value of the specified limit. For example, both 10.2 and 10.9 are considered as 10, by the Verilog-XL simulator.

+gui

The +gui plus option invokes Verilog-XL in the *SimVision* window of the SimVision graphical environment.

+incdir+

The +incdir+ plus option specifies the directories that Verilog-XL searches for the files that you have specified with the `include compiler directive.

Syntax:

+incdir+<directory1>+<directory2>+...<directoryN>

There is no limit to the number of +incdir+ plus options that you can specify. Verilog-XL searches for these directories in the order in which you list them on the command line.

Important

Verilog-XL does not check the characters between the two plus characters for errors. Verilog-XL assumes that all of these characters are part of the directory name.

+libext+

The +libext+ plus option specifies library directory file extensions.

Syntax:

```
+libext+<string1>+<string2>+...<stringN>
```

To specify a library directory file extension, you put +libext+ in the command line followed immediately by the characters that make up the extension, as demonstrated in the following example:

verilog sourcel.v -y /usr/me/proj/lib/cmos +libext+.v+

See <u>"Organizing Libraries" on page 100</u> for more information about file extensions in library directories using the old library scheme.

+libnonamehide

The +libnonamehide plus option reads only the necessary module and UDP definitions (as they are written in the file without appending character strings) to resolve instances. See <u>"Guidelines for Using the Default Method" on page 116</u> and <u>"+libnonamehide" on page 118</u> for more information on the +libnonamehide plus option in the old library scheme.

+liborder

The +liborder plus option scans libraries and directories as they follow on the command line and then wraps around to the preceding libraries that Verilog-XL has not yet visited, as shown in the following example:

verilog source1.v -v lib1.v source2.v -v lib2.v +liborder

See <u>"Library Scan Precedence: The Former Scheme</u>" on page 108 for more information on the +liborder plus option using the old library scheme.

+librescan

The +librescan plus option scans library files and directories to resolve all undefined module and UDP instances from source files and libraries. The behavior of this plus option depends on whether the undefined instance is located in a source file, library file, or a file within a library directory. See <u>"Library Scan Precedence: The Former Scheme"</u> on page 108 for more information on the +librescan plus option using the old library scheme.

+libverbose

The +libverbose plus option displays or prints information about the opening of files and the resolution of module and UDP definitions during the scanning of libraries. See <u>"Reporting of Resolution Paths" on page 101</u> for more information on the +libverbose plus option using the old library scheme.

+licq_all

The +licq_all plus option allows simulations to be queued and automatically activated as the following licenses become available:

- VERILOG-XL (Verilog-XL)
- VXL-LMC-HW-IF (Verilog-XL LMC Hardware Interface). This feature is activated during compilation whenever there is a LMSI (LMC hardware interface) system task, \$1m_*(), present in the design.

Requests in a queue are serviced on a first-in-first-out basis with all requests at the same priority level. See <u>Verilog-XL Licences</u> for more information on queuing license requests.

+licq_lmchwif

The +licq_lmchwif plus option allows simulations to be queued and automatically activated as the VXL-LMC-HW-IF license becomes available. Verilog-XL activates the VXL-LMC-HW-IF license during compilation whenever there is a LMSI system task, $lm_*()$, present in the design. See <u>Verilog-XL Licences</u> for more information on queuing license requests.

+licq_vxl

The +licq_vxl plus option allows simulations to be queued and automatically activated as the VERILOG-XL license becomes available. See <u>Verilog-XL Licences</u> for more information on queuing license requests.

+listcounts

The +listcounts plus option enables the \$listcounts system task, which is disabled by default, to accelerate simulation. The \$listcounts system task is also invoked if the +no_speedup plus option is on the command line.

+loadpli1

Syntax:

```
+loadpli1=<library>:<boot_strap_pointer>
```

The +loadpli plus option dynamically loads a specified PLI 1.0 library from the command line. The <library> name specifies a library that contains a PLI 1.0 application. Each library name automatically has an operating-specific suffix appended to it (.sl for HPPA, .so for Sun4V, and .dll for Windows NT). The <boot_strap_pointer> name is a function that returns a pointer to either a p_tfcell array or a veriuserfs array. Each array contains either the definitions of system tasks and functions or time/event-related callbacks. Each array must terminate with NULL.

The following example shows how to use the +loadplil plus option:

verilog -f run.f +loadpli1=Lib1:bootfunc

+loadvpi

Syntax:

+loadvpi=<library>:<boot_strap_pointer>

The +loadvpi plus option dynamically loads a specified VPI library from the command line. The <library> name specifies a library that contains a VPI application. Each library name automatically has an operating-specific suffix appended to it (.sl for HPPA, .so for Sun4V, and .dll for Windows NT).

The <boot_strap_function> name is a function that returns a pointer to either a vpi_register_systf() or a vpi_register_cb() function call that contains the definitions of system tasks and functions.

The following example shows how to use the +loadvpi plus option:

verilog -f run.f +loadvpi=Lib1:bootfunc

+maxdelays

The +maxdelays plus option selects the maximum delays for simulation. See <u>Describing</u> <u>Module Paths</u> for more information about using the +maxdelays plus option.

+max_err_count+

The +max_err_count plus option specifies the maximum number of error messages that Verilog-XL displays or prints during compilation. The default is 200. If the number of errors exceeds the specified number during compilation, the following error message is issued and compilation stops:

Error! Maximum error count <number> exceeded. Please use +max_err_count+<num> to modify maximum error count

+mindelays

The +mindelays plus option selects the minimum delays for simulation. See <u>Describing</u> <u>Module Paths</u> for more information about using the +mindelays plus option.

+multisource_int_delays

The +multisource_int_delays plus option provides transport delay functionality and full pulse control for all multi-source interconnect delays. This option also lets you specify unique source/load delays. MIPDs (module input port delays) are inserted on all single-source nets. Therefore, this plus option affects only nets with more than one source.

Using the +multisource_int_delays plus option with the +transport_int_delays plus option provides transport delay functionality and full pulse control for interconnect delays with one or more sources, using these two options also lets you delay each source-to-load path. For more information about interconnect delays, see <u>Chapter 16</u>, Interconnect Delays.

+neg_tchk

The <code>+neg_tchk</code> plus option enables negative timing check arguments in the <code>\$recovery</code> and <code>\$setuphold</code> timing checks.

See <u>\$recovery</u> for details on the *\$recovery* timing check and <u>\$setuphold</u> for details on the *\$setuphold* timing check.

+nolibcell

The +nolibcell plus option disables the automatic tagging of library modules as cells.

+notimingchecks

The +notimingchecks plus option disables timing checks.

+no_cancelled_e_msg

The +no_cancelled_e_msg plus option suppresses the warning message that accompanies the occurence of an e state when an event is cancelled and the display of cancelled schedules mode is selected. It does not, however, suppress the e state itself.

+no_charge_decay

The +no_charge_decay plus option causes Verilog-XL to ignore all the delay specifications for charge decay, as well as all the `default_decay_time compiler

directives with a numerical argument in the source description. See <u>trireg Net Charge</u> <u>Decay</u> for more information on trireg net charge decay.

+no_cond_event_error

The +no_cond_event_error plus option causes the following warning message to be issued if you attempt to condition an event in a timing check with more than one signal. However, simulation continues despite this warning.

Warning! Ignoring illegal conditioned event in timing check

If you attempt to condition an event in a timing check with more than one signal without the +no_cond_event_error plus option, Verilog-XL issues the following error message and simulation is halted:

Error! Illegal conditioned event in timing check

+no_notifier

The +no_notifier plus option prevents notifiers from changing their value to indicate timing check violations. Notifiers are registers passed as arguments for timing checks. See <u>Using Notifiers for Timing Violations</u> for information about notifiers.

+no_pulse_int_backanno

The +no_pulse_int_backanno plus option prevents the PLI backannotation of pulse limits for interconnect delays. Only one warning message is issued on the first attempt. See <u>Controlling MIPD and S/MITD Creation</u> for information on pulse control for interconnect delays.

+no_pulse_msg

The +no_pulse_msg plus option disables messages generated by the +pulse_e/n plus option. See <u>Specifying Global Pulse Control on Module Paths</u> for more information on the +no_pulse_msg plus option.

+no_show_cancelled_e

The +no_show_cancelled_e plus option disables the display of cancelled schedules. See <u>Pulse Filtering for Module Path Delays</u> for information about pulse filtering and cancelled schedules.

+no_speedup

The +no_speedup plus option disables the default acceleration of behavioral constructs. See <u>"Behavioral Performance Improvements" on page 307</u> for information about the default acceleration of behavioral constructs.

+no_tchk_msg

The $+no_tchk_msg$ plus option prevents timing check violation messages from displaying or printing.

+nowarn

The +nowarn plus option disables a specified type of warning by concatenating the warning's code to the end of the plus option. For example, to disable the Verilog-TFNPC warning message, issue the +nowarnTFNPC plus option. You can specify multiple +nowarn plus options.

+noxl

The +noxl plus option disables the current XL algorithm and applies the XL algorithm that existed for Verilog-XL Version 1.6c and previous versions.

Note: The -a (accelerate) option invokes the current XL algorithm for the entire design.

+password

The +password plus option enables you to launch a unique simulation run from your command-line, irrespective of the fact that the GUI is invoked or whether it is in the post processing mode (+ppe).

+pathpulse

The +pathpulse plus option enables the PATHPULSE\$ specparam, which narrows the scope of module path pulse control, to a specific module or to particular paths within modules. See <u>Specifying Local Pulse Control for Module Paths</u> for more information on pulse control for specific modules and module paths.
+ppe

The +ppe plus option enables the post processing mode from the command line directly once the simulator completes simulating the design. In PPE mode, you have access to the same tools as you would have during an interactive session.

For Verilog-XL, invoke SimVision with the +ppe option.

verilog +ppe source_filenames

+pre_16a_paths

The +pre_16a_paths plus option simulates SDPD paths as if their conditional expressions are always true, which is the default behavior in Verilog-XL Version 1.6a and in previous versions.

Choosing to simulate SDPDs as unconditional paths can introduce the following variations in a simulation:

- suppression of some error checking introduced in Verilog-XL 1.6a
- different results when multiple paths connect an input and an output

+profile

The +profile plus option enables you to use the behavior profiler in Turbo and Twin Turbo modes, in which the behavior profiler is disabled by default to increase performance.

+protect

The +protect plus option enables the protection of those regions in a source description that are bounded by `protect and `endprotect compiler directives. See <u>Chapter 9</u>, <u>"Source Protection,"</u> for more information.

+pulse_e/n and +pulse_r/m

The $+pulse_e/n$ plus option sets module output paths to e (error state) and the module path output pulses pass through. See <u>Specifying Global Pulse Control on Module Paths</u> for more information on the $+pulse_e/n$ plus option. The $+pulse_r/m$ plus option sets a limit for rejecting output pulses. See <u>Specifying Global Pulse Control on Module Paths</u> for more information on the $+pulse_r/m$ plus option.

+pulsestyle_ondetect

The +pulsestyle_ondetect plus option enables the on-detect style of pulse filtering that is described in <u>Pulse Filtering for Module Path Delays</u>.

+pulsestyle_onevent

The +pulsestyle_onevent plus option enables the on-event style of pulse filtering that is described in <u>Pulse Filtering for Module Path Delays</u>.

+pulse_int_e/n and +pulse_int_r/m

The +pulse_int_e/n plus option sets the module output paths for interconnect delays to e (error state) and lets module path output pulses pass through. The +pulse_int_r/m plus option sets a limit for rejecting output pulses for interconnect delays. For more information, see <u>Specifying Global Pulse Control on Module Paths</u>.

+save_twin_turbo

The <code>+save_twin_turbo</code> plus option enables the <code>\$save</code> system task, which saves simulation checkpoint files if you are running in Twin Turbo mode. Using the <code>+save</code> plus option increases memory usage by approximately 10%.

Note: You do not need the +save_twin_turbo plus option to restart a simulation using a previously saved file.

+sdf_cputime

The +sdf_cputime plus option logs the number of central processing unit (CPU) seconds that it takes for the SDF Annotator to complete the annotation. This CPU time is written to the log file.

Note: For complete information about the SDF Annotator, see the SDF Annotator Guide.

+sdf_error_info

The +sdf_error_info plus option displays PLI error messages.

Note: SDF Annotator errors are called fatal or nonfatal errors. Fatal errors cause the SDF Annotator to stop. Nonfatal errors do not stop the SDF Annotator, but cause it to skip the

action that caused the error. An example of a nonfatal error is when a condition specified in the SDF file cannot be matched in the Verilog description.

+sdf_file<filename>

The <code>+sdf_file</code> plus option with a corresponding appended filename (no space in between) specifies the SDF file that the SDF Annotator uses. For example, <code>+sdf_filemyfile.sdf</code>. This plus option overrides the file specified as an argument to the <code>\$sdf_annotate</code> system task. See the SDF Annotator Guide for more information.

Note: The <code>+sdf_file</code> option will only work if the <code>\$sdf_annotate</code> task is also specified in the design file.

+sdf_ign_timing_edge

Note: This option is applicable for Verilog-XL only.

The <code>+sdf_ign_timing_edge</code> plus option annotates the last edge without any extra overheads for <code>SETUP</code>, <code>HOLD</code> and <code>SETUPHOLD</code>. By default, Verilog-XL generates an error message during annotation if the verilog file contains a timing check without an edge and the sdf file contains an edge. To facilitate annotation, you can use the <code>+sdf_ign_timing_edge</code> plus option. Consider the example given below.

Verilog File:

```
$setup(data, clk, 2);
$hold(clk, data, 1);
```

SDF File:

```
(SETUP (posedge data) clk (3))
(SETUP (negedge data) clk (3))
(HOLD (posedge data) clk (2))
(HOLD (negedge data) clk (2))
```

In this example, the timing check signal data does not contain an edge in the Verilog file but contains both a posedge and a negedge in the SDF file. Using the

+sdf_ign_timing_edge plus option, the timing check signal data will first be annotated with a posedge and then a negedge. In effect, the last edge defined is annotated.

+sdf_nocheck_ celltype

The +sdf_nocheck_celltype plus option disables celltype validation between the SDF Annotator and the Verilog description. By default, the SDF Annotator validates the type

specified in the CELLTYPE construct against the type of the cell instance that is specified in the INSTANCE keyword construct. See the *SDF Annotator Guide* for more information

+sdf_no_errors

The +sdf_no_errors plus option disables error messages from the SDF Annotator.

+sdf_nomsrc_int

If you have no multisource interconnect transport delays (MITDs) in the design, use the <code>+sdf_nomsrc_int</code> plus option, which will increase performance and reduce memory consumption. This is because Verilog-XL normally maintains information about the various MITDs that map to the same port, causing the SDF Annotator to resolve delays prior to annotating the port. The SDF Annotator provides three resolution functions: AVERAGE, MAXIMUM, and MINIMUM. For the SDF Annotator to correctly resolve the delays, it must maintain the interconnect information until the end of annotation.

+sdf_no_warnings

The +sdf_no_warnings plus option disables warning messages from the SDF Annotator.

+sdf_split_two_timing_check +sdf_splitvlog_suh +sdf_splitvlog_recrem

Note: These options are applicable for Verilog-XL only.

SDF Annotator attempts to match the one-timing checks (SETUP, HOLD, REMOVAL, and RECOVERY) to their corresponding one-timing checks in the Verilog source. If no match is found, then the SDF annotator splits the two-timing checks (\$setuphold and \$recrem) in the Verilog source into corresponding one-timing checks and attempts to match. For example, \$setuphold is split into \$setup and \$hold and then matched to SETUP and HOLD.

You can split SDF two-timing checks (SETUPHOLD and RECREM) using the +sdf_split_two_timing_check plus option into their corresponding one-timing checks. The conditions specified with SETUPHOLD and RECREM are ignored after the split.

If you have used +sdf_split_two_timing_check plus option and no two-timing checks are found, the SDF Annotator reports errors in terms of corresponding split timing checks.

The options +sdf_splitvlog_suh and +sdf_splitvlog_recrem can be used to perform splitting of SETUPHOLD only and RECREM only respectively.

+sdf_verbose

The +sdf_verbose plus option instructs the SDF Annotator to write detailed information about the backannotation process to the annotation log file.

+show_cancelled_e

The <code>+show_cancelled_e</code> plus option displays cancelled schedules. See <u>Pulse Filtering for</u> <u>Module Path Delays</u> for information about pulse filtering and cancelled schedules.

+splitsuh

The +splitsuh plus option disables the default splitting of \$setuphold timing checks into \$setup and \$hold during compilation. By not splitting the timing checks, you can have a single handle for each \$setuphold check, and you can therefore simultaneously change both types of timing delays using the PLI acc_replace_delays routine.

Note: Splitting timing checks provides compatibility with applications (such as the SDF Annotator) that handle <code>\$setup</code> and <code>\$hold</code> timing checks separately. If <code>\$setuphold</code> checks are split, two calls to the routine (one to change the setup limit and one to change the hold limit) are required, and you cannot perform a single consistency check on the pair of limits.

+switchxl

The +switchxl plus option invokes the Switch-XL algorithm to accelerate the simulation of bidirectional switches. See <u>Chapter 8, "Switch-Level Simulation,"</u> for more information.

Note: The Switch-XL algorithm expects references to the terminals of switches to be expanded. Therefore, references to the terminals of switches cannot be references to register bit-selects when the +switchxl plus option is used.

+sxl_keep_all

The +sxl_keep_all plus option ensures that the Switch-XL algorithm does not remove any nets from channel-connected switch networks during compilation. See <u>"Optimization of Switch Networks" on page 152</u> for more information.

+sxl_keep_declared

The +sxl_keep_declared plus option ensures that the Switch-XL algorithm does not remove any explicitly declared nets from channel-connected switch networks during compilation. See <u>"Optimization of Switch Networks" on page 152</u> for more information.

+sxl_keep_minimum

The +sxl_keep_minimum plus option ensures that the Switch-XL algorithm does not remove any net that it does not need for some other purpose from a channel-connected switch network. See <u>"Optimization of Switch Networks" on page 152</u> for more information.

+sxl_unidirect

The +sxl_unidirect plus option converts all unidirectional switches in a source description to the turn on/turn off delay model. Use this plus option when you want to invoke Switch-XL, and when you do not want unidirectional switches with two kinds of delay timing models. See <u>"Conversion of Channel Delay to Turn-On/Turn-Off Delay"</u> on page 149 for more information.

+trace_twin_turbo

The $+trace_twin_turbo$ plus option displays the trace results in Twin Turbo mode whenever the settrace system task, the -t option, or the single step (,) command is used.

+transport_int_delays

The +transport_int_delays plus option provides transport delay functionality and full pulse control for interconnect delays with one or more sources. See <u>Chapter 16, Interconnect</u> <u>Delays</u> for details on interconnect delays.

Using the +transport_int_delays plus option with the +multisource_int_delays plus option provides transport delay functionality and full pulse control for interconnect delays with one or more sources using these two options also lets you have unique delays for each source-to-load path.

+transport_path_delays

The +transport_path_delays plus option enables full transport delay functionality for module path delays.

In Verilog-XL version 2.0 and earlier versions, module path delays may have limited transport delay functionality. See <u>"Pulse Handling in Verilog-XL 2.0 and Earlier Versions" on page 316</u> for a discussion of this limited implementation of transport delay functionality in path delays.

+turbo

The +turbo plus option improves behavioral simulation performance over the default Turbo mode by disabling the behavior profiler and the end-of-simulation event count. This plus option can also be used with the +twin_turbo plus option. See <u>Chapter F, "Verilog-XL</u> <u>Turbo and Twin Turbo Options,"</u> for details on Turbo and Twin Turbo.

+turbo+2

The +turbo+2 plus option increases behavioral simulation performance over the +turbo level by optimizing assignments and by converting scalar nets to compact nets if this conversion accelerates the simulation. This plus option can be used with the +twin_turbo plus option. See <u>Chapter F, "Verilog-XL Turbo and Twin Turbo Options,"</u> for details on Turbo and Twin Turbo.

+turbo+3

The +turbo+3 plus option increases behavioral simulation performance over the +turbo+2 level by evaluating the right-hand sides of assignments only when the assignments actually occur. This plus option can be used with the +twin_turbo plus option. See <u>Chapter F,</u> <u>"Verilog-XL Turbo and Twin Turbo Options,"</u> for details on Turbo and Twin Turbo.

+twin_turbo

The +twin_turbo plus option generates and uses compiled code for processing behavioral constructs in the selected Turbo mode (+turbo, +turbo+2, or +turbo+3). See <u>Chapter F,</u> <u>"Verilog-XL Turbo and Twin Turbo Options,"</u> for details on Turbo and Twin Turbo.

+typdelays

The +typdelays plus option selects typical delays for simulation. See <u>Specifying Transition</u> <u>Delays on Module Paths</u> for more information on the +typdelays plus option.

+vra

The +vra plus option creates a TMS file (an ASCII representation of the lexical and hierarchical information for a design) and places it in the SHM directory structure for the design. The Verilog-XL Results Analyzer (VRA) analyzes the results of your batch jobs using the TMS file.

+x_transport_pessimism

The +x_transport_pessimism plus option causes an x state for the output in cases where timing dilemmas are caused by event cancellation that occur when using transport delays or when using the accu_path delay selection algorithm. See <u>Simulating Distributed</u> <u>Delays as Inertial and Transport Delays</u> for more information on transport delays and +x_transport_pessimism.

User-Definable Command-Line Arguments

This feature allows you to test for command-line arguments within your Verilog HDL source description or within user tasks linked through the PLI. This provides you with a way to change the behavior of the simulation at invocation time.

You can only check for the plus (+) options; you cannot check for the standard command line (-) options. Each of these arguments is preceded by a plus (+) character. You can specify any string preceded by a plus on the command line and test for it in the source description or user tasks, thus creating a new option.

Testing for Plus Arguments

There are two ways to test for the presence of a plus argument on the command line. One is through a system function, \$test\$plusargs, and the other is through a PLI interface routine, mc_scan_plusargs.

The system function \$test\$plusargs takes one string argument and returns true (1) if the string appears at the beginning of one or more plus arguments on the command line, and false (0) if not. For example, consider the following command line:

```
verilog sourcel.v +reset
```

The Verilog HDL source description contained in file <code>source1.v</code> can test for the plus argument as shown in the following example:

```
initial
if ($test$plusargs("reset"))
        begin
            reset = 1;
            #100 reset = 0;
            end
else
            reset = 0;
```

The PLI interface routine mc_scan_plusargs allows the plus options to modify the behavior of routines linked through the PLI. The syntax of this routine is the following:

```
char *mc_scan_plusargs(startarg)
char *startarg;
```

This routine scans all plus arguments which have been specified on the command line. "startarg" is matched against the first characters of the plus options on the command line. If a match is found, then the routine returns a pointer to a string consisting of the remaining characters of the plus option. If an exact match is found and there are no remaining characters, a pointer to the C string terminator is returned. If no match is found, then the routine returns null.

For example, consider the following command line:

```
verilog source1.v +size64
```

A call to mc_scan_plusargs can detect the size specified in the plus argument as follows:

```
if (size= mc_scan_plusargs("size"))
    printf("size is %s\n",size);
```

Lack of Command-Line Syntax Checking

There is no way to check for syntax errors in command-line plus options. Verilog-XL allows any string that is preceded by a plus (+) character on the command line. If you make a syntax error, Verilog-XL simulates as though the incorrect plus option has not been specified. Because of this, you should use care when entering command-line plus options. Cadence suggests that you print out information regarding user-defined plus arguments.

Note: If you specify more than one of the same plus option when using the PLI mc_scan_plusargs routine to find the remaining characters of a plus option, only the first plus option occurrence is detected.

Compiler Directives

This section describes the directives that let you control what happens when Verilog-XL compiles and simulates a description. Because these directives are described in detail with the specific features that they control, this section gives only a brief description of each directive and tells you where to find more detailed information about it.

All Verilog-XL compiler directives are preceded by the accent grave (`) character. This character is also known as a "tick" character. An example follows:

`default_nettype trireg

'accelerate and 'noaccelerate

The `noaccelerate directive is pertinent to the default XL algorithm. `noaccelerate causes Verilog-XL to stop applying the XL algorithm to the modules following the directive. The `accelerate directive causes Verilog-XL to start applying the XL algorithm again. These directives can only be specified outside of module definitions, but you can specify as many of these directives as you want in your source description. For more information about these directives, see <u>Appendix C, "Maximizing Default Acceleration."</u>

'autoexpand_vectornets

The `autoexpand_vectornets directive lets the compiler expand vectors as needed to form proper connections between elements of the description. This is the default. The details of vector expansion are in <u>Port Collapsing</u> and <u>Port Connection Rules</u>.

'celldefine and 'endcelldefine

The directives `celldefine and `endcelldefine tag module instances as cell instances. Cells are used by certain PLI access routines for applications such as delay calculation.

All module instances that appear between `celldefine and `endcelldefine are treated as cell instances, particularly by PLI access routines that recognize cells. An example is acc_next_cell which scans the design hierarchy for cells. Macro modules, which are expanded inline, are not marked as cell instances. See <u>Macro Modules</u> for more information on macro modules.

Note: You do not need to apply these directives to cells extracted from libraries because Verilog-XL automatically tags them as cells unless invoked with the command-line option +nolibcell.

The following example shows how to use the `celldefine and `endcelldefine directives:

```
`celldefine
// GDA-Series AN3
// Three input and-nor
// Pin-to-pin delay model; times in 10ps
module AN3(I1,I2,I3,Y);
input I1, I2, I3;
output Y;
    // netlist
    and g1(m1,I1,I2);
    nor g2(Y,I3,m1);
    // specify block
    specify
        // path delays
        (I1, I2 => Y) = (15:50:88, 24:80:140);
        (I3 => Y) = (10:35:62,16:55:97);
        // delay constants x100
                                    Kd I1\$Y = 10, Kd I3\$Y = 8;
        parameter
        // loading and driving factors
        parameter ilf_I1 = 2, ilf_I2 = 2;
        parameter ilf_I3 = 1, odf_Y = 10;
    endspecify
endmodule
`endcelldefine
```

You should pair each `celldefine with an `endcelldefine. More than one of these pairs may appear in a single source description.

Refer to the *PLI 1.0 User Guide Reference* and the *VPI User Guide and Reference* for more information about access routines that recognize cells and the use of cells in delay calculation.

'default_decay_time

This compiler directive allows you to specify the decay time for triregs whose declarations do not include a decay time specification. This compiler directive applies to all of the triregs in all of the modules that follow it in the source description. The `default_decay_time compiler directive must include an argument that specifies the charge decay time. You can enter this argument as a constant integer, a real number, or the character string infinite. The character string infinite specifies no charge decay in the triregs that follow.

The following example shows a use of the `default_decay_time compiler directive with a numerical argument:

```
`default_decay_time 100
```

In this example, all triregs without a decay time specification in all the modules that follow this `default_decay_time compiler directive have a charge decay time of 100 time units.

The following example shows a use of the <code>`default_decay_time</code> compiler directive with the character string infinite argument:

```
`default_decay_time infinite
```

In the previous example, charge decay does not occur in the triregs without a decay time specification in the modules that follow this `default_decay_time compiler directive.

'default_nettype

The `default_nettype directive controls the net type for implicit net declarations. It can be used only outside of module definitions. It affects all modules that follow it, even across source file boundaries. Multiple `default_nettype directives are allowed. The most recent directive encountered controls the type of nets that are implicitly declared. The following net types can be specified:

wire	tri	tri0
wand	triand	tri1
wor	trior	trireg

See <u>Implicit Declarations</u> and <u>Implicit Net Declarations</u> for more information about implicit net declarations.

'default_rswitch_strength

This compiler directive specifies the default drive strength of resistive switches in simulations that invoke the Switch-XL algorithm. See <u>"Switch-XL Default Charge and Drive Strengths" on page 159</u> for more information on the <code>`default_rswitch_strength</code> compiler directive.

'default_switch_strength

This compiler directive specifies the default drive strength of switches in simulations that invoke the switch-XL algorithm. See <u>"Switch-XL Default Charge and Drive Strengths" on page 159</u> for more information on the <code>`default_switch_strength</code> compiler directive.

'default_trireg_strength

This compiler directive specifies the default charge strength of triregs in simulations that invoke the switch-XL algorithm. See <u>"Switch-XL Default Charge and Drive Strengths" on page 159</u> for more information on the `default_trireg_strength compiler directive.

'define

This compiler directive allows you to create macros for text substitution (see <u>Text</u> <u>Substitutions</u>) and macros to trigger the `ifdef compiler directive. You can use text macros both inside and outside module definitions.

'delay_mode_distributed

This compiler directive specifies the distributed delay mode for all modules that follow it in the source description. See <u>Setting a Delay Mode</u> for more information on the `delay_mode_distributed compiler directive.

'delay_mode_path

This compiler directive specifies the path delay mode for all modules that follow it in the source description. See <u>Setting a Delay Mode</u> for more information on the 'delay_mode_path compiler directive.

'delay_mode_unit

This compiler directive specifies the unit delay mode for all modules that follow it in the source description. See <u>Setting a Delay Mode</u> for more information on the `delay_mode_unit compiler directive.

'delay_mode_zero

This compiler directive specifies the zero delay mode for all modules that follow it in the source description. See <u>Setting a Delay Mode</u> for more information on the 'delay_mode_zero compiler directive.

'expand_vectornets and 'noexpand_vectornets

This `expand_vectornets directive causes all vector nets to be expanded into a group of scalar nets, except those with the keyword vectored in their declarations.

The `noexpand_vectornets directive causes no expansion to take place except where explicitly specified by the keyword scalared in a vector net declaration.

'ifdef, 'else, and 'endif

You use the conditional compilation (`ifdef, `else, and `endif) directives to optionally include lines of a Verilog HDL source description during compilation.

The `ifdef compiler directive checks for the definition of a variable name either in the source code or on the command line. If the variable name is defined, Verilog-XL includes the lines of the source description that follow the directive. This way, you can optionally include lines of code by specifying condition(s) that must be met.

There are two options for defining `ifdef variables. You can define and use a compiler directive (`define) or you can use a command-line plus argument (+define+) to define a text macro. See <u>"Conditional Compilation"</u> on page 235 for more information.

'include

Use the <code>`include</code> compiler directive when you want to insert the entire contents of a source file in another file during Verilog-XL compilation. Verilog-XL compiles as though the contents of the include source file appear in place of the <code>`include</code> command. You can use the <code>`include</code> compiler directive to include global or commonly-used definitions and tasks without encapsulating repeated code within module boundaries. See <u>"File Inclusion" on page 241</u> for more information on the <code>`include</code> compiler directive.

'pre_16a_paths and 'end_pre_16a_paths

Use the `pre_16a_paths and `end_pre_16a_paths directives to turn on and turn off the functionality of conditional paths that is characteristic of Verilog-XL prior to version 1.6a. In these versions, Verilog-XL treats conditional paths as if their conditional expressions are always true. Veritime observes a path's conditional state.

Thus, libraries written for use with both Veritime and Verilog-XL versions prior to 1.6a contain conditional paths, and users of those libraries have become accustomed to performing

Verilog-XL simulations in which paths described as conditional always simulate as if their conditions are true.

The `resetall compiler directive, like `end_pre_16a_paths, also turns off the functionality of conditional paths characteristic of prior versions.

The effect of the <code>`pre_16a_paths</code> compiler directive crosses file boundaries until Verilog-XL arrives at the <code>`end_pre_16a_paths</code> or <code>`resetall</code> compiler directive.

Note: Choosing to simulate SDPDs as unconditional paths can introduce the following variations in a simulation:

- □ suppression of some error checking introduced in Verilog-XL 1.6a
- different results when multiple paths connect an input and an output

'protect and 'endprotect

Use the `protect and `endprotect directives to mark the regions in a source description that you want Verilog-XL to protect when you invoke it with the +protect command-line option. See <u>"The 'protect and 'endprotect Compiler Directives</u>" on page 166 for more information on the `protect and `endprotect directives.

'protected and 'unprotected

Use the `protected and `unprotected directives before the module name and after the endmodule keyword when you do not want Verilog-XL to find the module definition. See <u>""protected and 'unprotected</u>" on page 121 for more information on the `protected and `unprotected directives.

'remove_gatenames and 'noremove_gatenames

The `remove_gatenames directive is very similar to `remove_netnames. It causes Verilog-XL to eliminate any gate instance names that have been specified in modules defined between the directives `remove_gatenames and `noremove_gatenames. The operation of these directives is detailed in <u>Gate and Net Name Removal</u>. These directives can only be specified outside module definitions.

'remove_netnames and 'noremove_netnames

The `remove_netnames directive causes Verilog-XL to eliminate the names of all nets from the data structure. Its operation is described in <u>Gate and Net Name Removal</u>. This directive cannot be used if it is necessary to refer to nets by hierarchical name, either from within the source description or interactively. This directive is incompatible with named port connections.The directive `noremove_netnames causes Verilog-XL to stop eliminating the names. These directives must be specified outside of modules. All the modules between `remove_netnames and `noremove_netnames are affected.

'resetall

This compiler directive resets all compiler directives, except `define compiler directive, that are active when it is encountered during compilation to their default values. This is useful for ensuring that only those directives that are desired in compiling a particular source file are active. The recommended usage is to place `resetall at the beginning of each source text file, followed immediately by the directives desired in that file. This directive is particularly important in library files and library directory files.

'switch default

This compiler directive enables the default algorithm for simulating networks composed of the bidirectional switches that follow the directive. A `switch compiler directive that enables another algorithm for switch-level simulation cuts off the effect of the `switch default compiler directive. See <u>Chapter 8</u>, "Switch-Level Simulation," for a unified discussion of the `switch compiler directive.

'switch XL

This compiler directive enables the Switch-XL algorithm for simulating networks composed of the unidirectional and bidirectional switches that follow the directive. A `switch compiler directive that enables another algorithm for switch-level simulation cuts off the effect of the `switch XL compiler directive. See <u>Chapter 8</u>, "Switch-Level Simulation," for a unified discussion of the `switch compiler directive.

'timescale

This directive specifies the time unit and time precision of the modules that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values. The time precision specifies the place value to which Verilog-XL rounds time values. The

rounded values that Verilog-XL uses are accurate to within the unit of time specified as the time precision. The details of the timescale constructs are in <u>Chapter 17, Timescales</u>

'unconnected_drive and 'nounconnected_drive

These directives cause unconnected input ports to automatically be pulled up (if pull1 is specified) or down (if pull0 is specified) instead of floating to the high impedance value z. Inputs are pulled up or down in all the modules between the directives 'unconnected_drive and 'nounconnected_drive. These directives must be specified outside modules only.

'undef

This directive lets you remove any definition of a text macro created by the `define compiler directive or the +define+ command-line plus option.

The `undef compiler directive can be used to undefine a text macro that you use in more than one file.

An example of this is as follows:





Note: If you use `undef to undefine a name not previously defined, Verilog-XL displays no error messages or warnings.

The `undef compiler directive must be followed by a text macro name. Otherwise compilation results in a syntax error.

Once you have undefined a text macro name, that name no longer shows up in the decompilation listing as shown in the following example.

Before Decompilation

After Decompilation



'uselib

This directive enables you to specify the paths that the compiler searches to find definitions of instantiations whose definitions are not part of the design description. These path specifications can include library files, library directories, and the extensions for the names of the files in library directories. See <u>"The Standard Library Management Scheme" on page 102</u> for a discussion of this compiler directive.

Conditional Compilation

You use the conditional compilation (<code>`ifdef</code>, <code>`else</code>, and <code>`endif</code>) compiler directives to optionally include lines of a Verilog HDL source description during compilation. The <code>`ifdef</code> compiler directive checks for the definition of a variable name either in the source code or on the command line. If the variable name is defined, Verilog-XL includes lines of the source description that follow the directive. This way, you can optionally include lines of code by specifying condition(s) that must be met. Situations in which you can use the <code>`ifdef</code>, <code>`else</code>, and <code>`endif</code> compiler directives include:

- selecting different representations of a module such as behavioral, structural, or switch level
- choosing different timing or structural information
- selecting different stimulus for a given run of Verilog-XL

Syntax

The <code>`ifdef</code>, <code>`else</code>, and <code>`endif</code> compiler directives have the following syntax:

The <text_macro_name> is a Verilog HDL identifier. The <first_group_of_lines> and the <second_group_of_lines> are any parts of a Verilog HDL source description. The `else compiler directive and the <second_group_of_lines> are optional.

How 'ifdef, 'else, and, 'endif Work

The `ifdef, `else, and `endif compiler directives work in the following manner:

- When Verilog-XL encounters an `ifdef, it tests the <text_macro_name > to see if it is defined as a text macro name using either a `define within the Verilog HDL source description or the +define+ command-line plus argument (entered interactively).
- If you define the <text_macro_name>, Verilog-XL compiles the <first_group_of_lines> as part of the source description. If there is an `else compiler directive, Verilog-XL ignores the <second_group_of_lines>.

The following example shows the `ifdef, `else, and `endif compiler directives in a module:

```
module and_op (a, b, c);
output a;
input b, c;
    'ifdef behavioral
    wire a = b & c;
    'else
        and (a,b,c);
    'endif
endmodule
```

Note: Verilog-XL does not check the syntax for any group of lines that the compiler ignores. However, even though Verilog-XL does not check the syntax of this text, the text must follow the Verilog-XL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

Nesting the 'ifdef, 'else, and 'endif Compiler Directives

You can nest the `ifdef, `else, and `endif compiler directives as shown in the following example:

```
module test(out);
output out;
'define wow
'define nest_one
'define second nest
'define nest_two
    `ifdef wow
        initial $display("wow is defined");
        'ifdef nest one
        initial $display("nest_one is defined");
             `ifdef nest two
                initial $display("nest_two is defined");
            `else
                initial $display("nest_two is not defined");
            `endif
        `else
            initial $display("nest one is not defined");
        `endif
    `else
        initial $display("wow is not defined");
        'ifdef second nest
            initial $display("nest_two is defined");
        `else
            initial $display("nest two is not defined");
        `endif
    `endif
endmodule
```

Defining Variable Names to Control Conditional Compilation

There are two options for defining `ifdef variables. You can use either a compiler directive (`define) or a command-line plus argument (+define+) to define a text macro. The +define+ command-line plus argument (+define+) can define an empty macro, which is discussed in this section, or a macro string, which is discussed in <u>"+define+" on page 207</u>.

The 'define compiler directive

The `define compiler directive allows you to create macros for text substitution. You can use text macros both inside and outside of module definitions. When Verilog-XL encounters the `ifdef compiler directive, it checks to see if its variable name matches a text macro name in a `define compiler directive. The syntax for this usage of the `define compiler directive is as follows:

`define <text_macro_name> [<macro_contents>]

For more information on the `define compiler directive, see <u>Text Substitutions</u>. The following is an example of the `define compiler directive within the Verilog HDL source description:

'define sun3

Note: Verilog-XL does not perform text macro substitution inside `ifdef blocks that are skipped.

In Verilog-XL, an accent grave (`) must precede a text macro name whenever you use it in the source description. The accent grave instructs Verilog-XL to substitute the macro text in place of the macro name. However, the `ifdef compiler directive does not allow macro substitution for a variable name. Do not precede the variable name(s) that the `ifdef compiler directive tests with the accent grave character or the conditional compilation may not work properly.

In the following example, when there is no `define for the `ifdef variable name rega and there is a text macro that redefines the `else compiler directive, Verilog-XL does not compile any of the statements in the `ifdef, `else, and `endif compiler directive block. Verilog-XL does not print either of the display statements.

```
//test.v
module test(out);
output out;
// no 'define for rega
'define my_else 'else
    `ifdef reqa
                                           // Verilog-XL does not include
        initial $display("part 1");
                                           // these statements during
                                           // compilation because you
    `my_else
        initial $display("part 2");
                                           // define a macro to be the
    `endif
                                           // 'else directive.
endmodule
```

The +define+ command-line plus argument

The +define+ command-line plus argument defines variable names as an empty text macro throughout the Verilog-XL compilation process. An empty text macro is a text macro you define as empty. The syntax for the +define+ command-line plus argument is as follows:

+define+<text_macro_name1>+<text_macro_name2>+..+<text_macro_nameN>

The <text_macro_name1>, <text_macro_name2>, and <text_macro_nameN> are the Verilog-XL identifiers you want to define. The plus sign (+) is a delimiter between each variable name.

The following is an example of a +define+ command-line plus argument. A single +define+ is the command line equivalent to the `define compiler directive as shown in the following example:

```
+define+sun3 // command-line plus argument `define sun3 // compiler directive
```

However, you can have multiple +define+ arguments on a command line, as follows:

+define+sun4+version3 +define+structural+sun4+

In this example, Verilog-XL defines the three variable names sun4, version3, and structural during compilation.

Important

When parsing the +define+ arguments for variable names, Verilog-XL assumes all characters between the two + characters are part of the variable name. There is no error checking.

If you define the same macro name differently in a command line +define+ option and a `define compiler directive, the command-line option overrides the compiler directive. See <u>"+define+" on page 207</u> for more information about such an override.

The Predefined Symbol for Conditional Compilation

Verilog-XL predefines the symbol verilog to create a standard and save you the trouble of defining a symbol with the 'define compiler directive or the +define+ command-line plus argument.

You can use the predefined verilog symbol and the `ifdef, `else, `endif, and `undef compiler directives to include code only in a Verilog-XL simulation and exclude that code from simulations involving other Veritools.

The existence of the verilog predefined symbol is convenient in writing libraries, in which a single UDP can need different versions to perform with different Veritools.



Libraries include the verilog symbol, so using the `undef compiler directive can yield unexpected and undesirable results.

Decompiling Source Descriptions

The <code>`ifdef, `else</code>, and <code>`endif</code> compiler directives do not appear in decompilation when you use a <code>\$list</code> system task to decompile the source description. When decompiling a module that contains the <code>`ifdef, `else</code>, and <code>`endif</code> compiler directives, the text that Verilog-XL includes during compilation is shown in the <code>\$list</code> output. However, the line numbers preceding the Verilog HDL statements correspond to the line numbers in the original file, as illustrated in the following example.

Original File	Decompilation of Module Test
`define foo module test;	2 module test; 3 initial
<pre>initial begin 'ifdef foo \$display("foo is defined"), 'else \$display("foo is not defined"); 'endif</pre>	3 begin 6 \$display("foo is defined"); 11 * \$list(test);
	12 end 13 endmodule
<pre>\$list(test); end endmodule</pre>	

Conditional Compilation Error Messages

When Verilog-XL finds a syntax error during compilation, an error message displays the line number of the sytax error from the original file.

When you use the `ifdef compiler directive without a text macro, Verilog-XL does the following:

■ Verilog-XL displays a syntax error message.

- Verilog-XL ignores the remainder of the current line.
- The compiler does not compile the <first_group_of_lines>, but it does compile the <second_group_of_lines>.

Conditional Compilation Source Protection

The <code>`ifdef, `else</code>, and <code>`endif</code> compiler directives affect source protection first when Verilog-XL encounters the compiler directives during source protection and again when Verilog-XL executes the protected source.

The following list describes how the <code>`ifdef</code>, <code>`else</code>, and <code>`endif</code> compiler directives affect source protection:

- Verilog-XL encodes the `ifdef, `else, and `endif compiler directives as well as the first and second groups of lines in your source description file.
- Verilog-XL copies the `ifdef, `else, and `endif compiler directives outside of the protected regions as unprotected source code to the protected file.

The following list describes what happens when Verilog-XL compiles the source protected `ifdef, `else, and `endif compiler directives:

- When Verilog-XL compiles the protected source, Verilog-XL evaluates the `ifdef compiler directive statement.
- Verilog-XL evaluates the condition and compiles either the first or second group of lines.

The following example shows a module before Verilog-XL compiles the source file:

```
// test.v
'protect
module test;
reg in1,in2;
'define foo;
    `ifdef foo
         'resetall
        initial $display("foo is defined\n");
    `endif
    and (out, in1, in2);
    initial $monitor($stime,"out=%b in1=%b in2=%b\n",out,in1,in2);
    initial #100 $finish;
    initial
    begin
        in1=0; in2=0;
        #10 in1=1;
        #10 in2=1;
        #10 $finish(2);
    end
```

```
endmodule
'endprotect
module and_op (a, b, c);
output a;
input b, c;
    'ifdef behavioral
    wire a=b && c;
    'else
        and (a, b, c);
    'endif
endmodule // test.v
```

The following example shows the effect that source protection has on the `ifdef, `else, and `endif compiler directives in the previous example:

```
//test.v
`protected
Bk]BDhU>2DjRll`mnZM`@Ee=a<@ULec[9cTlZNGW>:Yd_3qA<daHZmkMRdP@^0Aj
;3e6h6qE]Va;F>Kj_kBM90mW:U5jH_CoSU7DYN1cIiWIUbDKckKc^Xle<2<eJ0qB
1Ejn5p@j3]H[EeG[XD=k_TYk<QI:LpIK1=V?]>;\OmnUELn8[GpoGjDGIb25nFH]
1KVq7MR=de7R?cU_9Hpn>Zi9KBp^[_9YX7ALf9;<QVST?f`Ccc>[njI?3CQG8d>Z
PF5;PJfk?qZRLKDQGqSXGSObkG5RJTog6X?<YbiVVAR]TRF5q<;9`djadF2UTTPn
_>\KUcUCdWI9L?nW_Hk9>YH>e]`eNA`=JQ9mfU`k5b<WNn^[OTlmneVTq010CK2I
fnliaH7R;RMO3oc40FV>KNWXb^k3]]SSORFNnh\d>Xj6AYeF?qa40;?LA0JU[F:I
M><MdC=dMad?pRfT h53PBNcmPid0[q29Hb7aT[8p0X87TV4hqk^@aF3FDXCnT?F
Rd3o4n0a4213n5]HqH^L8G:@`c`S:519:FmoSiibpDh18cBU=FkTk>Ec4dVOD^jS
XqNJ^N [L1N24>@??RmIm45 1FqdAoN1V>jlJqhMncLancH7dU?^>6ZXPqXYloNm
pS;7k7K$
`endprotected
module and_op (a, b, c);
output a;
input b, c;
    'ifdef behavioral
        wire a=b && c;
    'else
        and (a, b, c);
    `endif
endmodule
```

File Inclusion

You use the file inclusion (<code>include</code>) compiler directive when you want to insert the entire contents of a source file into another file during Verilog-XL compilation. Verilog-XI behaves as though the contents of the included source file appear in place of the <code>include</code> command. You can use the <code>include</code> compiler directive to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

The advantages of using the *`include* compiler directive include the following:

- providing an integral part of configuration management
- improving the organization of Verilog HDL source descriptions

■ facilitating the maintenance of Verilog HDL source descriptions

Syntax of 'include

The syntax for the 'include compiler directive is as follows:

`include "<filename>"

You can specify the compiler directive `include anywhere within the Verilog-XL description. The <filename> is the name of the file that you want to include in your source file. The <filename> can be referenced from either the current directory, or a directory specified on the command line with the +include+<incdir> command-line option. The <filename> can be a full or relative path name, as in the following example:

```
`include ``/net/usr/working/lib/TECHLIB1/parts/count.v" // full path
`include ``lib/TECHLIB1/parts/count.v" // relative path
`include ``./lib/TECHLIB1/parts/count.v" // relative path
```

Note: The '/' character can be used as a path delimiter on both Windows NT and UNIX workstations.

Verilog-XL requires only blank spaces or a comment after the <filename>. Examples of legal `include compiler directives are as follows:

```
`include "fileB"
`include "fileB" // including fileB
```

Specifying Search Directories

You use the +incdir+ command-line plus argument to specify the directories you want Verilog-XL to search for an included file. The syntax for specifying the +incdir+ option on the command line is as follows:

+incdir+<directory1>+<directory2>+...<directoryN>

There is no limit to the number of +incdir+ options you can specify. Verilog-XL searches for these directories in the order in which you list them on the command line.



Verilog-XL does not check the characters between the two plus characters for errors. Verilog-XL assumes that all of the characters are part of the directory name.

How 'include Works in Verilog-XL

The 'include compiler directive works in the following way:

- 1. Verilog-XL searches for the file specified by the compiler directive, relative to the current working directory.
- 2. If the file is not found, Verilog-XL searches the directories specified in the +incdir+ command-line plus option. Verilog-XL searches these directories in the order in which you list them on the command line.
- 3. If Verilog-XL finds the file specified by the 'include compiler directive, it executes the source code in that file as though that code has replaced the 'include compiler directive.
- 4. If Verilog-XL searches all the directories that you specified in the +incdir+ commandline plus argument and the file is not found, then the search results in an error and compilation stops.

Nested 'include Compiler Directives

An `include file can contain other `include compiler directives. You can nest `includes up to a maximum of 8 levels. If Verilog-XL detects a recursive `include, then Verilog-XL displays an error message and compilation stops.

When Verilog-XL begins compiling an included file, Verilog-XL displays an informational message to inform you of the Verilog-XL compilation process. An example of this informational message is as follows:

Compiling Included source file <filename>

You can use a message like this to inform you of errors while processing nested `include directives.

When Verilog-XL continues compiling the file containing the <code>`include</code> compiler directive, Verilog-XL displays the following informative message:

Continuing compilation of source file <filename>

Decompiling Source Descriptions

Verilog-XL decompiles a source description whenever it displays the source description for a module. Every time Verilog-XL decompiles a module that contains an `include command,

the line numbers before and after the *`include* text correspond to the original files, as illustrated in the following example.



Decompilation can display the wrong filenames when splitting any of the following contructs across different file boundaries:

- an object or gate declaration
- a initial, fork/join, or specify block
- a task or function

In the previously mentioned cases, Verilog-XL does not print the filename as soon as the file boundary is crossed. Instead, Verilog-XL prints the filename at the start of another block (initial, fork/join, or specify block) in the second file as, shown in the following example:

```
// from file test.v
module test;
    initial
    'include "test2.v"
    initial
    $list (test);
endmodule
// from file test2.v
$display ("first initial");
    initial
$display ("second initial");
// the following is the source description decompilation:
first initial
second initial
 // test.v
 1 module test;
 2 initial
 1 $display("first initial");
 // test2.v <- Verilog-XL prints the name of the included file
 2 initial
 3 $display("second initial");
 // test.v
 4 initial
```

```
5* $list(test);
6 endmodule
```

'include Error Messages

All syntax error messages in a `include file display the correct filename and line number.

Any semantic error that is caught after the first compilation pass can display the wrong filenames when any of the following constructs is split across different file boundaries:

- an object or gate declaration
- a initial, fork/join, or specify block
- a task or function

Source Protection for Included Files

You can use the *`include* compiler directive either inside or outside of a source protected region.

When you specify the `include compiler directive within a protected region, the `include compiler directive is always copied unprotected to the protected file, even if the `include compiler directive appears in a protected region.

Verilog-XL does not protect the `include compiler directive or the `include file during source protection. To source protect the `include file, you must protect it separately.

The following example shows what a source-protected module looks like before and after Verilog-XL compiles a source file.

Before Source Protection:	After Source Protection:
<pre>`protect module test;</pre>	`protected @#\$%^&*%\$#&@%\$^(##@%#! + <first module="" of="" part="" protected=""> %\$#^&(#*@!%&#(&\$*(#(&@##*</td></tr><tr><td><first part of module test></td></tr><tr><td>`include "FileB"</td><td>endprotected Protect this file</td></tr><tr><td><second part of module test></td><td>'protected separately.</td></tr><tr><td>endmodule
`endprotect</td><td><pre>%\$#^&(#*@!%&#(&\$*(#(&@##*
<second part of protected module>
@#\$%^&*%\$#&@%\$^(##@%#! +
`endprotected</pre></td></tr></tbody></table></first>

:

Note: When Verilog-XL compiles the protected file, Verilog-XL includes the <code>`include</code> file and processes the <code>`include</code> file properly, whether the <code>`include</code> file is protected or unprotected.

Interactive Control and Debugging

This appendix describes the following:

- <u>Overview</u> on page 247
- <u>Getting Started</u> on page 248
- Interactive Recovery on page 250
- <u>Getting Help</u> on page 251
- <u>Selecting the Foci of a Debugging Session</u> on page 253
- Stepping through a Simulation on page 257
- Setting Breakpoints in a Simulation on page 261
- <u>Displaying Waveforms</u> on page 269

Overview

The Verilog-XL interactive control and debugging system allows you to interact very closely with the simulation process. Some of the major functions you can perform using this system are listed below:

- Issue interactive commands.
- Display the values of variables.
- Execute single statements.
- Take incremental dumps of the entire simulation state for later retrieval.
- Get full or selective tracing of source statement executions.
- Set foci for debugging.
- Step through source code.

■ Set breakpoints.

The Verilog-XL online facility is a full symbolic debugger, featuring debug commands that are fully compatible with Verilog HDL. All variable, module, task, function, and block names are accessible online in their original source description form.

The waveform postprocessor displays waveforms after simulation is complete. <u>"Displaying</u> <u>Waveforms" on page 269</u> provides a summary.

Getting Started

The Verilog-XL debugging environment enables you to interact with the simulation process. You can stop the simulation at predefined points or step through the simulation. Each time you stop the simulation, you enter the Verilog-XL interactive mode. In interactive mode the simulation is in a state of *frozen animation* that is, all objects are frozen in their current states.

Note: Verilog-XL is unable to debug source text that is protected.

In interactive mode, you can analyze the results of the simulation from the beginning of the simulation to the current simulation time. If you find that the simulation has proceeded incorrectly, you can temporarily patch the design and continue simulating.

To issue interactive commands, you must order the simulator to halt processing. You can effect either a synchronous halt using the Verilog \$stop system task or an asynchronous halt using the host operating system interrupt key (control-C under UNIX and VMS). Either way, the simulator halts processing and issues a prompt at the terminal indicating that it is ready to receive an interactive command.

The interactive command prompt appears in the following format:

C<command_number>>

Each interactive command that you enter is assigned a unique number. The <command_number> in the prompt indicates the number of the command you are currently entering. For example, command number 6 would be entered at the following prompt:

C6>

There are several different types of interactive commands. The first type of commands consists of any normal behavioral statements—that is, any valid procedural type statements, including assignments, enabling tasks, and block statements. However, behavioral statements issued at the interactive command prompt must comply with the following restrictions:

Named blocks are not allowed.

- The keywords initial and always are not allowed.
- If you use an if statement that is not inside a block, you must not omit the else part of the if statement. If no action is required in the else part, simply enter else followed by a semicolon.

Note: The interactive debugging environment system tasks are described later in this chapter. See the table in <u>"Getting Help"</u> on page 251 for a list of the debugging environment system tasks.

You do not have to type a behavioral statement on a single line; that is, white space (spaces, tabs, and new lines) may be used freely throughout the statement. As you type each line of the statement, the Verilog-XL interpreter parses the text and produces error messages in exactly the same manner as when it executes the main source description. If an error results, you must retype the entire statement. The interactive command is executed as soon you enter a complete, error-free statement.

You can also enter any compiler directive as an interactive command. However, most compiler directives only affect compilation, and therefore have no effect during an interactive session.

Syntax	Action	Description
<statement></statement>	command	Compile and execute a behavioral statement.
	continue	Continue with the simulation.
;	step	Step through a single statement.
1	trace-step	Step through and trace a single statement.
:	where	Print current location.
<number></number>	re-execute	Re-execute a previous command.
- <number></number>	disable	Disable a previous command.

Other types of interactive commands perform functions that normal statements cannot perform. A subset of interactive commands is shown in the following table:

The period (.) character issues the continue command. In response, the simulator switches from halt mode to run mode and continues the simulation run.

The semicolon (;) character issues the step command. This causes the simulator to execute the next statement in the source text file and immediately return control back to the user.

The comma (,) character issues the trace-step command. This command performs the same function as the step command, but also generates a trace message from the statement executed.

The colon (:) character issues the where command, which prints out information about the line that is currently being executed in the source text file. The information comprises the line number, source filename (for distinguishing between two or more source files), and the full hierarchical name of the module instance that contains the current line of source.

A decimal number issues the re-execute command, which executes a previously entered interactive command. Once you enter a command, you can repeat it any number of times simply by typing its command number.

A decimal number preceded by a dash (-) character issues the disable command, which disables a previously entered command that is still active. The number that you type specifies the command that Verilog-XL disables. For example, consider the following complex statement command:

C4> forever @s85.i85.acc \$stop;

This command comprises an infinite loop statement that continuously monitors the variable s85.i85.acc and stops the simulation whenever the variable changes. Commands such as this may be active throughout simulation. To kill command number 4, issue the disable command, as follows:

C9> -4

Continued use of the re-execute and disable commands provides a quick and easy method of enabling and disabling many different kinds of commands during a debugging session.

Note: Whenever you re-execute a previous command, the command is first disabled and then re-executed.

Many of the system tasks described in <u>Chapter 14</u>, "System Tasks and Funtions" have been designed for online debugging purposes. However, nothing prevents these tasks from being used in the main source description. If lengthy debugging statements are needed, it is usually better to put them into a file and input the file using the *sinput* system task, or to include the debugging statements in the main source text, as they can then serve as a more permanent debugging record.

Interactive Recovery

The key file, the -i input file option, the save task, and the -r restart file option together provide a very powerful interactive recovery mechanism.

Whenever Verilog-XL runs, two files are generated—a log file and a key file. The log file contains all of the text that has been printed to the standard output. Similarly, the key file contains all the text that has been entered interactively. (By default, the key file is named verilog.key, but this name can be changed at invocation using the -k option.) The key file can be used in a subsequent run of Verilog-XL to perform an exact 'replay' of the simulation run and interactive dialogue, including any asynchronous interrupts.

For example, suppose that you start Verilog-XL with the following invocation command:

verilog example.v -s

The -s option directs Verilog-XL to pause for interactive input before performing any simulation. Suppose that you enter some interactive commands and then resume the simulation using the continue command. When the simulation is completed, you could replay this run using Verilog-XL's interactive recovery mechanism. To do so, you must first save the key file under another name (say to example.key) as under the UNIX operating system, Verilog-XL overwrites the key file in every run. Then you can use the following invocation command to replay the previous run:

verilog example.v -si example.key

Here, the -s option tells Verilog-XL to enter interactive mode immediately following compilation, and the -i option tells Verilog-XL to execute the interactive commands contained in the file named example.key. After executing all of the stored commands, Verilog-XL prompts you for additional interactive input. To resume the simulation, enter the continue (.) command. If desired, you can break into the middle of this replay by typing the key combination for an operating system interrupt (Control-C under UNIX and VMS).

The key file recovery mechanism described above is most useful for backtracking several commands in order to recover a previous state of simulation. This is accomplished by editing the key file before the replay and deleting any commands that you wish to omit from the replay.

Getting Help

Use the db_help system task to see a list of the interactive debugging environment commands. The interactive debugging environment commands are a group of system tasks that let you step, trace, and set your debugging focus and breakpoints. The debugging environment system tasks begin with $db_$, as in the db_help system task described below:

Syntax:

\$db_help;

November 2008

The *sdb_help* system task outputs a list of all of the debugging system tasks. The list includes a description of the arguments and function of each debugging system task.

The following table shows a list of the debugging environment system tasks. For syntax descriptions, see the command descriptions of these system tasks in this chapter.

Task	Command	Description
getting help	\$db_help	Lists the debugging system tasks.
setting foci	\$db_setfocus	Lets you set the scope for debugging operations by adding one or more scopes to the focus list.
setting foci	\$db_deletefocus	Removes the foci that you specify from the focus list.
setting foci	\$db_enablefocus	Enables the foci that you specify.
setting foci	\$db_disablefocus	Disables the foci that you specify; the foci remain in the focus list.
setting foci	\$db_showfocus	Displays the focus list.
stepping	\$db_step	Steps through the source file one or more steps depending on the number of steps that you specify.
stepping	\$db_steptime	Steps through the simulation by the number of time units that you specify.
tracing	\$db_settrace	Turns trace stepping mode on; the decompilation and trace results are displayed at each step of the simulation.
tracing	<pre>\$db_cleartrace</pre>	Turns trace stepping mode off.
setting breakpoints	\$db_breakatline	Sets a breakpoint at the line you specify; Verilog-XL breaks each time it encounters the specified line.
setting breakpoints	\$db_breakbeforetime	Sets a breakpoint at the beginning of a time unit that you specify.
setting breakpoints	\$db_breakaftertime	Sets a breakpoint at the end of a time unit that you specify.
setting breakpoints	\$db_breakwhen	Sets a breakpoint dependent on the value of a given object; Verilog-XL breaks each time the object has the specified value.

252
Task	Command	Description
setting breakpoints	\$db_breakoncewhen	Sets a breakpoint dependent on the value of a given object; Verilog-XL breaks only the first time the object has the specifed value.
setting breakpoints	\$db_breakonposedge	Sets a transition-based breakpoint;Verilog-XL breaks each time a postive-edge transition occurs on the specified object.
setting breakpoints	\$db_breakonceonposedge	Sets a transition-based breakpoint;Verilog-XL breaks the first time a postive-edge transition occurs on the specified object.
setting breakpoints	\$db_breakonnegedge	Sets a transition-based breakpoint;Verilog-XL breaks each time a negative-edge transition occurs on the specified object.
setting breakpoints	\$db_breakonceonnegedge	Sets a transition-based breakpoint;Verilog-XL breaks each the first time a negative-edge transition occurs on the specified object.
setting breakpoints	\$db_deletebreak	Deletes a given set of breakpoints from the breakpoint list.
setting breakpoints	\$db_enablebreak	Enables a given set of breakpoints in the breakpoint list.
setting breakpoints	\$db_disablebreak	Disables a given set of breakpoints in the breakpoint list.
setting breakpoints	\$db_showbreak	Displays the breakpoint list.

Selecting the Foci of a Debugging Session

The *focus* of a debugging session defines the limits of the debugging activity during source stepping and tracing. It is similar in concept to the scope of a Verilog-XL hierarchical model. Selecting a focus enables you to *focus in* on the part of the hierarchy that you are interested in debugging.

The hierarchy of a Verilog-XL model is a tree structure in which each node or scope is defined by an instance of a module, task, function, or named block. You define a debugging focus as one scope or a set of scopes. If you do not set the focus for a debugging session, the default focus is the entire hierarchy.

It is important that you are aware of your current focus because this affects how you source step and trace through your design during debugging.

A list of the current foci is maintained by Verilog-XL. Foci can be added to and deleted from this list. However, in order for a focus in the list to be used by Verilog-XL, the focus must be *enabled*. If a focus appears in the list, but is not enabled, the focus is considered *disabled* and will be ignored. All foci are initially enabled.

Note: The list of foci can be saved with <code>\$save()</code> and restarted with <code>\$restart()</code>.

<u>"Source Stepping" on page 257</u> contains a list of the behavioral statements you can use in tracing and setting breakpoints.

\$db_setfocus

Syntax:

```
$db_setfocus(<scope> <,<scope>>*);
```

Purpose:

The \$db_setfocus system task adds the specified scope or set of scopes to the current focus list. The new focus or foci are automatically enabled by default. Each scope is the hierarchical name of a module instance, task, function, or named block. The scopes in the list must be separated by commas.

\$db_setfocus outputs the focus ID number of the new focus or foci.

Example:

```
C7 > $db_setfocus(named_fork);
Set focus (1) on scope ffnand_test.named_fork.
```

\$db_deletefocus

Syntax:

```
$db_deletefocus;
$db_deletefocus(<focus_list_entry>
    <,<focus_list_entry>>*);
    <focus_list_entry>
        ::= <focus_id>
        ::= <scope>
```

Purpose:

The \$db_deletefocus system task deletes the given set of foci from the focus list. You can specify each focus by either the focus ID number or the hierarchical scope name. You must separate the focus ID numbers and scope names in the list with commas. When no argument is provided, the system task deletes all foci in the focus list.

Example:

```
C8 > $db_deletefocus(named_fork);
Deleted focus (1) on scope ffnand_test.named_fork.
```

\$db_enablefocus

Syntax:

```
$db_enablefocus;
$db_enablefocus(<focus_list_entry>
    <,<focus_list_entry>>*);
    <focus_list_entry>
        ::= <focus_id>
        ::= <scope>
```

Purpose:

The \$db_enablefocus system task enables the given set of foci in the focus list. You can specify each focus by either the focus ID number or the hierarchical scope name. You must separate the focus ID numbers and scope names in the list with commas. When no argument is provided, the system task enables all foci in the focus list.

Example:

```
C4 > $db_enablefocus(1);
Enabled focus (1) on ffnand_test
```

\$db_disablefocus

Syntax:

```
$db_disablefocus;
$db_disablefocus(<focus_list_entry>
    <,<focus_list_entry>>*);
    <focus_list_entry
        ::= <focus_id>
        ::= <scope>
```

Purpose:

The \$db_disablefocus system task disables the given set of foci in the focus list. You can specify each focus by either the focus ID number or the hierarchical scope name. You must separate the focus ID numbers and scope names in the list with commas. When no argument is provided, the system task disables all foci in the focus list.

Example:

C10 > \$db_disablefocus(named_task);
Disabled focus (3) on scope ffnand_test.named_task.

\$db_showfocus

Syntax:

\$db_showfocus;

Purpose:

The \$db_showfocus system task outputs the focus list. Each entry in the list contains the following:

- Focus ID number
- Hierarchical scope name
- enabled **or** disabled

Example:

```
Cl1 > $db_showfocus;
Status enabled focus (2) on scope ffnand_test.named_fork.
```

Stepping through a Simulation

You use stepping to simulate from the current simulation time until either the next point of activity within the debugging focus or a specified simulation time. These two types of stepping are as follows:

- Source stepping
- Stepping in time

Source Stepping

You can perform source stepping using either single steps or multiple steps. Each single step or set of multiple steps runs the simulation for the specified number of steps and then returns you to interactive mode within the current focus. The simulation is interrupted either when one or more statements within a behavioral construct are about to be evaluated or after Verilog-XL has determined the value of the output of a gate or unidirectional switch.

The current focus defines the context for source stepping through a design; each step moves the simulation to the next point of activity within the focus.

Execution of the behavioral constructs in the following list is considered activity within the focus, and you can use these commands in setting breakpoints and tracing. <u>"Source Line-</u>

<u>Based Breakpoints</u>" on page 261 discusses the constructs that can generate more than one breakpoint.

assign delay statement assign event statement assign multi statement assign statement case statement casez statement deassign statement delay statement disable statement event generation statement event statement for statement force statement fork statement
function-call statement
if statement
if else statement
non-blocking delay statement
non-blocking multistatement
release statement
repeat statement
system function statement
task-call statement
user-defined function statement
wait statement
while statement

Stepping in Time

Stepping in time advances the simulation from the current simulation time to just before execution of simulation events at the specified simulation time.

Note: Stepping in time is not affected by the current focus.

Tracing

There are two stepping modes for both source stepping and stepping in time:

- stepping without tracing
- stepping with tracing

When you step with tracing, only the activity within the focus is traced. The trace information includes:

- the current simulation time
- a full decompilation of each executed statement
- the results of executing each statement

\$db_step

Syntax:

```
$db_step;
$db_step(<step_count>);
```

Purpose:

The \$db_step system task performs the specified number of source steps. When no argument is provided, the system task performs a single step. When there is no current focus, the source stepping is equivalent to the interactive ; command when the trace is not on. When there is no current focus, the source stepping is equivalent to a comma (,) when the trace is on. The task does not generate output at the time that you invoke it.

Example:

```
C1 > $db_step(1);
    ...
0 preset=1 clear=0 q=x qbar=x
Stepped to line 60, scope ffnand_test.named_begin, file code.doc, time 1.
```

\$db_steptime

Syntax:

```
$db_steptime(<time_units>);
```

Purpose:

The \$db_steptime system task performs stepping in time. The task advances the simulation to the point just before the simulation events execute at the simulation time equal to the value of the current simulation time plus the specified number of time units.

The specified time units are scaled according to the \$timeformat system task. If there is
no time format, the time units are scaled to the timescale of the current scope. If the current
scope does not have a timescale, the specified time units are assumed to be given in
simulation time units.

This task does not generate output at the time that you invoke it.

Example:

```
C1 > $db_steptime(10);
...
0 preset=1 clear=0 q=x qbar=x
1 preset=1 clear=0 q=x qbar=1
Stepped to time 10.
```

\$db_settrace

Syntax:

\$db_settrace;

Purpose:

The \$db_settrace system task turns on the trace stepping mode. When there is no focus selected, the system task is equivalent to the \$settrace system task. This task produces no output or messages.

Example:

C1 > \$db_settrace;

\$db_cleartrace

Syntax:

\$db_cleartrace;

Purpose:

The \$db_cleartrace system task turns off the trace stepping mode. When there is no focus selected, the system task is equivalent to the \$cleartrace system task. This task produces no output or messages.

Example:

C3 > \$db_cleartrace;

Setting Breakpoints in a Simulation

Setting a *breakpoint* enables you to interrupt the simulation and enter the Verilog-XL interactive mode at predefined points in the simulation. In order to debug your design using breakpoints, at least one breakpoint should be set before you start the simulation.

A list of the current breakpoints is maintained by Verilog-XL. Breakpoints can be added to and deleted from this list. However, in order for a breakpoint in the list to be used by Verilog-XL, the breakpoint must be *enabled*. If a breakpoint appears in the list, but is not enabled, the breakpoint is considered *disabled* and will be ignored. All breakpoints are initially enabled.

Note: The list of breakpoints can be saved with <code>\$save()</code> and restarted with <code>\$restart()</code>.

Each breakpoint in the breakpoint list has a unique ID number associated with it. They also appear in breakpoint messages that are generated by Verilog-XL.

There are four classes of breakpoints:

■ Time-Based Breakpoints

A *time-based breakpoint* is reached when the simulation progresses to a specific simulation time. You can specify whether you want the breakpoint interrupt to occur before or after the execution of simulation events at the specified simulation time. When setting a time-based breakpoint, you specify the simulation time as an absolute time (for example, at 5 simulation time units).

Transition-Based Breakpoints

A *transition-based breakpoint* is reached when a specific transition occurs. When setting a transition-based breakpoint, you specify the transition as the positive edge, the negative edge, or any edge of a specific object.

■ Value-Based Breakpoints

A *value-based breakpoint* is reached when the value of a specific object is changes to either the value you specify, or any value (default).

■ Source Line-Based Breakpoints

A *source line-based breakpoint* is reached when the simulation progresses to the specified source line. When setting a source line-based breakpoint, you specify the scope, filename, and line number.

As with source stepping, when a source line-based breakpoint is reached, Verilog-XL interrupts the simulation at the point where one or more statements within a behavioral

construct are about to be evaluated, or after Verilog-XL has determined the value of the output of a gate or unidirectional switch.

Source stepping is insensitive to the activity at levels below the current focus. This enables you to advance through the simulation at a high level, ignoring the fine details of activity that are occurring at the lower levels. If during debugging you wish to include lower hierarchical levels in the focus, you must explicitly set the new current focus to include these lower levels.

Assignment statements that generate two breakpoints

The following types of assignment statements generate a breakpoint both at the time that a right-hand side evaluation occurs *and* just before the assignment:

- assign delay statement
- non-blocking delay statement
- assign event statement
- non-blocking event statement
- assign multi statement
- non-blocking multi statement
- wait statement

Statements on the same line that generate two breakpoints

In most cases, if multiple statements are on a line, you can set a breakpoint only on the first statement. The following two statements are exceptions to this rule. If one of the following statements and the next statement are on the same line, setting a breakpoint on one of the following statements generates two breakpoints:

- delay statement
- event wait statement

The first breakpoint occurs before the delay statement or event wait statement. The second breakpoint occurs before the statement that follows the delay statement or event wait statement on the same line.

Constructs ineligible for setting breakpoints

You can not set breakpoints on the constructs in the following list. Attempting to set breakpoints on some of these constructs sets breakpoints later.

function
initial
join
macromodule declaration
macromodule instance
module declaration
module instance
primitive declaration
primitive instance
specify block
table definition
task

Statements generating breakpoints at the next statement

Attempting to set breakpoints on one of the following constructs sets a breakpoint at the next statement in the current scope at which you can set a breakpoint:

- initial
- always
- forever
- ∎ begin

Continuous and Non-Continuous Breakpoints

A *continuous breakpoint* remains enabled until you explicitly disable it. A *non-continuous breakpoint* remains enabled until it is reached the first time and is then automatically disabled. Disabled breakpoints remain in the breakpoint list until you explicitly delete them. By definition, all time-based breakpoints are non-continuous breakpoints. Transition-based, value-based, and source line-based breakpoints can be either continuous or non-continuous.

To set non-continuous breakpoints use the debugging system tasks that contain the word "once" in their names.

\$db_breakatline

Syntax:

```
$db_breakatline(<line_no> <,<scope> <,<filename>>? >? );
$db_breakonceatline(<line_no> <,<scope> <,<filename>>? >? );
```

Purpose:

The \$db_breakatline system task sets a continuous source line-based breakpoint at the specified line number and adds the breakpoint to the breakpoint list. The scope name and the filename are optional. If they are not supplied, Verilog-XL assumes the breakpoint is in the current Verilog-XL scope and file.

Example:

```
C4 > $db_breakatline(79, ffnand_test, "dbloc.v");
Set break (1) at line 79, scope ffnand_test, file dbloc.v.
...
Break (1) occurred at line 79, scope ffnand_test, file dbloc.v, time 20.
```

\$db_breakbeforetime

Syntax:

```
$db_breakbeforetime(<time>);
```

Purpose:

The \$db_breakbeforetime system task sets a time-based breakpoint at the specified simulation time and adds the breakpoint to the breakpoint list. The breakpoint occurs before Verilog-XL evaluates simulation events at the simulation time you specify.

The specified time is scaled according to the *stimeformat* system task. If there is no time format, the time is scaled to the timescale of the current scope. If the current scope does not have a timescale, the time is assumed to be given in simulation time units.

Example:

```
C1 > $db_breakbeforetime(10);
Set break (1) before time 10.
...
Break (1) occurred before time 10.
```

\$db_breakaftertime

Syntax:

```
$db_breakaftertime(<time>);
```

Purpose:

The \$db_breakaftertime system task sets a time-based breakpoint at the specified simulation time and adds the breakpoint to the breakpoint list. The breakpoint occurs before Verilog-XL evaluates simulation events at the simulation time you specify.

The specified time is scaled according to the *stimeformat* system task. If there is no time format, the time is scaled to the timescale of the current scope. If the current scope does not have a timescale, the time is assumed to be given in simulation time units.

Example:

```
C2 > $db_breakaftertime(5)
Set break (2) after time 5.
...
Break (2) occurred after time 5.
```

\$db_breakwhen

Syntax:

```
$db_breakwhen(<object> <,<value>>? );
$db_breakoncewhen(<object> <,<value>>? );
```

Purpose:

The \$db_breakwhen system task sets a continuous value-based breakpoint and adds the breakpoint to the breakpoint list. The breakpoint occurs whenever the object that you specify changes to the value that you specify.

The object you specify must be a net, register, integer, real, or event. If the object is outside the current scope, specify it with a hierarchical name.

You can specify the value as any valid expression. Verilog-XL evaluates the expression at the breakpoint, and the resulting value is used as the break value.

If you do not specify the value, the breakpoint occurs whenever the object that you specify changes value. In this case, the system task is equivalent to setting a continuous transition-based breakpoint in which you have specified the transition as any occurrence of the event.

When the object that you specify is an event, you need not specify the value.

Example:

```
C7 > $db_breakwhen(vrg, `b1);
Set break (3) when top.vrg = 0001
...
Break (3) occurred when top.vrg = 0001 at time 7.
```

\$db_breakonposedge

Syntax

```
$db_breakonposedge(<object>);
$db_breakonceonposedge(<object>);
```

Purpose:

The *\$db_breakonposedge* system task sets a continuous transition-based breakpoint and adds this breakpoint to the breakpoint list. The breakpoint occurs when a positive-edge transition occurs on the object you specify.

You can specify the object using the hierarchical name of a scalar net, a bit of an expanded vector net, or a single-bit register.

Example:

C4 > \$db_breakonposedge(rg); Set break (4) on pos edge top.rg. ... Break (4) occurred on pos edge top.rg at time 8.

\$db_breakonnegedge

Syntax:

```
$db_breakonnegedge(<object>);
```

\$db_breakonceonnegedge(<object>);

Purpose:

The \$db_breakonnegedge system task sets a continuous transition-based breakpoint and adds the breakpoint to the breakpoint list. The breakpoint occurs when a negative-edge transition occurs on the object you specify.

You can specify the object using the hierarchical name of a scalar net, a bit of an expanded vector net, or a single-bit register.

Example:

```
C6 > $db_breakonnegedge(rg);
Set break (4) on neg edge top.rg.
...
Break (4) occurred on neg edge top.rg at time 8.
```

\$db_deletebreak

Syntax:

```
$db_deletebreak;
$db_deletebreak(<break_id> <,<break_id>>*);
```

Purpose:

The \$db_deletebreak system task deletes the given set of breakpoints from the breakpoint list. You can specify each breakpoint by the breakpoint ID number. You must separate the breakpoint ID numbers in the list with commas. When no argument is provided, the system task deletes all of the breakpoints in the breakpoint list.

Example:

\$db_deletebreak(1); Deleted break (1) at line 79, scope ffnand_test, file dbloc.v.

\$db_enablebreak

Syntax:

```
$db_enablebreak;
$db_enablebreak(<break_id> <,<break_id>*);
```

Purpose:

The \$db_enablebreak system task enables the given set of breakpoints in the breakpoint list. You can specify each breakpoint by the breakpoint ID number. You must separate the breakpoint ID numbers in the list with commas. When no argument is provided, the system task enables all of the breakpoints in the breakpoint list.

Example:

```
C4 > $db_enablebreak(2);
Enabled break (2) at line 79, scope ffnand_test, file dbloc.v.
```

\$db_disablebreak

Syntax:

```
$db_disablebreak;
$db_disablebreak(<break_id> <,<break_id>>*);
```

Purpose:

The \$db_disablebreak system task disables the given set of breakpoints in the breakpoint list. You can specify each breakpoint by the breakpoint ID number. You must separate the breakpoint ID numbers in the list with commas. When no argument is provided, the system task disables all of the breakpoints in the breakpoint list.

Example:

```
$db_disablebreak(3);
disabled break (3) at line 79, scope ffnand_test, file dbloc.v.
```

\$db_showbreak

Syntax:

\$db_showbreak;

Purpose:

The $db_showbreak$ system task outputs the breakpoint list. Each entry in the list contains the following:

- breakpoint ID number
- enabled or disabled
- non-continuous (once) or continuous
- for time-based breakpoint: simulation time and before/after
- for source line-based breakpoint: line number, scope name, and file name
- for value-based breakpoint: object and value
- for transition-based breakpoint: object and positive/negative edge

Example:

```
C5 > $db_showbreak;
Status enabled break (2) after time 5.
Status enabled break (3) once when top.vrg = 0001.
Status enabled break (4) on pos edge top.rg.
```

Displaying Waveforms

This section summarizes the system tasks necessary for using the Simvision Waveform Viewer.

Simvision Waveform Viewer

The SimVision Waveform Viewer is the new powerful tool that you can use for analyzing simulation results and debugging designs. Its high-performance, waveform viewing technology enables you to analyze large amounts of complex simulation data quickly and accurately.

You can run the Waveform Viewer:

- Interactively from the SimVision analysis environment, which lets you view waveforms for your simulation results as they are generated.
- In the post-processing environment (PPE) mode from the SimVision analysis environment, which lets you view the simulation results that you have stored in a simulation database.
- Alone to view simulation results that you have stored in a database.

No matter how you run the Waveform Viewer, you can view the results of several simulations simultaneously, in either the same or separate Waveform windows. For details, refer to the <u>SimVision User Guide</u>.

SHM Tasks

The waveform viewer tool can be run in batch mode or interactively and are used to display waveforms. The simulation history manager (SHM) is a group of system tasks that control communication between the Verilog-XL simulation and a database that stores data for the display. You must use SHM tasks to prepare data for display. The following table summarizes the SHM system tasks:

SHM Task	Service that Task Provides
\$shm_open	Opens a database.
\$shm_probe	Specifies signals whose value changes enter the database.
\$shm_close	Terminates the simulation's connection to the database.
\$shm_suspend	Temporarily suspends dumping of values to the database.
\$shm_resume	Resumes dumping of values to the database.

Opening a Database with \$shm_open

Use the $\${\rm shm_open}$ system task to open a simulation database for display in waveform window.

The syntax of \$shm_open system task is as follows:

```
$shm_open (["db_name"], [<is_sequence_time>], [<database_size>],
        [<is_compression>])
```

The arguments are:

"db_name"	(Optional) Filename of the simulation database to be opened for display. If you do not specify the database name, waveform viewer searches for a database waves.shm in the current directory and if found, opens it.
<is_sequence_time></is_sequence_time>	(Optional) Sets to display all events or transitions of a signal that take place at the same time step. Specify 1 to enable or 0 to disable the feature. The default setting is 0 .
<database_size></database_size>	(Optional) Specify the maximum size (in bytes) of the transition file (.trn file) generated by Verilog-XL. You must specify a size of 2MB or more.
Note: It is recommend Verilog-XL may overwr	led that you specify the size as atleast 10MB. Otherwise ite the transition file if the needed size is more than your setting.

<is_compression> (Optional) Sets to compress the transition file (.trn file) generated by Verilog-XL. Specify 1 to compress. The default setting is 0 or no compression.

Probing Signals with \$shm_probe

The <code>\$shm_probe</code> system task lets you specify the signals whose value changes you want to record in your SHM database and lets you specify the nodes at which value changes are recorded.

The syntax for the <code>\$shm_probe</code> system task is as follows:

\$shm_probe([scope1, "node_specifier1", scope2, "node_specifier2", ...])

The arguments to <code>\$shm_probe</code> are optional, but the parentheses are not.

If you do not specify any arguments, \$shm_probe writes the value changes that occur at all inputs, outputs, and inouts in the current scope to your SHM database.

The arguments to the <code>\$shm_probe</code> system task are as follows:

■ scope1, scope2, ...

Specifies the scope or scopes whose signals you want to probe. If you do not specify a scope, <code>\$shm_probe</code> records the signal changes that occur in the current scope.

"node_specifier1", "node_specifier2", ...

Specifies one of five codes, called node specifiers, to indicate the nodes at which you want to record value changes for the specified signals.

Node specifiers apply to the specified scopes in order of appearance. That is, node_specifier1 applies to scope1, node_specifier2 applies to scope2, and so on. If a node specifier does not have a corresponding scope, it applies to the current scope. If a scope does not have a corresponding node specifier, \$shm_probe records value changes at all inputs, outputs, and inouts in that scope.

The five node specifiers are:

Node Specifier	Signals that enter the database
"A"	All nodes (including inputs, outputs and inouts) of the specified scope.
"S″	Inputs, outputs, and inouts of the specified scope, and in all instantiations below it, except inside library cells.
"C"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below it, including inside library cells.
"AS"	All nodes (including inputs, outputs and inouts) of the specified scope, and in all instantiations below it, except inside library cells.
"AC"	All nodes (including inputs, outputs and inouts) in the specified scope and in all instantiations below it, even inside library cells.

The following examples show you how to use <code>\$shm_probe</code> to choose the signals and nodes whose value changes you want to record in your SHM database

- To record value changes at all inputs, outputs, and inouts in the current scope. \$shm_probe();
- To record value changes at all nodes in the current scope.

```
$shm_probe("A");
```

■ To record value changes at all the inputs, outputs and inouts in the scopes alu and adder:

```
$shm_probe(alu, adder);
```

To record value changes at all the inputs, outputs, and inouts in the current scope, and below, excluding those in library cells. Also, record all value changes at all the nodes in the scope top.alu and in all scopes below top.alu, including those nodes in the library cells:

```
$shm_probe("S", top.alu, "AC");
```

Using \$shm_suspend and \$shm_resume

Use the <code>\$shm_suspend</code> system task to temporarily suspend dumping the values of signals to the SHM database. To resume dumping the values of signals, use the <code>\$shm_resume</code> system task.

Note: \$shm_suspend applies to all the probed signals. It is not possible to suspend signals selectively.

When viewing signals in the waveform viewer, values of signals generated between the execution of the system tasks <code>\$shm_suspend</code> and <code>\$shm_resume</code> are displayed as undefined.

Consider the following code segment:

```
initial
begin
$shm_open(...);
$shm_probe(....);
reset = 0;
# 100;
reset = 1;
/* Values of signals in above 100 timesteps are dumped */
$shm_suspend;
# 500;
/* Values of signals above 500 timesteps are NOT dumped */
$shm_resume;
# 500;
/* Values of signals for these 500 timesteps are dumped */
end
```

In the above example, all values of signals generated in the first 100 timesteps are dumped whereas values of signals generated for the next 500 timesteps are not dumped. Dumping of the values of signals are resumed only when the <code>\$shm_resume</code> system task is encountered.

Using \$recordvars and Related Tasks

If you have Verilog code that contains calls to \$recordvars and related tasks
(\$recordfile, \$recordsetup, and so on), you can use these calls to record data in an
SHM (SST2) database. The PLI tasks that were used for recording data during a Verilog
simulation have been implemented as system tasks native to the simulator. That is, it is not
necessary to write PLI code and to link this into the simulator.

The following PLI tasks have been implemented as system tasks native to the simulator:

- srecordvars
- <u>\$recordfile</u>

- <u>\$recordsetup</u>
- <u>\$recordon/\$recordoff</u>
- \$recordclose
- srecordabort

Only the tasks that are used for recording data have been implemented as system tasks. In addition, some options to the \$record* tasks have not been implemented. These are the options that have to do with writing incremental files (incsize, incci, incci, incci, and nosummary). Using these options will generate warning messages.

There are a few differences between the database that is dumped using the new built-in system tasks and the database that is dumped using the PLI interface support. These differences include the following:

- The database dumped using the PLI interface support includes the highconns for ports. These are not always included in the database that is dumped using the new built-in system tasks.
- The database dumped using the PLI interface support includes continuous assignments that are not always included in the database that is dumped using the new built-in system tasks.
- For primitive terminals, the database that is dumped using the new built-in system tasks always dumps the signal to which the terminal is connected.

\$recordvars

The only system task required to record data to an SHM (SST2) database is \$recordvars.

Syntax:

```
$recordvars[("options")];
```

If you do not specify any options, value changes on all signals in the design hierarchy (with no driver or primitive information) are recorded.

Only one database may be written at a time, but you can add variables to be recorded to the database at any time by using another *\$recordvars* task.

The following table lists the options that you can use with *\$recordvars*. Options apply to all following variables and scopes in the call, or to the default scopes if none are specified.

Option	Effect	Default
"depth=n"	Limit the depth if a scope is specified. If "depth=1", no child scopes are included.	"depth=0"
"drivers"	Record drivers (an output terminal of a primitive). Drivers are recorded for all recorded variables that have more than one driver.	"nodrivers"
"primitives"	Record primitives. For all scopes that are recorded, record their primitives in addition to their variables.	"noprimitives"
"nocells"	Do not record variables within a cell, or within any scopes below the cell.	"cells"
	By default, modules defined in a library are cells and other modules are not cells. A non-library module can be defined as a cell using the Verilog `celldefine directive.	
"noports"	Do not record port connectivity information.	"ports"
	This option is used primarily to work around a simulator defect that affects some designs. If this option is used, Simvision does not display ports in different colors, and the Add Trace and Add Module Inputs commands do not display ports.	
"trace"	Record statement trace information.	
	If you use this option, you must also use the "sequence" option on either the <pre>\$recordfile</pre> or <pre>\$recordsetup task.</pre>	
	Recording statement trace information is independent of what variables you are recording. You must record variables in separate \$recordvars task statements. Do not specify other options in the same \$recordvars statement where you specify the "trace" option.	
Any variable	Record a variable. A variable can be a net, register, integer, time, real, or named event. For example, top.ul.u32.a.	

Option	Effect	Default
Any scope	Record a scope. By default, all variables within the scope and all variables in all child scopes are recorded in the database. Use "depth= n " to limit the depth.	All top-level modules and all subscopes.
	A scope can be a module, task, function, or named block. For example, top.control.	

If you open an SHM database with the precord* system task in your Verilog code, the name of the database that is created is preceded by an underscore character. For example, assuming that the top-level module is called top, the following system task opens a database called _top.

\$recordvars;

The following system tasks open a database called _results.

```
$recordfile("results");
$recordvars;
```

This lets you interact with databases opened with <code>\$record*</code> in the same way that you interact with databases that you open with the <code>database</code> command or with <code>\$shm_open</code>. That is, you can use the <code>database</code> command to disable, enable, or display information about the databases.

The \$recordvars task generates two output files:

• A design file (.dsn), which contains information about the design.

By default, the name of this file is <design_name-version_name.dsn>. For example, top-1.dsn.

■ A transition file (.trn), which contains the transition information.

By default, the name of this file is <design_name-version_namerun_name.trn>. For example, top-1-1.trn.

Use the \$recordsetup task to override the default design_name, version_name, and run_name.

Note: If you revert to using the PLI interface support, the file naming convention is the same as that described above if you do not include a \$recordfile task to specify the name of the database. If you use \$recordfile to specify a database name, the files are called database_name.dsn and database_name.trn. These files are overwritten every time

the simulator is run. For example the following code generates $\tt results.dsn$ and <code>results.trn:</code>

```
$recordfile("results");
$recordvars;
```

Example 1:

In the following example, no options, variables, or scopes are specified in the *precordvars* call. All top-level modules are used by default, and all variables in the design are recorded.

```
module record;
.....
initial $recordvars;
endmodule
```

Example 2:

In the following example:

- The first \$recordvars records all variables within scope top.mod1 and all of its descendants, but records only the variables three levels deep for scope top.mod2.
- The second \$recordvars illustrates how options apply to all variables specified later in that \$recordvars task unless overridden. This task records driver information for variables in mod1 and driver and primitive information for variables in mod2.
- The third \$recordvars records two explicitly named variables.

```
module record;
.....
initial
    begin
      $recordvars(top.mod1, "depth = 3", top.mod2);
      $recordvars("drivers", mod1, "primitives", mod2);
      $recordvars(top.io.mux1.q0, top.io.mux2.q0);
      end
endmodule
```

Example 3:

In the following example:

- The first \$recordvars records three levels of variables within scope top.mod1. Drivers are also recorded if the variables have more than one driver. Scope top.mod2 is not depth restricted and no driver information is recorded for variables in this scope.
- The second \$recordvars records driver information for variables in mod1 and driver and primitive information for variables in mod2.

- The third \$recordvars records top.middle.clock and all variables in module2 and its subscopes.
- The fourth \$recordvars records statement trace information. Recording statement trace information is independent of what variables you are recording. You must record variables in \$recordvars task statements, and specify the trace option in a separate \$recordvars statement.
- The \$recordsetup task in this example specifies the recording of sequence information. Sequence information is needed to correlate statements and transitions. If you collect trace information but do not collect sequence information, you will receive a warning message during simulation.

The recording of statement trace information is either on or off for the entire simulation. The \$recordon and \$recordoff statements have no effect on recording statement trace information. Other \$recordvars options, such as specifying depth or scopes, have no effect on how much statement trace information is recorded.

```
module record;
.....
initial
begin
    $recordsetup("design = mydesign", "sequence");
    $recordvars("depth = 3", "drivers", top.mod1,
                      "depth = 0", "nodrivers", top.mod2);
    $recordvars("drivers", mod1, "primitives", mod2);
    $recordvars(top.middle.clock, module2);
    $recordvars(top.middle.clock, module2);
    $recordvars("trace"); // Must be alone
    end
endmodule
```

\$recordfile

The \$recordfile task records basic design information and sets up the recording options
for variables recorded with \$recordvars. This task is optional. If you use it, the task should
be placed before the first \$recordvars task.

Syntax:

```
$recordfile ( <filename> [,"options"] );
```

```
<filename>
```

The name of the database. This can be a string enclosed in double quotes, or the name of a variable that contains the file name. Although not required, the extension .trn is recommended to identify the transition database. A .dsn file is also created with the same base name as the .trn file.

The following table lists the options that you can use with the \$recordfile task.

Note: The following *precordfile* options, all of which have to do with writing incremental files, are not supported except the summary [*file*] option. Using them will generate a warning message.

"incsize = <i>size"</i>			
<pre>"inctime = sin</pre>	<pre>"inctime = simtime"</pre>		
■ "inccpu = <i>cpu</i>	"inccpu = <i>cputime"</i>		
■ "incfiles = co	ount"		
■ "summary[=file	e]" and "nosummary"		
Option	Effect	Default	
"wrapsize= <i>size"</i>	Limit the size of the .trn file before data is wrapped into another file.		
	The size argument is a number followed by B (bytes), κ (kilobytes), M (megabytes), or G (gigabytes). The default is M.		
	When the transition data exceeds the specified size, the oldest transitions are overwritten by newer transitions. However, transitions are written to the file, and discarded from the file, in blocks of about 4-5 Mb. This means that the actual size of the database can be considerably larger than, or smaller than, the specified size.		
	It is recommended that the maximum size be at least 10 Mb, if specified.		
"sequence"	Save sequence information (the sequence in which events occurred). This is necessary for tracing.	"nosequence"	
"compress"	Compress the database. Sequence information is not compressed.	"nocompress"	

Example:

In the following example, a design file named adder-1.dsn and a transition file named adder-1-1.trn is created. The transition file is compressed. Sequence time is recorded in the database. You can record sequence information with either the \$recordfile or the \$recordsetup task.

You can use the "compress" and "sequence" options together, but only transition information is compressed; sequence information is not compressed.

```
module record;
.....
initial $recordfile("adder", "compress", "sequence");
$recordvars;
endmodule
```

\$recordsetup

The *\$recordsetup* task records basic design and hierarchy information and sets up the recording options for variables recorded with *\$recordvars*. This task is optional. If you use it, the task should be placed before the first *\$recordvars* task.

When *\$recordsetup* is called, the scope hierarchy is recorded in the design file immediately. However, primitives and variables are not recorded until *\$recordvars* is called.

Syntax:

```
$recordsetup( ["options"]);
```

The following table lists the options that you can use with the *\$recordsetup* task.

Note: The following *frecordsetup* options, all of which have to do with writing incremental files, are not supported. Using them will generate a warning message.

- "incsize = *size*"
- "inctime = simtime"
- "inccpu = cputime"
- "incfiles = count"
- "summary[=file]" and "nosummary"

Option	Effect	Default
"design= <i>name"</i>	Create a name for the design.	Name of the first top scope found.

Option	Effect	Default
"version= <i>name"</i>	Name this version of the design.	Next number (based on the files in the current directory or the directory specified with the "directory" option).
"run= <i>name"</i>	Name this particular simulation run.	Next number (based on the files in the current directory or the directory specified with the "directory" option).
"directory=path"	Specify the directory where the files will be saved. If the specified directory does not exist, it is created for you.	Current working directory.
"wrapsize= <i>size"</i>	Limit the size of the .trn file before data is wrapped into another file.	
	The size argument is a number followed by B (bytes), K (kilobytes), M (megabytes), or G (gigabytes). The default is M.	
	When the transition data exceeds the specified size, the oldest transitions are overwritten by newer transitions. However, transitions are written to the file, and discarded from the file, in blocks of about 4-5 Mb. This means that the actual size of the database can be considerably larger than, or smaller than, the specified size.	
	It is recommended that the maximum size be at least 10 Mb, if specified.	
"sequence"	Save sequence information (the sequence in which events occurred). This is necessary for tracing.	"nosequence"

Option	Effect	Default
"compress"	Compress the database. Sequence information is not compressed.	"nocompress"

Example 1:

In the following example, a design file named data/adder-1.dsn is created. If adder-1.dsn already exists in the data directory, adder-2.dsn is created. A transition file named data/adder-1-1.trn is created. If this file already exists, a file called adder-2-1.trn is created.

```
module record;
.....
initial
begin
$recordsetup("directory = data", "design = adder");
$recordvars;
end
endmodule
```

Example 2:

In the following example, a design file named data/adder-algo1.dsn is created, or replaced if it exists. The database is compressed.

\$recordon/\$recordoff

Use the *\$recordon* and *\$recordoff* tasks to turn recording on or off, respectively. Recording can be turned on or off at selected times or based on conditions in Verilog.

The *frecordoff* task does not close the database file. Variable transitions are not recorded during the period where recording is off. All recorded variables are updated to their current values when recording is turned back on.

Example:

In the following example, the *\$recordon* and *\$recordoff* tasks are used to record variables for a portion of the total simulation time.

```
module record;
. . . . .
. . . . .
  initial
    begin
      $recordvars;
      $recordoff;
    end
endmodule
module top;
reg clock;
  initial
    begin
      #0 clock=0;
      #100 clock=1;
      #100 clock=1;
      #100 clock=0; $recordon;
      #100 clock=1;
      #100 clock=0;
      #100 clock=1; $recordoff;
      #100 clock=0;
      #100 clock=1;
    end
endmodule
```

\$recordclose

Use the *\$recordclose* task to close an open database. This task stops the recording of data, flushes buffered data to the database, and closes the database.

Example:

```
module record;
.....
initial $recordclose;
endmodule
```

\$recordabort

Use the *\$recordabort* task to abort recording to a database that is no longer wanted. Any buffered information not yet written to the database is discarded, and the database is deleted. Any current interactive connection to waveform viewer is also aborted.

Example:

```
module record;
......
```

initial \$recordabort; endmodule

Maximizing Default Acceleration

This appendix describes the following:

- <u>Overview</u> on page 285
- <u>Controlling the Application of the Default XL Algorithm</u> on page 286
- Items Supported by the Default XL Algorithm on page 287
- <u>Items Unsupported by the Default XL Algorithm</u> on page 288
- <u>Differences between Default XL and Non-XL Algorithms</u> on page 290
- Potential Problems with Default XL Algorithm on page 291
- Measuring and Optimizing Code on page 292
- <u>Hardware Upgrades</u> on page 303
- <u>Reducing Executed Code</u> on page 305
- <u>Behavioral Performance Improvements</u> on page 307

Overview

Every Verilog-XL simulation employs two methods of acceleration by default:

- the XL algorithm
- the behavioral performance improvements

The default XL algorithm provides accelerated gate-level and switch-level simulation that is 10 to 20 times faster than simulation without the XL algorithm. The default XL algorithm accelerates nets declared without the vectored keyword, standard primitives, or user-defined primitives. The plus option +caxl accelerates the continuous assignments that conform to the restrictions discussed in <u>"Chapter 5, Assignments"</u>.

The behavioral performance improvements are most beneficial to sequential blocks that do not include timing controls.

As with any system, the maximum performance that Verilog-XL can deliver is limited by the weakest component. The complete simulation process involves the following dynamic activities, each of which affects overall system performance:

- application of stimuli to the circuit being simulated
- activity within the circuit being simulated
- determination of the circuit response and the reporting of that response in a form recognizable to you

Accelerated logic simulation runs fastest when the circuit is self-stimulated and no reporting of circuit response is required. The reason: Logic simulation accelerators accelerate only the simulation of the circuit. As you increase the number of stimulus patterns to be applied or the number of responses to be reported, the memory overhead that these activities contribute can significantly reduce the overall throughput achievable from a logic simulation accelerator. This degradation of throughput becomes noticeable when the overhead of first and third items in the previous list begins to approach the time involved in second item.

Thus, in order to get the most out of a logic simulation accelerator, you must be cognizant of the factors affecting throughput. This chapter explains how you can minimize the overhead involved in first and third items previously listed in order to maximize overall throughput.

Controlling the Application of the Default XL Algorithm

Verilog-XL simulations invoke the XL algorithm by default, eliminating the need to use -a command-line option which existed in versions before Verilog-XL 1.7. The +noxl command-line option makes Verilog-XL operate as it did in Verilog-XL 1.6c and earlier versions; that is, the XL algorithm is disabled by default, and the -a command-line option is required to invoke the algorithm.

Another method of controlling the application of the XL algorithm is a pair of compiler directives, `accelerate and `noaccelerate as shown in the following example:

```
`noaccelerate
module a;
    ...
endmodule
`accelerate
module b;
    ...
endmodule
```

module c; ... endmodule

The example shows these compiler directives in a description. Module a is not accelerated, but modules b and c are accelerated. These compiler directives can only be specified outside of module definitions. Therefore either all instances of a particular module are accelerated or none are accelerated.

In situations where the command line includes the +noxl option and the -a option and the description includes these compiler directives, the -a option takes precedence, and the XL algorithm accelerates the entire design.

Items Supported by the Default XL Algorithm

The default XL algorithm accelerates these primitives and net types.

nand	notif1	rcmos
nmos	or	rnmos
nor	pmos	rpmos
not	pulldown	xnor
notif0	pullup	xor
	nand nmos nor not notif0	nand notif1 nmos or nor pmos not pulldown notif0 pullup

Supported net types

supply0	triO	trior	wire
supply1	tri1	trireg	wor
tri	triand	wand	

The default XL algorithm also accelerates combinational and sequential UDPs.

There are some restrictions on acceleration discussed in other sections of this chapter. Any unsupported items are automatically processed by the non-XL simulation algorithm. Specifying the driving strength of a gate has no impact on whether or not the default XL algorithm can accelerate it.

"Keeping Primitives Accelerated" on page 295 discusses how to keep primitives accelerated.

Items Unsupported by the Default XL Algorithm

The non-XL algorithm automatically processes items that the default XL algorithm cannot accelerate.

The following restrictions determine unsupported items:

The following bidirectional primitives are accelerated either by a default algorithm or by invoking the Switch-XL algorithm. <u>Chapter 8, "Switch-Level Simulation,"</u> discusses the algorithms that accelerate bidirectional elements.

tran	tranif0	rtranif1
tranif1	rtran	rtranif0

- buf and not gates with more than one output are not accelerated.
- Generally, nets with non-zero delay are not accelerated. The only exception is a trireg net specified with a charge decay time. (See the discussion of trireg nets in trireg Net Charge Decay.)
- The default XL algorithm can support no more than 32,767 distinct gate delays. Once this limit is reached during compilation, any subsequent gates with distinct delays cannot be accelerated. The following example shows what makes one set of delays distinct from another:

```
#(1,4) is distinct from #(1,3)
#(2,3) is distinct from #(3,2)
#(2,2) is distinct from #2
```

- Primitives that have non-constant (dynamic) delay expressions are not accelerated.
- Primitives with an expression involving any kind of operator on an input are not accelerated unless the +cax1 option is used. See "<u>Accelerated Continuous</u> <u>Assignments</u>".
- Primitives that have an input connected to an expression are not accelerated.
- Vectored nets (nets declared with the vectored keyword) are not accelerated.
- Any net that has a continuous assignment made to it is not accelerated unless the +cax1 option is used.
- Any forced net is not accelerated. A net may start out as an accelerated net but once a force statement is activated on it, it can no longer be accelerated, even after it is released.
- Gates with inputs connected to bit selects of vector registers are not accelerated, but primitives connected to bit-selects of wires are accelerated.
Reporting Non-XL Structures Using \$shownonxl

Minimizing the number of non-XL events in a simulation maximizes the benefit of the default XL algorithm. Locating non-XL structures and optimizing them is part of reducing the number of non-XL events. Reducing the number of events that occur in non-XL structures is the other part of this process.

The \$shownonxl() system task locates non-XL structures.

The parentheses in the <code>\$shownonxl()</code> system task can contain no argument, or they can hold the instance name of a module. If the parentheses contain no argument, the <code>\$shownonxl</code> task searches your entire simulation for non-XL structures. If the parentheses contain the instance name of a module, the scope of the <code>\$shownonxl</code> task is limited to the module named in the parentheses.

The report that <code>\$shownonxl</code> issues contains the following types of entries:

■ GATE NOT ACCELERATED

Gates in this category are not accelerated by the XL algorithm. Bidirectional primitives are accelerated by a default algorithm or by invoking the switch-XL algorithm. <u>Chapter 8,</u> <u>"Switch-Level Simulation,"</u> discusses the algorithms that accelerate bidirectional elements. Multi-output buf gates and gates with non-constant delay expressions are not accelerated by the XL algorithm.

■ GATE DECELERATED

Gates in this category are accelerated by the XL algorithm, but have been decelerated. One reason for this deceleration is that an expression is connected to one terminal of the gate. Another possible reason is that the output net of the gate has been forced while you were in interactive mode.

■ NET (XL->NORMAL)

Nets in this category are evaluated in XL, but they pass value changes out of XL for normal processing. If the value on the output net of a gate is used in a procedural expression, display statement, or monitor statement, the net is in this category.

■ NET (NORMAL->XL)

Nets in this category are evaluated outside XL, but they pass value changes into XL. An example is a net that passes the value that it receives through a non-accelerated continuous assignment to the input terminal of an accelerated primitive.

Differences between Default XL and Non-XL Algorithms

When processing multiple events that occur at the same simulation time, disabling the default XL algorithm may cause Verilog-XL to process events in a different order. Disabling the default XL algorithm may therefore create certain redundant events. This difference can be seen in traces produced by the *ssettrace* system task.

Because the default XL and non-default XL algorithms may process simultaneous events in a different order, it is possible for zero delay oscillations to occur when using one of the algorithms but not when using the other. Consider the simple latch shown in the following example and subsequent diagram:

```
module latch(q, nq, set1, set2, reset1, reset2);
output q, nq;
input set1, set2, reset1, reset2;
nand
    gq (q, nq, w1),
                                        //these two gates form a zero delay
                                         //loop which can cause an oscillation
    gnq (nq, q, w2);
nand
    g1 (w1, set1, set2),
    g2 (w2, reset1, reset2);
endmodule
    set1
                     w1
              q1
    set2
                             gq
                                         q
 reset1
                     w2
                             gnq
                                         nq
              g2
 reset2
```

The feedback loop between q and nq has been specified with zero delay. If input changes cause both w1 and w2 to go from 0 to 1 at the same simulation time, then a race situation is created. The race can be resolved in one of three ways, depending on how the simultaneous events are processed by the simulation algorithm:

- The latch could settle in the "0" state.
- The latch could settle in the "1" state.
- A zero delay oscillation could be triggered.

Because the default XL and non-XL algorithms may process simultaneous events in a different order, the two algorithms could produce different results in this type of situation. The best remedy for this type of problem is to avoid using any feedback loops that are made up entirely of zero delay gates.

Another case in which the default XL and non-XL algorithms may differ is when a pulse passes through a gate and the width of the pulse is equal to the gate's delay. Using the non-XL algorithm, such a pulse always passes through the gate. Using the default XL algorithm, there is no such guarantee; depending on the order that simultaneous events are processed by the default XL algorithm, such a pulse sometimes passes through a gate and at other times does not. This could produce different results in potential race situations, such as when a unit delay flip-flop is clocked with a unit delay pulse.

Finally, the default XL and non-XL algorithms differ in how they respond to interrupts. An accelerated event is defined as a value change on an accelerated gate or scalar net. A normal event is any other simulation event, of which there are many kinds. If you want to interrupt the simulation to enter interactive mode, you can only do so upon the execution of a normal event. Thus, in non-XL mode it is possible to interrupt the simulator manually or at breakpoints at any time. In XL mode, however, an interrupt can only occur at a timestep.

Potential Problems with Default XL Algorithm

The following is a list of potential problems to be aware of when running Verilog-XL in the default XL mode:

- When you use the -i command-line option to replay an interactive dialog from a previously saved key file, you must use the same acceleration mode that was used in the original run if the key file contains asynchronous interrupts. Otherwise, the interrupts in the replay occur at points different from the original run.
- If you perform single step tracing in interactive mode while accelerated events are being executed, you may get a large amount of trace output from the accelerated events. Moreover, if all the trace messages are from accelerated events, you can not control the amount of information displayed.

A related problem occurs when you perform an asynchronous interrupt during a full trace: You may get a large amount of trace output from accelerated events before the interrupt is acted upon. These problems occur because it is not possible to stop in interactive mode during an accelerated event.

Sometimes asynchronous interrupt requests do not work if you are simulating a selfcontained circuit model that requires little or no external stimulus (for example, one with a clock stimulus produced by accelerated gate-level logic). One way to solve this problem is to specify a statement like the following in your source description:

always #50;

In interactive mode, the same effect can be achieved by entering this command:

C9> forever #50;

Both solutions cause a normal event to take place every 50 time units, providing frequent opportunities for asynchronous interrupts to be recognized. The actual delay you choose depends on the characteristics of the circuit being simulated and how responsive you require the simulator to be. However, using too small a delay could produce significant memory overhead, which may adversely affect the simulator's performance.

Measuring and Optimizing Code

Some of the many different areas in which the performance of a given model can be improved are methodology changes, accuracy trade-offs, and modeling style. The following sections describe the available improvements, the amount of work required for each improvement, and the advantages and disadvantages of each improvement.

Estimating Model Speed

"How fast will my model run?" This is one of the most commonly asked questions about Verilog-XL. It is also the hardest to answer. The design complexity, methodology and level of detail, the experience of the designers, and simulation platform all influence the speed of a model.

Note: While processor simulations may range from 10 to 0.1 cycles per second, a model should run at more than one cycle per second to be useful. Running any slower does not allow sufficient time for verification; running much faster means that the model may not be accurate enough. Next generation designs are created on current generation processors. Because the rate of change of CPU performance and complexity has remained the same, the metric described above has remained the same in several designs studied over the last few years.

An easily used but often misunderstood measure of simulator performance is events per second (e/s). Most gate simulators have similar event counts when running a fixed circuit and stimuli. Thus, the metric of e/s was created to compare simulator performance.

An event that evaluates an AND gate takes about the same amount of time as an OR gate event. Behavioral events can take widely differing amounts of time, such as an integer compare versus a procedure call with a dozen parameters. Behavioral events can be faster or slower than structural events. The event count for a logic function depends on the model style.

Establishing a Metric

Before you start making any model changes, you should pick a representative simulation run and measure both the wall clock time and the CPU time. Use this run to judge whether a change improves performance. The Verilog-XL system function <code>\$simevents</code> returns the cumulative number of events and the function <code>\$cputime</code> returns the cumulative CPU time in tenths of a second.

A pure RTL model can have a higher e/s rate than a structural model with the same functionality. For example, an RTL processor model may have an e/s rate down in the structural range. Changing the latches from UDPs to behavioral models (without applying the default behavioral improvements) doubles both the e/s and the run time. In this case, judging simulation performance solely by events per second shows that the model speeds up when in fact, it really slows down.

There are three basic ways to maximize the performance benefits that the default XL algorithm offers:

- Changing from a slow set of events to a faster set for example, using more accelerated events.
- Reducing the total number of events while maintaining the same functionality for example, replacing several gates with a UDP.
- Changing the functionality of the model to reduce or eliminate unnecessary sections.

Modeling at Different Levels

Model performance depends on the level at which the model is written. A system-level model runs quickly, but has little detail, while a structural model runs much more slowly, but has greater accuracy. Your model should be written at the highest level dictated by the need for accuracy.

A Verilog system-level model generally runs slower than one written in a high-level language such as C or Pascal. The advantage of using Verilog is that the model can be reused at lower levels.

Because the first users of Verilog used extensive gate-level modeling, the default XL algorithm, UDPs, and other features were created to speed up these models. The performance of a behavioral model may therefore be comparable to the equivalent structural model, especially if the behavioral model has not been optimized and runs without the default behavioral improvements.

RTL models can have worse performance than their equivalent behavioral and gate models. The following are typical sources of overhead that can retard RTL models:

- A continuous assignment statement may be triggered multiple times in a single time slot as each bit of a multi-bit signal changes.
- Large logic blocks may be modeled with functions that need multiple signals passed in and that need several signals concatenated together to act as the function result.

Reducing Memory Overhead from Switching Algorithms

Verilog-XL places each event that it simulates in an accelerated queue or an unaccelerated queue for the appropriate time step in the simulation. During any time step, Verilog-XL processes all events in the accelerated queue before the events in the unaccelerated queue. The processing of events in one queue may produce more events during a time step than processing of events in the other queue. Verilog-XL continues to go back and forth between the queues until they are both empty. Once all events scheduled for a time step have been completed, Verilog-XL advances to the time at which the next event is scheduled. The processing of event queues then takes place in the new time step.

Verilog-XL incurs some overhead when switching between the default XL and non-XL algorithms. This overhead was reduced in version 1.6, but it may still cause an accelerated model to run slower than a non-accelerated model. This difference in efficiency often happens in a model that is mostly behavioral or RTL with a small percentage of structural constructs. In such a model, the gain from speeding up a small number of gates is lost because of the overhead of switching between the two modes. This effect is most pronounced when gates are intermixed with behavioral code because this causes frequent mode switching. Concentrating gates in discrete sections of a model reduces the switching overhead, because Verilog-XL stays in the XL mode longer.

Note: An easy way to increase simulation speed is to run a fixed length test without XL (+nox1), then one with XL (the default) and finally one with continuous assignment statements accelerated (+cax1). Measure the CPU time for all three runs and use the combination of switches that gives the best result.

If your model runs faster in non-XL mode (with +noxl), you may want to change non-XL events, such as behavioral code, to XL events, such as gates and UDPs. Whenever you make changes to a model, test whether it runs faster in XL or in non-XL mode.

Keeping Primitives Accelerated

In a gate-level net list, most gates are connected to other gates and are therefore accelerated. Gates may be unaccelerated at locations where they are connected to either behavioral models or the stimulus.

An easy way to find the unaccelerated primitives in a design is to run Verilog-XL with tracing turned on using *\$settrace*. All accelerated events use the keyword FROMXL. Locate gate events without FROMXL. Then determine why the default XL algorithm cannot accelerate each unaccelerated gate.

When an unaccelerated event is swapped for an accelerated event, performance for that event typically improves tenfold. The improvement is greatest when one or both identifiers come from or go to another accelerated primitive. This is due to the overhead involved when propagating events from one algorithm to the other.

The remaining discussions in this section describe techniques for keeping gates accelerated.

Do not use forced net initialization.

The following example shows a replacement for forced net initialization:

```
// Unaccelerated
initial
begin
    force a = 1;
    #100 \text{ force } a = 0;
    #100 release a;
end
// Accelerated
reg f_a, r_a
bufif1 (a, r a, f a);
initial
begin
    f_a = 1;
    r a = 1;
    \#100 r_a = 0;
    \#100 f_a = 0;
end
```

Rewrite a pullup on a wire.

The following continuous assignment of 1 with a pull strength to the net signal is not efficient for the default XL algorithm:

assign (pull1,pull0) signal = 1;

Replacing the preceding code with the following code increases the efficiency:

pullup (signal);

Use unidirectional switches, not bidirectionals.

Bidirectionals are essential only if signals must pass through a switch in both directions, or for modeling turn-on and turn-off delays. The default XL algorithm cannot accelerate gates whose outputs connect to bidirectional switches.

Make sure inputs are wires or scalar registers.

For default XL algorithm acceleration, each gate input should connect to a wire, a tri or a single-bit register.

Buffer behavioral outputs connected to primitive outputs.

The following figure shows how to avoid making a connection between a gate output and a register that can prevent the default XL algorithm from accelerating the gate.

reg a is the output for a block of behavioral code This buf isolates the gate from the register, which makes it possible for the XL algorithm to accelerate the gate. If the buf is driven by a bit-select of type reg, additional buffering is required.

Delays on primitives must be constant expressions.

Parameterized or backannotated expressions maintain constant values during simulation, and they do not prevent the default XL algorithm from accelerating primitives. Note that using reals to specify delays on accelerated primitives requires the same amount of simulation time as using integers.

Expressions on inputs are similar to continuous assignments.

The default XL algorithm can accelerate an expression that is an input for a gate if the +caxl algorithm can accelerate a continuous assignment in which the expression to be accelerated is on the right-hand side.

Modeling Clock Generators

Clocks are generally the most active part of a simulation and can be responsible for the majority of the unaccelerated events in the system. Converting behavioral clock descriptions to gate-level oscillators can save between five and twenty percent of the CPU time.

The following example shows how to convert a behavioral clock (unaccelerated) to a gatelevel clock (accelerated):

```
// Unaccelerated
reg clk;
always
begin
    #low clk=1;
    #high clk= 0;
end
// Accelerated
req rst;
nand #(high, low) (clk, clk, rst);
initial
begin
    rst = 0;
    #low #0
    rst = 1;
end
```

Using Behavioral Profiler

The Behavioral Profiler is the best tool for finding behavioral model performance problems. The following is one way to use it:

- 1. Start Verilog-XL, preferably under a scrolling xterm window.
- 2. Run the model through any initialization and reset code, to prevent incorrectly biasing the Profiler.
- 3. Turn on the Profiler with \$startprofile.
- 4. Run the model for approximately one minute of CPU time. This should be enough to run approximately one hundred clock cycles, so that most of the model code has been exercised. You may want to run more cycles initially to get a more accurate measurement.

- 5. Look at the performance measurements with *performance* measurements with *performance* need to look at the first dozen statements and modules.
- 6. Get a list of the code and counts for a module with the command *\$listcounts* <*module_name*>. You may now look at the report and the module listing together to see the bottlenecks. Be sure to include the +listcounts option on the command line to enable the task.

A *bottleneck* is any line that takes more than two percent of the total execution time or any module that takes five percent of the total execution time. These are good targets for optimization.

The Behavioral Profiler breaks a model into lines of code and module instances, but it does not show a summary for all instances of a particular module. For example, if a latch is used 1000 times in a design, the Behavioral Profiler will probably not list any single instance of it. However the latch is an excellent target for optimization. Use the Verilog-XL system task \$showallinstances to see which modules have been instantiated the most. Converting modules that can be modeled as UDPs can save at least five or more gates. Look for registers and latches modeled at the gate or switch level. These types of models usually have more activity in them compared to other models.

If the profiler report contains entries for continuous assignments, then they are not being accelerated. You may want to investigate why the statements are unaccelerated by checking the statements against the latest set of accelerated operators in Verilog.

If your model has no obvious bottlenecks, you may need to restructure it to make broad changes, or you may require a faster machine to gain additional performance.

Using Different Coding Methods

The inner workings of event simulation are not always obvious. When making optimizations you should run small experiments to determine how Verilog-XL performs. Keep in mind that the relative performance of various constructs often changes in a new release of Verilog-XL.

Once you find a bottleneck, change the model style to use either more efficient events or fewer events. In general, the best way to speed up a model is to reduce the number of events. There are two ways to accomplish this: execute more efficient statements, or execute fewer statements.

Verilog does not perform all the code optimizations that high-level language compilers perform. You can move constant expressions outside of loops and simplify highly active continuous assignment statements to reduce the number of events and speed up your model.

The three most common operators that slow down models are:

- bit extracts
- bit inserts
- concatenation

A common coding mistake using these operators is to make separate statements for every bit of an expression. The following examples show two inefficient ways to model a single bit signal ANDed with an 8-bit bus:

```
for(i=0; i<8; i=i+1) result[i] = cond & bus[i];
or
result[0] = cond & bus[0];
result[1] = cond & bus[1];
...
result[7] = cond & bus[7];</pre>
```

A faster method is to use the select operator as follows:

result = cond ? bus : 0;

There is always a trade-off between speed, accuracy, and the ability to maintain a design. Continuous assignment statements may use the select operator to efficiently model a multiplexer, but a complex, multilevel multiplexer may be more understandable if it is broken out into a separate function, using if and case statements.

The next example shows a method of measuring the relative speed of Verilog-XL statements. Various versions of a piece of logic are put inside each of the loops, and the CPU time used by each loop, minus the overhead, is reported. The example compares a 32-bit addition with a 33-bit addition. The initialization code sets the number of loops and performs a stop to put Verilog-XL into interactive mode. You then run the timer task by typing timer; This task should be run several times to obtain an average.

```
module timing;
integer i, start, overhead, diff, maxloop;
req [31:0] a, b;
reg [32:0] c, d;
initial
begin
    maxloop = 10000;
    $display("Timing model initialized");
    $stop;
end
task timer;
begin
    start = $cputime;
    for (i = 1; i < maxloop; i = i + 1)
        begin
            // Do nothing
        end
    overhead = $cputime - start;
    $display("Reference loop took %0d ticks", overhead);
```

```
start = $cputime;
    for (i = 1; i < maxloop; i = i + 1)
        begin
            a = a + b;
        end
   diff = $cputime - start - overhead;
    $display("32 bit add. took %0d ticks", diff);
    start = $cputime;
    for (i = 1; i < maxloop; i = i + 1)
        begin
            c = c + di
        end
    diff = $cputime - start - overhead;
    $display("33 bit add. took %0d ticks", diff);
    $stop;
end
endtask // timer
endmodule // timing
```

Using UDPs

User Defined Primitives (UDPs) allow you to compress a piece of logic into a single primitive that executes as fast as a single Verilog-XL primitive such as an AND gate. UDPs thus execute faster than the equivalent behavior statements.

Note: Whenever possible, use UDPs to replace blocks of logic. For example, if a design uses behavioral latches extensively, changing the latches to UDPs can make the model run four times faster. This is because triggering the behavioral latch takes several events to process; triggering the UDP latch takes only one event.

Verilog-XL includes an extensive library of over 400 UDPs including models for flip-flops, latches, multiplexers, and many others. Cadence developed these UDPs for its own model libraries, and they are carefully written to reduce pessimism.

It is a mistake to avoid UDPs because they seem limited by having only a single output. UDPs are so efficient that even if you have to use a separate one for each output, the model is still faster. For example, a D flip flop has q and qbar outputs. The Cadence UDP library includes a UDP to generate the q output and another to generate the qbar output. A model that uses both UDPs is still faster than one built from NAND gates. Another alternative is to use a UDP to create the q output and an inverter to make the qbar output, although this complicates assigning lumped delays.

Using Event Controls

Selecting and ordering event controls can reduce the number of times a procedural block is executed. If a set of event controls times a block's execution, consider the relative activity of the signals in the event controls. The least active signal should be in the first event control.

For example, the following inefficient description evaluates the expression expr on every positive edge of the clock:

```
always @(posedge clock)
if (expr)
begin
...
end
```

The following efficient description evaluates the expression only when an identifier in the expression changes:

```
always wait (expr)
@(posedge clock)
if (expr)
begin
...
end
```

Event controls are also useful in determining when to stop a simulation. An inefficient way to do this is to count the clock cycles and then compare the count to some limit as in the following example:

You can then remove the comparison at each cycle if you know the clock period as in the following example:

```
task run;
input [31:0] cycles;
#(cycles * clock_period)
    $stop;
endtask
```

Using Aliases

There are some cases in which it is helpful to have a simple name that represents a group of signals, as in following example:

```
assign R12 = \{r.a, r.b, r.c\};
```

This is a reasonable way to model R12 if the right-hand side of the expression changes as often as R12 is used. Sometimes the alias is created for debugging purposes and is not used

that often. A better way to model this would be to use the compiler directive `define. The expression defined by `R12 is only evaluated as needed.

```
`define R12 {r.a,r.b,r.ac}
```

Using Level-Sensitive Behavior

Modeling level-sensitive behavior in is not always straightforward. Most Verilog HDL constructs are best for combinatorial or edge-triggered logic. Procedural continuous assignments (assign and deassign used with registers) are intended to model level-sensitive logic, but they must be placed in level-sensitive constructs to do this efficiently.

Procedural continuous assignments must be placed inside either an initial or an always block of code, each of which is an edge-triggered construct. Consider the following piece of code that models a transparent latch:

```
always
begin
if(!clk)
R = data
#1;
end
```

If the clock period is 100 time-units long, the if statement is executed 100 times, and the assignment is made 50 times each clock period. However 99 of the if statement executions and 49 of the assignments are unnecessary to model the level-sensitive functionality. The following latch model is more efficient than the preceding code because its wait statements are level-sensitive. They minimize simulator activity by describing only the necessary functionality.

```
always
begin
    wait (!clk);
    assign    R = data;
    wait (clk);
    deassign R;
end
```

The most efficient way to model a transparent latch is to avoid procedural continuous assignment statements and to use only procedural statements. This avoids the overhead of forcing signals. Consider the following example:

```
always @(data)
if (!clk)
R = data;
```

You can use the assign and deassign statements for a level-sensitive set of reset signals, but do not use these statements if other procedural statements can accomplish the same results.

Hardware Upgrades

The simplest method of speeding up a simulation is to use more hardware. The increase in speed that you get depends on the type of model and the hardware changes you make. Simulation of a self-stimulating model (such as a processor executing a test out of a memory) is typically limited first by CPU speed, then by physical memory size, and lastly by I/O bandwidth. Models driven from stimulus files may be CPU or I/O limited, but they are rarely limited by the amount of RAM in the system.

Upgrading the CPU, often by running a simulation on a server instead of on a workstation, is only successful if the server has the capacity to run the additional jobs, and if it is not already fully loaded by other processes. Current servers only offer an incremental performance improvement over a workstation for a single process. If that process has to compete with several other processes, it may run slower than it does on a workstation.

If a job can be split onto several processors by breaking a long test into smaller pieces, the wall clock time can drop below the CPU time, but at the expense of requiring more attention by the designer. Time spent developing an automated regression testing capability has a high payback.

Memory limitations are rarely seen with smaller ASICs but are becoming increasingly more common with simulations of large processors. Compounding this problem, the effects of paging and swapping on a workstation are not as likely to be noticed or understood today by someone unfamiliar with virtual memory concepts as they were in the past when mainframes and minicomputers had system administrators who understood the details of the operating system.

What are paging and swapping and how do they affect simulation? In a virtual memory operating system such as UNIX[®], the virtual memory requirements of each process have to be balanced with the physical memory available in the processor. When a single process needs more virtual memory than the available physical memory, UNIX must move pages from physical memory to disk and back again. This is called *paging*. The first symptom of paging is that the disk drive chatters while the program is running because of the added disk traffic. *Swapping* occurs when all the processes together use more virtual memory than available physical memory, and so UNIX must swap out the inactive processes, that is copy their data out to disk to make room for the active processes.

The net result of paging and swapping is that the wall clock time for a run always increases; depending on the operating system, the CPU time charged to a process may also increase.

Because Verilog-XL usually runs as the primary process, it suffers from paging when the simulation data structure grows too large. Large simulation jobs have poor virtual memory behavior, because they tend to run through the entire simulation data structure from

beginning to end every clock cycle with no locality. This results in *thrashing*, in which the next page needed is often the one just swapped out.

To monitor paging in UNIX, use the command vmstat as follows:

- **1.** In a windowing environment such as X, open two UNIX windows.
- 2. In one window, type the UNIX command vmstat 5. This will print virtual memory statistics every 5 seconds. The important data is in the column under "fre", for free page list size in kilobytes, "po" for page out (paging rate), and the last three columns that list the percentage of CPU time spent in user mode (Verilog-XL), system mode (UNIX), and idle.
- 3. In the other window, start Verilog-XL with the -s switch to stop after compilation. The free list size should drop during compilation, but remain above a certain minimum. Under SUN-O/S, paging may begin when the free list length falls below 300 kilobytes, depending on the system's parameters. At this point, the paging rate will rise sharply because the system is thrashing, and performance will plummet.

Note: To compute the minimum physical memory needed to simulate a model, compile the model and use the *\$stop(2)* command to obtain the memory usage of the simulation. Add between three and five megabytes for the Verilog-XL image, UNIX, and the windowing software. The CPU should have at least this much physical memory to prevent paging when running a single Verilog-XL job.

The UNIX command ps u displays information about all of your processes. The column "SZ" shows the virtual memory used by the process while the column "RSS" shows the amount in physical memory; both are listed in kilobytes.

Another way to gather statistics on a job is the UNIX C shell command time. If you place the following command in the script for a simulation run, UNIX prints information after Verilog-XL terminates:

set time=(3 "Wall:%E Sys:%S User:%U %P Mem:%Mk PgFlts:%F Swaps:%W")

The 3 tells the C shell to display data about any command that takes more than three seconds of CPU time.

In addition to virtual memory thrashing, medium and large simulations may also suffer *cache thrashing*. Caches depend on code and data locality, but when the data structure size grows larger than the cache size, the cache can actually reduce performance. This causes model performance to drop suddenly when the model's memory size grows larger than the cache. Unfortunately, it is difficult to switch off the cache on most machines.

Reducing Executed Code

This section presents methods of making a model run faster by making high-level changes.

Simplifying the Model

If a model is running too slowly, it may be simulating unnecessary parts of the design, or it may be simulating some parts at unnecessarily low level.

For example, a project may need to run extensive tests on a processor model which contains a core set of three chips: the CPU, the Cache, and the FPU. Measurements of the behavioral model show that the FPU uses more simulation time than the rest of the model. However, only a specific subset of tests execute floating-point instructions, so the extra simulation time is wasted.

The model can be changed so that the FPU code is not activated when it is disabled in the processor status register. In this case, the FPU runs only during initialization and during tests that explicitly enable it. This results in the model run time dropping by a factor of four for tests that do not use floating-point code.

Much of the above model still sits idle while read and write requests are processed by the detailed memory model. If the designers write an "instant" memory model that accepts write requests every cycle, and returns read data in the minimum time allowed by the bus protocol. The model runs twice as quickly because very little time is spent simulating the memory subsystem, and tests run in fewer cycles. Tests that cover CPU memory interactions, then, turn off the instant memory and turn on the detailed model.

Changing Your Debugging Style

The methods used to debug a model can impact its performance. The primary mistake that users make is to save too much information just in case the information is needed when a bug occurs. Verilog-XL owes much of its speed to the fact that it saves *only* the simulation history that is specifically requested.

One common debugging style is to have either the test code or the model code self-checking so that the simulation will stop when a bug is encountered. At this point the waveforms are rolled back to find the sequence of events that may have caused the bug.

In this situation, a trade-off occurs. Is the time spent saving waveform data for all the tests greater than the time spent to restart the failing test and run it with waveforms displayed? It may be that only the last section of the testbench needs to run to produce the bug.

Another debugging style is saving the simulation data at fixed intervals until a bug occurs, and then restarting the simulation with the last simulation data file. The test can repeatedly restart from the last save point until the design error is identified. The ideal save interval is a balance between the time required to save and the time required for simulation. Saving simulation data can slow down Verilog-XL, and the save files can use a large amount of disk space. The Verilog-XL system task \$incsave saves disk space because it saves only data that has changed since the last \$save command.

Capturing Simulation Data

You can capture information from a simulation for post-processing or for later simulation runs. Verilog-XL offers several methods for saving simulation values to a file.

The simplest and slowest technique uses the *fdisplay* task to write data directly to a file. This file is often contains only the lists of values written at the end of every clock cycle. You or a C program can easily read these lists of values, but they are not compact.

The following table shows the data saved cycle by cycle for a bus simulation. The CPU requests the bus in cycle 101, is granted the bus in 102, starts a read in 103, and the data is finally returned in 107, with the memory signaling success.

cmd	address	data		request	grant	terminator
idle	0		0			
idle	0		0	cpu		
idle	0		0	cpu	grant	
read	1200		0			
read	1200		0			
read	1200		0			
read	1200		0			
read	1200	3322110	00			ready
	cmd idle idle read read read read read	cmd address idle 0 idle 0 idle 0 read 1200 read 1200 read 1200 read 1200 read 1200 read 1200 read 1200	cmd address data idle 0 idle 0 idle 0 idle 0 read 1200 read 1200 read 1200 read 1200	cmd address data idle 0 0 idle 0 0 idle 0 0 idle 0 0 read 1200 0	cmd address data request idle 0 0 cpu read 1200 0 read read 1200 0 read read 1200 0 read	cmd address data request grant idle 0 0 cpu idle 0 0 cpu grant idle 0 0 cpu grant idle 1200 0 read 1200 0 read 1200 0 read 1200 0 read 1200 0 read 1200 0 read 1200 33221100 1 1 1

The Value Change Dump (VCD) function in Verilog-XL is faster than the \$fdisplay task, but a post-processor for the VCD's output file must be more sophisticated. The VCD monitors a list of signals that you supply. When one signal changes value, Verilog-XL writes a coded signal name and its new value to the VCD file. See <u>Chapter 20</u>, "The Value Change Dump <u>File</u>" for more information on the VCD.

The fastest method of capturing simulation data is using the Value Change Link (VCL) to link the C routines that you write into Verilog-XL. Like the VCD, the VCL traces a list of signals. When a signal changes value, a C routine is called to process the event. If these routines preprocess the data and reduce the amount written to the file, the VCL reduces the overhead of saving simulation data. For example, the bus trace in the previous table can be compressed into a transaction file as follows:

Bus request @101 by cpu, granted in 102 Read started @103, addr=1200, @107 data=33221100, success Refer to the Value Change Link Application Note for more information and examples.

Reducing Compilation Time

Verilog-XL's compilation speed is fast, but for large models, compilation still takes a significant amount of time. The primary cause of slow compilation is insufficient physical memory, so check for paging when compile times start to rise.

Note: A gate-level model takes approximately five to ten times as much memory to compile as to simulate. A behavioral model with the default behavioral improvements uses approximately five times as much memory to compile as to simulate. Reducing the levels of hierarchy by eliminating the lowest level can significantly reduce the data structure size by reducing the number of modules and ports.

Parsing the input file takes only a fraction of the compilation time; most of the time is spent in allocating and initializing virtual memory. The <code>\$save</code> and <code>\$restart</code> system tasks provide only limited help with this process because a restart must also create the internal data structures in virtual memory. Files created by <code>\$save</code> are beneficial only if the model has a lengthy reset and initialization phase, so that restoring an initialized model provides a significant improvement over compiling, resetting, and initializing.

The *\$reset* task sets the simulation back to its state just after compilation finishes. This system task can be used to run multiple jobs on a single model without recompiling the model between steps. The *\$reset* task is typically much faster than either a compile or a *\$restart*, and it uses much less memory than a compilation.

Behavioral Performance Improvements

Verilog-XL's default behavioral performance improvements reduce behavioral simulation time. The greater the proportion of behavioral statements in your source description, the faster your source description simulates with the improvements. The $+no_speedup$ command-line option disables the improvements. The effects of the improvements are similar to the effects of the +speedup option in Verilog-XL 1.6c.

You can increase the effectiveness of the behavioral performance improvements by limiting the use of the following:

- sequential blocks containing few procedural statements
- statements that involve procedural timing controls

Avoid writing a large number of begin-end blocks that contain a small number of procedural statements. The improvements work best with a small number of begin-end blocks that contain more procedural statements.

Try to limit the number of procedural timing controls in your source description. The following constructs that precede a procedural statement are procedural timing controls:

```
# <number>
@ <identifier>
wait (<condition_expression>)
```

Combining the behavioral performance improvements with the *\$listcounts* system task and the *+listcounts* command-line option can degrade performance significantly. Using the improvements with the following system tasks can cause minor performance degradation:

```
$startprofile
$reportprofile
$stopprofile
```

Simulating with the behavioral performance improvements can differ from simulating without the improvements in the following ways:

- a different simulation event count
- a different order of simulation events for constructs with zero delays
- greater memory requirements
- a non-XL event count reported as zero
- an increase in link time

D

Stochastic Analysis

This appendix describes the following:

- <u>Overview</u> on page 309
- <u>Queue Management</u> on page 309
- Probabilistic Distribution Functions on page 312

Overview

The development of new computer systems often includes some investigation into the performance ramifications of architectural decisions. This can be accomplished using queueing models which use predicted job creation and processing rates to simulate the operation of the system under varying load. Verilog-XL supports this technique through a set of system tasks and functions that manage queues and provide you with random numbers that have specific distributions.

Queue Management

The set of tasks and functions that create and manage queues are:

- <u>"\$q_initialize"</u> on page 310
- <u>"\$q add"</u> on page 310
- <u>"\$q_remove"</u> on page 311
- <u>"\$q full"</u> on page 311
- <u>"\$q_exam"</u> on page 311

\$q_initialize

Syntax:

\$q_initialize (<q_id>, <q_type>, <max_length>, <status>)

Purpose:

The $q_initialize$ system task creates new queues. The q_id parameter is an integer input that must uniquely identify the new queue. The q_type parameter is an integer input. The value of the q_type parameter specifies the way in which the queue functions. You can specify this value as one of the following integers:

- 1 The queue is arranged first-in/first-out
- 2 The queue is arranged last-in/first-out

The <max_length> parameter is an integer input that specifies the maximum number of entries that are allowed in the queue. The success or failure of the creation of the queue is returned as an integer in <status>. The error conditions and corresponding values of <status> are described in <u>"Meaning of the status parameter" on page 312</u>.

\$q_add

Syntax:

\$q_add (<q_id>, <job_id>, <inform_id>, <status>)

Purpose:

The q_add system task places a job in a queue. The q_id parameter is an integer input that indicates the queue to which Verilog-XL adds the job. The job_id parameter is an integer input that identifies the job.

The <inform_id> parameter is an integer input that the queue manager stores in the memory. Its meaning is user-defined. An example of a parameter is the job execution time in a CPU model. The <status> parameter reports on the success of the operation. Error conditions and corresponding values of <status> are described in <u>"Meaning of the status</u> parameter" on page 312.

\$q_remove

Syntax:

\$q_remove (<q_id>, <job_id>, <inform_id>, <status>)

Purpose:

The $q_id>$ parameter is an integer input that indicates the queue from which Verilog-XL removes the job. The $<job_id>$ parameter is an integer output that identifies the job being removed. The $<inform_id>$ parameter is an integer output that the queue manager stores in memory during the q_add . Its meaning is user-defined. The <status> parameter reports on the success of the operation. Error conditions and corresponding values of <status> are described in "Meaning of the status parameter" on page 312.

\$q_full

Syntax:

```
$q_full (<q_id>, <status>)
```

Purpose:

The q_full system function checks whether there is room for another job on a queue. It returns 0 when the queue is not full, and 1 when the queue is full.

\$q_exam

Syntax:

\$q_exam (<q_id>, <q_stat_code>, <q_stat_value>, <status>)

Purpose:

The q_exam system task provides statistical information about activity in the queue $<q_id>$. The task returns a value in $<q_stat_value>$ depending on the information requested in the $<q_stat_code>$ parameter. The values of $<q_stat_code>$ are

described in the following section, and the corresponding information returned in $<q_stat_value>$ is described in the following table:

lf <q_stat_code> is</q_stat_code>	Then <q_stat_value> returns</q_stat_value>
1	current queue length
2	mean inter-arrival time
3	maximum queue length
4	shortest wait time ever
5	longest wait time for jobs still in the queue
6	average wait time in the queue

Meaning of the status parameter

All of the queue management tasks and functions return an output *<status>* parameter. The *<status>* parameter values and corresponding information are described as follows:

- 0 OK
- 1 queue full, cannot add
- 2 undefined <q_id>
- 3 queue empty, cannot remove
- 4 unsupported queue type, cannot create queue
- 5 specified length <= 0, cannot create queue
- 6 duplicate $< q_i d >$, cannot create queue
- 7 not enough memory, cannot create queue

Note: The system tasks \$save and \$restart cannot be invoked after one of the queueing tasks or functions has been invoked.

Probabilistic Distribution Functions

Performing system analysis using queueing models requires that job arrival rates and processing times be represented by random functions which approximate the real world. Experience has shown that certain probabilistic distributions occur often in real systems.

Verilog-XL provides random number generators that return integer values distributed according to standard probabilistic functions. The syntax of these system functions is as follows:

```
$dist_uniform(<seed>, <start>, <end>)
$dist_normal(<seed>, <mean>, <standard_deviation>)
$dist_exponential(<seed>, <mean>)
$dist_poisson(<seed>, <mean>)
$dist_chi_square(<seed>, <degree_of_freedom>)
$dist_t(<seed>, <degree_of_freedom>)
$dist_erlang(<seed>, <k_stage>, <mean>)
```

All parameters must be declared as integer values. For the exponential, poisson, chi-square, t, and erlang functions, the parameters <mean>, $<degree_of_freedom>$, and $<k_stage>$ must be greater than 0.

The nature of these functions is well-documented in the literature. For exact descriptions, refer to any good textbook covering probability and statistics.

For each system function, the seed parameter is an *in-out parameter* — that is, a value is passed to the function and a different value is returned. These functions always return the same value given the same seed. This facilitates debugging by making the operation of the system repeatable. The argument for the seed parameter should be an integer variable that is initialized by the user and that is only updated by the system function. This ensures that the desired distribution is always achieved.

In the \$dist_uniform function, the <start> and <end> parameters are integer inputs
that bound the values returned. The <start> value must therefore be smaller than the
<end> value.

The < mean > parameter is an integer input that forces the average value returned by the function to approach the value specified. This average can only be achieved after many calls to the function.

The *standard_deviation* parameter used with the *sdist_normal* function is an integer input that helps determine the shape of the density function. Higher numbers spread the returned values over a wider range.

The <degree_of_freedom> parameter used with the \$dist_chi_square and \$dist_t functions is an integer input that helps determine the shape of the density function. Higher numbers spread the returned values over a wider range.

Verilog-XL User Guide Stochastic Analysis

Ε

Software Behavior and Recommendations

This appendix describes the following:

- <u>Overview</u> on page 315
- Platform- and Version-Specific Behavior on page 316
- <u>Use of PLI Routines</u> on page 318
- <u>Macro Modules and Port Collapsing</u> on page 319
- Module Paths and Path Simulation on page 321
- <u>Using Module Input Port Delays (MIPDs)</u> on page 322
- <u>Conditional Statements</u> on page 323
- Using the 'timescale Compiler Directive on page 324
- Changing a Parameter During Simulation on page 324
- Performing Modulo Division on \$random Outputs on page 324
- <u>Defining Vector Indices Across Module Boundaries</u> on page 325
- <u>Syntax Recommendations</u> on page 326

Overview

This appendix helps you avoid potential problems when using Verilog-XL. Each section in this chapter alerts you to limitations, unexpected behaviors and, where possible, workarounds for specific problems.

Platform- and Version-Specific Behavior

Restarting from \$save Files Created on Incompatible Hosts

The following rule determines the compatibility of \$save files from different hosts: If the hosts use different Verilog-XL executables, then their \$save files are incompatible.

Typically, \$save files are not compatible across host types. Restarting from \$save files created on incompatible hosts results in a *fatal* run-time error.

System 5 UNIX C Shell Scripts Running Verilog-XL

On some platforms, when you use the interrupt key to terminate a Verilog-XL simulation started by a System 5 UNIX C shell script, the script terminates, but the simulation continues. In this case, you cannot interact with the simulation. To terminate the simulation either let the simulation run to its completion or obtain its process number, and include the number in a UNIX kill command.

The interrupt key has a normal effect on scripts in the Bourne shell that initiate Verilog-XL simulations.

The UNIX command stty -a identifies the interrupt key. The stty command sets terminal characteristics such as the interrupt key. For example, to bind the interrupt to control-c, enter the following command:

stty intr ^C

Pulse Handling in Verilog-XL 2.0 and Earlier Versions

In Verilog-XL 2.0 and earlier versions, the simulator functions only as an inertial delay simulator. This means that during the period of a module path delay, only one transition passes through the path's module input and propagates from the module output.

Inertial delay is the default behavior of the simulator in version 2.1, but you can enable transport delay functionality with the following plus options:

+transport_path_delays

Enables transport delay functionality and pulse control for module path delays.

+transport_int_delays

Enables transport delay functionality and pulse control for interconnect delays.

+multisource_int_delays

Enables transport delay functionality and pulse control for interconnect delays and allows you to have unique source-load delays in multi-source nets.

In versions up to Verilog-XL version 2.1, if you do not use one of the three plus options shown above, PATHPULSE\$ with the argument (0,0) enables you to pass one complete pulse through a module during the period of a path delay.

The following example illustrates passing the second transition composing a pulse during a scheduled delay of the first transition:

```
module pulser;
req pulse a, pulse b;
inner one (pulse_a,pulse_b,out_a,out_b);
initial
begin
    $monitor ($time,,,"pulse_a=%b out_a=%b ``,pulse_a, out_a,
    " pulse b=%b out b=%b ", pulse b, out b);
        pulse a=1'b0;
        pulse_b=1'b0;
    #133 pulse_a=1'b1;
        pulse b=1'b1;
    #2 pulse_a=1'b0;
        pulse_b=1'b0;
    #165
        pulse_a=1'b1;
        pulse b=1'b1;
    #100 pulse_a=1'b0;
        pulse_b=1'b0;
    #100;
        pulse_a=1'b1;
        pulse_b=1'b1;
    #99 pulse_a=1'b0;
        pulse_b=1'b0;
    #201;
    $finish;
end
endmodule
                      // pulser
module inner (pulse_a,pulse_b,out_a,out_b);
input pulse_a,pulse_b;
output out_a,out_b;
buf (out_a,pulse_a),
 (out_b,pulse_b);
    specify
        specparam period=100;
        specparam PATHPULSE$pulse a$out a=(0,0);
        (pulse_a=>out_a)=period;
        (pulse_b=>out_b)=period;
    endspecify
endmodule
```

The following are the results of the preceding simulation:

0	pulse_a=0	out_a=x	pulse_b=0	out_b=x
100	pulse_a=0	out_a=0	pulse_b=0	out_b=0
133	pulse_a=1	out_a=0	pulse_b=1	out_b=0
135	pulse_a=0	out_a=0	pulse_b=0	out_b=0

233	pulse_a=0	out_a=1	pulse_b=0	out_b=0
235	pulse_a=0	out_a=0	pulse_b=0	out_b=0
300	pulse_a=1	out_a=0	pulse_b=1	out_b=0
400	pulse_a=0	out_a=1	pulse_b=0	out_b=1
500	pulse_a=1	out_a=0	pulse_b=1	out_b=0
599	pulse_a=0	out_a=0	pulse_b=0	out_b=0
600	pulse_a=0	out_a=1	pulse_b=0	out_b=0
699	pulse_a=0	out_a=0	pulse_b=0	out_b=0

Registers pulse_a and pulse_b drive signals through out_a and out_b respectively. The paths from the registers to the outputs have delays of 100 time units.

Registers pulse_a and pulse_b have the value of 0 at time 100. At times 133 and 135, the registers change value, propagating a two-time-unit pulse. The signal from register pulse_b is not modeled with the (0,0) specification for the PATHPULSE\$ specparam. Therefore neither transition of the two-time-unit pulse reaches out_b. The signal from register pulse_a is modeled with the (0,0) specification for the PATHPULSE\$ specparam. Thus both transitions of the two-time-unit pulse reach out_a.

At times 300 and 400, the registers change value, propagating a 100-time-unit pulse. Both transitions of the 100-time-unit pulse reach out_a and out_b. The pulse period is equal to the path delay. Verilog-XL can schedule both transitions composing the pulse without any PATHPULSE\$ specification if the pulse length is equal to or greater than the module path delay. At times 500 and 599, the registers change value, propagating a 99-time-unit pulse. The pulse appears at out_a but not at out_b, because the pulse is shorter than the module path delay.

Use of PLI Routines

Calls to PLI Annotation and \$reset

If a Verilog description contains a call to any PLI annotation routine (such as \$sdf_annotate) and you invoke \$reset, the simulation does not work properly. The \$reset does not set the delays back to the pre-annotated values.

Any PLI call to annotate delays must be run only once. The *sreset* must not cause an annotation to be run again. The solution to this problem is to add the following code to an annotation call:

if (\$reset_count == 0) \$<annotate_task>;

PLI and Pulse Control

When PLI routines change the delays of module paths, the reject and error value ranges for signals passing though these paths retain their previously- determined values. These previously-determined values are not the values produced by applying the reject% and error% values that you specified in the pulse control options to the new delays supplied by the PLI routines. The acc_set_pulsere routine specifies the reject and error value ranges that you want in this situation. Thus, you do not use the acc_set_pulsere routine, delays may be inconsistent with pulse limits.

Note: When SDF annotation changes module path delays, the reject and error values change proportionally, conforming to the specified reject% and error%.

Macro Modules and Port Collapsing

Terminal and Port Lists in Macro Modules

In macro modules, the terminal lists in gate instances and the port lists in UDP instances can contain only the following constructs:

- scalar net identifiers
- literal constants

If these lists contain any other values—such as parameters, bit selects, part-selects, or expressions—Verilog-XL does not expand the macro module and instead treats it as though it were a normal module.

Effect of Port Collapsing on Net Delays

When Verilog-XL collapses a port, it does not add the delays on all the items connected across the port to determine the delay on the resulting net. The collapsed net acquires the characteristics of the dominant net, including its delay as shown below:



To ensure that a net retains its characteristics, connect its ends to continuously enabled MOS gates.

For more information about port collapsing, see <u>Chapter 11, "Hierarchical Structures"</u> of the *Verilog-XL Reference*.

Port Collapsing and 'default_nettype Specifications

Collapsing a port in which the internal net type is specified by the `default_nettype compiler directive creates a single net of the same type as the external net. To avoid this result

and to have the normal results of port collapsing, you must explicitly declare the internal net type.

Module Paths and Path Simulation

Rules for Path Destination Signals

Verilog-XL requires that path destination signals follow the same rules that govern accelerated nets.

Specifically, a path destination signal must be a net driven only by a gate-level primitive. The only restriction is that the primitive must be a bidirectional transfer gate. For a complete discussion of the rules governing accelerated nets, see <u>Appendix C, "Maximizing Default</u><u>Acceleration."</u>

Path Output Nets With Multiple Drivers in One Module

Nets used as path destinations in a single module cannot be wired together—inside or outside the module—if they have different drivers within the same module. However there are some situations in which you can overcome this limitation by replacing wired logic with gated logic to create a single driver to the path output. See <u>"Driving Wired Logic Outputs"</u> in *Verilog-XL Reference* for more information on driving wired logic.

Note: Although path output nets *inside* a module may have only one driver, nets *outside* the module may have multiple drivers as long as they are in different module instances.

Path Outputs That Drive Other Path Outputs

If one path output drives another path output through a gate, the driving path must have the smaller delay. Otherwise, the simulator schedules an event on the driven path output later than expected—at the same time as the event on the driving path output.

You can circumvent this problem by placing a buffer on the driving output, as described in <u>"Simulating Path Outputs that Drive Other Path Outputs</u>" in *Verilog-XL Reference*.

Strength Changes That Occur on Path Inputs

Strength changes always propagate through a circuit using internal gate and net delays, *not* path delays. Therefore, when Verilog-XL schedules path output events, it does not take into account the time at which the strength change occurs.

Annotation of Multiple Paths with the Same Delay

If you assign the same delay to multiple paths in a single assignment statement, you can not backannotate delay values on those paths individually. Therefore, we recommend that you use separate assignment statements for all paths whose delay values you plan to annotate, even though you may want to assign them the same value.

Here is a sample path delay assignment statement that assigns two paths the same delay:

```
(clr, pre*> q) = 10 ;
```

However, if you want to annotate the paths (clr*> q) and (pre*> q) with different delays, they must receive their initial delay assignments separately, as shown:

```
(clr*> q) = 10 ;
(pre*> q) = 10 ;
```

Using Module Input Port Delays (MIPDs)

MIPDs delay the propagation of input signals that drive accelerated primitives from input or inout ports.

MIPDs appear at the outputs of the loads connected to module ports. Therefore the \$monitor system task cannot display the signal delays caused by MIPDs until the signals propagate from the loads.

A MIPD that applies to signals driving a load input through a module port also applies to all signals driving that load input, whether the driving signals originate inside or outside the module containing the load.

Only one MIPD can apply to a module port. The following figure shows the effects of MIPDs on input and output signals:



Conditional Statements

Conditional Statements in Interactive Mode

Verilog-XL does not behave predictably in interactive mode unless each conditional statement conforms to one or both of the following rules:

- The conditional statement must be in a sequential (begin-end) procedural block or a parallel (fork-join) procedural block.
- The conditional statement must include an else statement.

Evaluation of Expressions in Conditional Statements

When Verilog-XL evaluates conditional expressions, it assumes that the left- and right-hand sides have the same number of bits. Therefore, you must specify the number of bits for all known values.

For example, in the following statement, Verilog-XL evaluates the left-hand side a^~b as an integer because the right-hand side 1 is an integer. This evaluation expands the left-hand side to 32 bits, insuring that the statement is false.

if ((a^~b)==1)

To make this statement function properly, you must make the right-hand side a single bit, as the following statement demonstrates:

if ((a^~b)==1'b1)

Using the 'timescale Compiler Directive

Place the `timescale compiler directive *either* in all of your source files or in none of them. When you use the `timescale compiler directive, place a `resetall compiler directive followed by the `timescale compiler directive at the top of each library file or library directory file.

Note: All Cadence-supplied libraries currently contain the `timescale compiler directive.

Changing a Parameter During Simulation

Changing the value of a parameter during simulation produces unpredictable results and is not recommended.

Performing Modulo Division on \$random Outputs

Unless you use the concatenation operator as discussed in this section, performing modulo division on the output of \$random task can result in a negative number because \$random outputs are signed.

For example, if b is greater than 0, the expression (\$random \$ b) gives a number in the following range: [(-b+1): (b-1)]. The following code fragment shows an example of such a random number generation:

```
reg [23:0] rand;
rand = $random % 60;
```

The preceding example gives rand a value between -59 and 59. The following example shows how adding the concatenation operator to the preceding example gives rand a positive value from 0 to 59:

```
reg [23:0] rand;
rand = {$random} % 60;
```
Performing Bit Swaps in Module Instance Vector Ports

The following example shows an improper method and a proper method of swapping bits in module instance vector ports:

```
/*
   This Does Not Work: */
module X(a, b, c);
   input [0:15] a;
   input b,c;
     b b1(a[15:0], c, b);
endmodule
module b (a,b,c);
   input [0:15] a;
   input b,c;
endmodule
/*
  Successful Method: */
module X(a, b, c);
   input [0:15] a;
   input b,c;
      b b1(a, c, b);
endmodule
module b (a,b,c);
   input [15:0] a;
   input b,c;
endmodule
```

Defining Vector Indices Across Module Boundaries

Bit selects of vector nets that are collapsed across module boundaries may cause undesirable results if the indices are ordered in opposite directions.

Consider this example:



If the nets shown above are collapsed onto a, each bit select of b uses the corresponding index of a. As a result, the reference b[0] actually produces the value of b[3]. To avoid such

unexpected simulation results, we recommend that you keep vector indices in the same order across module boundaries, as follows:



Note: Verilog-XL does not expand macro modules that contain specify blocks.

Syntax Recommendations

Do Not Use => for Full Connections on Paths

Currently, you do not receive a compiler error if you use the operator => to set up full path connections in these situations:

- between one scalar and one vector
- between one scalar and multiple path sources or destinations

However, **we do not recommend this practice** because using => to set up any kind of full connection may cause a compiler error in future releases of Verilog-XL.

Do not Use Keywords for 'rs_technology in Other Compiler Directives

The keywords for the `rs_technology compiler directive (shown in the following table) must not appear in any other compiler directives:

cdiff	default	highthresh	mapcap	resistance
cgo	deltal	ldiff	mapres	xa
COX	deltaw	lowthresh	name	

Avoid Radix Format Specifications for Character Strings

Make sure that the format specifications you use in the \$display, \$monitor, \$strobe and \$write system tasks are appropriate for their arguments. The format specifications of a radix—%b, %d, %h and %o—must not be entered for character string arguments. This is

because radix format specification for a character string argument in a \$display or \$write system task outputs a number instead of a character string.

F

Verilog-XL Turbo and Twin Turbo Options

This appendix describes the following:

- <u>Overview</u> on page 329
- <u>Turbo Option</u> on page 329
- <u>Twin Turbo Option</u> on page 330
- Invoking Turbo and Twin Turbo on page 330
- <u>Combining Non-Turbo with Turbo</u> on page 332
- <u>Twin Turbo Restrictions</u> on page 333
- <u>Achieving Optimal Performance</u> on page 334

Overview

The Turbo and Twin Turbo options provide performance enhancements for behavioral simulation.

Turbo Option

Versions of Verilog-XL prior to the 1.7 release are based solely on an interpreted simulation implementation. At compile and link time, the design and stimulus descriptions are parsed and used to build an internal representation, or data structure, of the design's functionality. During simulation, these operations representing the design's behavior are interpreted and processed one at a time. A significant amount of time is spent manipulating the internal data structure tree in order to fetch and store simulation values.

The Turbo option, introduced in Verilog-XL version 1.7, enhances performance by adding highly efficient function calls or pseudocode to the data structure and optimizing memory transfers. After compiling and linking, it rewrites the code that executes behavioral statements to maximize its efficiency.

Rewriting the code that executes behavioral statements requires some additional link time. However, the reduction in simulation time with the Turbo option more than compensates for this additional link time.

Twin Turbo Option

The Twin Turbo option, introduced in the Verilog-XL 2.1 release, provides additional performance enhancements over Turbo by augmenting Turbo's interpreted simulation method with a new compiled code implementation. In Twin Turbo mode, all behavioral source code is converted directly into machine code prior to simulation. During simulation, machine instruction sequences representing behavioral code execute at the system speed of the host computer. This compiled code executes considerably faster than the interpreted pseudocode used in Turbo mode, yet all of the flexibility and interactive debugging capabilities are maintained.

Because machine code instructions are generated in Twin Turbo mode to represent sequences of behavioral code statements, the link time of the simulation increases approximately 20% to 60%, with an additional increase in memory size of approximately 3% to 5%. The increase in simulation performance, however, easily makes up for the additional link time and memory requirements.

Twin Turbo is an extension of the Verilog-XL Turbo capability.

Twin Turbo option is not available on Windows NT or AIX platforms.

Invoking Turbo and Twin Turbo

The Turbo mode is available as default. Use the +twin_turbo option to invoke the Twin Turbo extension of default Turbo.

There are three command line options that you can use to invoke levels of performance superior to the Turbo default. The Twin Turbo functionality operates in conjunction with each of the three Turbo levels. Specifying the Twin Turbo option causes the simulation to generate and use compiled code for processing behavioral constructs in the selected Turbo mode. The following table summarizes the Turbo and Twin Turbo command line options.

Turbo	Twin Turbo	Effects
No plus option.	+twin_turbo	Invokes the default Turbo optimizations.
+turbo	+twin_turbo with +turbo	Improves performance by turning off the behavior profiler and the event count at the end of simulation.

Turbo	Twin Turbo	Effects
+turbo+2	+twin_turbo with +turbo+2	Increases performance over +turbo while guaranteeing event order matching Verilog-XL. Tracing gives different results.
+turbo+3	+twin_turbo with +turbo+3	Increases performance over +turbo+2, but may order events in the same time unit differently from Verilog-XL.

+turbo

The +turbo level improves on the performance without altering simulation results by disabling the end-of-simulation event count and the behavior profiler. Use +twin_turbo with +turbo to generate and use compiled code for processing behavioral constructs.

+turbo+2

The +turbo+2 level increases performance over +turbo level without altering simulation results by:

Converting scalar nets to compact nets if this conversion accelerates the simulation.

The PLI tf_nodeinfo routine cannot return strength for a compact net or values for the individual bits of a vector compact net. This limitation may require rewriting the PLI code with the acc_get_value routine.

Optimizing assignments.

Attempting to observe the assignments optimized by +turbo+2 with the \$settrace system task or the -t command-line option generates a message that indicates that an assignment is occurring, but that does not show the value assigned:

L9 "assignment_1": b = a + c; Not traceable due to +turbo+2

Use +twin_turbo with +turbo+2 to generate and use compiled code for processing behavioral constructs.

+turbo+3

The +turbo+3 option delivers better performance than +turbo+2 level by evaluating the right-hand sides of assignments only when the assignments actually occur.

The $\pm turbo+3$ option may change event ordering, and designs that are dependent on event ordering may produce different simulation results. The changes are most likely to occur in designs that demonstrate differences in event ordering dependent on the presence of default structural acceleration and $\pm cax1$. Any event ordering differences appear in the ordering of simultaneous events. Event-reordering typically does not occur in synchronous designs, in which nonblocking assignments have been used to allow all the inputs of the behavioral statements to be updated before the corresponding outputs are evaluated. If result mismatches are observed when comparing the output of two engines, a more severe problem may exist with the design. If event reordering presents a problem, simulate the design using the second-level Turbo optimizations ($\pm turbo+2$), which executes statements exactly like Verilog-XL, but at the cost of some performance.

Use +twin_turbo with +turbo+3 to generate and use compiled code for processing behavioral constructs.

+no_speedup

The $+no_speedup$ option disables all behavioral performance improvements, including the default performance improvements in non-Turbo Verilog-XL. This makes behavioral performance similar to that provided by Verilog-XL 1.6c, but guarantees identical results to an old simulation.

The following examples show how to create command line aliases that use the Turbo command line options to meet your differing needs:

```
alias vlog_faster /net/machine/exe/vlog +turbo+2
alias vlog_even_faster /net/machine/exe/vlog +turbo+3
alias vlog_fastest /net/machine/exe/vlog +twin_turbo +turbo+3
```

Combining Non-Turbo with Turbo

If a design simulates faster with the XL algorithm prior to using the Turbo option, as a general rule it runs faster with the Turbo option and the XL algorithm combined. The following table summarizes the non-Turbo accelerations that can be used with Turbo accelerations to improve performance.

Acceleration	Effects
default structural	Accelerates structural items with the XL algorithm by default in Verilog-XL.
+switchxl	Invokes the Switch-XL algorithm, which accelerates the simulation of bidirectional switches and provides two strength models.

Verilog-XL User Guide Verilog-XL Turbo and Twin Turbo Options

Acceleration	Effects
+caxl	Accelerates continuous assignments.

There is an overhead associated with switching between the XL algorithm and any non-XL algorithm, but sufficient XL algorithm activity compensates for the overhead. The Turbo option improves communication between the XL algorithm and non-XL algorithms, so it is beneficial to invoke the XL algorithm even when it accelerates only a small portion of the design.

The +noxl option makes simulation proceed without the XL algorithm, which could be preferable for three reasons:

- Completely behavioral designs can simulate slightly faster.
- The XL algorithm can yield different results, largely due to changes in event ordering.
- Memory requirements are smaller.

The Switch-XL algorithm applies only to structural constructs that the Turbo option does not accelerate. If the +switchxl option is beneficial outside the Turbo option, combining it with the Turbo option yields even better results. The Switch-XL algorithm provides unique strength models which can make it essential for some switch simulations.

Whether to use the +caxl option involves a more complex analysis. Both the Turbo option and the +caxl option accelerate continuous assignments. The +caxl option is more effective than Turbo in accelerating continuous assignments, but it can change the order of event evaluations, and some designs run more slowly with +caxl. Generally, if you would use the +caxl option without the Turbo option, it is preferable to use it with the Turbo option.

Twin Turbo Restrictions

There are some restrictions when using Twin Turbo. These restrictions do not apply to any Verilog hardware description language construct, but affect simulation checkpointing and single-step tracing.

The checkpointing system tasks \$save and \$restart and the -r command-line option are supported in Twin Turbo mode. However, to save simulation checkpoint files, you must include the +save_twin_turbo option on the command line. You do not need to use this option to restart simulation using a previously saved file. Using the +save_twin_turbo option increases memory usage by approximately 10%. See <u>"Saving and Restarting Simulations"</u> for more information about saving and restarting a simulation.

- In Twin Turbo mode, there is no way to perform incremental saves. The \$incsave system task behaves like \$save. Both system tasks save the complete data structure.
- The \$settrace system task, the -t command-line option, and the single step (,) interactive command display correct results in Twin Turbo mode. However, some events may be optimized away. If you want to make sure that you see complete results, use the +trace_twin_turbo command-line option. Using this option has some impact on performance.
- In <u>"+turbo+3" on page 331</u>, it was noted that using the third-level Turbo mode (+turbo+3) may create event ordering differences compared to the regular Verilog-XL behavioral engine. All Twin Turbo levels can result in event ordering differences compared to the corresponding Turbo mode results. For example, results may be different between using the second level Turbo mode (+turbo+2) and the corresponding Twin Turbo level (+turbo+2 with +twin_turbo). To ensure compatibility of results, use the +compat_twin_turbo command-line option. Using this option has some impact on performance.

Achieving Optimal Performance

Several optimizations can be implemented to improve the performance of the design. In general, the focus should be on keeping the number of scheduled events as low as possible. <u>Appendix C, "Maximizing Default Acceleration,"</u> provides details on many ways to improve performance. This section lists several tasks you can do to optimize a design for Turbo simulation.

Run the behavior profiler to identify the lines of code that consume large portions of simulation time.

In Turbo mode, the behavior profiler is disabled by default. To run the profiler in Turbo mode, invoke Verilog-XL with the +profile and +listcounts options. The +listcounts option enables the \$listcounts system task, which decompiles the design and displays the number of times each line of code was sampled while executing. This information is extremely useful for tracing potential inefficiencies down to individual lines of code. You can then try to rewrite these sections of code to increase simulation performance.

Using the *\$listcounts* task to generate code execution counts incurs some overhead, but you can leave this task in the source code and disable it by not including the *+listcounts* option on the command line.

Running the behavior profiler is also useful for quantifying simulation performance by displaying exactly how much simulation time is spent in various portions of the design. The sreportprofile task identifies the percentage of total CPU time spent

processing behavioral constructs and modules in the source description. You can then use these numbers to calculate the effect of the gate-level portion of the design compared to the behavioral portion.

See <u>"Chapter 19, The Behavior Profiler</u>" of the *Verilog-XL Reference* for details on the behavior profiler and for examples of profiler output.

- Minimize the use of file I/O and debug statements that write to a file or that display information, such as \$monitor, \$fmonitor, \$display, \$fdisplay, etc. Because of the relatively slow access to the computer's devices, calls to file I/O system tasks can significantly affect simulation performance.
- Reduce the simulator's workload by careful use of flow control statements. For example, if possible, combine several if statements into a single case statement, as shown in the following figure. The simulator will then evaluate the case statement once to understand the branching condition rather than evaluating several conditions repeatedly.

```
always(clk)
begin
                              always@(clk)
    if (status == 2'b00)
                              case(status)
        out = Y;
                                  2'b00: out = Y;
    if (status == 2'b01)
                                  2'b01: out = A;
        out = A;
                                  2'b10: out = 1'bz;
    if (status == 2'b10)
                                  2'b11: out = B;
       out = 1'bz;
                              endcase
    if (status == 2'b11)
        out = B;
end
```

Where possible, consolidate multiple procedural blocks that are triggered by the same conditions. For example, in synchronous designs many procedural blocks that are triggered only by the positive edge of the system clock can be consolidated. The following example shows how procedural blocks can be consolidated for more efficient processing.

```
always@(posedge clk)
Y = C + D;
always@(posedge clk)
Z = A * B;
always@(posedge clk)
X = E * F;
end
```

Use event controls efficiently. Inefficient use of event controls may cause the simulator to execute a particular statement or group of statements unnecessarily. In the following example, the code has been rewritten to specify more exactly when the assignment is to

be executed. Remember that if you use a synthesis tool, the HDL coding rules must not be violated

Limit the use of nonbehavioral constructs. Because nonbehavioral constructs are not accelerated by Turbo and Twin Turbo modes, they detract from the simulation's overall performance gain in these modes. Two common sets of constructs that detract from Turbo and Twin Turbo performance are instances of gate and UDP primitives and system tasks and functions, including PLI applications. If speed does not improve as expected, the simulation must be profiled using the behavior profiler to determine where the simulator is spending its time.

Code Examples

This appendix describes the following:

- <u>Overview</u> on page 337
- <u>Code Examples</u> on page 337
- <u>Sample Outputs</u> on page 348
- <u>Circuit Diagrams</u> on page 348
- Graphical Output on page 351

Overview

This appendix lists the code examples, sample outputs and circuit diagrams referred by other chapters of this user guide.

Code Examples

conditional_drive.v

```
// conditional_drive.v
module conddrive;
reg clk; // = 1'hx, x
reg clr; // = 1'hx, x
reg [3:0] data; // = 4'hx, x
wire [3:0] q; // = 4'hx, x (scalared)
hardreg h1(data, clk, clr, q);
initial
        begin
            clr = 1;
            clk = 0;
        end
always
            #50 clk = ~clk;
initial
            begin
            data = 4'b0;
```

Verilog-XL User Guide

Code Examples

#100 data = 4'b1; #100 data = 4'b10; \$finish; end endmodule // conddrive

counter.v

```
module m16(value, clock, fifteen, altFifteen);
      output [3:0] value;
      output fifteen, altFifteen;
                     clock;
      input
     dEdgeFF a(value[0], clock, ~value[0]),
            b(value[1], clock, value[1] ^ value[0]),
            c(value[2], clock, value[2] ^ &value[1:0]),
            d(value[3], clock, value[3] ^ &value[2:0]);
      assign fifteen = value[0] & value[1] & value[2] & value[3];
      assign altFifteen = &value;
endmodule // m16 counter
```

dff.v

```
primitive dff_udp(q, d, clk, rst);
   output q;
input d, clk, rst;
reg q;
    table
    // d
             clk
                         : q_old : q_new
                  rst
                             ? : 0;
        ?
             ?
                   0
                        :
        ?
             ?
                        :
                             0
                                 : 0;
                    x
        0
             (01)
                        :
                    1
                             ?
                                 : 0;
                   1
        1
             (01)
                        :
                             ?
                                 :
                                    1;
        ?
             (10)
                   1
                        :
                           :
                             ?
                                 :
                                    -;
                               ?
                        ?
                  ?
                                   :
        (??)
                                        -;
                   (??) : ?
                                : -;
        ?
              ?
    endtable
endprimitive
```

dff debug.v

```
module dff(q, qb, d, clk, rst);
     output q, qb;
     input clk, d, rst;
     nand n1 (cf, dl, cbf);
nand n2 (cbf, clk, cf, rst);
nand n3 (dl, d, dbl, rst);
nand n4 (dbl, dl, clk, cbf);
     nand n5 (q, cbf, qb);
     nand n6 (qb, dbl, q, rst);
endmodule // dff
```

dff_test.v

```
module DFF_test;
    reg clk, clr, d;
    wire q, qb;
    DFF dff1 (d, clk, clr, q, qb);
    initial
        begin
            clr=0; d = 0; clk = 0;
            $monitor("time = %0t, q = %b", $stime, q);
        end
    initial
        begin
            #80 d = 1;
            #100 \ clr = 1;
            #10 d = 0;
            \#100 d = 1;
            #100 $finish(2);
        end
    always #50 clk = ~clk;
endmodule // DFF_test
```

flipflop.v

```
// Model RS Flip Flop
module flipflop (clock, data, qa, qb);
    input clock,data;
    output qa, qb;
    nand #10 ndl (a, data, clock),
    nd2 (b, ndata, clock),
    nd3 (qa, a, qb),
    nd4 (qb, b, qa);
    mynot nt1 (ndata, data);
endmodule // flipflop
module mynot (out, in);
    output out;
    input in;
    not(out,in);
endmodule // mynot
```

flop.v

```
module flop (data, clock, clear, q, qb);
input data, clock, clear;
output q, qb;
wire data, clock, clear, q, qb;
nand #10 nd1 (a, data, clock, clear),
nd2 (b, ndata, clock),
nd4 (d, c, b, clear),
nd5 (e, c, nclock),
nd6 (f, d, nclock),
nd8 (qb, q, f, clear);
nand #9 nd3 (c, a, d),
nd7 (q, e, qb);
not #10 iv1 (ndata, data),
```

```
iv2 (nclock, clock);
endmodule // flop
```

flop_model.v

```
// Flop model
'delay_mode_distributed
module flop (data, clock, clear, q, qb);
    input data, clock, clear;
    output q, qb;
    nand #10 nd1 (a, data, clock, clear),
        nd2 (b, ndata, clock),
        nd4 (d, c, b, clear),
        nd5 (e, c, nclock),
        nd6 (f, d, nclock),
        nd6 (f, d, nclock),
        nd8 (qb, q, f, clear);
    nand #9 nd3 (c, a, d),
        nd7 (q, e, qb);
    not #10 iv1 (ndata, data),
        iv2 (nclock, clock);
endmodule // flop
```

flop_test.v

```
// Flop Text Fixture (flop_test.v)
`resetallv
module flop_test;
   reg clk, clr;
   reg data;
    wire q;
    flipflop f1 (data, clk, clr, q,);
initial
    begin
        clr = 1;
                    clk = 0;
        $monitor("time = %0t, data = %b, q = %b", $stime, data, q);
    end
always #50 clk = ~clk;
initial
    begin
        data = 0;
        #100 data = 1;
        #100 data = 0;
        #100 \ clr = 0;
        #100 data = 1;
        #100 data = 0;
        #100 $finish;
    end
endmodule // flop_test
```

full_adder.v

```
module full_adder (sum,c_out,a,b,c_in);
input a, b, c_in;
output sum, c_out;
```

```
half_adder m1 (si, ci1, a, b),
m2 (sum, ci2, si, c_in);
or m3 (c_out, ci2, ci1);
endmodule // full_adder
```

guarantee_order.v

```
module guarantee_order;
    reg clk, oldvalue;
    always @ clk
        #0 oldvalue = clk;
    always @ (posedge clk)
        if ((oldvalue == 0) & (clk == 1))
            $display ("true rising edge at %d", $time);
    initial
        begin
            #1 clk = `bz;
            #1 clk = 1;
            #1 \ clk = bx;
            #1 clk = 1;
            #1 clk = 0;
            #1 clk = 1;
        end
endmodule // guarantee_order
```

half_adder.v

```
module half_adder (sum, c_out, a, b);
    input a, b;
    output sum, c_out;
    xor #10 n1 (sum, a, b);
    and #10 n2 (c_out, a, b);
endmodule // half_adder
```

harddrive.v

```
module harddrive;
    reg clr, clk;
    reg [3:0] data;
    wire [3:0] q;
    `define stim #100 data = 4'b
    event end_first_pass;
   hardreg h1 (data, clk, clr, q);
initial
   begin
        clr = 1;
        clk = 0;
    end
always #50 clk = ~clk;
always @(end_first_pass)
    begin
        clr = ~clr;
        $stop;
    end
always @(posedge clk)
```

Verilog-XL User Guide Code Examples

\$strobe("at time %0d clr =%b data=%d q=%d",\$time, clr, data, q); initial begin // repeat repeat (2) begin // data assignment data = 4'b0000;`stim 0001; `stim 0010; `stim 0011; `stim 0100; stim 0101; `stim 0110; `stim 0111; `stim 1000; `stim 1001; `stim 1010; `stim 1011; `stim 1100; `stim 1101; `stim 1110; `stim 1111; #200 ->end first pass; end //data assignment \$finish; end // repeat endmodule //harddrive

hardreg.v

```
module hardreg (d, clk, clrb, q);
input clk, clrb;
input [3:0] d;
output [3:0] q;
flop f1 (d[0], clk, clrb, q[0], ),
f2 (d[1], clk, clrb, q[1], ),
f3 (d[2], clk, clrb, q[2], ),
f4 (d[3], clk, clrb, q[3], );
endmodule // hardreg
```

monitor.key

register.v

and a1(load, clk, ena); dff_udp d0 (r[0], data[0], load, rst), d1 (r[1], data[1], load, rst), d2 (r[2], data[2], load, rst), d3 (r[2], data[3], load, rst), d4 (r[4], data[4], load, rst), d5 (r[5], data[5], load, rst), d6 (r[6], data[6], load, rst), d7 (r[7], data[7], load, rts); endmodule // register

register_debug.v

```
* 8-bit register with heirarchy *
    module register(r, clk, data, ena, rst);
   output [7:0] r;
   input [7:0] data;
   input clk, ena, rst;
   wire [7:0] data, r;
   and al(load, clk, ena);
       d0 (r[0], data[0], load, rst),
   dff
       d1 (r[1], data[1], load, rst),
       d2 (r[2], data[2], load, rst),
       d3 (r[2], data[3], load, rst),
       d4 (r[4], data[4], load, rst),
       d5 (r[5], data[5], load, rst),
       d6 (r[6], data[6], load, rst),
d7 (r[7], data[7], load, rts);
endmodule // register
```

register_fixed.fm

```
* 8-bit register with heirarchy *
module register(r, clk, data, ena, rst);
    output [7:0] r;
    input [7:0] data;
    input clk, ena, rst;
         [7:0] data, r;
    wire
    and al(load, clk, ena);
    dff d0 (r[0], data[0], load, rst),
        d1 (r[1], data[1], load, rst),
        d2 (r[2], data[2], load, rst),
        d3 (r[3], data[3], load, rst),
        d4 (r[4], data[4], load, rst),
d5 (r[5], data[5], load, rst),
d6 (r[6], data[6], load, rst),
d7 (r[7], data[7], load, rst);
endmodule // register
```

register_test_debug.v

```
* Stimulus for testing the 8-bit Register
module test;
   wire [7:0] reg_out;
                          //declare vector for register output
   reg [7:0] data;
reg ena, rst, start, error;
   register r1(reg_out, clk, data, ena, rst);
   nand #10 (clk, clk, start); //clock oscilator
initial
                          //start the clock oscillator
   begin
       start = 0;
       #10 start = 1;
   end
initial
                           //apply stimulus to register inputs
   begin
       rst = 0;
                              //should reset register to hex 00
       ena = 1; data = 8'hff; //prepare to load the register
       @(posedge clk) ; //should NOT load data with reset asserted
       #2 rst = 1;
                              //de-assert reset
                              //should load data (hex FF)
       @(posedge clk) ;
       @(negedge clk) data = 8'h55;
           //sync to negedge clock to meet setup spec.
       @(posedge clk) ;
                                  //should load data (hex 55)
       #10 ena = 0; data = 8'hff; //de-assert enable
       @(posedge clk) ; //should NOT load data with ena de-asserted
       #20 $stop; $finish;
    end
initial
   begin
       error=0;
       #20 if(reg out !== 8'h00)
       begin
           $display("reg_out should be 8'b00000000 at time",$stime);
           $display("reg_out is 8'b%b", reg_out, "\n");
           error=1;
       end
       #20 if(reg_out !== 8'hff)
           begin
               $display("reg_out should be 8'b11111111 at time",$stime);
               $display("reg_out is 8'b%b", reg_out, "\n");
               error=1;
           end
       #20 if(reg_out !== 8'h55)
           begin
               $display("reg_out should be 8'b01010101 at time",$stime);
               $display("reg_out is 8'b%b", reg_out, "\n");
               error=1;
           end
           #1 if (!error)
                   $display("******* TEST PASSED ********");
               else
                   $display("******* TEST FAILED ********");
   end
endmodule // test
```

reregister_fixed.v

shortdrive.v

```
module shortdrive;
    reg [3:0] data;
    reg clk, clr;
    wire [3:0] q;
    hardreg h1 (data, clk, clr, q);
    initial
        begin
             repeat (2)
                 begin
                     data = 4'b0;
                      #100 data = 4'b1;
                      #100 \text{ data} = 4'b10;
                 end
             $finish;
        end
endmodule // shortdrive
```

step.v

```
module step;
initial
begin
$display ("First Statement");
$display ("Second Statement");
$display ("Third Statement");
$display ("Fourth Statement");
$display ("Last Statement");
end
endmodule // step
```

test.v

```
module test;
    reg [1:0] bus_a, bus_b;
    reg c_in;
    wire [1:0] sum;
    wire c_out;
    two_bit_adder under_test (sum, c_out, bus_a, bus_b, c_in);
    initial
        begin
            c_in = 1'b0;
            bus_a = 2'b00;
            bus_b = 2'b01;
            #2000 bus_a = 2'b01;
            #1000 c_in = 2'b01;
            #2000 bus_a = 2'b10;
            #1000 c_in = 2'b00;
        end
endmodule // test
```

test_flop.v

```
// Test fixture for flop
module test_flop;
    reg data, clock;
    flop f1 (clock, data, qa, qb);
    initial
        begin
            clock = 0; data = 0;
            #10000 $finish;
        end
    initial
        begin
            $shm_open("db1.shm");
            $shm_probe(clock);
            $shm_probe(data,qa,qb);
            $shm_probe(f1.nt1);
            $shm_close();
        end
    always #100 clock = ~clock;
    always #300 data = ~data;
endmodule // test_flop
```

tester.v

```
module tester;
reg [1:0] bus_a, bus_b;
reg c_in;
wire [1:0] sum;
wire c_out;
two_bit_adder under_test (sum, c_out, bus_a, bus_b, c_in);
always @(sum or c_out)
#100 // wait for output to stabilize
```

```
$strobe(" %b + %b (+ %b) = %b%b\n", bus_a, bus_b, c_in, c_out, sum);
initial
    begin
        $display("bus_a + bus_b (+ c_in) = result (c_out and sum)");
        c_in = 1'b0;
        bus_a = 2'b00;
        bus_b = 2'b01;
        #2000 bus_a = 2'b01;
        #1000 c_in = 2'b01;
        #1000 c_in = 2'b10;
        #1000 c_in = 2'b00;
    end
endmodule // tester
```

time_flop.v

```
module time_flop;
    reg clk, clr, data;
    wire q, qb;
    flop f1 (data, clk, clr, q, qb);
always #50 clk = ~clk;
initial
   begin
        clr = 1; clk = 0; data = 0;
        #100
                   data = 1;
                   data = 0;
        #49
        #102
                    data = 1;
        #49
                   clr = 0;
                   data = 0;
        #49
                    $finish;
        #100
    end
endmodule // time_flop
```

two_bit_adder.v

Sample Outputs

ex_signal_values

```
C1 > $display("clk, = ", clk,);
$display("clr, = ", clr,);
$display("data, = ", data,);
$display("q[3],q[2],q[1],q[0] = ", q[3],q[2],q[1],q[0]);
$display("clk, = ", clk,);
clk, = x
C2 > $display("clr, = ", clr,);
clr, = x
C3 > $display("data, = ", data,);
data, = x
C4 > $display("q[3],q[2],q[1],q[0] = ", q[3],q[2],q[1],q[0]);
C1 > $display("clk, = ", clk,);
$display("clr, = ", clr,);
$display("data, = ", data,);
$display("q[3],q[2],q[1],q[0] = ", q[3],q[2],q[1],q[0]);
$display("clk, = ", clk,);
clk, = x
C2 > $display("clr, = ", clr,);
clr, = x
C3 > $display("data, = ", data,);
data, = x
C4 > $display("q[3],q[2],q[1],q[0] = ", q[3],q[2],q[1],q[0]);
```

Circuit Diagrams





Verilog-XL User Guide Code Examples

half_adder circuit



two_bit_adder



Verilog-XL User Guide Code Examples

two_bit_adder under test



Graphical Output

two_bit_adder under test

	SimVision: Si	gnal Flow Browser 1	•
<u>F</u> ile	<u>E</u> dit <u>V</u> iew <u>E</u> ×plore	<u>F</u> ormat S <u>i</u> mulation <u>W</u> indows	<u>H</u> elp
۲	a 🗑 A 🐝	🔹 🚸 🛛 Send To: 💐 🖺 📰 🛞 🖡	•
Ъ Г	x2 TimeA 💌 = 0	💽 s 💽 🔂 t 🖓 🧆 💁 🤇 Se	arch Ti
Trac	ce: simulator::test.under_t	est.p0.m1 ▼ [= St ×	
	3		<u> </u>
	Driver	Value	
→	and n2(c_out, a, b)	St ×	
	Contributing Signal	Value	
→	test.under_test.p0.m1.a	St ×	
2	test.under_test.p0.m1.b	St x	
	Driver	Value	
→	INPUT a		
	Contributing Signal	Value	
 →	test.bus_a	×	
	Driver	Value	
1	bus_a = 'b00	00	
2	bus_a = 'b01	01	
3	bus_a = 'b10	[10	

Verilog-XL User Guide Code Examples

Η

Veriog-XL Messages

This appendix describes the following:

- <u>Overview</u> on page 353
- <u>Message Syntax</u> on page 353
- <u>Message Levels</u> on page 354
- <u>Compilation Error Messages</u> on page 356

Overview

Verilog-XL displays messages during compilation and execution. This section discusses the following about Verilog-XL messages:

- <u>"Message Syntax"</u> on page 353
- <u>"Message Levels"</u> on page 354
- "Compilation Error Messages" on page 356

Message Syntax

The syntax and format of a Verilog-XL message are as follows:

Verilog-XL User Guide

Veriog-XL Messages

The meaning for each message field is described in the following table

Verilog-XL message field	Field description
<level></level>	The class of information. There are five classes: message, warning, error, internal, and system. See <u>"Message Levels"</u> on page 354 for more information.
<message text=""></message>	The explanation for the Verilog-XL message.
<facility></facility>	Usually the name of the software tool that issues the message. It can also be a string argument that you define when creating your own messages.
<code></code>	A character code three to six characters long that is unique for each Verilog-XL message. You can use this code to get additional information from the technical support hotline if you need it. The code can also be a string argument that you define when creating your own messages.
<filename></filename>	The name of the file that contains the error. The filename is optional and is not included in every Verilog-XL message.
<line number=""></line>	The line location of the error. The line number is optional and is not included in every Verilog-XL message.
<source/>	The variable or operation that is causing the error. The source message field is optional and is not included in every Verilog-XL message.

Message Levels

There are five Verilog-XL message levels: Informational, Warning, Error, System, and Internal.

Informational message

Informational messages provide information about your source description. For example, you might want to print a message when your model reaches a certain point in the simulation. Informational messages do not contain a code field.

Verilog-XL User Guide

Veriog-XL Messages

In the following example, two conflicting source protection command-line options, +protect and +autoprotect, are specified. The software resolves this conflict by continuing the simulation in +autoprotect mode and informing you of this action.

Message! Both '+protect' and '+autoprotect' specified; continuing in '+autoprotect' mode [Verilog]

Warning message

Warning messages indicate that there may be something wrong with your model. Your model will continue to run, but the results of the simulation may not be reliable.

In the following warning message example, the message text informs you that more than one delay mode has been selected. The software will select the delay mode with the highest priority and continue simulating.

```
Warning! More than one delay mode selected, highest priority one used [Verilog-MDMS]
```

Error message

Error messages indicate that a user error has occurred at compile time or at run time. In this example, the Verilog-XL source description file test.v contains an error. The line number indicates that the error is on line 4. The tool that generates this error message is Verilog-XL and the code is DISIA. The source field indicates that there is a missing argument in system task \$display for the string expression num=%b.

Syntax error messages do not have a code field. Only the facility is enclosed in square brackets. The message text field indicates that the error is a syntax error.

In the following example, the message states that there is a syntax error in the Verilog-XL source description file t1.v. The line number indicates that the error is on line 2. The source shows that the parameter listing for not gate n1 is missing a right parenthesis.

Error! syntax error [Verilog] "t1.v", 2: not n1 (out,in;<-</pre>

System message

System messages report errors that return from the operating system. For example, an operating system error can occur if you try to access a file that does not exist or run a program when the system is out of memory. In the following example, the operating system cannot open the Verilog-XL file example.v because the file does not exist:

System! Source file "example.v" cannot be opened for reading [Verilog-SFCOR]

Internal message

Internal messages indicate a software bug in the application. The simulation stops, and you are instructed to call customer support for assistance. Be sure to provide the technical support hotline with the code for prompt assistance.

In the following example, the Verilog routine pe_gen_index came across a value it did not expect:

Internal! Routine: pe_gen_index - a character other than 0,1 or x found [Verilog-COVF] Please contact Cadence Design Systems about this problem and provide the above information.

Compilation Error Messages

The compilation process consists of more than one pass through your source description. If Verilog-XL finds errors in one of its passes through your source description, it displays compilation error messages and stops compilation.

Note: Because Verilog-XL does not proceed to its next pass through your source description after it finds an error, correcting errors in your source description can lead to the display of a new set of errors.

Syntax error messages

During Verilog-XL's first pass through your source description, syntax errors are reported by giving the following items:

- the line number
- the line of source text up to the offending token
- the two characters <- indicating the offending token

If you are unable to discern the reason for a syntax error, then you should examine the syntax definition for the construct in question. Note that the syntax error may print out a perfectly correct statement construct. In this case, you should scan back through the text to the previous line of text for the error. For example, the following two lines of source text:

pc = {regb, regc} // load program counter @clock busa = data;// wait for clock before loading bus

produce the following syntax error message:

```
"example1.v", 284: syntax error: @<-
```

where 284 is the line number beginning with <code>@clock</code>, and <code>example.v</code> is the source text filename. Searching back a token from the token specified by the error message takes you back to the end of the previous line where the source of the problem is a missing assignment semi-colon terminator.

Another example of a syntax error message that makes it hard to spot the source of the problem is the error message that results from the following text:

```
wires
    w1, w2, w3;
nand #1
    g1 (w1, w2, w3);
```

This text results in the following error message:

"example2.v", 3: syntax error: w1,<-</pre>

This syntax error has pointed to what seems to be a perfectly correct comma separating a pair of nets. The problem is that the compiler thinks that wires is a module name (not the keyword wire), and module instances require an opening parenthesis after the instance name.

Common compilation error messages

The following is a summary of error messages that can come from the Verilog-XL compiler. The list is not meant to be complete. Other messages that may appear are either self-explanatory or similar to an error message given here.

The following messages indicate that a name has been used for identifying something that has previously been used to identify something elseL:

"element instance name (<instance_name>) previously declared"
"module name (<module_name>) previously declared"
"task or function name (<tf_name>) previously declared"
"gate name (<gate_name>) previously declared"
"(<variable_name>) previously declared"
"block name (<block_name>) previously declared "

The following messages indicate that an identifier has been used that has not been declared or defined. This situation usually arises because of name misspelling.

```
"identifier (<variable_name>) not declared"
"task or function (<tf_name>) not defined"
"identifier (<name>) not defined"
"port (<port_name>) not defined as input, output or inout"
```

The following messages occur because un-lengthed numbers default to the size of integers (usually 32-bits); they may not be specified in concatenations. For example, $\{a, 'b010, b\}$ is illegal; you must use $\{a, 3'b010, b\}$ instead.

"concatenations may not have un-lengthed based numbers" "concatenations may not have un-lengthed numbers"

The following messages indicate that a gate output or a bidirectional terminal has been specified incorrectly. Such a terminal can only be connected to a scalar net of the appropriate type.

"gate (<gate_name>) has illegal output specification"
"gate (<gate_name>) has illegal inout specification"

The following messages indicate that an input or an inout has been declared as an incompatible type. For example, the first message is issued if you declared an input to be of type reg.

```
"incompatible declaration, (<variable_name>) defined as input at line <number>" "incompatible declaration, (<variable_name>) defined as inout at line <number>"
```

```
"component name (<name>) not a module, task, function or block".
"component name (<name>) not declared"
```

The following message indicates that only memory elements can be referenced in an expression with respect to memories:

"illegal reference to memory (<memory_name>)"

The following messages are issued when an identifier has been used incorrectly:

```
"(<variable_name>) is illegal, constant expected"
"illegal reference in constant expression"
"illegal reference to event (<event_name>)"
"illegal use of identifier (<name>)"
"identifier (<name>) not a task or function"
```

The following message indicates that strings must be enclosed in double-quote characters and contained on one line:

"literal string not terminated"

The following messages indicate that the /* */ type of comment may have been used without the */ ending. In this situation, all the text from the incorrect comment to the next comment of the same type is treated as a comment. The compiler therefore issues the following error message when it detects /* within a comment or when it encounters the end of the source text before the end of comment:

```
"/* found in comment"
"end of file while in comment"
```

Verilog-XL User Guide Veriog-XL Messages

This message is issued if modules have not been correctly completed using the endmodule keyword, or if the end of the source text has been reached without finishing the correct syntax constructs:

"Premature end of file"

The following message means that a module instance has been given more port connections than are specified in the module description:

```
"too many port connections"
```

The following message indicates that a module instance has been given fewer port connections than are specified in the module description. This warning can be suppressed with the -w command-line option.

"too few module port connections"

The following message indicates that the signal connected to a module port has a different number of bits than the port to which it is connected. This warning can be suppressed with the -w command-line option.

"port sizes differ in port connection (port <number>)"

The following messages indicate that output and inout ports to modules cannot be expressions:

```
"illegal output port specification (port <number>)"
"illegal inout port specification (port <number>)"
```

The following message appears when modules have been embedded within one another, forming an infinite loop of modules within modules:

"recursive instantiation of module (<module_name>)"

The following message is issued when a module instantiation statement references an object that is not a module:

"identifier (<name>) not a module"

The following messages indicate that certain statements, such as time control statements and enabling task statements, are not allowed inside a function body definition:

```
"statement in function is illegal"
"task enabling in a function is illegal"
"output or inout declaration not allowed in function"
```

The following warning indicates that the indicated element in a hierarchical name is at a higher level in the hierarchy than the statement which caused the warning. The reference will work as a consequence of the scope rules.

"component name (<name>) not on a downward path".

The following message indicates that the left-hand-side of a continuous assign statement is not a net type or a construct that Verilog-XL can handle:

"illegal left-hand-side continuous assignment"

The following messages are issued due to illegal data types being used in certain constructs:

"illegal left-hand side assignment" "illegal left-hand side data type in assign statement" "illegal data type in deassign statement" "illegal left-hand side initial assignment in for statement" "illegal left-hand side step assignment in for statement"

The following messages are issued when either task or function parameters do not match in number, or an illegal left-hand side (lhs) expression is used for output or inout task parameters:

"incompatible number of task or function parameters" "illegal output or inout task parameter (position <number>)"

The following message indicates that the identifier in a disable statement may only refer to a task or block:

"identifier (<name>) not a task or block"

The following message indicates that the character given is not valid for the decimal or based number being specified:

"illegal digit *<character>* in decimal number" "illegal digit *<character>* in based number"
Index

Symbols

\$bidirectional network reporting by \$showvars 147 \$db_breakaftertime 265 syntax 265 \$db breakatline <u>264, 265</u> syntax 264 \$db_breakbeforetime 264 syntax <u>264</u> \$db_breakonceatline syntax <u>264</u> \$db_breakonceonnegedge syntax 267 \$db_breakonceonposedge 266 \$db_breakoncewhen syntax 265 \$db_breakonnegedge 266 syntax 266 \$db_breakonposedge syntax 266 \$db_breakwhen 265 syntax 265 \$db_cleartrace 260 syntax <u>260</u> \$db_deletebreak <u>267</u> syntax <u>267</u> \$db_deletefocus 255 syntax 255 \$db_disablebreak 268 syntax <u>268</u> \$db_disablefocus 256 syntax 256 \$db_enablebreak 268 syntax 268 \$db_enablefocus 255 syntax <u>255</u> \$db_help <u>251</u> syntax <u>251</u> \$db_setfocus 254 syntax 254 \$db_settrace 260 syntax <u>260</u> \$db_showbreak 269 syntax 269 \$db_showfocus 256

syntax 256 \$db_step 259 syntax 259 \$db_steptime 259 syntax 259 \$display accessing protected data <u>171</u> \$dist chi square syntax 313 \$dist_erlang syntax 313 \$dist exponential syntax <u>313</u> \$dist normal syntax 313 \$dist_poisson svntax 313 \$dist_t syntax 313 \$dist uniform syntax 313 \$dumpvars accessing protected data 171 \$gr waves accessing protected data <u>171</u> \$incsave and Twin Turbo 334 efficiency of 306 \$list and source protection 170 library definition renaming <u>102</u> \$listcounts and performance 308 \$monitor accessing protected data 171 \$omiCommand <u>197</u> \$q add 310 syntax 310, 311 \$q_exam <u>311</u> syntax 311 \$q full 311 syntax 311 \$q_initialize <u>310</u> \$q_remove <u>311</u> syntax 311 \$random, modulo division on output 324

Srecordabort 273 \$recordclose 273 \$recordfile 273 \$recordoff 273 \$recordon 273 \$recordsetup <u>273</u> \$recordvars <u>273</u> \$reportfile 298 \$reportprofile and performance 308 \$reset and performance 307 and PLI calls 318 \$restart and performance 307 and queueing tasks 312 and Twin Turbo 333 \$save and interactive recovery 250 and performance 307 and queueing tasks 312 and Twin Turbo 333 files from incompatible hosts <u>316</u> \$scope accessing protected data 171 \$settrace and acceleration 290 and finding unaccelerated primitives 295 and source protection 170 in Twin Turbo 334 to trace from start of simulation 203 \$shm_close 270 \$shm_open 270 \$shm_probe 270 \$shm_resume 270, 273 \$shm_suspend 270, 273 \$showallinstances and resolving modules 112 library definition renaming 102 using with the Behavioral Profiler 298 \$showexpandednets and source protection 170 \$shownonxl 289 \$showportsnotcollapsed and source protection 170 \$showvars and source protection 170 reporting on bidirectional networks <u>147</u> \$sreadmemb and protected data 177 to 179

\$sreadmemh and protected data 177 to 179 \$startprofile and performance 308 how to use 297 \$stop description of use 248 \$stopprofile and performance 308 \$strobe accessing protected data 171 \$test\$plusargs 224 to ??, 224, ?? to 225 \$write accessing protected data 171 199 -a +accnoerr 205, 214 +accu_path_delay 205 +alt_path_delays 206 +annotate_any_time 206 +autoprotect 173, 206 how to use 168 -c 200 and library files 102 use with +protect or +autoprotect 176 use with -r 200 +caxl 206 use with Turbo 333 +compat_twin_turbo 206, 334 -d 200 and source protection 170 +define+ and 'define 207 and empty macros 238 and library search paths 103 and macro strings 207 +delay_mode_distributed 208 +delay mode path 208 +delay_mode_unit 208 +delay_mode_zero 208 +err_line_length+ 209 -f 200 to 201 +gui 209 -i 201 and interactive recovery 250 +incdir+ 210, 242 error checking for 210 202 -k -1 202 +libext+ 210 how to use 107 +libnonamehide

how to use 119 use with +librescan 119 +liborder 211 how to use <u>110</u> to <u>112</u>, <u>119</u> use with +librescan 112 +librescan 211 how to use 112 +libverbose 211 and resolving modules <u>112</u> +licq_all <u>211</u> +licq_lmchwif 212 +licq_vxl 212 +listcounts 211, 212 and performance 308 use with the behavioral profiler 298 +loadpli 212 +loadvpi 213 +max_err_count 213 +maxdelays 213 +mindelavs 213 +multisource_int_delays 214 +neg_tchk 214 +no_cancelled_e_msg 214 +no_charge_decay 214 +no_cond_event_error 215 +no pulse int backanno 215 +no_pulse_msg 215 +no_show_cancelled_e 215 +no_speedup 216, 332 and performance 307 +no_tchk_msg 216 +nolibcell 214 +nowarn 216 +noxl 216 use with Turbo 333 +pathpulse 216 +pre 16a paths <u>217</u> +profile $\frac{217}{217}$ +protect 217and library searches 173 how to use 167 use with \$readmemb and \$readmemh <u>177</u> to <u>179</u> +pulse_e/n 217 +pulse_e_style_ondetect 218 +pulse_e_style_onevent 218 +pulse_int_e/n 218 +pulse int r/m 218 -q 202 202 -r and interactive recovery 250

use with -c 200 +rsw_opt_stack 218 -s 203 +save_twin_turbo 218, 333 +sdf_cputime 218 +sdf_error_info 218 +sdf_file 219 +sdf_ign_timing_edge 219 +sdf_no_warnings 220 +sdf_nocheck_celltype 219 +sdf_nomsrc_int 220 +sdf verbose 221 +show_cancelled_e 221 +splitsuh 221 +switchxl 221 how to use 143 use with Turbo 332 +sxl_keep_all 221 affect on optimization 155 example of use <u>156</u> +sxl_keep_declared <u>222</u> affect on optimization 155 example of use 156 +sxl_keep_minimum 222 affect on optimization 155 +sxl unidirect 222 -t 203 +trace_twin_turbo 222, 334 +transport_int_delays 222 +transport_path_delays 223 +turbo 223, 330, 331 +turbo+2 223, 331 +turbo+3 223, 331 +twin_turbo <u>223</u>, <u>330</u> +typdelays 224 -u 203 -v 203, 204 syntax 203, 204, 212 204 -W -x 204 +x_transport_pessimism 224 205 -y continue command 249 interactive command 249 step command 249 <and compiler error messages <u>356</u>

in compiler directives 226 'accelerate 226 controlling application of the XL algorithm 286 'autoexpand_vectornets 226 celldefine 226 'default_decay_time 227 'default_nettype 228 collapsing internal nets 320 'default_rswitch_strength 228 how to use 159 'default_switch_strength <u>228</u> how to use 159 'default_trireg_strength 229 how to use 159 'define 229 and library search paths 103 'delay_mode_distributed 229 'delay_mode_path 229 'delay_mode_unit 229 'delay_mode_zero 229 'else 230 how to use <u>235</u> to <u>241</u> 'end_pre_16a_paths 230 'endcelldefine 226 'endif 230 how to use 235 to 241 endprotect 230 example of use 171 how to use 166 to 167 use with +autoprotect 169 'endprotected example of use 172 inserted by +autoprotect 169 to 170 inserted by +protect 167 'expand_vectornets 230 'ifdef 230 how to use 235 to 241 'include 230 how to use 241 to 246 'noaccelerate 226 controlling application of the XL algorithm 286 'noexpand_vectornets <u>23</u>0 'noremove_gatenames 231 'noremove_netnames 232 'nounconnected_drive 233 'pre_16a_paths 230 protect 230 example of use 171 how to use 166 to 167

use with +autoprotect 169 protected 231 affect on library scanning 121 example of use 172 inserted by +autoprotect 168 to 170 inserted by +protect 167 'remove_gatenames 231 'remove_netnames 232 'resetall and library files 121 'resetall and how to use 232 'rs_technology 232 keywords in other compiler directives 318 'switch 143 'switch default 232 'switch XL 232 timescale 232 placement <u>326</u> 'unconnected_drive 233 'undef 233 'unprotected 231 affect on library scanning 121 'uselib 234 defining macros for use with 103 how to use 102 to 106

trace-step command 249

Α

acc_user.h 130 acceleration 285 to 308 accelerated primitives and scalar nets 287 and key files containing asynchronous interrupts 291 and tracing 291 behavioral performance improvements 307 to 308 non-accelerated items 288 of events 291 of primitives for performance 295 potential problems 291 to 292 processing simultaneous events 290 when pulse width equals gate delay 291 XL algorithm and non-XL simulation 290 to 291 access routines 130

activating commands <u>46</u> active commands showing <u>48</u> alias performance <u>301</u>

В

begin-end block statement acceleration 307 Behavior Profiler 182 behavioral modeling +no_speedup 307 acceleration 307 behavioral performance improvements 307 to 308 bidirectional network force and release 144 wired logic 145 bit swaps in vector ports 325 bit-select performance 298 breakpoints 261 continuous versus non-continuous 263 setting code line 42 source line-based 261 time-based 261 transition-based 261 value-based 261

С

cache thrashing and performance 304 Cadence Model Manager for Quickturn options 196, 197 CAI configuration See configuration 125 CAI simulation 126 canceling commands 46 capturing simulation data 306 Cell definition of 122 cell 122 channel delay timing model 149 charge decay 214 ignoring specifications for 214 specifying a default decay time 227 circular library scan order in the old library scheme 110

clearing trace 260 clock generators and performance 297 closing simulation data files - 30 code measuring 292 optimizing 292 optimizing for behavioral performance improvements 307 reducing executed code 305 command file option 200 command line options 199 to 205 199 -a 200 -C use with +protect and +autoprotect 176 -d 200 200 -f 201 -i -k 202 -1 202 202 -q 202 -r 203 -S specifying on command line 23 to 24 -t 203 203 -u <u>203, 204</u> -V 204 -W 204 -X -y <u>2</u>05 plus options 205 to 224 +accnoerr 205 +accu path delay 205 +alt_path_delays 206 +annotate_any_time +autoprotect 206 206 how to use 168 +caxl 206 use with Turbo 333 +compat_twin_turbo 206, 334 +define+ and 'define 207, 238 and empty macros 238 and library search paths 103 and macro strings 207 +delay_mode_distributed 208 +delay_mode_path 208

+delay_mode_unit 208 +delay_mode_zero 208 +err_line_length+ 209 +incdir+ 210, 242 error checking for 210 +libext+ 210 how to use <u>107</u> +libnonamehide 210 how to use 119 use with +librescan 119 +liborder 211 example of use <u>119</u> to <u>120</u> how to use 110 to 112 use with librescan 112 +librescan 211 how to use 112 +libverbose 211 +licq_all 211 +licq_Imchwif 212 +licq_vxl 212 +listcounts 211, 212 use with the behavioral profiler 298 +loadpli 212 +loadvpi 213 +max_err_count 213 +maxdelays <u>213</u> +mindelays <u>213</u> +multisource_int_delays 214 +neg_tchk 214 +no_charge_decay 214 +no_cond_event_error 215 +no_notifier 215 +no_pulse_int_backanno 215 +no_pulse_msg <u>215</u> +no_speedup <u>216, 332</u> effect on performance 307 +nolibcell 214 +nowarn 216 +noxl use with Turbo 333 +pathpulse 216 +pre_16a_paths <u>217</u> +protect <u>217</u> and library searches <u>173</u> how to use 167 use with \$readmemb and \$readmemh 178 +pulse_int_e/n 218 +pulse_int_r/m 218 +pulse_r/m 217

+rsw_opt_stack 218 +save_twin_turbo 218, 333 +sdf_cputime 218 +sdf_error_info 218, 221 +sdf file 219 +sdf_no_errors 220 +sdf_no_warnings 220 +sdf_nocheck_celltype 219 +sdf_nomsrc_int 220 +sdf_verbose 221 +splitsuh 221 +switchxl 221 how to use 143 +sxl_keep_all 221 affect on optimization 155 example of use <u>156</u> +sxl_keep_declared <u>222</u> affect on optimization 155 example of use 156 +sxl_keep_minimum 222 affect on optimization 155 +sxl_unidirect 222 +trace_twin_turbo 222, 334 +transport_int_delays 222 +transport_path_delays 223 +turbo 223, 330, 331 +turbo+2 223, 331 +turbo+3 223, 331 +twin_turbo 223, 330 +typdelays 224 +vcw 209 +x_transport_pessimism 224 command line arguments testing for plus arguments 87 command line options storing in a file 94 commands plus options for Shell Generator <u>194</u> specifying from a file 92 compilation ?? to 246 command line 23 compile only option syntax checking in library files 102 compile only option -c option 200 compiling source files 24 to 25 directives 226 error messages 356 to 360 performance 307 compiling source code continually 85 concatenation of library extensions to module and UDP

names 107 performance 299 conditional compilation 238 conditional compilation symbol 238 conditional statement evaluation of expressions 323 in interactive mode 323 configuration <u>122</u> continue 249 control-C 248 control-d 83 controling how interactive time values are interpreted 72 cosimulation Verilog-XL and Quickturn, overview of 187

D

Databases using \$recordvars 273 deactivated commands showing 48 deactivating commands 46 debug system tasks 251 \$db breakaftertime 265 \$db_breakatline 264, 265 \$db_breakbeforetime 264 \$db_breakonceonposedge 266 \$db_breakonnegedge 266 \$db_breakwhen 265 \$db cleartrace 260 \$db deletebreak 267 \$db_deletefocus 255 \$db_disablebreak 268 \$db disablefocus 256 \$db_enablebreak 268 \$db enablefocus 255 \$db_help 251 \$db_setfocus 254 \$db settrace 260 \$db_showbreak 269 \$db_showfocus 256 \$db_step 259 \$db_steptime 259 debugging 247 to 251 disabled by protection 248 initial wide scope 57 style and performance 305 tools 137

decompile option 200 decompiling source code 48 default acceleration maximization 285 to 308 behavioral 307 to 308 XL algorithm 287 to 297 default library scan precedence 108 to 110 delay trireg charge decay 214 ignoring 214 specifying a default 227 delavs specifing the delay type (min, typ, max) <u>95</u> specifying the delay mode 96 deleting breakpoint 267 foci 255 **Design libraries** example structure 122 design library 121 directory library <u>100</u> able <u>249, 250</u> disable disabling breakpoints 268 commands 46 foci 256 disabling warnings with +nowarn 216 drivers showing 62 driving your design

Ε

emulator database generating <u>189</u> enabling breakpoints <u>268</u> foci <u>255</u> enabling commands <u>46</u> ending simulations <u>83</u> EOF character (control-d) <u>83</u> error handling <u>137</u> error messages <u>355</u> compiler <u>356</u> to <u>360</u> effect of source protection <u>176</u> syntax <u>356</u>

creating test fixtures 27

establishing a metric 293 estimating model speed 292 event accelerated 291 event control performance 300 event order observing in a time slot 67 event-triggered breakpoints 36 examining code 48 signals 62 simulation effects 48 examples accelerating specific modules 286 command argument file 201 command line 23 +liborder <u>110</u> +librescan 112 library directory option 106 library file option 106 library options <u>107, 110, 112</u> multiple library directory file extensions 107 null library file extensions 107 options 24, 205 compilation with +protect 173 error message syntax 356 interactive command prompt 248 level-sensitive latch 290 protected source description output 173 specifying a cell 227 testing plus options 225 zero-delay oscillation 290 execution ?? to 234 expanding source code 48 expressions evaluation in conditional statements 323 extensions for files in library directories 107 to 108

F

fanin showing <u>62</u> file inclusion <u>241</u> to <u>246</u> files command line argument files <u>94</u>

extensions in library 107 to 108 directories extensions of protected source files 167 including in other files 88 including source-code files 88 input files <u>92</u> key files 91 library <u>100</u> log files <u>89</u> writing simulation data to 30 Finish command 83 finishing simulations 83 floating license 23 foci 253 to 257 <u>58</u> focusing a trace forcing an output $\underline{64}$ formatting time display <u>72</u>

G

gotolinkfitwin \$vlogref/chap13.fm firstpage <u>177</u> graphical environment invoking <u>209</u>

Η

hardware performance optimization <u>303</u> help debugging <u>251</u> hierarchy and source protection <u>173</u> to <u>176</u> in libraries <u>106</u> traversing <u>52</u>

implicit declarations net type created for <u>228</u> included files in other files <u>88</u> input file option <u>201</u> and interactive recovery <u>250</u> input files <u>92</u> inserting a file into another file <u>88</u> interactive

conditional statement <u>323</u> control and debugging 247 to 251 mode and source protection 171 prompt 248 recovery 250 to 251 interactive commands continue 249 disable <u>249, 250</u> list of 249 re-execute 249, 250 step 249 syntax 249 trace-step 249 and source protection 171 251 interactive debugging environment interactive sessions reproducing 91 interconnect delays and +multisource_int_delays 214 intermediary files 24 interrupt accelerated simulation 291 with \$stop 248 with control-C <u>248</u> invoking Verilog-XL ?? to 24

K

key file and interactive recovery <u>250</u> to <u>251</u> changing the default name with -k <u>202</u> potential problem using <u>291</u> key files generating <u>91</u>

L

level-sensitive performance optimization of models <u>302</u> Libraries Library.Cell View structure example <u>122</u> libraries <u>99</u> to <u>101</u> and 'resetall <u>232</u> circular scan order in the old library scheme <u>110</u> controlling scan precedence <u>108</u>

creating unique identifiers for multiple modules and UDPs with the same name 111 definition renaming 102 directories 106 to 108 directory file extensions 107 to 108 files 100 forced scan precedence <u>110 to 112</u> how extensions interact with scan order <u>109</u> to <u>111</u> new scheme 102 to 106 'uselib definition of search paths 102 search order with 'uselib paths 105 unresolvable instantiations 105 null extensions 107 renaming definitions 102 reporting resolution paths 101 syntax checking in files 101 use of compiler directives with <u>120</u> library design 121 reference 121 work 121 library directory option description and syntax 205 library file option 203, 204 Library.Cell View 121 license obtaining during invocation 23 queuing and +licq_all 211 and +licq_lmchwif 212 and +licq_vxl 212 linking to Verilog-XL 326 listing activated and deactivated commands 48 log file option 202 changing the default name with -I 202 log files generating 89 specifying the name of 89

Μ

macro module does not support specify blocks <u>326</u> terminals not expressions <u>319</u> macros

and +define+ 207 and 'define 229 defining for 'uselib 103 passing values from the command line 84 showing expanded 50 maximizing default acceleration 285 to 308 behavioral <u>307</u> to <u>308</u> XL algorithm 287 to 297 mc_scan_plusargs 224 to ??, 225, ?? to 225 MCDs (multi-channel descriptors) 30 measuring code 292 memory limitations and performance 303 memory usage 181 messages <u>353</u>, <u>355</u>, ?? to <u>360</u> error 355 informational 354 levels 354 to 356 MIPDs effects 322 model hierarchy traversing 52 model manager for Verilog-XL and Quickturn cosimulation 187 modeling simplification for performance 305 modeling level and performance 293 module library definition renaming 102 top-level avoiding unnecessary creation of 99 module path pulse control and PLI interface routines 323 monitoring signals 60 wire values periodically 59 multiple library directory file extensions 107 multisource interconnect delays and +multisource_int_delays 214

Ν

names making multiple definitions of like-named modules and UDPs unique <u>111</u> netlist for Verilog-XL and Quickturn cosimulation, creation of <u>189</u> network resetting <u>73</u> null extensions <u>107</u>

0

opening files for writing simulation data <u>30</u> optimizing code <u>292</u> for Twin Turbo <u>334</u> options command line <u>199</u> to <u>205</u> invoking on command line <u>23</u> to <u>24</u> specifying on command line <u>23</u> to <u>24</u> overhead <u>294</u> in simulation <u>286</u> overriding macro definitions <u>84</u>

Ρ

parameter changing in simulation 324 patching a model 64 resetting prior to 73 path delay accuracy and +accu_path_delay 205 path simulation special considerations 321 performance accelerated primitives 295 accelerating behavioral code 307 aliases 301 behavioral performance improvements 307 to 308 cache thrashing 304 capturing simulation data 306 clock generators 297 coding tricks 298 compilation 307 debugging style 305 establishing a metric 293 estimating model speed 292 event controls 300

hardware 303 keeping primitives accelerated 295 level-sensitive behavior modeling 302 measuring and optimizing code 292 to 308 memory limitations 303 model simplification 305 modeling level 293 overhead due to switching algorithms 294 overview 285 reducing executed code <u>305</u> UDPs 300 performance, simulation 182 PLI interface routines and pulse control 323 and +no_pulse_int_backanno 215 back annotation of multiple paths 322 mc_scan_plusargs 224 to 225 with \$reset 318 plus arguments testing for presence of 87 plus options <u>205</u> to <u>224</u> +accnoerr 205, 214 +accu_path_delay 205 +alt_path_delays 206 +annotate_any_time 206 +autoprotect 206and library searches <u>173</u> how to use 168 +caxl 206 use with Turbo 333 +compat_twin_turbo 206, 334 +define+ and 'define 207, 238 and empty macros 238 and library search paths 103 and macro strings 207 +delay_mode_distributed 208 +delay_mode_path 208 +delay_mode_unit 208 +delay_mode_zero 208 +err_line_length+ 209 +incdir+ 210, 242 error checking for <u>210</u> +libext+ 210 how to use 107 +libnonamehide how to use 119 use with +librescan <u>119</u> +liborder 211

example of use <u>119</u> to <u>120</u> how to use 110 to 112 use with +librescan 112 +librescan 211 how to use 112 +libverbose 211 and resolving modules 112 +licq_all 211 +licq_Imchwif 212 +licq_vxl <u>212</u> +listcounts 211, 212 and performance <u>308</u> use with the behavioral profiler 298 +loadvpi 212, 213 +max_err_count 213 +maxdelays 213 +mindelays 213 +multisource_int_delays 214 +neg_tcheck 214 +no_charge_decay 214 +no_cond_event_error 215 +no_notifier 215 +no_pulse_int_backanno 215 +no_pulse_msg <u>215</u> +no_speedup <u>216, 332</u> effect on performance 307 +no_tchk_msg 216 +nolibcell <u>214</u> +nowarn <u>216</u> +noxl 216 use with Turbo 333 +pathpulse 216 +pre_16a_paths 217 +protect 217 and library searches 173 how to use 167 use with \$readmemb and \$readmemh 178 to 179 +pulse_int_e/n 218 +pulse_int_r/m 218 +pulse_r/m 217 +rsw_opt_stack 218 +save_twin_turbo 218, 333 +sdf_cputime 218 +sdf_error_info <u>218</u>, <u>221</u> +sdf_file 219 +sdf_no_errors 220 +sdf no warnings 220 +sdf_nocheck_celltype 219 +sdf_nomsrc_int 220 +sdf_verbose 221

+splitsuh 221 +switchxl 221 how to use 143 use with Turbo 332 +sxl_keep_all 221 affect on optimization 155 example of use 156 +sxl_keep_declared 222 affect on optimization 155 example of use 156 +sxl_keep_minimum 222 affect on optimization 155 +sxl_unidirect <u>222</u> +trace_twin_turbo <u>222, 334</u> +transport int delays 222 +transport_path_delays 223 +turbo <u>223, 330, 331</u> +turbo+2 <u>223, 331</u> +turbo+3 223, 331 +twin_turbo <u>223, 330</u> +typdelays <u>224</u> +vcw 209 +x_transport_pessimism 224 for Shell Generator 194 no error checking 108, 225 predefined 205 to 224 specifying on command line 23 to 24 testing 224 to 225 user-defined specifying at invocation 23 to 24 to test for command line arguments 224 plus plus sign(++) to specify null extensions 107 plus sign(+) separator for +libext arguments 107 port connecting named ports stop net name removal 232 probabilistic distribution functions 312 to 313 \$dist_chi_square 313 \$dist_erlang 313 \$dist_exponential 313 \$dist_normal 313 \$dist_poisson 313 \$dist t 313 \$dist_uniform 313 procedural timing controls reducing behavioral acceleration <u>308</u> pulse handling 316

Q

queue management 309 to 312 \$q_add <u>310</u>, <u>311</u> \$q exam description 311 syntax 311 \$q_full description 311 syntax 311 \$q_initialize 310 \$a remove description 311 syntax 311 queueing tasks and \$restart 312 queueing tasks and \$save 312 status codes 312 status parameters <u>312</u> queueing models 309 Quickturn cosimulation with Verilog-XL 187 model manager for 187 simulating with Verilog-XL 198 Quickturn modes clocked 196 clocked mode options 196 event 195 quiet option 202

R

race condition created when default XL algorithm disabled 290 random number generators probabilistic distribution functions <u>313</u> recordabort See \$recordabort 273 recordclose See \$recordclose 273 recordfile See \$recordfile 273 recordoff See \$recordoff 273 recordon See \$recordon 273 recordsetup

See \$recordsetup 273 recordvars See \$recordvars 273 re-execute 249, 250 reference library 121 registers monitoring 60 reinitializing the network and clock 73 replaying a simulation run 251 reporting non-xl structures 289 resetting the network and clock 73 resolving modules and UDPs 102, 106 to 112 restart file option and -c 200 and compile only option 200 and interactive recovery 250 syntax 202 restarting a simulation 76 restrictions on interactive commands 248 to 249 run-time behavior modifying 87

S

saving a simulation 76 saving simulation data in Twin Turbo 333 scoping model hierarchy 52 SDF annotation +annotate_any_time 206 SDF Annotator errors 218 SDPDs simulating as unconditional paths 230 search paths for included files 88 seed 313 sequential block statement acceleration 307 sessions, interactive reproducing 91 saving interactive input from <u>91</u> Set Breakpoint command code line breakpoints 42 setting foci 253 to 257

trace 260 shell file 194 Shell Generator command 190 command arguments 191 shell generator in Verilog-XL and QuickTurn cosimulation, use of 188 Shell Generator plus options <u>194</u> shellgen command 190 command arguments 191 in Verilog-XL and QuickTurn cosimulation, use of 188 SHM 270 SHM databases using \$recordvars 273 Show History command 46 showing activated and deactivated commands 48 all simulation events 57 breakpoints 269 decompiled source code 50 event order in a time slot 67 expanded macros 50 fanin of signals 62 foci 256 model hierarchy 52 restricted set of simulation events 58 signals values now 62 source code 48 time format 72 wire values periodically 59 wires when they change value 60 signals monitoring 60 simulating CAI 126 Verilog-XL and Quickturn 198 simulation capturing data 306 23 command line list of activities 286 overhead 286 response 286 stimulus 286 simulation history manager (SHM) 270 simulation shell generating 190 simulation time, See time, simulation 29

373

simulations ending 83 modifying behavior at run time 87 performance of 182 restarting 76 76 saving stepping 81 stopping at the beginning 78 stopping during 79 tracing 81 single-stepping a focused area of source code 58 through all source code 57 software behavior \$save files and incompatible hosts 316 'rs_technology keywords in compiler directives 318 timescale placement <u>326</u> back annotation of multiple paths <u>322</u> bit swaps in vector ports 325 changing parameters <u>324</u> collapsing internal nets specified by 'default_nettype <u>320</u> conditional statements in interactive mode 323 expressions in conditional statements 323 linking to Verilog-XL 326 macro module and specify blocks <u>326</u> macro module terminals not <u>31</u>9 expressions MIPDs' effects 322 modulo division on \$random output 324 path destination signals 326 PLI and pulse control 323 PLI calls with \$reset 318 pulse handling 316 radix specifications for strings 326 special considerations for paths <u>321</u> specifying full connections on paths 326 system 5 UNIX c shell scripts 325 vector indices and module boundaries 325 source code conditionally compiling 85 single-stepping through 57 source description file 23 source protection 165 accessing protected

information <u>170</u> to <u>173</u> disables debugging 248 displaying hierarchical path names 173 to 176 effect of timing checks 176 effect on simulation 170 to 173 error messages 176 file extensions 167 protecting all modules and UDPs in a source description 168 protecting data in memory 177 protecting selected regions 166 to 167 specify block path destination signals acceleratable 326 specifying full connections on paths 326 states resetting 73 step 249 stepping 257 in time definition 258 description, syntax, and example 259 through a simulation 81 stochastic analysis 309 probabilistic distribution functions <u>312</u> to <u>313</u> queue management 309 to 312 stop option 203 stopping simulations at the beginning 78 during the simulation 79 strings radix specifications for tasks 326 strobing signals 59 swapping bits in vector ports 325 Switch XL conversion of channel delay to turn on/ turn off delay 149 switch-level simulation 139 <u>13</u>9 algorithms algorithms' major features 140 choosing an algorithm 142 default algorithm 144 to ?? locally enabling algorithms 143 networks 139 Switch-XL <u>148</u> to <u>162</u> default charge and drive strength 159 net removal 152

optimization <u>152</u> to <u>157</u> purpose 141 strength mapping 160 strength reduction 159 Switch-XL algorithm 148 to 162 Switch-XL strength model 157 to 162 symbolic debugging 247 to 251 syntax \$db breakaftertime 265 \$db breakatline 264 \$db_breakbeforetime 264 \$db breakonceatline 264 \$db_breakonceonnegedge 267 \$db_breakoncewhen 265 \$db breakonnegedge 266 \$db_breakonposedge 266 \$db_breakwhen 265 \$db_cleartrace 260 \$db_deletebreak 267 255 \$db deletefocus \$db_disablebreak 268 \$db_disablefocus 256 \$db_enablebreak 268 \$db_enablefocus 255 \$db_help 251 \$db setfocus 254 \$db settrace 260 \$db_showbreak 269 \$db_showfocus 256 \$db_step 259 \$db_steptime 259 \$dist_chi_square <u>313</u> \$dist_erlang 313 \$dist_exponential 313 \$dist_normal 313 \$dist_poisson 313 \$dist t 313 \$dist_uniform <u>313</u> \$q_add <u>310, 311</u> \$q_exam <u>311</u> \$q_full <u>311</u> \$q_initialize <u>310</u> \$q_remove <u>311</u> \$test\$plusargs 224 +define+ 207 +incdir+ 210 +libext+ 210 'switch 143 errors and <-356 -i option 200, 201 interactive commands 249

-k option <u>202</u> -l option <u>202</u> mc_scan_plusargs <u>225</u> -r option <u>202</u> Switch-XL strength model <u>158</u> -v option <u>203, 204, 212</u> -y option <u>204</u> system task associating your C routine with a <u>135</u> system tasks effect of source protection <u>170 to 173</u>

Т

test fixtures creating 27 testbench modifying 190 text macro substitutions and +define+ 207 and 'define 229 and undef 233 time controling display and interpretation of 72 resetting 73 time slot showing event order in 67 time, simulation defining variables for 29 displaying 29 getting with system tasks -29 timing checks \$recovery and +neg_tchk 214 \$setuphold and +neg_tchk 214 and +splitsuh 221 top-level module avoiding unnecessary creation of 99 trace \$settrace in Twin Turbo 334 and acceleration 291 option 203 trace-step 249 tracing 258 event order in a time slot 67 simulation events 57 through a simulation 81

traversing model hierarchy <u>52</u> Turbo <u>329</u> data structure <u>329</u> invoking <u>330</u> turn on/turn off delay timing model <u>149</u> Twin Turbo <u>329, 330</u> and Turbo compatibility of results <u>334</u> invoking <u>330</u> optimizing a design for <u>334</u> restrictions <u>333</u> saving and restarting <u>333</u>

U

UDPs performance <u>300</u> ULM <u>23</u> upper case option <u>203</u> user-defined options <u>224</u> no error checking <u>225</u> specifying on command line <u>23</u> to <u>24</u> testing <u>224</u> to <u>225</u> Using <u>273</u> utility routines <u>130</u>

V

values passing from the command line 84 vconfig 131, 134 vectors and module boundaries 325 and vector net expansion 204 Verilog-XL simulating with Quickturn 198 task flow 20 verinuser.c purpose of <u>130</u> veriuser.h 130 View definition of <u>122</u> view <u>122</u> vpi_user.c 131 vpi_user.h <u>131</u> vpi user cds.h 131

W

warning messages <u>355</u> disabling with +nowarn <u>216</u> warning suppression <u>220</u> warning suppression for SDF annotation <u>220</u> warning suppression option <u>204</u> warnings disabling with +nowarn <u>216</u> what if asking <u>64</u> wires monitoring <u>60</u> work library <u>121</u> writing simulation data to files <u>30</u>

Χ

XL algorithm <u>285</u> to <u>307</u> accelerated primitives and scalar nets <u>287</u> and 'accelerate <u>226</u> and key files containing asynchronous interrupts <u>291</u> and tracing <u>291</u> controlling application <u>286</u> to <u>287</u> non-accelerated items <u>288</u> non-XL simulation <u>290</u> to <u>291</u> potential problems <u>291</u> to <u>292</u> processing simultaneous events <u>290</u> when pulse width equals gate delay <u>291</u>

Ζ

zero-delay oscillation <u>290</u>