
Verilog[®]-XL Reference

Product Version 3.4
January 2002

© 1990-2002 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) 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 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. 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. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. 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

<u>Introduction</u>	19
<u>Overview</u>	19
<u>The Verilog Hardware Description Language</u>	19
<u>The Verilog-XL Logic Simulator</u>	21
<u>Major Features of Verilog-XL</u>	21
<u>Verilog-XL Licenses</u>	22

2

<u>Lexical Conventions</u>	23
<u>Overview</u>	23
<u>Operators</u>	23
<u>White Space and Comments</u>	24
<u>Numbers</u>	24
<u>Strings</u>	26
<u>String Variable Declaration</u>	26
<u>String Manipulation</u>	26
<u>Special Characters in Strings</u>	27
<u>Identifiers, Keywords, and System Names</u>	28
<u>Escaped Identifiers</u>	28
<u>Keywords</u>	29
<u>Text Substitutions</u>	29

3

<u>Data Types</u>	31
<u>Overview</u>	31
<u>Value Set</u>	31
<u>Registers and Nets</u>	32
<u>Nets</u>	32
<u>Registers</u>	32

Verilog-XL Reference

<u>Signed Objects</u>	33
<u>Net and Register Declaration Syntax</u>	35
<u>Declaration Examples</u>	37
<u>Vectors</u>	37
<u>Specifying Vectors</u>	38
<u>Vector Net Accessibility</u>	38
<u>Strengths</u>	38
<u>Charge Strength</u>	39
<u>Drive Strength</u>	39
<u>Implicit Declarations</u>	39
<u>Net Initialization</u>	40
<u>Net Types</u>	40
<u>wire and tri Nets</u>	40
<u>Wired Nets</u>	41
<u>trireg Net</u>	41
<u>tri0 and tri1 Nets</u>	45
<u>Supply Nets</u>	45
<u>Memories</u>	46
<u>Integers and Times</u>	47
<u>Real Numbers</u>	48
<u>Real Number Declaration Syntax</u>	48
<u>Specifying Real Numbers</u>	48
<u>Operators and Real Numbers</u>	49
<u>Conversion</u>	49
<u>Parameters</u>	50

4

<u>Expressions</u>	51
<u>Overview</u>	51
<u>Operators</u>	52
<u>Binary Operator Precedence</u>	53
<u>Numeric Conventions in Expressions</u>	54
<u>Arithmetic Operators</u>	54
<u>Arithmetic Expressions with Registers and Integers</u>	55
<u>Relational Operators</u>	56

Verilog-XL Reference

<u>Equality Operators</u>	56
<u>Logical Operators</u>	57
<u>Bit-Wise Operators</u>	58
<u>Reduction Operators</u>	59
<u>Syntax Restrictions</u>	60
<u>Shift Operators</u>	60
<u>Arithmetic Shift Operators for Signed Objects</u>	60
<u>Conditional Operator</u>	61
<u>Concatenations</u>	62
<u>Operands</u>	62
<u>Net and Register Bit Addressing</u>	63
<u>Memory Addressing</u>	64
<u>Strings</u>	64
<u>String Operations</u>	65
<u>String Value Padding and Potential Problems</u>	65
<u>Null String Handling</u>	66
<u>Minimum, Typical, Maximum Delay Expressions</u>	66
<u>Expression Bit Lengths</u>	67
<u>An Example of an Expression Bit Length Problem</u>	67
<u>Verilog Rules for Expression Bit Lengths</u>	68

5

<u>Assignments</u>	71
<u>Overview</u>	71
<u>Continuous Assignments</u>	72
<u>The Continuous Assignment Statement</u>	73
<u>The Net Declaration Assignment</u>	73
<u>Delays</u>	74
<u>Strength</u>	77
<u>Calling Functions in a Continuous Assignment</u>	78
<u>Procedural Assignments</u>	79
<u>Accelerated Continuous Assignments</u>	80
<u>Restrictions on Accelerated Continuous Assignments</u>	80
<u>Controlling the Acceleration of Continuous Assignments</u>	86
<u>The Effects of Accelerated Continuous Assignments</u>	87

Verilog-XL Reference

<u>Procedural Continuous Assignments</u>	93
<u>The assign and deassign Procedural Statements</u>	94
<u>The force and release Procedural Statements</u>	95

6

Gate and Switch Level Modeling..... 97

<u>Overview</u>	98
-----------------------	----

<u>Gate and Switch Declaration Syntax</u>	98
---	----

<u>The Gate Type Specification</u>	100
--	-----

<u>The Drive Strength Specification</u>	100
---	-----

<u>The Delay Specification</u>	101
--------------------------------------	-----

<u>The Primitive Instance Identifier</u>	101
--	-----

<u>The Range Specification</u>	101
--------------------------------------	-----

<u>Primitive Instance Connection List</u>	102
---	-----

<u>Rules for Using an Array of Instances</u>	102
--	-----

<u>and, nand, nor, or, xor, and xnor Gates</u>	105
--	-----

<u>buf and not Gates</u>	106
--------------------------------	-----

<u>bufif1, bufif0, notif1, and notif0 Gates</u>	107
---	-----

<u>MOS Switches</u>	108
---------------------------	-----

<u>Bidirectional Pass Switches</u>	110
--	-----

<u>cmos Switches</u>	112
----------------------------	-----

<u>pullup and pulldown Sources</u>	113
--	-----

<u>Implicit Net Declarations</u>	113
--	-----

<u>Logic Strength Modeling</u>	114
--------------------------------------	-----

<u>Strengths and Values of Combined Signals</u>	116
---	-----

<u>Combined Signals of Unambiguous Strength</u>	116
---	-----

<u>Ambiguous Strengths: Sources and Combinations</u>	117
--	-----

<u>Ambiguous Strength Signals and Unambiguous Signals</u>	123
---	-----

<u>Wired Logic Net Types</u>	127
------------------------------------	-----

<u>Strength Resolution for Continuous Assignments</u>	129
---	-----

<u>Mnemonic Format</u>	130
------------------------------	-----

<u>Strength Reduction by Non-Resistive Devices</u>	131
--	-----

<u>Strength Reduction by Resistive Devices</u>	131
--	-----

<u>Strengths of Net Types</u>	131
-------------------------------------	-----

<u>tri0 and tri1 Net Strengths</u>	131
--	-----

Verilog-XL Reference

<u>trireg Strength</u>	132
<u>supply0 and supply1 Net Strengths</u>	132
<u>Gate and Net Delays</u>	132
<u>min/typ/max Delays</u>	136
<u>trireg Net Charge Decay</u>	137
<u>Gate and Net Name Removal</u>	140

7

User-Defined Primitives (UDPs)

<u>Overview</u>	143
<u>UDP Syntax</u>	144
<u>UDP Definition</u>	145
<u>UDP Terminals</u>	146
<u>UDP Declarations</u>	146
<u>Sequential UDP initial Statement</u>	146
<u>UDP State Table</u>	146
<u>Summary of UDP Symbols</u>	148
<u>Combinational UDPs</u>	148
<u>Level-Sensitive Sequential UDPs</u>	150
<u>Edge-Sensitive UDPs</u>	150
<u>Sequential UDP Initialization</u>	151
<u>Mixing Level-Sensitive and Edge-Sensitive Descriptions</u>	154
<u>Level-Sensitive Dominance</u>	155
<u>UDP Instances</u>	156
<u>Compilation</u>	157
<u>Reducing Pessimism</u>	158
<u>Processing of Simultaneous Input Changes</u>	159
<u>Memory Usage and Performance Considerations</u>	160
<u>UDP Examples</u>	161

8

Behavioral Modeling

<u>Overview</u>	163
<u>Structured Procedures</u>	164
<u>always Statement</u>	165

Verilog-XL Reference

<u>initial Statement</u>	165
<u>Procedural Assignments</u>	166
<u>Blocking Procedural Assignments</u>	167
<u>Non-Blocking Procedural Assignments</u>	167
<u>Processing Blocking and Non-Blocking Procedural Assignments</u>	173
<u>Conditional Statements</u>	174
<u>Multi-Way Decision Statements</u>	175
<u>if-else-if Statements</u>	175
<u>case Statements</u>	176
<u>Using case Statements with Inconsequential Conditions</u>	178
<u>Looping Statements</u>	179
<u>forever Loop</u>	180
<u>repeat Loop</u>	180
<u>while Loop</u>	181
<u>for Loop</u>	181
<u>Procedural Timing Controls</u>	182
<u>Delay Control</u>	183
<u>Zero-Delay Control</u>	183
<u>Event Control</u>	184
<u>Named Events</u>	184
<u>Event OR Construct</u>	185
<u>Level-Sensitive Event Control</u>	186
<u>Intra-Assignment Timing Controls</u>	186
<u>Block Statements</u>	189
<u>Sequential Blocks</u>	189
<u>Parallel Blocks</u>	191
<u>Block Names</u>	192
<u>Start and Finish Times</u>	192
<u>Behavior Model Examples</u>	193

9

<u>Tasks and Functions</u>	197
<u>Overview</u>	197
<u>Distinctions Between Tasks and Functions</u>	197
<u>Tasks and Task Enabling</u>	198

Verilog-XL Reference

<u>Defining a Task</u>	198
<u>Task Enabling and Argument Passing</u>	199
<u>Task Example</u>	200
<u>Effect of Enabling an Already Active Task</u>	201
<u>Functions and Function Calling</u>	201
<u>Defining a Function</u>	201
<u>Returning a Value from a Function</u>	202
<u>Calling a Function</u>	202
<u>Function Rules</u>	203
<u>Function Example</u>	203

10

<u>Disabling of Named Blocks and Tasks</u>	205
<u>Overview</u>	205
<u>Syntax</u>	205
<u>disable Statement Examples</u>	206

11

<u>Hierarchical Structures</u>	209
<u>Overview</u>	209
<u>Modules</u>	210
<u>Top-Level Modules</u>	211
<u>Module Instantiation</u>	211
<u>Module Definition and Instance Example</u>	211
<u>Overriding Module Parameter Values</u>	213
<u>Using the defparam Statement</u>	214
<u>Using Module Instance Parameter Value Assignment</u>	215
<u>Parameter Dependence</u>	216
<u>Macro Modules</u>	216
<u>Constructs Allowed in Macro Modules</u>	216
<u>Specifying Macro Modules</u>	217
<u>Instances of Macro Modules</u>	217
<u>Using Parameters with Macro Modules</u>	217
<u>Effect on Decompile and Tracing</u>	218
<u>Ports</u>	219

Verilog-XL Reference

<u>Port Definition</u>	219
<u>Port Declarations</u>	220
<u>Connecting Module Ports by Ordered List</u>	220
<u>Connecting Module Ports by Name</u>	221
<u>Real Numbers in Port Connections</u>	223
<u>Port Collapsing</u>	223
<u>Port Connection Rules</u>	224
<u>Port Connections in Macro Modules</u>	227
<u>Hierarchical Names</u>	228
<u>Data Structures</u>	231
<u>Macro Modules and Hierarchical Names</u>	231
<u>Upwards Name Referencing</u>	232
<u>Automatic Naming</u>	233
<u>Scope Rules</u>	234

12

<u>Using Specify Blocks and Path Delays</u>	237
<u>Understanding Specify Blocks</u>	237
<u>Specparam Declarations</u>	238
<u>Understanding Path Delays</u>	239
<u>Driving Wired Logic Outputs</u>	242
<u>Simulating Distributed Delays as Inertial and Transport Delays</u>	244
<u>Simulating Path Delays</u>	244
<u>Describing Module Paths</u>	247
<u>Establishing Parallel or Full Connections</u>	248
<u>Specifying Transition Delays on Module Paths</u>	251
<u>Calculating Delay Values for X Transitions</u>	253
<u>Specifying Module Path Polarity</u>	254
<u>Using Path Delays in Behavioral Descriptions</u>	255
<u>Simulating Path Outputs that Drive Other Path Outputs</u>	256
<u>Understanding Strength Changes on Path Inputs</u>	257
<u>Specifying Global Pulse Control on Module Paths</u>	257
<u>Specifying Local Pulse Control for Module Paths</u>	259
<u>Pulse Filtering for Module Path Delays</u>	260
<u>Pulse Filtering and Cancelled Schedules</u>	262

Verilog-XL Reference

<u>Pulse Filtering and Cancelled Schedule Dilemmas</u>	266
<u>Using State-Dependent Path Delays (SDPDs)</u>	269
<u>Evaluating SDPD Expressions</u>	270
<u>Using Edge Keywords in SDPDs</u>	273
<u>Making SDPDs Function as Unconditional Delays</u>	274
<u>Working with Distributed Delays and SDPDs</u>	274
<u>Working with Multiple Path Delays</u>	275
<u>Effects of Unknowns on SDPDs</u>	276
<u>Effects of Unknowns on Edge-Sensitive Delays</u>	277
<u>Possible Effects of Internal Logic</u>	277
<u>Enhancing Path Delay Accuracy</u>	279
<u>Invoking the <code>accu_path</code> Algorithm</u>	279
<u>Comparing the Default and <code>accu_path</code> Delay Selection Algorithms</u>	281
<u>Limits of the <code>accu_path</code> Algorithm</u>	285

13

<u>Timing Checks</u>	289
<u>Overview</u>	289
<u>Using Timing Checks</u>	289
<u>Understanding Timing Violation Messages</u>	290
<u>Using Edge-Control Specifiers</u>	291
<u>Using Notifiers for Timing Violations</u>	292
<u>Enabling Timing Checks with Conditioned Events</u>	293
<u>Using the Timing Check System Tasks</u>	295
<u><code>\$hold</code></u>	295
<u><code>\$nochange</code></u>	297
<u><code>\$period</code></u>	298
<u><code>\$recovery</code></u>	299
<u><code>\$recrem</code></u>	301
<u><code>\$removal</code></u>	304
<u><code>\$setup</code></u>	305
<u><code>\$setuphold</code></u>	307
<u><code>\$skew</code></u>	310
<u><code>\$width</code></u>	311
<u>Using Negative Timing Check Limits in <code>\$setuphold</code> and <code>\$recrem</code></u>	313

Verilog-XL Reference

<u>Effects of Delayed Signals on Timing Checks</u>	314
<u>Calculation of Delayed Signals and Limit Modification</u>	316
<u>Explicitly Defining Delayed Signals</u>	318
<u>Non-Convergence in Timing Checks</u>	319
<u>Explicitly Defining Delayed Signals</u>	326
<u>Effects of Delayed Signals on Path Delays</u>	327
<u>Restrictions</u>	328
<u>Exception Handling</u>	330

14

<u>System Tasks and Functions</u>	331
<u>Filename Parameters</u>	332
<u>Display and Write Tasks</u>	333
<u>Escape Sequences for Special Characters</u>	334
<u>Format Specifications</u>	334
<u>Size of Displayed Data</u>	336
<u>Unknown and High-Impedance Values</u>	337
<u>Strength Format</u>	338
<u>Hierarchical Name Format</u>	340
<u>String Format</u>	340
<u>Strobed Monitoring</u>	340
<u>Continuous Monitoring</u>	341
<u>Monitoring Interconnect Delay Signal Values</u>	342
<u>File Output</u>	343
<u>Default Base</u>	345
<u>Signed Expressions</u>	346
<u>Simulation Time</u>	346
<u>Stop and Finish</u>	347
<u>Random Number Generation</u>	347
<u>Tracing</u>	348
<u>Saving and Restarting Simulations</u>	352
<u>Incremental Save and Restart</u>	354
<u>Command-Line Restart</u>	355
<u>Limitations for Saving and Restarting</u>	355
<u>Command History</u>	355

Verilog-XL Reference

<u>Command Input Files</u>	356
<u>Log File</u>	356
<u>Key File</u>	357
<u>Setting the Interactive Scope</u>	358
<u>Showing the Hierarchy</u>	358
<u>Showing Variable Status</u>	358
<u>Showing Net Expansion Status</u>	359
<u>Showing Module Port Status</u>	360
<u>Showing Number of Drivers</u>	360
<u>Displaying the Delay Mode</u>	362
<u>Storing Interactive Commands</u>	362
<u>Interactive Source Listing—Decompilation</u>	363
<u>\$list</u>	363
<u>\$listcounts</u>	363
<u>\$list forces</u>	364
<u>Disabling and Enabling Warnings</u>	365
<u>\$disable warnings</u>	366
<u>\$enable warnings</u>	367
<u>Loading Memories from Text Files</u>	368
<u>Setting a Net to a Logic Value</u>	369
<u>Fast Processing of Stimulus Patterns</u>	370
<u>Incremental Pattern File Tasks</u>	372
<u>\$incpattern write</u>	372
<u>\$incpattern read</u>	373
<u>\$compare</u>	375
<u>\$strobe compare</u>	376
<u>Examples of Response Checking</u>	378
<u>Functions and Tasks for Reals</u>	380
<u>Functions and Tasks for Timescales</u>	380
<u>Protecting Data in Memory</u>	381
<u>Value Change Dump File Tasks</u>	382
<u>Running the Behavior Profiler</u>	382
<u>\$startprofile</u>	383
<u>\$reportprofile</u>	383
<u>\$listcounts</u>	384
<u>\$stopprofile</u>	384

Verilog-XL Reference

<u>Resetting Verilog-XL—Starting Simulation Over Again</u>	384
<u>\$reset</u>	385
<u>\$reset count</u>	390
<u>\$reset value</u>	391
<u>SDF Annotation</u>	392
<u>\$sdf annotate</u>	393
<u>Controlling \$sdf annotate Output</u>	395
<u>\$sdf annotate Examples</u>	395
<u>Annotating Path Delay or Timing Check Vector Bits in Specify Blocks</u>	399
<u>Using the \$dlc System Task</u>	403
<u>Using the \$system System Task</u>	404

15

<u>Programmable Logic Arrays</u>	405
<u>Overview</u>	405
<u>Syntax</u>	405
<u>Array Types</u>	406
<u>Array Logic Types</u>	406
<u>Logic Array Personality Declaration and Loading</u>	407
<u>Logic Array Personality Formats</u>	407
<u>PLA Examples</u>	409
<u>Synchronous Example</u>	409
<u>And-Or Array Example</u>	410
<u>PAL16R8 Example</u>	411
<u>PAL16R4 Example</u>	416

16

<u>Interconnect Delays</u>	421
<u>Overview</u>	421
<u>Module Import Port Delays (MIPDs)</u>	423
<u>How MIPDs Work</u>	423
<u>Specifying MIPDs</u>	428
<u>Restrictions on Ports for MIPDs</u>	429
<u>Monitoring Nets Internal to MIPDs</u>	430
<u>Displaying Status Information for Nets Internal to MIPDs</u>	430

Verilog-XL Reference

<u>An Application of MIPDs</u>	431
<u>Single-Source/MultiSource Interconnect Transport Delays (S/MITDs)</u>	432
<u>Controlling MIPD and S/MITD Creation</u>	433
<u>S/MITDs and Pulse Handling</u>	436
<u>Resolving Ambiguous S/MITD Events</u>	436
<u>PLI Tasks for S/MITDs</u>	438

17

Timescales

<u>Overview</u>	439
<u>The 'timescale Compiler Directive</u>	440
<u>Usage Rules</u>	440
<u>Syntax</u>	440
<u>Effects of Timescales on Simulation Performance</u>	442
<u>Timescale System Functions</u>	442
<u>\$time</u>	443
<u>\$realtime</u>	444
<u>\$scale</u>	444
<u>The Timescale System Tasks</u>	445
<u>\$printtimescale</u>	446
<u>\$timeformat</u>	446
<u>Timescales Examples</u>	449

18

Delay Mode Selection

<u>Overview</u>	453
<u>Delay Modes</u>	454
<u>Unit Delay Mode</u>	454
<u>Zero Delay Mode</u>	454
<u>Distributed Delay Mode</u>	455
<u>Path Delay Mode</u>	455
<u>Default Delay Mode</u>	456
<u>Reasons to Select a Delay Mode</u>	456
<u>Setting a Delay Mode</u>	456
<u>Compiler Directives</u>	456

Verilog-XL Reference

<u>Command-Line Plus Options</u>	457
<u>Precedence in Selection</u>	457
<u>Timescales and Simulation Time Units</u>	458
<u>Overriding Delay Values</u>	459
<u>PLI 1.0 or VPI Access Routines and Delays</u>	459
<u>Parameter Attribute Mechanism</u>	460
<u>Delay Mode Example</u>	461
<u>Decompiling with Delay Modes</u>	462
<u>\$showmodes</u>	462
<u>acc fetch delay mode Access Routine</u>	462
<u>Macro Module Expansion and Delay Modes</u>	462
<u>Summary of Delay Mode Rules</u>	463

19

<u>The Behavior Profiler</u>	465
<u>How the Behavior Profiler Works</u>	465
<u>Behavior Profiler System Tasks</u>	467
<u>\$startprofile</u>	467
<u>\$reportprofile</u>	468
<u>\$stopprofile</u>	469
<u>\$listcounts</u>	469
<u>Behavior Profiler Data Report</u>	471
<u>Profile Ranking by Statement</u>	471
<u>Profile Ranking by Module Instance</u>	474
<u>Profile Ranking by Statement Class</u>	475
<u>Profile Ranking by Statement Type</u>	476
<u>Recommended Modeling Practices</u>	483
<u>Invoke the Behavior Profiler After You Initialize Your Design</u>	483
<u>Put Statements on Separate Lines</u>	483
<u>How Verilog-XL Affects Profiler Results</u>	483
<u>Using a Variable to Drive Multiple Module Instances</u>	483
<u>Expanded Vector Nets</u>	483
<u>Accelerated Events</u>	484
<u>Behavior Profiler Example</u>	484

20

<u>The Value Change Dump File</u>	493
<u>Overview</u>	493
<u>Creating the Value Change Dump File</u>	493
<u>Specifying the Dump File Name (\$dumpfile)</u>	494
<u>Specifying Variables for Dumping (\$dumpvars)</u>	495
<u>Stopping and Resuming the Dump (\$dumpoff/\$dumpon)</u>	496
<u>Generating a Checkpoint (\$dumpall)</u>	496
<u>Limiting the Size of the Dump File (\$dumplimit)</u>	497
<u>Reading the Dump File During Simulation (\$dumpflush)</u>	497
<u>Sample Source Description Containing VCD Tasks</u>	498
<u>Format of the Value Change Dump File</u>	498
<u>Contents of the Dump File</u>	498
<u>Structure of the Dump File</u>	499
<u>Formats of Dumped Variable Values</u>	499
<u>Using Keyword Commands</u>	500
<u>Description of Keyword Commands</u>	501
<u>Syntax of the VCD File</u>	506
<u>Value Change Dump File Format Example</u>	507
<u>Using the \$dumpports System Task</u>	510
<u>\$dumpports Syntax</u>	510
<u>\$dumpports Output</u>	511
<u>\$dumpports Restrictions</u>	515
<u>\$dumpports close</u>	515

A

<u>Formal Syntax Definition</u>	517
<u>Summary of Syntax Descriptions</u>	517
<u>Source Text</u>	518
<u>Declarations</u>	521
<u>Primitive Instances</u>	523
<u>Module Instantiations</u>	523
<u>Behavioral Statements</u>	524
<u>Specify Section</u>	526

Verilog-XL Reference

<u>Expressions</u>	529
<u>General Syntax Definition</u>	530
<u>Switch-Level Modeling</u>	531

B

<u>Verilog-XL Keywords</u>	533
<u>Keywords from Compiler Directives</u>	533
<u>Keywords from Specify Blocks</u>	535
<u>Keywords from Neither Compiler Directives nor Specify Blocks</u>	535

C

<u>Verilog-XL and Standards' Compliance</u>	539
<u>Supported Standards</u>	539
<u>Known Exceptions</u>	539
<u>VPI Routines</u>	539
<u>Wire with same name as a Port</u>	540

<u>Index</u>	541
--------------------	-----

Introduction

This chapter describes the following information:

- [Overview](#) on page 19
- [The Verilog Hardware Description Language](#) on page 19
- [The Verilog-XL Logic Simulator](#) on page 21

Overview

This reference manual describes the features of the Verilog-XL digital logic simulator and the Verilog Hardware Description Language you use to model a design for simulation by Verilog-XL.

The Verilog Hardware Description Language

The Verilog Hardware Description Language (HDL) describes a hardware design or part of a design. Verilog models are descriptions of designs in the Verilog HDL. The Verilog HDL is both a behavioral and a structural language. Models in the Verilog HDL can describe both the function of a design and the components and the connections to the components in a design.

Verilog models can be developed for different levels of abstraction. These levels of abstraction and their corresponding model types are described in [Table](#) on page 19.

Table 1-1 Verilog models and their level of abstraction

algorithmic	a model that implements a design algorithm in high-level language constructs
RTL	a model that describes the flow of data between registers and how a design processes that data
gate-level	a model that describes the logic gates and the connections between logic gates in a design

Verilog-XL Reference

Introduction

Table 1-1 Verilog models and their level of abstraction

switch-level	a model that describes the transistors and storage nodes in a device and the connections between them
--------------	---

The basic building block of the Verilog-XL HDL is the module. The module format facilitates top-down and bottom-up design. A module contains a model of a design or part of a design. Modules can incorporate other modules to establish a model hierarchy that describes how parts of a design are incorporated in an entire design. The constructs of the Verilog HDL, such as its declarations and statements, are enclosed in modules.

The Verilog HDL behavioral language is structured and procedural like the C programming language. The behavioral language constructs are for algorithmic and RTL models. The behavioral language provides the following capabilities:

- structured procedures for sequential or concurrent execution
- explicit control of the time of procedure activation specified by both delay expressions and by value changes called event expressions
- explicitly named events to trigger the enabling and disabling of actions in other procedures
- procedural constructs for conditional, if-else, case, and looping operations
- procedures called tasks that can have parameters and non-zero time duration
- procedures called functions that allow the definition of new operators
- arithmetic, logical, bit-wise, and reduction operators for expressions

The Verilog HDL structural language constructs are for gate-level and switch-level models. The structural language provides the following capabilities:

- a complete set of combinational primitives
- primitives for bidirectional pass and resistive devices
- the ability to model dynamic MOS models with charge sharing and charge decay

Verilog structural language models can accurately model signal contention. In the Verilog HDL, structural modeling accuracy is enhanced by primitive delay and output strength specification. Signal values can have different strengths and a full range of ambiguous values to reduce the pessimism of unknown conditions.

The Verilog-XL Logic Simulator

The Verilog-XL digital logic 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 HDL and then supply Verilog-XL with the file names that contain these models. You also need a Verilog-XL license. This section describes the major features of Verilog-XL and the Verilog-XL license.

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

Verilog-XL Reference

Introduction

- investigating the performance ramifications of architectural decision—stochastic modeling
- simulating with SimVision, the Verilog-XL graphical user interface

Verilog-XL Licenses

To use the Verilog-XL logic simulator you need a license. The SoftShare application handles all licenses for Verilog-XL and provides a variety of license management tools and options.

When you invoke Verilog-XL, SoftShare searches for a license file and checks out a license for you if one is available. You can queue a request for a license if one is not currently available. When you queue license requests, you can automatically run a number of simulations as licenses become available. Requests in the queue are first-in-first-out with all requests at the same priority level. There is no time-out on queues, meaning that you cannot wait for a license for a fixed time. To remove a request from the queue, you must provide an interrupt signal.

The following license features can be queued:

- `VERILOG-XL` (Verilog-XL)
- `VXL-LMC-HW-IF` (Verilog-XL LMC Hardware Interface). This feature is checked out during compilation whenever there is a LMSI (LMC Hardware Interface) system task, `$lm_*()`, present in the design.

To enable the queuing, use the following command-line plus options:

- `+licq_vxl` (queue only the `VERILOG-XL` license)
- `+licq_lmchwif` (queue only the `VXL-LMC-HW-IF` license)
- `+licq_all` (queue all of the above licenses)

Lexical Conventions

This chapter describes the following:

- [Overview](#) on page 23
- [Operators](#) on page 23
- [White Space and Comments](#) on page 24
- [Numbers](#) on page 24
- [Strings](#) on page 26
- [Identifiers, Keywords, and System Names](#) on page 28
- [Text Substitutions](#) on page 29

Overview

Verilog language source text files are a stream of lexical tokens. A token consists of one or more characters, and each single character is in exactly one token. The layout of tokens in a source file is free format — that is, spaces and newlines are not syntactically significant. However, spaces and newlines are very important for giving a visible structure and format to source descriptions. A good style of format, and consistency in that style, are an essential part of program readability.

This manual uses a syntax formalism based on the Backus-Naur Form (BNF) to define the Verilog language syntax. [Appendix A, “Formal Syntax Definition”](#) contains the complete set of syntax definitions in this format, plus a description of the BNF conventions used in the syntax definitions.

Operators

Operators are single, double, or triple character sequences and are used in expressions. [Chapter 4, “Expressions,”](#) discusses the use of operators in expressions.

Unary operators appear to the left of their operand. Binary operators appear between their operands. A ternary operator has two operator characters that separate three operands. The Verilog language has one ternary operator the—conditional operator. See “[Conditional Operator](#)” on page 61 for an explanation of the conditional operator.

White Space and Comments

White space can contain the characters for blanks, tabs, newlines, and formfeeds. The Verilog language ignores these characters except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

The Verilog language has two forms to introduce comments. A one-line comment starts with the two characters `//` and ends with a newline. A block comment starts with `/*` and ends with `*/`. Block comments cannot be nested, but a one-line comment can be nested within a block comment.

Numbers

You can specify constant numbers in decimal, hexadecimal, octal, or binary format. The Verilog language defines two forms to express numbers. The first form is a simple decimal number specified as a sequence of the digits 0 to 9 which can optionally start with a plus or minus. The second takes the following form:

```
<size><base_format><number>
```

The `<size>` element contains decimal digits that specify the size of the constant in terms of its exact number of bits. For example, the `<size>` specification for two hexadecimal digits is 8, because one hexadecimal digit requires four bits. The `<size>` specification is optional. The `<base_format>` contains a letter specifying the number’s base, preceded by the single quote character (`'`). Legal base specifications are one of `d`, `h`, `o`, or `b`, for the bases decimal, hexadecimal, octal, and binary respectively. (Note that these base identifiers can be upper or lowercase.)

The `<number>` element contains digits that are legal for the specified `<base_format>`. The `<number>` element must physically follow the `<base_format>`, but can be separated from it by spaces. No spaces can separate the single quote and the base specifier character.

Alphabetic letters used to express the `<base_format>` or the hexadecimal digits `a` to `f` can be in upper- or lowercase.

The following example shows *unsized* constant numbers.

```
659           // is a decimal number
'h 837FF      // is a hexadecimal number
```


Verilog-XL Reference

Lexical Conventions

```
'o7460      // is an octal number
4af         // is illegal (hexadecimal format requires 'h)
```

The following example shows *sized* constant numbers

```
4'b1001      // is a 4-bit binary number
5 'D 3       // is a 5-bit decimal number
3'b01x       // is a 3-bit number with the least significant bit unknown
12'hx        // is a 12-bit unknown number
16'hz        // is a 16-bit high-impedance number
```

In the Verilog language a plus or minus preceding the size constant is a sign for the constant number—the size constant does not take a sign. A plus or minus between the *<base_format>* and the *<number>* is illegal syntax. In the following example, the first expression is a syntax error. The second expression legally defines an 8-bit number with a value of minus 6.

```
8 'd -6      // this is illegal syntax
-8 'd 6      // this defines the two's complement of 6,
              // held in 8 bits—equivalent to -(8'd 6)
```

The number of bits that make up an un-sized number (which is a simple decimal number or a number without the *<size>* specification) is the host machine word size—for most machines this is 32 bits.

In the Verilog language, an *x* expresses the unknown value in hexadecimal, octal, and binary constants. A *z* expresses the high-impedance value. See “[Value Set](#)” on page 31 for a discussion of the Verilog value set. An *x* sets four bits to unknown in the hexadecimal base, three bits in the octal base, and one bit in the binary base.

Similarly, a *z* sets four, three, and one bit, respectively, to the high-impedance value. If the most significant specified digit of a constant number is an *x* or a *z*, then Verilog-XL automatically extends the *x* or *z* to fill the higher order bits of the constant. This makes it easy to specify complete vectors of the unknown and the high-impedance values. The following example illustrates this value extension:

```
reg [11:0] a;
initial
begin
  a = 'h x;           // yields xxx
  a = 'h 3x;         // yields 03x
  a = 'h 0x;         // yields 00x
end
```

The question mark (?) character is a Verilog HDL alternative for the *z* character. It sets four bits to the high-impedance value in hexadecimal numbers, three in octal, and one in binary. Use the question mark to enhance readability in cases where the high-impedance value is a don't-care condition. See the discussion of *casez* and *casex* in “[case Statements](#)” on page 176 and the discussion on personality files in “[Logic Array Personality Formats](#)” on page 407.

Verilog-XL Reference

Lexical Conventions

The underline character is legal anywhere in a number except as the first character. Use this feature to break up long numbers for readability purposes. The following example illustrates this.

```
27_195_000
16'b0011_0101_0001_1111
32'h 12ab_f001
```

Underline characters are also legal in numbers in text files read by the `$readmemb` and `$readmemh` system tasks.

Note: A sized negative number is not sign-extended when assigned to a register data type.

Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Verilog treats strings used as operands in expressions and assignments as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. The following shows examples of strings:

```
"this is a string"
"print out a message\n"
"bell!\007"
```

String Variable Declaration

To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character.

For example, to store the string "Hello world!" requires a register 8*12, or 96 bits wide, as follows:

```
reg [8*12:1] stringvar;
initial
begin
    stringvar = "Hello world!";
end
```

String Manipulation

Verilog permits strings to be manipulated using the standard Verilog HDL operators. Keep in mind that the value being manipulated by an operator is a sequence of 8-bit ASCII values, with no special termination character.

Verilog-XL Reference

Lexical Conventions

The code in the following example declares a string variable large enough to hold 14 characters and assigns a value to it. The code then manipulates this string value using the concatenation operator.

```
module string_test;
reg [8*14:1] stringvar;
  initial
  begin
    stringvar = "Hello world";
    $display("%s is stored as %h",stringvar,stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h",stringvar,stringvar);
  end
endmodule
```

Note: When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

The following strings display as the result of executing Verilog-XL in the previous example:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

Special Characters in Strings

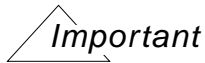
Certain characters can only be used in strings when preceded by an introductory character called an *escape character*. The following table lists these characters in the right-hand column with the escape sequence that represents the character in the left-hand column.

Specifying special characters in strings

Escape String	Character Produced by Escape String
<code>\n</code>	new line character
<code>\t</code>	tab character
<code>\\</code>	slash (<code>\</code>) character
<code>\"</code>	double quote (<code>"</code>) character
<code>\ddd</code>	a character specified in 1-3 octal digits ($0 \leq d \leq 7$)
<code>%%</code>	percent (<code>%</code>) character

Identifiers, Keywords, and System Names

An identifier is used to give an object, such as a register or a module, a name so that it can be referenced from other places in a description. An identifier is any sequence of letters, digits, dollar signs (\$), and the underscore (_) symbol.



The first character must not be a digit or \$; it can be a letter or an underscore.

Upper- and lowercase letters are considered to be different (unless the uppercase option is used when compiling). Identifiers can be up to 1024 characters long. Examples of identifiers follow:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

Escaped Identifiers

Escaped identifiers start with the backslash character (\) and provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal). An escaped identifier ends with white space (blank, tab, newline). Note that this also applies when using bit- or part-selects on the escaped identifier, in which case the bit- or part- select operator must be preceded by a space.

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier.

The primary application of escaped identifiers is for translators from other hardware description languages and CAE systems, where special characters may be allowed in identifiers; do not use escaped identifiers under normal circumstances.

Examples of escaped identifiers follow:

```
\busa+index
\clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
\p1$i14/data [2]
```

Note: Remember to terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

Keywords

Keywords are predefined non-escaped identifiers that are used to define the language constructs. A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only and therefore must be typed in lowercase in source files (unless the `-u` uppercase option is used when compiling).

See [Appendix B, “Verilog-XL Keywords,”](#) for a complete list of Verilog-XL keywords.

Text Substitutions

You can define a text macro name, assign a value to it, and use the name repetitively throughout your design. Verilog-XL substitutes the assigned value whenever it encounters the text macro name. To change the value of the text macro throughout the design, you need only modify the definition statement. Text macros are especially useful for constant values.

You can also define and use text macros in the interactive mode. For example, you can assign the value of often-used interactive commands to a text macro.

The syntax for text macro definitions is as follows:

```
<text_macro_definition>  
 ::= `define <text_macro_name> <macro_text>  
<text_macro_name>  
 ::= <IDENTIFIER>
```

The syntax for using a text macro is as follows:

```
<text_macro_usage>  
 ::= `<text_macro_name>
```

The accent grave (`'`), also called “tick”, must precede the text macro name.

You can reuse names that are used as identifiers elsewhere. For example, `signal_name` and `'signal_name` are different.

Do not use compiler directive keywords as text macro names. For example, ``define` `define` and ``define` `accelerate` are illegal because ``define` and ``accelerate` are compiler directives.

The value for `<macro_text>` is any text specified on the same line as the `<text_macro_name>`. A one-line comment (specified with the characters `//`) does not become part of the text substituted. The text for `<macro_text>` can be blank, in which case the text macro is defined to be empty and no text is substituted when the macro is used.

Verilog-XL Reference

Lexical Conventions

Once you define a text macro name, you can use it anywhere in a source description or in an interactive command; there are no scope restrictions.

The following example shows how to define and use two text macros called `wordsize` and `typ_nand`. The macro `wordsize` has a value of 8. The macro `typ_nand` has a value of `nand #5`.

```
`define wordsize 8 // assign a value of 8 to the wordsize macro
    reg [1:`wordsize] data;
    // translates to "reg [1:8] data;"
`define typ_nand nand #5 // define a nand gate with typical delay
    `typ_nand g121 (q21, n10, n11);
    // translates to "nand #5 g121 (q21, n10, n11);"
```

Do not split the text specified for `<macro_text>` across the following lexical tokens:

- comments
- numbers
- strings
- identifiers
- keywords
- double or triple character operators

For example, the following is illegal syntax in the Verilog language because it is split across a string:

```
`define first_half "start of string
$display(`first_half end of string"); // illegal syntax
```

You can redefine text macros; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

Data Types

This chapter describes the following information:

- [Overview](#) on page 31
- [Value Set](#) on page 31
- [Registers and Nets](#) on page 32
- [Vectors](#) on page 37
- [Strengths](#) on page 38
- [Implicit Declarations](#) on page 39
- [Net Initialization](#) on page 40
- [Net Types](#) on page 40
- [Memories](#) on page 46
- [Integers and Times](#) on page 47
- [Real Numbers](#) on page 48
- [Parameters](#) on page 50

Overview

The set of Verilog HDL data types is designed to represent the data storage and transmission elements found in digital hardware.

Value Set

The Verilog HDL value set consists of four basic values:

- 0 – represents a logic zero, or false condition

Verilog-XL Reference

Data Types

- 1 – represents a logic one, or true condition
- x – represents an unknown logic value
- z – represents a high-impedance state

The values 0 and 1 are logical complements of one another.

When the z value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an x value. Notable exceptions are the MOS primitives, which can pass the z value.

Almost all of the data types in the Verilog language store all four basic values. The exceptions are the `event` data type, (which has no storage), and the `triereg` net data type, (which retains its first state when all of its drivers go to the high-impedance value), and z. All bits of vectors can be independently set to one of the four basic values.

The language includes strength information in addition to the basic value information for scalar net variables. This is described in detail in [Chapter 6, “Gate and Switch Level Modeling.”](#)

Registers and Nets

There are two main groups of data types: the register data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.

Nets

The `net` data types represent physical connections between structural entities, such as gates. A `net` does not store a value (except for the `trieregnet`, discussed in [“triereg Net”](#) on page 41). Instead, it must be driven by a driver, such as a gate or a continuous assignment. See [Chapter 6, “Gate and Switch Level Modeling.”](#) and [Chapter 5, “Assignments.”](#) for definitions of these constructs. If no driver is connected to a `net`, its value will be high-impedance (z)—unless the `net` is a `triereg`.

Registers

A register is an abstraction of a data storage element. The keyword for the register data type is `reg`. A register stores a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the register. The Verilog language has powerful constructs that allow you to control when and if these assignment statements

Verilog-XL Reference

Data Types

are executed. Use these control constructs to describe hardware trigger conditions, such as the rising edge of a clock, and decision-making logic, such as a multiplexer. [Chapter 8, Behavioral Modeling](#) describes these control constructs.

The default initialization value for a `reg` data type is the unknown value, `x`.

Caution

Registers can be assigned negative values, but, when a register is an operand in an expression, its value is treated as an unsigned (positive) value. For example, a minus one in a four-bit register functions as the number 15 if the register is an expression operand. See [“Numeric Conventions in Expressions”](#) on page 54 for more information on numeric conventions in expressions.

Signed Objects

You can type any object as signed (except for user system functions) using the `signed` keyword in a type declaration (see [“Net and Register Declaration Syntax”](#) on page 35). The value of signed quantities are represented with two’s complement notation. A signed value will not cross hierarchical boundaries. If you want a signed value in other modules in a hierarchy, you must declare them in each of the modules where signed arithmetic is necessary. The following example shows some sample declarations.

```
wire signed [3:0] signed_wire; // range -8 <-> +7
reg signed [3:0] signed_reg; // range -8 <-> +7
reg signed [3:0] signed_mem [99:0] // 100 words range -8 <-> +7
function signed [3:0] signed_func; // range -8 <-> +7
```

You can type a based constant by prepending the letter `s` to the base type as shown in the following example.

```
module test;

reg signed [3:0] sig_reg;
reg [3:0] unsig_reg;

initial
begin
    $monitor($time,, "sig_reg=%d unsig_reg=%d (-4'd1)=%d (-4'sd1)=%d",
             sig_reg, unsig_reg, -4'd1, -4'sd1);
    #0 sig_reg = -4'd1;
      unsig_reg = -4'd1;
    #10 sig_reg = -4'sd1;
       unsig_reg = -4'sd1;
end

endmodule
```

The output would be as follows:

Verilog-XL Reference

Data Types

0 sig_reg= -1 unsig_reg=15 (-4'd1)=15 (-4'sd1)= -1

The following rules determine the resulting type of an expression:

- The expression type depends only on the operands. It does not depend on the left-hand side (LHS) (if any).
- Decimal numbers are signed.
- If any operand is real, the result is real.
- If all operands are signed, the result is signed, regardless of operator.
- The following list shows objects that are unsigned regardless of the operands:
 - The result of any expression where any operand is unsigned
 - Based numbers
 - Comparison results (1, 0)
 - Bit select results
 - Part select results
 - Concatenate results
- If a signed operand is to be resized to a larger signed width and the value of the sign bit is `x` or `z`, the resulting value will be a bit filled with an `x` value.
- If any nonlogical operation has a bit with a signed value of `x` or `z`, then the result is `x` for the entire value of the expression.

Nets as signed objects only have significance in an expression, in which case the entire expression is considered a signed value.

Expressions on ports are typed, sized, evaluated, and assigned to the object on the other side of the port using the same rules as expressions in assignments.

Verilog-XL uses the following steps for evaluating an expression:

1. Determine the right-hand side (RHS) type, then coerce all RHS operands to this type.
2. Determine the largest operand size, including the LHS (if any), then resize all RHS operands to this size.
3. Evaluate the RHS expression, producing a result of the type found in step 1 and the size found in step 2.
4. If there is a LHS,

Verilog-XL Reference

Data Types

- ❑ Resize the result to the LHS size.
- ❑ Coerce the result to the LHS type.

For information about arithmetic shift operators for signed objects, see [“Arithmetic Shift Operators for Signed Objects”](#) on page 60.

Net and Register Declaration Syntax

The following syntax is for net and register declarations.

```
<net_declaration>
    ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
    ||= triereg <charge_strength>? <expandrange>? <delay>? <list_of_variables> ;
    ||= <NETTYPE> <drive_strength>? <expandrange>? <delay>?
    <list_of_assignments>;
    ||= <NETTYPE> <drive_strength>? <signed_keyword>?
        <expandrange>? <delay>? <list_of_assignments> ;

<reg_declaration>
    ::= reg <range>? <list_of_register_variables> ;

<list_of_variables>
    ::= <name_of_variable> <,<name_of_variable>>*>

<name_of_variable>
    ::= <IDENTIFIER>

<list_of_register_variables>
    ::= <register_variable> <,<register_variable>>*>

<register_variable>
    ::= <name_of_register>

<name_of_register>
    ::= <IDENTIFIER>

<expandrange>
    ::= <range>
    ||= scalared <range>
    iff [the data type is not a triereg]
        the following syntax is available:
    ||= vectored <range>

<range>
    ::= [ <constant_expression> : <constant_expression> ]

<list_of_assignments>
    ::= <assignment> <,<assignment>>*>

<charge_strength>
    ::= ( <CAPACITOR_SIZE> )
```

Verilog-XL Reference

Data Types

```
<drive_strength>
    ::= ( <STRENGTH0> , <STRENGTH1> )
    ||= ( <STRENGTH1> , <STRENGTH0> )
```

The following definitions are for net declaration syntax.

<NETTYPE> is one of the following keywords:

- wire
- wand
- wor
- supply0
- supply1
- tri
- tri0
- tril
- triand
- trior
- trireg

<IDENTIFIER> is the name of the net that is being declared.

See [Chapter 2, Lexical Conventions](#) for a discussion of identifiers.

<delay> specifies the propagation delay of the net (as explained in [Chapter 6, “Gate and Switch Level Modeling.”](#)) or, when associated with a <list_of_assignments>, it specifies the delay executed before the assignment (as explained in [Chapter 5, “Assignments.”](#)).

<CAPACITOR_SIZE> is one of the following keywords:

- small
- medium
- large

<STRENGTH0> is one of the following keywords:

- supply0
- strong0

Verilog-XL Reference

Data Types

- pull0
- weak0
- highz0

<STRENGTH1> is one of the following keywords:

- supply1
- strong1
- pull1
- weak1
- highz1

Declaration Examples

The following are examples of register and net declarations:

Register and net declarations

```
reg a;                // a scalar register
wand w;              // a scalar net of type 'wand'
reg[3:0] v;           // a 4-bit vector register made up of
                    // (from most to least significant):
                    // v[3], v[2], v[1] and v[0]

tri [15:0] busa;      // a tri-state 16-bit bus
reg [1:4] b;          // a 4-bit vector register
reg signed [0:3] signed_reg; // 4-bit signed register with a range of -8 to +7
reg signed [0:3] signed_mem [99:0] // 100 words with a range of -8 to +7
triereg (small) storeit; // a charge storage node of strength small
```

If a set of nets or registers shares the same characteristics, you can declare them in the same declaration statement. The following is an example:

```
wire w1, w2;         // declares 2 wires
reg [4:0] x, y, z;   // declares 3 5-bit registers
```

Vectors

A net or reg declaration without a <range> specification is one bit wide; that is, it is scalar. Multiple bit net and reg data types are declared by specifying a <range>, and are known as vectors.

Specifying Vectors

The *<range>* specification gives addresses to the individual bits in a multi-bit net or register. The most significant bit (msb) is the left-hand value in the *<range>* and the least significant bit (lsb) is the right-hand value in the *<range>*.

The range is specified as follows:

```
[ <msb_expr> : <lsb_expr> ]
```

Both *<msb_expr>* and *<lsb_expr>* are non-negative constant expressions. There are no restrictions on the values of the indices. The msb and lsb expressions can be any value, and *<lsb_expr>* can be a greater value than *<msb_expr>*, if desired.

Vector nets and registers obey laws of arithmetic modulo 2 to the power *n*, where *n* is the number of bits in the vector. Vector nets and registers are treated as unsigned quantities.

Vector Net Accessibility

A vector net can be used as a single entity or as a group of *n* scalars, where *n* is the number of bits in the vector net. The keyword `vector`d allows you to specify that a vector net can be modified only as an indivisible entity. The keyword `scalar`d explicitly allows access to bit and parts. The Verilog-XL process of accessing bits within a vector is known as vector expansion. Declaring a net with neither the `scalar`d nor the `vector`d keyword makes a net that Verilog-XL treats as an indivisible entity unless the simulation requires an expanded net.

Only when a net is not specified as `vector`d can bit selects and part selects be driven by outputs of gates, primitives, and modules—or be on the left-hand side of continuous assignments. You cannot declare a trireg with the `vector`d keyword.

The following are examples of vector net declarations:

```
tril scalar [63:0] bus64;      //a bus that will be expanded
tri vector [31:0] data;       //a bus that will not be expanded
```

Note: The keywords `scalar`d and `vector`d apply only to vector nets and do not apply to vector registers.

Strengths

There are two types of strengths that can be specified in a net declaration. They are as follows:

- charge strength—used when declaring a net of type trireg

- drive strength—used when placing a continuous assignment on a net in the same statement that declares the net

Gate declarations can also specify a drive strength. See [Chapter 6, “Gate and Switch Level Modeling.”](#) for more information on gates and for important information on strengths.

Charge Strength

The `<charge_strength>` specification can be used only with `triereg` nets. A `triereg` net is used to model charge storage; `<charge_strength>` specifies the relative size of the capacitance. The `<CAPACITOR_SIZE>` declaration is one of the following keywords:

- `small`
- `medium`
- `large`

When no size is specified in a `triereg` declaration, its size is `medium`.

The following is a syntax example of a strength declaration:

```
triereg (small) st1 ;
```

A `triereg` net can model a charge storage node whose charge decays over time. The simulation time of a charge decay is specified in the `triereg` net’s delay specification, discussed in [“triereg Net Charge Decay”](#) on page 137.

Drive Strength

The `<drive_strength>` specification allows a continuous assignment to be placed on a net in the same statement that declares that net. See [Chapter 5, Assignments](#) for more details.

Net strength properties are described in detail in [Chapter 6, Gate and Switch Level Modeling.](#)

Implicit Declarations

The syntax shown in [“Net and Register Declaration Syntax”](#) on page 35 is used to explicitly declare variables. In the absence of an explicit declaration of a variable, statements for gate, user-defined primitive, and module instantiations assume an implicit variable declaration. This happens if you specify a variable that has not been explicitly declared previously in one of the declaration statements of the instantiating module in the terminal list of an instance of a gate, a user-defined primitive, or a module.

These implicitly declared variables are scalar nets of type `wire`.

Net Initialization

The default initialization value for a net is the value `z`. Nets with drivers assume the output value of their drivers, which defaults to `x`. The `triereg` net is an exception to these statements. The `triereg` defaults to the value `x`, with the strength specified in the net declaration (`small`, `medium`, or `large`).

Net Types

There are several distinct types of nets. Each is described in the sections that follow.

wire and tri Nets

The `wire` and `tri` nets connect elements. The net types `wire` and `tri` are identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A `wire` net is typically used for nets that are driven by a single gate or continuous assignment. The `tri` net type may be used where multiple drivers drive a net.

Logical conflicts from multiple sources on a `wire` or a `tri` net result in unknown values unless the net is controlled by logic strength.

The following is a truth table for `wire` and `tri` nets. Note that it assumes equal strengths for both drivers. Please refer to [“Logic Strength Modeling”](#) on page 114 for a discussion of logic strength modeling.

Truth table for wire and tri nets

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Wired Nets

Wired nets are of type `wor`, `wand`, `trior`, and `triand`, and are used to model wired logic configurations. Wired nets resolve the conflicts that result when multiple drivers drive the same net. The `wor` and `trior` nets create wired `or` configurations, such that when any of the drivers is 1, the net is 1. The `wand` and `triand` nets create wired `and` configurations, such that if any driver is 0, the net is 0.

The net types `wor` and `trior` are identical in their syntax and functionality—as are the `wand` and `triand`. The following figure gives the truth tables for wired nets. Note that it assumes equal strengths for both drivers. Please refer to [“Logic Strength Modeling”](#) on page 114 for a discussion of logic strength modeling.

Truth table for `wand/triand` nets

<code>wand/triand</code>	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Truth table for `wor/trior` nets

<code>wor/trior</code>	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

`triereg` Net

The `triereg` net stores a value and is used to model charge storage nodes. A `triereg` can be one of two states:

- **The Driven State**—When at least one driver of a `triereg` has a value of 1, 0, or x, that value propagates into the `triereg` and is the driven value of a `triereg`.

Verilog-XL Reference

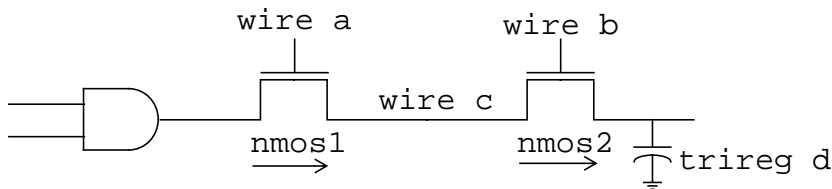
Data Types

- **Capacitive State**—When all the drivers of a `triereg` net are at the high-impedance value (`z`), the `triereg` net retains its last driven value; the high-impedance value does not propagate from the driver to the `triereg`.

The strength of the value on the `triereg` net in the capacitive state is `small`, `medium`, or `large`, depending on the size specified in the declaration of the `triereg`. The strength of a `triereg` in the driven state is `strong`, `pull`, or `weak` depending on the strength of the driver. You cannot declare a `triereg` with the `vectored` keyword.

The following figure shows a schematic that includes the following items: a `triereg` net whose size is `medium`, its driver, and the simulation results.

Simulation values of a `triereg` and its driver



simulation time	wire a	wire b	wire c	triereg d
0	1	1	strong 1	strong 1
10	0	1	HiZ	medium 1

Simulation of the design in this figure reports the following results:

1. At simulation time 0, `wire a` and `wire b` have a value of 1. A value of 1 with a `strong` strength propagates from the AND gate through the NMOS switches connected to each other by `wire c`, into `triereg d`.
2. At simulation time 10, `wire a` changes value to 0, disconnecting `wire c` from the AND gate. When `wire c` is no longer connected to the AND gate, its value changes to `HiZ`. The value of `wire b` remains 1 so `wire c` remains connected to `triereg d` through the NMOS2 switch. The `HiZ` value does not propagate from `wire c` into `triereg d`. Instead, `triereg d` enters the capacitive state, storing its last driven value of 1 with a `medium` strength.

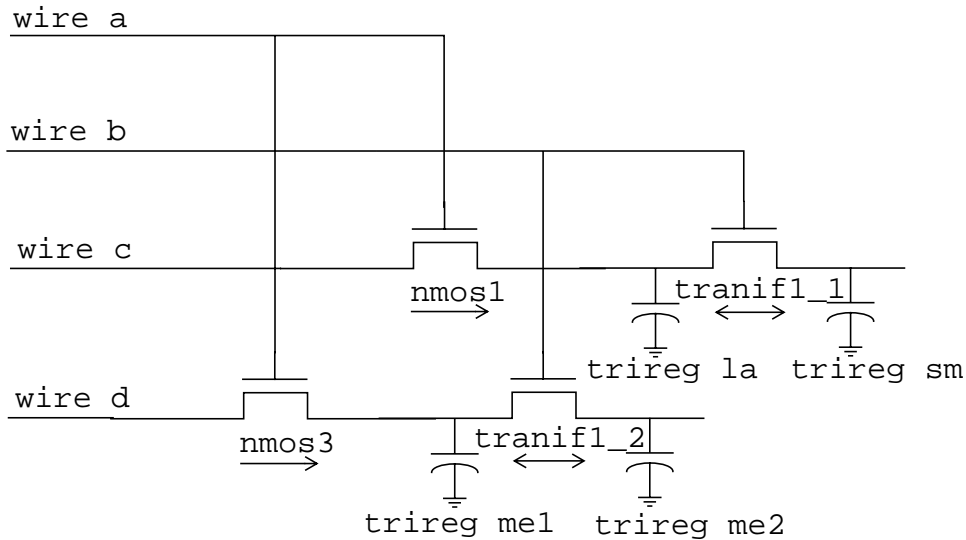
Capacitive networks

A capacitive network is a connection between more than one `triereg`. In a capacitive network where more than one `triereg` are in the capacitive state, logic and strength values can propagate between `triereg`s. The following figure shows a capacitive network in which the

Verilog-XL Reference Data Types

logic value of some `triregs` change the logic value of other `triregs` of equal or smaller size.

Simulation results of a capacitive network



simulation time	wire a	wire b	wire c	wire d	trireg la	trireg sm	trireg me1	trireg me2
0	1	1	1	1	1	1	1	1
10	1	0	1	1	1	1	1	1
20	1	0	0	1	0	1	1	1
30	1	0	0	0	0	1	0	1
40	0	0	0	0	0	1	0	1
50	0	1	0	0	0	0	x	x

In [Simulation results of a capacitive network](#) figure on page 43, the size of `trireg la` is large, the size of `triregs me1` and `me2` are medium, and the size of `trireg sm` is small. Simulation reports the following sequence of events:

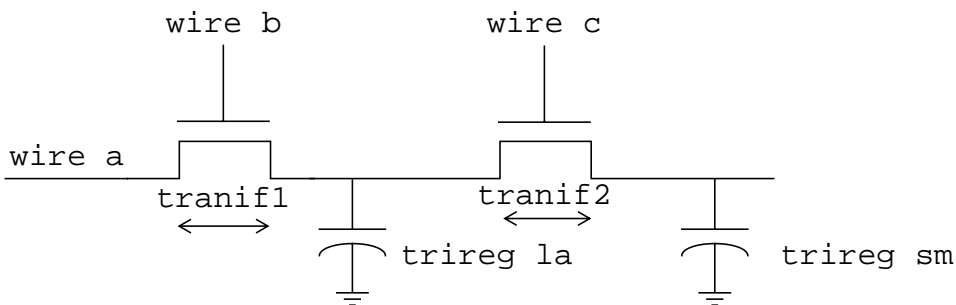
1. At simulation time 0, wire a and wire b have a value of 1. The wire c drives a value of 1 into `triregs la` and `sm`, wire d drives a value of 1 into `triregs me1` and `me2`.
2. At simulation time 10, the value of wire b changes to 0, disconnecting `trireg sm` and `me2` from their drivers. These `triregs` enter the capacitive state and store the value 1; their last driven value.
3. At simulation time 20, wire c drives a value of 0 into `trireg la`.

Verilog-XL Reference Data Types

4. At simulation time 30, wire `d` drives a value of 0 into `triereg me1`.
5. At simulation time 40, the value of wire `a` changes to 0, disconnecting `triereg la` and `me1` from their drivers. These `triereg`s enter the capacitive state and store the value 0.
6. At simulation time 50, the value of wire `b` changes to 1. This change of value in wire `b` connects `triereg sm` to `triereg la`; these `triereg`s have different sizes and stored different values. This connection causes the smaller `triereg` to store the larger `triereg` value and `triereg sm` now stores a value of 0. This change of value in wire `b` also connects `triereg me1` to `triereg me2`; these `triereg`s have the same size and stored different values. The connection causes both `triereg me1` and `me2` to change value to `x`.

In a capacitive network, charge strengths propagate from a larger `triereg` to a smaller `triereg`. “Simulation results of charge sharing” on page 44 shows a capacitive network and its simulation results.

Simulation results of charge sharing



simulation time	wire a	wire b	wire c	triereg la	triereg sm
0	strong 1	1	1	strong 1	strong 1
10	strong 1	0	1	large 1	large 1
20	strong 1	0	0	large 1	small 1
30	strong 1	0	1	large 1	large 1
40	strong 1	0	0	large 1	small 1

In this figure, the size of `triereg la` is large and the size of `triereg sm` is small. Simulation reports the following results:

Verilog-XL Reference

Data Types

1. At simulation time 0, the value of wire `a`, `b`, and `c` is 1 and wire `a` drives a strong 1 into `triereg la` and `sm`.
2. At simulation time 10, the value of wire `b` changes to 0, disconnecting `triereg la` and `sm` from wire `a`. The `triereg la` and `sm` enter the capacitive state. Both `triereg la` and `sm` share the large charge of `triereg la` because they remain connected through `tranif2`.
3. At simulation time 20, the value of wire `c` changes to 0, disconnecting `triereg sm` from `triereg la`. The `triereg sm` no longer shares the large charge of `triereg la` and now stores a small charge.
4. At simulation time 30, the value of wire `c` changes to 1, connecting the two `triereg`s. These `triereg`s now share the same charge.
5. At simulation time 40, the value of wire `c` changes again to 0, disconnecting `triereg sm` from `triereg la`. Once again, `triereg sm` no longer shares the large value of `triereg la` and now stores a small charge.

Ideal capacitive state and charge decay

A `triereg` net can retain its value indefinitely or its charge can decay over time. The simulation time of charge decay is specified in the `triereg` net's delay specification.

`tri0` and `tri1` Nets

The `tri0` and `tri1` nets model nets with resistive `pulldown` and resistive `pullup` devices on them. When no driver drives a `tri0` net, its value is 0. When no driver drives a `tri1` net, its value is 1. The strength of this value is `pull`. See [Chapter 6, "Gate and Switch Level Modeling,"](#) for a description of strength modeling.

Supply Nets

The `supply0` and `supply1` nets model the power supplies in a circuit. The `supply0` nets are used to model `Vss` (ground) and `supply1` nets are used to model `Vdd` or `Vcc` (power). These nets should never be connected to the output of a gate or continuous assignment, because the strength they possess will override the driver. They have `supply0` or `supply1` strengths.

Memories

The Verilog HDL models memories as an array of register variables. You can use these arrays to model read-only memories (ROMs), random access memories (RAMs), and register files. Each register in the array is known as an *element* or *word* and is addressed by a single array index. There are no multiple dimension arrays in the Verilog language.

Memories are declared in register declaration statements by specifying the element address range after the declared identifier.

The following example gives the syntax for a register declaration statement. Note that this syntax extends the `<register_variable>` definition given in “[Net and Register Declaration Syntax](#)” on page 35.

```
<register_variable>
  ::= <name_of_register> <signed_keyword>?
     || = <name_of_memory>
     [ <constant_expression> : <constant_expression> ]

<constant_expression>
  ::= <expression>

<name_of_memory>
  ::= <IDENTIFIER>
```

The following example illustrates a memory declaration:

```
reg[7:0] mema[0:255];
```

This example declares a memory called `mema` consisting of 256 eight-bit registers. The indices are 0 through 255. The expressions that specify the indices of the array must be constant expressions.

Note that you can declare both registers and memories within the same declaration statement. This makes it convenient to declare both a memory and some registers that will hold data to be read from and written to the memory in the same declaration statement, as in the following example.

```
parameter                               // parameters are run-time constants - see Parameters
wordsize = 16,
memsize = 256;

// Declare 256 words of 16-bit memory plus two registers
reg [wordsize-1:0]           // equivalent to [15:0]
  mem [memsize-1:0],       // equivalent to [255:0]
  writereg,
  readreg;
```

Note that a memory of n 1-bit registers is different from an n -bit vector register, as shown in the following example:

Verilog-XL Reference

Data Types

```
reg [1:n] rega;           // an n-bit register is not the same
reg mema [1:n];         // as a memory of 1-bit registers
```

An n -bit register can be assigned a value in a single assignment, but a complete memory cannot; thus the following assignment to `rega` is legal and the succeeding assignment that attempts to clear all of the memory `mema` is illegal:

```
rega = 0;                // legal syntax
mema = 0;                //illegal syntax
```

To assign a value to a memory element, you must specify an index as shown in the following example:

```
mema[1] = 0;             //assign 0 to the first element of mema
```

The index can be an expression. This option allows you to reference different memory elements, depending on the value of other registers and nets in the circuit. For example, you can use a program counter register to index into a RAM.

Integers and Times

In addition to modeling hardware, there are other uses for variables in an HDL model. Although you can use the `reg` variables for general purposes such as counting the number of times a particular net changes value, the `integer` and `time` register data types are provided for convenience and to make the description more self-documenting.

The syntax for declaring `integer` and `time` variables is as follows:

```
<time_declaration>
    ::= time <list_of_register_variables> ;

<integer_declaration>
    ::= integer <list_of_register_variables> ;
```

The `<list_of_register_variables>` item is defined in “[Net and Register Declaration Syntax](#)” on page 35.

Use a `time` variable for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. You use this data type typically in conjunction with the `$time` system function. The size of a `time` variable is 64 bits.

Use an `integer` as a general purpose variable for manipulating quantities that are not regarded as hardware registers. The size of an `integer` variable is 32 bits.

You can use arrays of `integer` and `time` variables. They are declared in the same manner as arrays of `reg` variables, as in the following example:

Verilog-XL Reference

Data Types

```
integer a[1:64];           // an array of 64 integers
time change_history[1:1000]; // an array of 1000 times
```

Assign values to the `integer` and `time` variables the same manner as `reg` variables. Use procedural assignments to trigger their value changes.

`Time` variables behave the same as 64 bit `reg` variables. They are unsigned quantities, and unsigned arithmetic is performed on them. In contrast, `integer` variables are signed quantities. Arithmetic operations performed on `integer` variables produce 2's complement results.

Real Numbers

The Verilog HDL supports real number constants and variables in addition to integers and time variables. The syntax for real numbers is the same as the syntax for register types, and is described in [“Real Number Declaration Syntax”](#) on page 48.

Except for the following restrictions, you can use real number variables in the same places that integers and time variables are used.

- Not all Verilog HDL operators can be used with real number values. See the tables in [“Operators”](#) on page 52 for lists of valid and invalid operators for real numbers.
- Ranges are not allowed on real number variable declarations.
- Real number variables default to an initial value of zero.

Real Number Declaration Syntax

The syntax for declaring real number variables is as follows:

Syntax for real number variable declarations

```
<real_declaration>
    ::=real<list_of_variables>;
```

The `<list_of_variables>` item is defined in [“Net and Register Declaration Syntax”](#) on page 35.

Specifying Real Numbers

You can specify real numbers in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the 8th power). Real

Verilog-XL Reference

Data Types

numbers expressed with a decimal point must have at least one digit on each side of the decimal point.

The following are some examples of valid real numbers in the Verilog language:

```
1.2
0.1
2394.26331
1.2E12 (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12 (underscores are ignored)
```

The following are invalid real numbers in the Verilog HDL because they do not have a digit to the left of the decimal point:

```
.12
.3E3
.2e-7
```

Operators and Real Numbers

The result of using logical or relational operators on real numbers is a single-bit scalar value. Not all Verilog operators can be used with real number expressions. “Operators” on page 52 lists the valid operators for use with real numbers.

Real number constants and real number variables are also prohibited in the following contexts:

- edge descriptors (`posedge`, `negedge`) applied to real number variables
- bit-select or part-select references of variables declared as `real`
- real number index expressions of bit-select or part-select references of vectors
- real number memories (arrays of real numbers)

Conversion

The Verilog language converts real numbers to integers by rounding a real number to the nearest integer, rather than by truncating it. For example, the real numbers 35.7 and 35.5 both become 36 when converted to an integer, and 35.2 becomes 35. Implicit conversion takes place when you assign a real number to an integer.

Parameters

Verilog parameters do not belong to either the register or the net group. Parameters are not variables, they are constants. The syntax for parameter declarations is as follows:

```
<parameter_declaration>  
 ::= parameter <list_of_assignments> ;
```

The *<list_of_assignments>* is a comma-separated list of assignments, where the right-hand side of the assignment must be a constant expression, that is, an expression containing only constant numbers and previously defined parameters. The following shows examples of parameter declarations:

```
parameter msb = 7; // defines msb as a constant value 7  
parameter e = 25, f = 9; // defines two constant numbers  
parameter average_delay = (r + f) / 2;  
parameter byte_size = 8, byte_mask = byte_size - 1;  
parameter r = 5.7; // declares r as a 'real' parameter  
parameter [15:0] p = 'hedle; // Parameter with a bit range  
parameter strparm = "Hello world" // Converts "Hello world" to  
 // 48656c6c6f20776f726c64
```

Note: In the previous example, the last parameter assignment converts the ASCII characters (Hello world) to their hexadecimal integer notation (48656c6c6f20776f726c64) and does not store an actual text string. For information about strings in Verilog-XL, see [“Strings”](#) on page 26.

Even though they represent constants, you can modify Verilog parameters at compilation time to have values that are different from those specified in the declaration assignment. This allows you to customize module instances. You can modify the parameter with the `defparam` statement, or you can modify the parameter in the module instance statement. Typical uses of parameters are to specify delays and width of variables.

You can access bits and parts of parameters declared in this way and use them in assignments and logic operations. The following line shows an assignment of a parameter part-select to a register:

```
preg = p[11:8];
```

Do not attempt to write to parameters after time zero.

See [Chapter 11, “Hierarchical Structures”](#) for more details on parameter value assignment.

Expressions

This chapter describes the following:

- [Overview](#) on page 51
- [Operators](#) on page 52
- [Operands](#) on page 62
- [Minimum, Typical, Maximum Delay Expressions](#) on page 66

Overview

An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Alternatively, an expression is any legal operand—for example, a net bit-select. Wherever a value is needed in a Verilog HDL statement, an expression can be given. However, several statement constructs limit an expression to a constant expression. A constant expression consists of constant numbers and predefined parameter names only, but can use any of the operators defined in “[Operators](#)” on page 52.

For their use in expressions, `integer` and `time` data types share the same traits as the data type `reg`. Descriptions pertaining to register usage apply to integers and times as well.

An operand can be one of the following:

- number (including `real`)
- net
- register, integer, time
- net bit-select
- register bit-select
- net part-select

Verilog-XL Reference Expressions

- register part-select
- memory element
- a call to a user-defined function or system-defined function that returns any of the above

Operators

The symbols shown in [Table 4-1](#) on page 52 are Verilog HDL operators, which are similar to those in the C programming language. Operators that are valid for real expressions are marked Y in the third column.

Table 4-1 Operators

Operators	Type	Reals
{}	concatenation	N
+ - * /	arithmetic	Y
%	modulus	N
> >= < <=	relational	Y
!	logical negation	Y
&&	logical and	Y
	logical or	Y
==	logical equality	Y
!=	logical inequality	Y
===	case equality	N
!==	case inequality	N
~	bit-wise negation	N
&	bit-wise and	N
	bit-wise inclusive or	N
^	bit-wise exclusive or	N
^~ or ~^	bit-wise equivalence	N
&	reduction and	N
~&	reduction nand	N

Verilog-XL Reference Expressions

Table 4-1 Operators, *continued*

Operators	Type	Reals
	reduction or	N
~	reduction nor	N
^	reduction xor	N
~^ or ^~	reduction xnor	N
<<	shift left	N
>>	shift right	N
<<<	arithmetic shift left	N
>>>	arithmetic shift right	N
? :	conditional	Y

The result of using logical or relational operators on real numbers is a single-bit scalar value. See [“Real Numbers”](#) on page 48 for more information on use of real numbers.

Binary Operator Precedence

The precedence order of binary operators (and the ternary operator ? :) is the same as the precedence order for the matching operators in the C programming language. Verilog has two equality operators not present in C; they are discussed in [“Equality Operators”](#) on page 56. The following example summarizes the precedence rules for Verilog’s binary and ternary operators.

Precedence rules for operators

```

!      ~      %      highest precedence
*      /
+      -
<<     >>     <<<     >>>
<      <=    >      >=
==     !=     ===    !==
&
^      ^~
|
&&
||
?:     (ternary operator)    lowest precedence

```

Verilog-XL Reference

Expressions

Operators on the same line of the previous list have the same precedence. Rows are in order of decreasing precedence, so, for example, `*`, `/`, and `%` all have the same precedence, which is higher than that of the binary `+` and `-` operators.

All operators associate left to right. Associativity refers to the order in which a language evaluates operators having the same precedence. Thus, in the following example, `B` is added to `A` and then `C` is subtracted from the result of `A+B`.

```
A + B - C
```

When operators differ in precedence, the operators with higher precedence apply first. In the following example, `B` is divided by `C` (division has higher precedence than addition) and then the result is added to `A`.

```
A + B / C
```

Parentheses can change the operator precedence.

```
(A + B) / C          // not the same as A + B / C
```

Numeric Conventions in Expressions

Operands can be expressed as based and sized numbers—with the following restriction: The Verilog language interprets a number of the form `sss 'f nnn`, *when used directly in an expression*, as the *unsigned* number represented by the two's complement of `nnn`. The following example shows two ways to write the expression “minus 12 divided by 3.” Note that `-12` and `-d12` both evaluate to the same bit pattern, but in an expression `-d12` loses its identity as a signed, negative number.

```
integer IntA;
IntA = -12 / 3;      // The result is -4.
IntA = -'d 12 / 3;  // The result is 1431655761
```

Arithmetic Operators

The binary arithmetic operators are the following:

```
+   -   *   /   % (the modulus operator)
```

The unary arithmetic operators take precedence over the binary operators. The unary operators are the plus (+) and minus (-) signs.

For the arithmetic operators, if any operand bit value is the unknown value `x`, then the entire result value is `x`.

Integer division truncates any fractional part.

Verilog-XL Reference

Expressions

The modulus operator—for example, $y \% z$, gives the remainder when the first operand is divided by the second, and thus is zero when z divides y exactly. The result of a modulus operation takes the sign of the first operand. [Table 4-1](#) on page 55 gives examples of modulus operations.

Table 4-2 Examples of Modulus Operator

Modulus Expression	Result	Comments
<code>10 % 3</code>	1	10/3 yields a remainder of 1
<code>11 % 3</code>	2	11/3 yields a remainder of 2
<code>12 % 3</code>	0	12/3 yields no remainder
<code>-10 % 3</code>	-1	the result takes the sign of the first operand
<code>11 % -3</code>	2	the result takes the sign of the first operand
<code>-4'd12 % 3</code>	1	-4'd12 is seen as a large, positive number that leaves a remainder of 1 when divided by 3

Arithmetic Expressions with Registers and Integers

An arithmetic operation on a register data type behaves differently than an arithmetic operation on an integer data type.

The Verilog language sees a register data type as an unsigned value and an integer data type as a signed value. As a result, when you assign a value of the form `-<size><base_format><number>` to a register and then use that register as an expression operand, you are actually using a positive number that is the two's complement of `nnn`. In contrast, when you assign a value of the form `-<size><base_format><number>` to an integer and then use that integer as an expression operand, the expression evaluates using signed arithmetic. The following example shows various ways to divide minus twelve by three using integer and register data types in expressions.

```
integer intA;
reg [15:0] regA;
intA = -4'd12;
regA = intA / 3;           // Result is 65532, which is the bit pattern
                          // for 16 bit -4 assigned to an unsigned register
regA = -4'd12;           // Result is 65524 because regA is unsigned
intA = regA / 3;

intA = -4'd12 / 3;       // Result is 1431655761 because it is
                          // evaluated to 32 bits
regA = -12 / 3;          // Result is 65532 because regA is unsigned.
```

Relational Operators

The following examples define the relational operators.

```
a < b      // a less than b
a > b      // a greater than b
a <= b     // a less than or equal to b
a >= b     // a greater than or equal to b
```

The relational operators all yield the scalar value 0 if the specified relation is false, or the value 1 if the specified relation is true. If, due to unknown bits in the operands, the relation is ambiguous, then the result is the unknown value (x).

Note: If Verilog-XL tests a value that is x or z, then the result of that test is false.

All the relational operators have the same precedence. Relational operators have lower precedence than arithmetic operators. The following examples illustrate the implications of this precedence rule:

```
a < size - 1          // this construct is the same as
a < (size - 1)       // this construct, but . . .
size - (1 < a)       // this construct is not the same
size - 1 < a         // as this construct
```

Note that when `size - (1 < a)` evaluates, the relational expression evaluates first and then either zero or one is subtracted from `size`.

When `size - 1 < a` evaluates, the `size` operand is reduced by one and then compared with `a`.

Equality Operators

The equality operators rank just lower in precedence than the relational operators. The following examples list and define the equality operators.

```
a === b    // a equal to b, including x and z
a !== b    // a not equal to b, including x and z
a == b     // a equal to b, result may be unknown
a != b     // a not equal to b, result may be unknown
```

All four equality operators have the same precedence. These four operators compare operands bit for bit, with zero filling if the two operands are of unequal bit-length. As with the relational operators, the result is 0 if false, 1 if true.

For the `==` and `!=` operators, if either operand contains an x or a z, then the result is the unknown value (x).

Verilog-XL Reference

Expressions

For the `===` and `!==` operators, the comparison is done just as it is in the procedural case statement. Bits that are `x` or `z` are included in the comparison and must match for the result to be true. The result of these operators is always a known value, either 1 or 0.

Logical Operators

This section describes logical operators.

Logical AND and Logical OR Operators

The operators logical AND (`&&`) and logical OR (`||`) are logical connectives. Expressions connected by `&&` or `||` are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known. The result of the evaluation of a logical comparison is one (defined as *true*), zero (defined as *false*), or, the unknown value (`x`) if either operand is ambiguous.

For example, if register `alpha` holds the integer value 237 and `beta` holds the value zero, then the following examples perform as described:

```
regA = alpha && beta;           // regA is set to 0
regB = alpha || beta;         // regB is set to 1
```

The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators. The following expression ANDs three sub-expressions without needing any parentheses:

```
a < size-1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of the above example:

```
(a < size-1) && (b != c) && (index != lastone)
```

Logical Negation Operator

A third logical operator is the unary logical negation operator `!`. The negation operator converts a non-zero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value remains as `x`. A common use of `!` is in constructions such as the following:

```
if (!inword)
```

In some cases, the preceding construct makes more sense to someone reading the code than the equivalent construct shown below:

```
if (inword == 0)
```

Verilog-XL Reference Expressions

Constructions like `if (!inword)` can be read easily (“if not inword”), but more complicated ones can be hard to understand. The first form is slightly more efficient in simulation speed than the second.

Bit-Wise Operators

The bit operators perform bit-wise manipulations on the operands—that is, the operator compares a bit in one operand to its equivalent bit in the other operand to calculate one bit for the result.

Be sure to distinguish the bit-wise operators `&` and `|` from the logical operators `&&` and `||`. For example, if `x` is 1 and `y` is 2, then `x & y` is 0, while `x && y` is 1. When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

The following logic tables show the results for each possible calculation.

Bit-wise operators logic tables

bit-wise unary negation		bit-wise binary AND operator					bit-wise binary inclusive OR operator			
~		&	0	1	x		0	1	x	
0	1	0	0	0	0	0	0	1	x	
1	0	1	0	1	x	1	1	1	1	
x	x	x	0	x	x	x	x	1	x	

bit-wise binary exclusive OR operator				bitwise binary exclusive NOR operator			
^	0	1	x	^~	0	1	x
0	0	1	x	0	1	0	x
1	1	0	x	1	0	1	x
x	x	x	x	x	x	x	x

Verilog-XL Reference Expressions

Reduction Operators

The unary reduction operators perform a bit-wise operation on a single operand to produce a single bit result. The first step of the operation applies the operator between the first bit of the operand and the second—using the following logic tables. The second and subsequent steps apply the operator between the one-bit result of the prior step and the next bit of the operand—still using the same logic table.

Reduction Operators logic tables

reduction unary AND operator				reduction unary inclusive OR operator				reduction unary exclusive OR operator			
&	0	1	x		0	1	x	^	0	1	x
0	0	0	0	0	0	1	x	0	0	1	x
1	0	1	x	1	1	1	1	1	1	0	x
x	0	x	x	x	x	1	x	x	x	x	x

Note that the reduction unary NAND and reduction unary NOR operators operate the same as the reduction unary AND and OR operators, respectively, but with their outputs negated. The effective results produced by the unary reduction operators are listed in the following two tables.

Results of AND, OR, NAND, and NOR unary reduction operations

Operand	&		~&	~
no bits set	0	0	1	1
all bits set	1	1	0	0
some bits set but not all	0	1	1	0

Results of Exclusive OR and exclusive NOR unary reduction operations

Operand	^	~^
odd number of bits set	1	0
even number of bits set (or none)	0	1

Syntax Restrictions

The Verilog language imposes two syntax restrictions intended to protect description files from a typographical error that is particularly hard to find. The error consists of transposing a space and a symbol. Note that the constructs on line 1 below do *not* represent the same syntax as the similar constructs on line 2.

```
1.   a & &b   a | |b
2.   a && b   a | | b
```

To protect users from this type of error, Verilog requires the use of parentheses to separate a reduction `or` or `and` operator from a bit-wise `or` or `and` operator.

The following table shows the syntax that requires parentheses:

Syntax equivalents for syntax restriction

Invalid Syntax	Equivalent Syntax
<code>a & &b</code>	<code>a & (&b)</code>
<code>a b</code>	<code>a (b)</code>

Shift Operators

The shift operators, `<<` and `>>`, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators fill the vacated bit positions with zeroes. The following example illustrates this concept.

```
module shift;
    reg [3:0] start, result;
    initial
        begin
            start = 1;           // Start is set to 0001
            result = (start << 2); // Result is 0100
        end
endmodule
```

In this example, the register `result` is assigned the binary value `0100`, which is `0001` shifted to the left two positions and zero filled.

Arithmetic Shift Operators for Signed Objects

The arithmetic shift operators (`<<<` and `>>>`) work the same as regular shift operators on unsigned objects. However, when used on *signed* objects, the following rules apply:

Verilog-XL Reference

Expressions

- Arithmetic shift left ignores the signed bit and shifts bit values to the left (like a regular shift left operator), filling the open bits with zeroes.
- Arithmetic shift right propagates all bits, including the signed bit, to the right while maintaining the signed bit value.

The following example illustrates these concepts.

```
module shift;
  reg signed [3:0] start, result;
  initial
  begin
    start = -1;           // Start is 1111
    result = (start <<< 2); // Result is 1100
    result = (result <<< 1); // Result is 1000
    start = 5;           // Start is 0101
    result = (start <<< 2); // Result is 0100
    start = -3;          // Start is 1101
    result = (start >>> 1); // Result is 1110
    result = (result >>> 1); // Result is 1111
    result = (result >>> 1); // Result is 1111
    start = 3;           // Start is 0011
    result = (start >>> 1); // Result is 0001
    result = (result >>> 1); // Result is 0000
  end
endmodule
```

Conditional Operator

The conditional operator has three operands separated by two operators in the following format:

```
<cond_expr> ? <true_expr> : <>false_expr>
```

If *<cond_expr>* evaluates to false, then *<>false_expr>* is evaluated and used as the result. If the conditional expression is true, then *<true_expr>* is evaluated and used as the result. If *<cond_expr>* is ambiguous, then both *<true_expr>* and *<>false_expr>* are evaluated and their results are compared, bit by bit, using the following table to calculate the final result. If the lengths of the operands are different, the shorter operand is lengthened to match the longer and zero filled from the left (the high-order end).

Conditional operator ambiguous condition results

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Verilog-XL Reference

Expressions

The following example of a tri-state output bus illustrates a common use of the conditional operator.

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

The bus called `data` is driven onto `busa` when `drive_busa` is 1. If `drive_busa` is unknown, then an unknown value is driven onto `busa`. Otherwise, `busa` is not driven.

Concatenations

A concatenation is the joining together of bits resulting from two or more expressions. The concatenation is expressed using the brace characters `{` and `}`, with commas separating the expressions within. The next example concatenates four expressions:

```
{a, b[3:0], w, 3'b101}
```

The previous example is equivalent to the following example:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

Unsize constant numbers are not allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

Concatenations can be expressed using a repetition multiplier as shown in the next example.

```
{4{w}} // This is equivalent to {w, w, w, w}
```

The next example illustrates nested concatenations.

```
{b, {3{a, b}}} // This is equivalent to  
// {b, a, b, a, b, a, b}
```

The repetition multiplier must be a constant expression.

Operands

As stated before, there are several types of operands that can be specified in expressions. The simplest type is a reference to a net or register in its complete form—that is, just the name of the net or register is given. In this case, all of the bits making up the net or register value are used as the operand.

If just a single bit of a vector net or register is required, then a bit-select operand is used. A part-select operand is used to reference a group of adjacent bits in a vector net or register.

A memory element can be referenced as an operand.

Verilog-XL Reference

Expressions

A concatenation of other operands (including nested concatenations) can be specified as an operand.

A function call is an operand.

Net and Register Bit Addressing

Bit-selects extract a particular bit from a vector net or register. The bit can be addressed using an expression. The next example specifies the single bit of `acc` that is addressed by the operand `index`.

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of `acc`. For instance, each of the declarations of `acc` shown in the next example causes a particular value of `index` to access a *different* bit:

```
reg [15:0] acc;  
reg [1:16] acc;
```

If the bit select is out of the address bounds or is `x`, then the value returned by the reference is `x`.

Several contiguous bits in a vector register or net can be addressed, and are known as *part-selects*. A part-select of a vector register or net is given with the following syntax:

```
vect[ms_expr:ls_expr]
```

Both expressions must be constant expressions. The first expression must address a more significant bit than the second expression. Compiler errors result if either of these rules is broken.

The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit register called `vect` and initializes it to a value of 4. The bullet items describe how the separate bits of that vector can be addressed.

```
reg [7:0] vect;  
vect = 4;
```

- if the value of `addr` is 2, then `vect[addr]` returns 1
- if the value of `addr` is out of bounds, then `vect[addr]` returns `x`
- if `addr` is 0, 1, or 3 through 7, `vect[addr]` returns 0
- `vect[3:0]` returns the bits 0100
- `vect[5:1]` returns the bits 00010
- `vect[<expression that returns x>]` returns `x`

Verilog-XL Reference

Expressions

- `vect [<expression that returns z>]` returns `x`
- if any bit of `addr` is `x/z`, then the value of `addr` is `x`

Memory Addressing

This section discusses memory addressing. “[Memories](#)” on page 46 discussed the declaration of memories. The next example declares a memory of 1024 8-bit words:

```
reg [7:0] mem_name[0:1023];
```

The syntax for a memory address consists of the name of the memory and an expression for the address—specified with the following format:

```
mem_name[addr_expr]
```

The `addr_expr` can be any expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory indirection:

```
mem_name[mem_name[3]]
```

In the above example, `mem_name[3]` addresses word three of the memory called `mem_name`. The value at word three is the index into `mem_name` that is used by the memory address `mem_name[mem_name[3]]`. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the index is out of the address bounds or is `x`, then the value of the reference is `x`.

There is no mechanism to express bit-selects or part-selects of memory elements directly. If this is required, then the memory element has to be first transferred to an appropriately sized temporary register.

Strings

String operands are treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character, with no special termination character.

Any Verilog HDL operator can manipulate string operands. The operator behaves as though the entire string were a single numeric value.

The following example declares a string variable large enough to hold 14 characters and assigns a value to it. The example then manipulates the string using the concatenation operator.

Note that when a variable is larger than required to hold the value being assigned, the contents after the assignment are padded on the left with zeros. This is consistent with the padding that occurs during assignment of non-string values.

Verilog-XL Reference

Expressions

```
module string_test;
  reg [8*14:1] stringvar;
  initial
    begin
      stringvar = "Hello world";
      $display("%s is stored as %h", stringvar, stringvar);
      stringvar = {stringvar, "!!!"};
      $display("%s is stored as %h", stringvar, stringvar);
    end
endmodule
```

The result of running Verilog on the previous description is:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

String Operations

The common string operations *copy*, *concatenate*, and *compare* are supported by Verilog operators. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators. The examples in [“Strings”](#) on page 64 and [“String Value Padding and Potential Problems”](#) on page 65 illustrate assignment, concatenation, and comparison of strings.

When manipulating string values in vector variables, at least $8 \cdot n$ bits are required in the vector, where n is the number of characters in the string.

String Value Padding and Potential Problems

When strings are assigned to variables, the values stored are padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators do not distinguish between zeros resulting from padding and the original string characters.

The following example illustrates the potential problem.

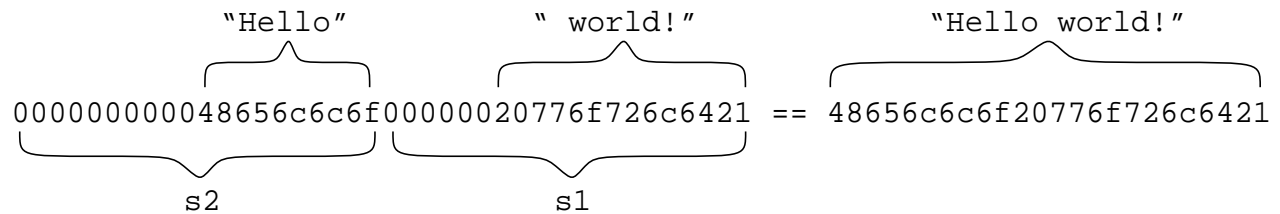
```
reg [8*10:1] s1, s2;
initial
  begin
    s1 = "Hello";
    s2 = " world!";
    if ({s1,s2} == "Hello world!")
      $display("strings are equal");
  end
```

Verilog-XL Reference

Expressions

The string variables `s1` and `s2` are padded with zeroes but the string "Hello world" contains no zero padding. Therefore, the comparison (`{s1,s2} == "Hello world!"`) fails, as demonstrated in the following example:

```
s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421
```



Null String Handling

The null string (" ") is equivalent to the value zero (0).

Minimum, Typical, Maximum Delay Expressions

Verilog HDL delay expressions can be specified as three expressions separated by colons. This triplet is intended to represent minimum, typical, and maximum values—in that order. The appropriate expression is selected by the compiler when Verilog-XL is run. The user supplies a command-line option to select which of the three expressions to use on a global basis. In the absence of a command-line option, Verilog-XL selects the second expression (the "typical" delay value). The syntax is as follows:

Syntax for `<mintypmax_expression>`

```
<mintypmax_expression>
  ::= <expression>
  ||= <expression1> : <expression2> : <expression3>
```

The three expressions follow these conventions:

- `<expression1>` is less than or equal to `<expression2>`
- `<expression2>` is less than or equal to `<expression3>`

Verilog models typically specify three values for delay expressions. The three values allow a design to be tested with minimum, typical, or maximum delay values. In the following example, one of the three specified delays will be executed before the simulation executes the assignment; if you do not select one, the simulator takes the default.

Verilog-XL Reference

Expressions

```
always @A
X = #(3:4:5) A;
```

The command-line option `+mindelays` selects the minimum expression in all expressions where `min:typ:max` values have been specified. Likewise, `+typdelays` selects all the typical expressions and `+maxdelays` selects all the maximum expressions. Verilog-XL defaults to the second value when a two or three-part delay expression is specified.

Values expressed in `min:typ:max` format can be used in expressions. The next example shows an expression that defines a single triplet of delay values. The minimum value is the sum of `a+d`; the typical value is `b+e`; the maximum value is `c+f`, as follows:

```
x_delay = (a:b:c) + (d:e:f)
```

The next example shows some typical expressions that are used to specify `min:typ:max` format values:

```
x_delay = (val - 32'd 50): 32'd 75: 32'd 100
```

The `min:typ:max` format can be used wherever expressions can appear, both in source text files and in interactive commands.

Expression Bit Lengths

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. Some situations have a simple solution, for example, if a bit-wise operation is specified on two 16-bit registers, then the result is a 16-bit value. However, in some situations it is not obvious how many bits are used to evaluate an expression, what size the result should be, or whether signed or unsigned arithmetic should be used.

For example, when is it necessary to perform the addition of two 16-bit registers in 17 bits to handle a possible carry overflow? The answer depends on the context in which the addition takes place. If the 16-bit addition is modeling a real 16-bit adder that loses or does not care about the carry overflow, then the model must perform the addition in 16 bits. If the addition of two 16-bit unsigned numbers can result in a significant 17th bit, then assign the answer to a 17-bit register.

An Example of an Expression Bit Length Problem

This section describes an example of the problems that can occur during the evaluation of an expression. During the evaluation of an expression, interim results take the size of the largest operand (in the case of an assignment, this also includes the left-hand side). You must therefore take care to prevent loss of a significant bit during expression evaluation.

Verilog-XL Reference

Expressions

The expression `(a + b >> 1)` yields a 16-bit result, but cannot be assigned to a 16-bit register without the potential loss of the high-order bit. If `a` and `b` are 16-bit registers, then the result of `(a+b)` is 16 bits wide—unless the result is assigned to a register wider than 16 bits.

If `answer` is a 17-bit register, then `(answer = a + b)` yields a full 17-bit result. But in the expression `(a + b >> 1)`, the sum of `(a + b)` produces an interim result that is only 16 bits wide. Therefore, the assignment of `(a + b >> 1)` to a 16-bit register loses the carry bit *before* the evaluation performs the one-bit right shift.

There are two solutions to a problem of this type. One is to assign the sum of `(a+b)` to a 17-bit register before performing the shift and then shift the 17-bit answer into the 16-bits that your model requires. An easier solution is to use the following method.

The problem:

Evaluate the expression `(a+b)>>1`, assigning the result to a 16-bit register without losing the carry bit. Variables `a` and `b` are both 16-bit registers.

The solution:

Add the `integer` zero to the expression. The expression evaluates as follows:

1. `0 + (a+b)` evaluates—the result is as wide as the widest term, which is the 32-bit zero.
2. The 32-bit sum of `0 + (a+b)` is shifted right one bit

This method preserves the carry bit until the shift operation can move it back down into 16 bits.

Verilog Rules for Expression Bit Lengths

In the Verilog language, the rules governing the expression bit lengths have been formulated so that most practical situations have an obvious solution.

The number of bits of an expression (known as the size of the expression) is determined by the operands involved in the expression and the context in which the expression is given.

A **self-determined expression** is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A **context-determined expression** is one where the bit length of the expression is determined by the bit length of the expression *and* by the fact that it is part of another

Verilog-XL Reference

Expressions

expression. For example, the bit size of the right-hand side expression of an assignment depends on itself and the size of the left-hand side.

Table 4-3 on page 69 shows how the form of an expression determines the bit lengths of the results of the expression. In the following example, i , j , and k represent expressions of an operand, and $L(i)$ represents the bit length of the operand represented by i .

Table 4-3 Bit lengths resulting from expressions

Expression	Bit Length	Comments
unsized constant number	same as integer (usually 32)	
sized constant number	as given	
i op j where op is: + - * / % & ^ ^~	$\max(L(i), L(j))$	
+ i and - i	$L(i)$	
~ i	$L(i)$	
i op j where op is: === !== == != && > >= < <=	1 bit	all operands are self-determined
op i where op is: & ~& ~ ^ ~^	1 bit	all operands are self-determined
i >> j i << j	$L(i)$	j is self-determined
i ? j : k	$\max(L(j), L(k))$	i is self-determined
{ i, \dots, j }	$L(i) + \dots + L(j)$	all operands are self-determined
{ i { j, \dots, k } }	$i * (L(j) + \dots + L(k))$	all operands are self-determined

Verilog-XL Reference

Expressions

Assignments

This chapter describes the following:

- [Overview](#) on page 71
- [Continuous Assignments](#) on page 72
- [Procedural Assignments](#) on page 79
- [Accelerated Continuous Assignments](#) on page 80
- [Procedural Continuous Assignments](#) on page 93

Overview

The assignment is the basic mechanism for getting values into nets and registers. An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equal sign (=). The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the variable that the right-hand side is to be assigned to. The left-hand side can take one of the following forms, depending on whether the assignment is a continuous assignment or a procedural assignment.

Table 5-1 Legal left-hand side forms in assignment statements

Statement	Left-hand side
continuous assignment	<ul style="list-style-type: none">■ net (vector or scalar)■ constant bit-select of a vector net■ constant part-select of a vector net■ concatenation of any of the above

Verilog-XL Reference

Assignments

Statement	Left-hand side
procedural assignment	<ul style="list-style-type: none">■ register (vector or scalar)■ bit-select of a vector register■ constant part-select of a vector register■ memory element■ concatenation of any of the above four items

Continuous Assignments

Continuous assignments drive values onto nets, both vector and scalar. The word *continuous* is used to describe this kind of assignment because the assignment is always active. Whenever simulation causes the value of the right-hand side to change, the assignment is re-evaluated and the output is propagated.

Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net. The expression on the right-hand side of the continuous assignment is not restricted in any way, and can even contain a reference to a function. Thus, the result of a `case` statement, `if` statement, or other procedural construct can drive a net.

See “[Calling Functions in a Continuous Assignment](#)” on page 78 for details on using functions in a continuous assignment statement.

The syntax for continuous assignments is as follows:

```
<net_declaration>
 ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
 || = trireg <charge_strength>? <expandrange>? <delay>?
    <list_of_variables> ;
 || = <NETTYPE> <drive_strength>? <expandrange>? <delay>?
    <list_of_assignments> ;

<continuous_assign>
 ::= assign <drive_strength>? <delay>? <list_of_assignments> ;

<expandrange>
 ::= <range>
 || = scalared <range>
 || = vectored <range>

<range>
 ::= [ <constant_expression> : <constant_expression> ]

<list_of_assignments>
 ::= <assignment> <,<assignment>> *

<charge_strength>
 ::= ( small )
```


Verilog-XL Reference Assignments

```
|| = ( medium )  
|| = ( large )  
  
<drive_strength>  
::= ( <STRENGTH0> , <STRENGTH1> )  
|| = ( <STRENGTH1> , <STRENGTH0> )
```

The Continuous Assignment Statement

The *<continuous_assign>* statement places a continuous assignment on a net that has been previously declared, either explicitly by declaration or implicitly by using its name in the terminal list of a gate, a user-defined primitive, or module instance. The following is an example of a continuous assignment to a net that has been previously declared:

```
wire a;  
assign a = b | (c & d);
```

In these statements, *a* is declared as a net of type *wire*. The right-hand side of the assignment statement can be thought of as a logic gate whose output is connected to wire *a*. The assignment is continuous and automatic. This means that whenever *b*, *c*, or *d* changes value during simulation, the whole right-hand side is reevaluated and assigned to the wire *a*.

The following is an example of the use of a continuous assignment to model a four-bit adder with carry.

```
module adder (sum_out, carry_out, carry_in, ina, inb);  
output [3:0]sum_out;  
input [3:0]ina, inb;  
output carry_out;  
input carry_in;  
wire carry_out, carry_in;  
wire[3:0] sum_out, ina, inb;  
    assign  
        {carry_out, sum_out} = ina + inb + carry_in;  
endmodule
```

The Net Declaration Assignment

[“The Continuous Assignment Statement”](#) on page 73 discusses placing continuous assignments on nets that were previously declared. An equivalent way of writing these two statements is through a net declaration assignment, which allows a continuous assignment to be placed on a net in the same statement that declares that net.

```
wire a = b | (c & d);
```

Here is another example of a net declaration assignment:

```
wire (strong1, pull0) mynet = enable;
```

A net declaration can be declared once for a specific net. This contrasts with the continuous assignment statement, where one net can receive multiple assignments of the continuous

Verilog-XL Reference

Assignments

assignment form. Also note that the assignment in the `adder` module in the previous section could not be specified directly in the declaration of the nets because it requires a concatenation on the left-hand side.

The following description contains examples of continuous assignments to a net (`data`) that was previously declared and an example of a net declaration assignment to a 16-bit output bus (`busout`). The module selects one of four input busses and connects the selected bus to the output bus.

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
    parameter n = 16;
    parameter Zee = 16'bz;
    output [1:n] busout;
    input [1:n] bus0, bus1, bus2, bus3;
    input enable;
    input [1:2] s;

    tri [1:n] data; // Net declaration.
    tri [1:n] busout = enable ? data : Zee; // Net declaration with
                                           // continuous assignment.

    assign // Assignment statement with
           // 4 continuous assignments.
        data = (s == 0) ? bus0 : Zee,
        data = (s == 1) ? bus1 : Zee,
        data = (s == 2) ? bus2 : Zee,
        data = (s == 3) ? bus3 : Zee;

endmodule
```

The following sequence of events takes place during simulation of the description in the `select_bus` module in the previous example:

1. The value of `s`, a bus selector input variable, is checked in the `assign` statement. Based on the value of `s`, the net `data` receives the data from one of the four input busses.
2. The setting of `data` triggers the continuous assignment in the net declaration for `busout`; if `enable` is set, the contents of `data` is assigned to `busout`; if `enable` is clear, the contents of `Zee` is assigned to `busout`.

Note that the parameter `Zee` has an *explicit* width specification on the high impedance value. This is recommended practice, because it avoids mistakes where extra bits of a value would cause erroneous results. The default width of the high-impedance value (`z`) is the word size of the host machine, typically 32 bits.

Delays

A delay given to a continuous assignment specifies the time duration between a right-hand side operand value change and the assignment made to the left-hand side. If the left-hand side references a scalar net, then the delay is treated in the same way as for gate delays—that is, different delays can be given for the output rising, falling, and changing to high impedance (see [Chapter 6, “Gate and Switch Level Modeling”](#)).

Verilog-XL Reference Assignments

The following can all have one, two, or three delays:

- a continuous assignment whose left-hand side references a vector net, such as:

```
wire [3:0] b;  
    assign #(5,20) b=a;
```

- a net declaration assignment, such as:

```
wire [3:0] #(5,20) b=a;
```

- a declaration of a net that is used in a continuous assignment, such as:

```
wire [3:0] #(5,20) b;  
    assign b=a;
```

When a continuous assignment whose left-hand side references a vector net or when a vector net declaration assignment includes delays, the following rules determine which delay controls the assignment. The following rules also determine the net delay that controls a continuous assignment to an unexpanded vector net.

- If the right-hand side *lsb* remains a 1 or becomes 1, then the rising (first) delay is used.
- If the right-hand side *lsb* remains 0 or becomes 0, then the falling (second) delay is used.
- If the right-hand side *lsb* remains z or becomes z, then the turn-off (third) delay is used.
- If the right-hand side *lsb* is an x or becomes an x, then the smallest of the delay values is used.

The results of the continuous assignment in the following example show how these rules operate on a delay that is based on the value or value change of the least significant bit.

```
module least_significant_bit (out);  
output [3:0] out;  
reg [3:0] a;  
wire [3:0] b;  
  
    assign #(10,20) b = a;  
  
    initial  
        begin  
            a = 'b0000;  
            #100 a = 'b1101;  
            #100 a = 'b0111;  
            #100 a = 'b1110;  
        end  
  
    initial  
        begin  
            $monitor($time, , "a=%b, b=%b", a, b);  
            #1000 $finish;  
        end  
endmodule
```

Compiling source file
Highest level modules:

Verilog-XL Reference Assignments

least_significant_bit

```
    0 a=0000, b=xxxx
    20 a=0000, b=0000
    100 a=1101, b=0000
    110 a=1101, b=1101 // LSB is high so uses a rise delay
    200 a=0111, b=1101
    210 a=0111, b=0111 // LSB is high so uses a rise delay
    300 a=1110, b=0111
    320 a=1110, b=1110 // LSB is low so uses a fall delay
```

The following example shows how multiple continuous assignments can model delays for discrete bits whose durations are determined by the types of transitions. A continuous assignment to an expanded vector net that has net delays in its declaration also models discrete bit delays whose values are dependent on the types of transitions.

```
module least_significant_bit (out);
output [3:0] out;
reg [3:0] a;
wire [3:0] b;
    assign #(10,20) b[0] = a[0],
                b[1] = a[1],
                b[2] = a[2],
                b[3] = a[3];

    initial
        begin
            a = 'b0000;
            #100 a = 'b1101;
            #100 a = 'b0111;
            #100 a = 'b1110;
        end
    initial
        begin
            $monitor($time, , "a=%b, b=%b", a, b);
            #1000 $finish;
        end
endmodule
```

Compiling source file
Highest level modules:
least_significant_bit

```
    0 a=0000, b=xxxx
    20 a=0000, b=0000
    100 a=1101, b=0000
    110 a=1101, b=1101 // rise delay of 10 time units
    200 a=0111, b=1101
    210 a=0111, b=1111 // rise delay of 10 time units
    220 a=0111, b=0111 // fall delay of 20 time units
    300 a=1110, b=0111
    310 a=1110, b=1111 // rise delay of 10 time units
    320 a=1110, b=1110 // fall delay of 20 time units
```

Delays in continuous assignments and in net declaration assignments behave differently from net delays because they do not add to the delays of other drivers on the net to make a longer delay.

Verilog-XL Reference

Assignments

In situations where a right-hand side operand changes before a previous change has had time to propagate to the left-hand side, the latest value change is the only one to be applied. That is, only one assignment occurs. This effect is known as **inertial delay**.

The following example implements a vector exclusive `or`. The size and delay of the operation are controlled by parameters, which can be changed when instances of this module are created. See [“Overriding Module Parameter Values”](#) on page 213 for details.

```
module modxor(axorb, a, b);
    parameter size = 8, delay = 15;
    output [size-1:0] axorb;
    input [size-1:0] a, b;
    wire [size-1:0] #delay axorb = a ^ b;
endmodule
```

Strength

You can specify the driving strength of a continuous assignment in either a net declaration or in a standalone assignment using the `assign` keyword. This applies only to assignments to scalar nets of the types listed below:

- `wire`
- `wand`
- `tri`
- `triereg`
- `wor`
- `triand`
- `tri0`
- `trior`
- `tril`

The strength specification, if provided, must immediately follow the keyword (either the keyword for the net type or the `assign` keyword) and must precede any specified delay. Whenever the continuous assignment drives the net, the strength of the value simulates as specified.

A `<drive_strength>` specification contains one strength value that applies when the value being assigned to the net is 1 and a second strength value that applies when the assigned value is 0. The following keywords specify the strength value for an assignment of 1:

- `supply1`

Verilog-XL Reference Assignments

- `strong1`
- `pull`
- `weak1`
- `highz1`

The following keywords specify the strength value for an assignment of 0:

- `supply0`
- `strong0`
- `pull0`
- `weak0`
- `highz0`

The order of the two strength specifications is arbitrary. The following two rules constrain the use of drive strength specifications:

- The strength specifications (`highz1`, `highz0`) and (`highz0`, `highz1`) are illegal language constructs.
- The keyword `vectored` is ignored when it is specified together with a specification of strength on a continuous assignment.

Calling Functions in a Continuous Assignment

The right-hand side expression of a continuous assignment can contain a reference to a function. For example, module `test` in the following example contains a continuous assignment that calls function `dumb`, which returns a value that is driven onto `sig_out_tmp`. See [“Functions and Function Calling”](#) on page 201 for details on defining and calling functions.

```
module test(sig_in, p_reset, sig_out);
    input sig_in, p_reset;
    output sig_out;
    wire sig_out_tmp;
buf (sig_out, sig_out_tmp);
assign sig_out_tmp = dumb(sig_in, p_reset);
function dumb;
input a, b;
begin
    if (b) // i.e., if p_reset
        dumb = 0;
    else
        dumb = a;
endfunction
endmodule
```

Verilog-XL Reference

Assignments

```
    end
endfunction
endmodule
```

In this example, the call to function `dumb` triggers an assignment to `sig_out_tmp` if `sig_in` or `p_reset` changes because both signals appear on the sensitivity list of the function call and because two variables are declared as local to the function.

In the next example, only `a` is declared as local to the function. The variable `p_reset` is passed to the function as a global variable. Even though the function can use and modify the variables within the boundary of the containing module (see “[Scope Rules](#)” on page 234), a change in `p_reset` will not trigger an assignment to `sig_out_tmp`.

```
module test(sig_in, p_reset, sig_out);
    input sig_in, p_reset;
    output sig_out;
    wire sig_out_tmp;

    buf (sig_out, sig_out_tmp);
    assign sig_out_tmp = dumb(sig_in);
    function dumb;
    input a;
    begin
        if (p_reset)
            dumb = 0;
        else
            dumb = a;
    end
endfunction
endmodule
```

Procedural Assignments

[Chapter 8, “Behavioral Modeling”](#) discusses procedural assignments in detail. However, a description of the basic ideas here highlight the differences between continuous assignments and procedural assignments.

As stated in “[Continuous Assignments](#)” on page 72, continuous assignments drive nets in a manner similar to the way gates drive nets. The expression on the right-hand side can be thought of as a combinational circuit that drives the net continuously. Continuous assignments cannot be disabled.

In contrast, procedural assignments can only assign values to registers or memory elements, and the assignment (that is, the loading of the value into the register or memory) is done only when control is transferred to the procedural assignment statement.

Procedural assignments occur only within procedures, such as `always` and `initial` statements (see “[always Statement](#)” on page 165 and “[initial Statement](#)” on page 165), and in functions and tasks (see [Chapter 9, “Tasks and Functions.”](#)), and can be thought of as

triggered assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, `if` statements, `case` statements, and looping statements can all be used to control whether assignments get evaluated.

Accelerated Continuous Assignments

This section describes how the `+cax1` command-line plus option accelerates continuous assignments to make your designs simulate faster. This section also explains the following:

- the restrictions on accelerated continuous assignments
- how to accelerate continuous assignments
- the kinds of designs that simulate faster with this feature and the kind of design that simulates slower
- how accelerated continuous assignments affect simulation

Restrictions on Accelerated Continuous Assignments

The `+cax1` option accelerates continuous assignments only if they meet the restrictions described in this section. These restrictions apply to the following syntax elements of a continuous assignment statement:

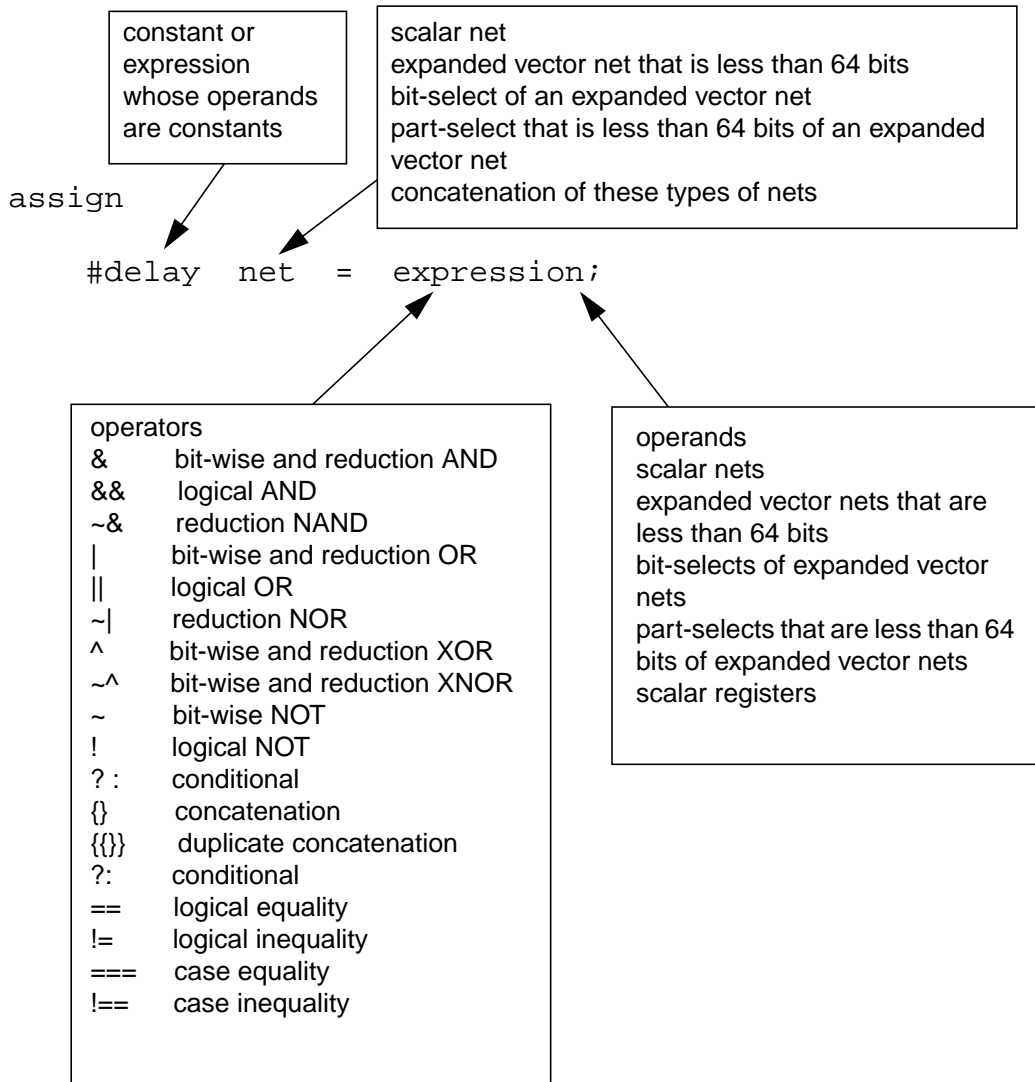
- the types of nets on the left-hand side of the assignment operator
- the operators and operands in the expressions on the right-hand side of the assignment operator
- the contents and use of a delay expression

[“Syntax elements of an accelerated continuous assignment”](#) on page 81 summarizes the valid syntax elements in accelerated continuous assignments. The sections following the syntax elements provide details and examples for each restriction.

Verilog-XL Reference

Assignments

Syntax elements of an accelerated continuous assignment



Left-hand side restrictions

The `+cax1` option can accelerate a continuous assignment to one of the following types of nets:

- a scalar net
- an expanded vector net that contains less than 64 bits
- a bit-select of an expanded vector net

Verilog-XL Reference

Assignments

- a part-select that is less than 64 bits of an expanded vector net

The `+cax1` option can also accelerate a continuous assignment to a concatenation of these types of nets, provided that the concatenation contains fewer than 64 bits. An expanded vector net is a vector net that Verilog-XL converts to a group of scalar nets. This group contains one scalar net for each bit of the vector net. Verilog-XL automatically converts or “expands” a vector net for a number of reasons, which include the following:

- to handle bit-selects and part-selects
- to improve performance and to accelerate continuous assignments

You can require Verilog-XL to expand a vector net by including the keyword `scalared` in the net’s declaration.

An unexpanded vector net is a vector net that Verilog-XL does not convert to scalar nets. You can prevent Verilog-XL from expanding a vector net by including the keyword `vectored` in its declaration.

The following example shows continuous assignments that the `+cax1` option can accelerate because the left-hand side meets these restrictions.

```
module aca1;
reg r1,r2,r3,r4;
wire c;
wire [3:0]a,d;
wire scalared [3:0] e,f,g;
assign      #5 c=a[0],           // CONTINUOUS ASSIGNMENTS TO...
           d={r1,r2,r3,r4},     // a scalar net
           e={r1,r2,r3,r4},     // an expanded vector net
           f[0]=r2,             // an expanded vector net
           f[3:2]={r3,r4},     // a bit-select of an expanded vector net
           {f[1],g[2:0]}=d;     // a part-select of an expanded vector net
           // a concatenation of valid nets
           ...
endmodule
```

Verilog-XL cannot accelerate a continuous assignment to the following types of vector nets:

- vector nets with 64 or more bits
- vector nets declared with the keyword `vectored`

To accelerate a continuous assignment to a vector net, Verilog-XL must expand that vector net. If you declare a vector net with the keyword `vectored`, Verilog-XL cannot accelerate a continuous assignment to it.

The following example shows continuous assignments that `+cax1` cannot accelerate because the left-hand side does not meet these restrictions.

```
module aca2;
reg r1,r2,r3,r4;
```

Verilog-XL Reference Assignments

```
wire [63:0] a;
wire vectored [3:0] b;
assign a = r1, // unaccelerated continuous assignment to
           // an expanded vector net with mor than 63 bits
       b={r1,r2,r3,r4}; // unaccelerated continuous assignment to
                       // an unexpanded vector net
...
endmodule
```

Right-hand side restrictions

The `+caxl` option can accelerate a continuous assignment only if the expression on the right-hand side contains certain operands and operators. The right-hand expression of a continuous assignment can contain any of the following operands:

- scalar nets
- expanded vector nets that contain less than 64 bits
- bit-selects of expanded vector nets
- part-selects that are less than 64 bits of expanded vector nets
- scalar registers
- constants

The `+caxl` option can also accelerate a continuous assignment where the right-hand side is a concatenation of these types of nets, provided that the concatenation contains fewer than 64 bits.

The following example shows continuous assignments that `+caxl` can accelerate because the operands in the expression on the right-hand side meet these restrictions. All operands are less than 64 bits in the example..

```
module aca3;
reg r1;
wire a,b;
wire [3:0] c,d;
wire scalared [3:0] e,f;
wire scalared [31:23] g,h;
assign
  h[31]=a & b, // operands a and b are scalar nets
  h[31:28]=c | d, // operands c and d are vector nets
  h[27]=e[0] ^ f[1], // operands e and f are bit-selects
                  // of expanded vector nets
  h[26:24]=e[2:0], // operand e is a part-select of an
                  // expanded vector net
  h[23]=r1; // operand is a scalar reg
...
endmodule
```

The prohibited operands are as follows:

Verilog-XL Reference Assignments

- expanded vector nets that contain more than 63 bits
- unexpanded vector nets
- bit-selects of unexpanded vector nets
- part-selects of unexpanded vector nets
- vector registers
- bit-selects of vector registers
- part-selects of vector registers

The following example shows continuous assignments that `+cax1` cannot accelerate because the operands in the expression of the right-hand side do not meet these restrictions.

```

module aca4;
reg [7:0]a,b;
wire vectored [7:0] c;
wire vectored [4:0] d;
wire [7:0] e,f;
wire [3:0] g,h,i;
wire [63:0] j;
assign
    i = j,           // unaccelerated because operand is
                    // a vector net with more than 63 bits
    e=c,            // unaccelerated because operand is
                    // an unexpanded vector net
    f[0]=c[0] & d[0], // unaccelerated because operands are
                    // bit-selects of an unexpanded vector net
    g=d[3:0],       // unaccelerated because operand is
                    // a part-select of an unexpanded vector net
    e=a,            // unaccelerated because operand is a vector reg
    g[0]=b[1],      // unaccelerated because operand is
                    // a bit-select of a vector reg
    h=b[3:0];       // unaccelerated because operand is
                    // a part-select of a vector reg
    ...
endmodule

```

The expression on the right-hand side of a continuous assignment can only contain the following operator

Table 5-2 Valid Operators

&	bit-wise and reduction AND	!	logical NOT
&&	logical AND	{}	concatenation
~&	reduction NAND	{{}}	duplicate concatenation
	bit-wise and reduction OR	?:	conditional
	logical OR	==	logical equality

Verilog-XL Reference Assignments

<code>~ </code>	reduction NOR	<code>!=</code>	logical inequality
<code>^</code>	bit-wise and reduction XOR	<code>===</code>	case equality
<code>~^</code>	bit-wise and reduction XNOR	<code>!==</code>	case inequality
<code>~</code>	bit-wise NOT		

The following example shows continuous assignments that `+caxl` can accelerate because the operators in the expression on the right-hand side meet these restrictions.

```

module aca5;
reg r1,r2,r3,r4,r5,r6,r7;
wire a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,s,t;
wire [1:0]u,v,w,y;
assign+
    a=r1 & r2,           // bit-wise AND operator
    b=&s,                // unary reduction AND operator
    c=r1 && r2,          // logical AND operator
    d=~&u,              // unary reduction NAND operator
    e=r2 | r3,          // bit-wise OR operator
    f=r3 || r4,         // logical OR operator
    g=|u,               // unary reduction OR operator
    h=~|u,              // unary reduction NOR operator
    i=r4 ^ r5,          // bit-wise XOR operator
    j=^u,               // unary reduction XNOR operator
    k=r5 ~^r6,          // bit-wise XNOR operator
    l=~^v,              // unary reduction XNOR operator
    m=~j,               // bit-wise NOT operator
    n = !r1,            // logical NOT operator
    w={a,b},            // concatenation operator
    y={2{r7}},          // duplicate concatenation operator
    q=r1 ? a : b,       // conditional operator
    s= r1 == r2,        // logical equality operator
    t= r3 != r4;        // logical inequality operator
endmodule

```

You can enter other operators in the right-hand side of accelerated continuous assignments, but only in an expression or sub-expression whose operands are constants. (A sub-expression is a part of an expression that Verilog-XL can evaluate separately.) The prohibition against other operators does not apply in these expressions or sub-expressions because Verilog-XL evaluates them at compile time. The following example shows how you can use other operators in accelerated continuous assignments.

```

module aca6;
parameter p1=8,p2=15;
reg r1;
wire [3:0] a,b,c;
assign
    a = 1 + p1,          // 1 + p1 expression with addition
                          // operator and constant operands
    b = r1 | (p2 << 1), // p2 << 1 sub-expression with shift
                          // operator and constant operands
    c {r1,(p2 % p1)};   // p2 % p1 sub-expression with
                          // modulo operator and constant operands

```

Verilog-XL Reference Assignments

```
endmodule
```

The following example shows continuous assignments that `+caxl` cannot accelerate because they use other operators with variable operands.

```
module aca7;
  reg r1,r2;
  reg [3:0] r3;
  wire a,b,c;
  wire [4:0] d;
  wire [31:0] e;
  assign
    e = r1 * r2,          // r1 * r2 expression with an arithmetic
                          // operator and variable operands
    a = (b <= c),        // b <= c expression with a relational
                          // operator and variable operands
    d = r3 << 1;         // r3 << 1 expression with a shift operator
                          // and variable operands
endmodule
```

Delay expression restrictions

The `+caxl` option can accelerate a continuous assignment that includes a delay only if that delay is a constant or an expression whose operands are constants.

The following example shows continuous assignments that `+caxl` can accelerate because the delay expression meets this restriction.

```
module aca8;
  reg r1,r2;
  wire a,b,q,qb;
  parameter p=10;
  assign #p q = ~(a & qb);          // #p delay is a constant
  assign #(p+1) qb = ~(b & q);     // #(p+1) delay expression with constant operands
  ...
endmodule
```

The following example shows continuous assignments that `+caxl` cannot accelerate because the delay expression does not meet this restriction.

```
module aca9;
  wire a,b,c,d;
  reg r1,r2;
  assign #r1 a=c;                  // #r1 delay is not a constant
  assign #(a & r2) b=d;           // operands a and r2 in delay expression are variables
  ...
endmodule
```

Controlling the Acceleration of Continuous Assignments

The `+caxl` command-line option accelerates continuous assignments subject to control by two compiler directives: ``accelerate` and ``noaccelerate`.

Verilog-XL Reference Assignments

- The ``accelerate` compiler directive makes continuous assignments that follow it acceleratable by the XL algorithm. A later ``noaccelerate` compiler directive cuts off the effect of a previous ``accelerate` compiler directive.
- The ``noaccelerate` compiler directive prevents the XL algorithm from accelerating continuous assignments that follow it; a later ``accelerate` compiler directive cuts off the effect of a previous ``noaccelerate` compiler directive.

The following example shows the region of a sample design, delimited by ``accelerate` and ``noaccelerate`, whose continuous assignments are accelerated if the `+caxl` option is on the command line.

In the following example, `+caxl` accelerates continuous assignments in the middle section, but the other continuous assignments cannot be accelerated.

Controlling acceleration of continuous assignments

```
`noaccelerate           // --- prevents   ---
module mod2;           // --- accelerated ---
  ...                 // --- continuous ---
assign d = e | f;      // --- events     ---
  ...                 // --- in         ---
  ...                 // --- this      ---
endmodule              // --- region    ---

`accelerate            // +++ permits    +++
module mod3 (v,l,g);  // +++ accelerated +++
  ...                 // +++ continuous +++
assign g = h ^ i;     // +++ assignments +++
  ...                 // +++ in        +++
  ...                 // +++ this     +++
endmodule              // +++ region    +++

`noaccelerate           // --- prevents   ---
module mod4(j,t,v);  // --- accelerated ---
  ...                 // --- continuous ---
assign j = e ~^ k;.  // --- events     ---
  ...                 // --- in         ---
  ...                 // --- this      ---
endmodule              // --- region    ---
```

The Effects of Accelerated Continuous Assignments

Accelerating continuous assignments can have the following effects on your simulation. These effects are described in the following sections.

- faster simulation
- slightly slower compilation

Verilog-XL Reference Assignments

- slightly more memory use
- alteration in simulation results

Simulation speed

Accelerating continuous assignments does not increase the simulation speed of all designs. The types of designs that simulate faster, and the one type that simulates slower, are described in this section.

Designs that simulate faster

The following is a list of the kinds of designs that simulate faster when `+cax1` accelerates continuous assignments:

- designs that consist entirely of accelerated continuous assignments to scalar nets
- designs that are a combination of gate-level and accelerated continuous assignments
- gate-level designs that are stimulated by accelerated continuous assignments
- designs that consist of accelerated continuous assignments to large vector nets

The following are examples of these designs and an explanation of how accelerated continuous assignment increases their simulation speed.

1. Accelerating continuous assignments is what most increases the simulation speed of designs that consist entirely of accelerated continuous assignments to scalar nets. These designs simulate approximately eight times faster when you accelerate all their continuous assignments. The following source description shows a design of a multiplexer that consists of accelerated continuous assignments to scalar nets:

```
module aca10 (op1,op2,s1,s2,out,cr);
input op1,op2,s1,s2;
output out,cr;
wire nop1,nop2,mx1,mx2;
assign
  nop1 = ~op1,
  nop2 = ~op2,
  mx1 = ((op1 & s1) | (nop1 & ~s1)),
  mx2 = ((op2 & s2) | (nop2 & ~s2)),
  out = mx1 ^ mx2,
  cr = mx1 & mx2;
endmodule
```

2. Accelerating continuous assignments also increases the simulation speed of designs whose logic is a combination of gate-level and accelerated continuous assignments.

Verilog-XL Reference Assignments

How much the acceleration of the continuous assignments increases the simulation speed depends on the proportion of continuous assignments to gate instances.

The following example shows a design that is a combination of accelerated continuous assignments and gate instances. In this source description, data flows from gates to continuous assignments and back to gates.

```
module acall (op1,op2,s1,s2,out,cr);
input op1,op2,s1,s2;
output out,cr;
wire nop1,nop2,mx1,mx2;
assign
  mx1 = ((op1 & s1)|(nop1 & ~s1)),
  mx2 = ((op2 & s2)|(nop2 & ~s2));
  not nt1 (nop1,op1),
  nt2 (nop2,op2);
  xor xr1 (out,mx1,mx2);
  and ad1 (cr,mx1,mx2);
endmodule
```

3. Accelerating continuous assignments also increases the simulation speed of gate-level designs that are stimulated by accelerated continuous assignments. How much the acceleration of the continuous assignments increases the simulation speed of these designs depends on the proportion of continuous assignments to gate instances.

The source description shown in the following example includes one accelerated continuous assignment. Accelerating this continuous assignment does little to increase the design's simulation speed because the accelerated continuous assignment is such a small proportion of this design.

```
module acal3;
reg r1,r2,r3;
wire a;
assign a=r3; // one accelerated continuous assignment...
tway t1 (r1,r2,a,o);
initial
  ...
endmodule

module tway(r1,r2,a,o);
input r1,r2;
output o;
inout a; // ...that drives this inout port
  bufif1(a,r1,r2);
  bufif0(o,a,r2);
endmodule
```

4. Accelerating the continuous assignments in designs that consist of continuous assignments to large vector nets results in the smallest increase in simulation speed. Continuous assignments to vector nets 64 bits wide and larger cannot be accelerated. The closer the left-hand side of a continuous assignment comes to this limit of 63 bits, the more time the XL algorithm needs to simulate the continuous assignment.

Verilog-XL Reference Assignments

The source description in the following example shows the continuous assignment of expressions with large operands and several operators to very large vector nets. The complexity of the expressions on the right-hand side and large vector nets on the left-hand side result in a limited increase in simulation speed.

```
module aca14;
wire [30:0]
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;
wire [61:0] m1,m2,m3,m4,m5;
assign m1=~(( {a,b}&{d,e} ) | ( {c,d}^{e,f} )),
        m2={e,f}&{h,i},
        m3=~{i,j},
        m4=~( {m,n} | {a,b} ),
        m5=((q & r)^{p | t})~^{q,r});
endmodule
```

Designs that simulate slower

Not all designs with continuous assignments that can be accelerated simulate faster with the XL algorithm. XL speeds up the simulation when it processes a continuous assignment, but transitions between the XL and non-XL algorithms slow down the simulation. A large number of transitions can make a simulation run slower than if no part of it is simulated by the XL algorithm. The following is a list of designs that contain continuous assignments that you can accelerate, but which simulate faster without accelerating these continuous assignments.

1. Designs whose data flows many times from accelerated to nonaccelerated continuous assignments simulate at a slower speed than if you did not accelerate any continuous assignment. This slower speed is caused by the performance cost of a large number of transitions between algorithms. The following source description shows data flowing through a path of continuous assignments that cause Verilog-XL to transition frequently between algorithms.

```
module aca15;
wire a,b,c,d,e,f,g,h,i,j,k,l,m,n,o;
assign
    a = b & c, // accelerated continuous assignment
    b = d + e, // nonaccelerated continuous assignment
    d = f | g, // accelerated continuous assignment
    f = h - i, // nonaccelerated continuous assignment
    h = j ^ k, // accelerated continuous assignment
    j = l * m, // nonaccelerated continuous assignment
    l = n ~^ o; // accelerated continuous assignment
    ...
endmodule
```

2. Designs whose data flows many times from accelerated continuous assignments to procedural assignments also simulate at a slower speed than if you did not accelerate any continuous assignment. This slower speed is also caused by transitions between algorithms. In the following source description, data flows between both kinds of assignments.

Verilog-XL Reference Assignments

```
module aca16;
reg r1,r2,r3,r4,r5;
wire a,b,c,d,e;

assign
  a = r1,           // accelerated
  b = r2,           // continuous
  c = r3,           // assignments
  d = r4,           // ...
  e = r5;           // ...

always
  begin
    #10 r1 = b;     // procedural
    #10 r2 = c;     // assignments
    #10 r3 = d;     // ...
    #10 r4 = e;     // ...
    #10 r5 = ~r5;   // ...
  end

initial
  begin
    r5=1;
    ...
  end
endmodule
```

In this source description, a value of 1 propagates through several wires and registers. Data flow begins with a procedural assignment to reg `r5`, then through a path of registers and wires that are driven by alternating continuous and procedural assignments.

Compilation speed

During compilation, Verilog-XL processes accelerated continuous assignments so that they can be simulated by the XL algorithm. Therefore, compilation time increases as the number of accelerated continuous assignments increases. A design that consists entirely of continuous assignments that can be accelerated takes approximately twice as long to compile if you accelerate these continuous assignments.

Memory usage

Accelerated continuous assignments cause Verilog-XL to use more memory at compile time, but less memory at run time.

Verilog-XL needs more memory to compile a design with accelerated continuous assignments. A design that consists entirely of accelerated continuous assignments needs 20% more memory to compile.

Accelerated continuous assignments reduce Verilog-XL's memory requirements during simulation.

Verilog-XL Reference

Assignments

The possibility of different results

Accelerating continuous assignments to vector nets when these continuous assignments include delay expressions can produce simulation results that differ from the results produced without accelerating these continuous assignments. This possible difference is caused by the difference between how the XL and non-XL algorithm simulate these continuous assignments.

In both the XL and non-XL algorithms, when a continuous assignment statement includes a delay expression, Verilog-XL evaluates the right-hand side and schedules the assignment to occur after the delay elapses. In the non-XL algorithm, if any of the bits of the right-hand side change before the delay elapses, Verilog-XL re-evaluates the entire right-hand side and reschedules the assignment. In the XL algorithm, if any of the bits of the right-hand side change before the delay elapses, Verilog-XL schedules a subsequent assignment to those bits.

The following two examples show how accelerating continuous assignments can produce different simulation results.

The first of two examples shows a module that contains accelerated and unaccelerated continuous assignments that assign the same values and include the same delay expression. The accelerated continuous assignments propagate value changes at simulation times when the unaccelerated continuous assignments do not propagate these value changes.

```
module dif;
wire [1:0] a1, a2;
wire vectored [1:0] b1;
reg c1,c2;
reg [1:0] d1;

assign      #10 a1 = {c1,c2};                // accelerated continuous
assignment

assign
    #10 b1 = {c1,c2},                        // unaccelerated continuous
        a2 = d1;                             // assignments

initial
begin
    $monitor("At simulation time %0d\n",
    $time,
    " accelerated a1=%b\n",a1,
    "unaccelerated b1=%b a2=%b\n\n",b1,a2);
    #25 c1 = 0;                               // procedural assignments of the
        d1[1] = 0;                             // same values to the bits of the
    #5 c2 = 0;                                 // right-hand side of all three
        d1[0] = 0;                             // continuous assignments.
end
endmodule
```

In the previous example, the continuous assignment to wire `a1` of the concatenation of scalar registers `c1` and `c2` can be accelerated. The continuous assignment to wire `b1` cannot be accelerated because it assigns a value to an unexpanded vector net, and the continuous

Verilog-XL Reference Assignments

assignment to wire `a2` cannot be accelerated because its operand is a vector register. The delay expression in these continuous assignments is 10 time units.

Procedural assignments assign the same values to the right-hand sides of these continuous assignments. These procedural assignments specify a five time unit interval between bit changes of the right-hand sides of the continuous assignments.

The XL algorithm schedules the propagation of all bit changes as they occur; the non-XL algorithm does not. The difference in simulation results between the accelerated and unaccelerated continuous assignments is shown in [Figure 5-1](#) on page 93.

Figure 5-1 Difference in Simulation results

Highest level modules:

`dif`

At simulation time 0

accelerated `a1=xx`

unaccelerated `b1=xx a2=xx`

At simulation time 35

accelerated `a1=0x`

unaccelerated `b1=xx a2=xx`

At simulation time 40

accelerated `a1=00`

unaccelerated `b1=00 a2=00`

The XL algorithm assigns values at simulation times 35 and 40.

The non-XL algorithm assigns values only at simulation time 40.

In the previous example, the XL algorithm assigns values to `a1` at simulation times 35 and 40. The non-XL algorithm waits until simulation time 40 to assign values.

Procedural Continuous Assignments

Continuous assignment statements allow you to describe combinational logic whose output is to be computed anytime any input changes. Procedural continuous assignments are procedural statements that allow for continuous assignments to be made to registers or nets for certain specified periods of time. Because the assignment is not in force forever, as with continuous assignments, procedural continuous assignments are sometimes called **quasi-continuous assignments**.

Verilog-XL Reference Assignments

The syntax for these assignment statements is as follows:

```
<statement>
    ::= assign <assignment> ;
<statement>
    ::= deassign <lvalue> ;
<force_statement>
    ::= force <assignment> ;
<release_statement>
    ::= release <lvalue> ;
```

The left-hand side of the assignment in the `assign` statement is restricted to be a register reference or a concatenation of registers. It cannot be a memory element (array reference), or a bit-select or a part-select of a register. In contrast, the left-hand side of the assignment in the `force` statement can be a register reference, a net reference, or a bit-select or part-select of an expanded vector net. It can be a concatenation of any of the above. Bit-selects and part-selects of vector registers or unexpanded vector nets are not allowed, and result in an error.

The assign and deassign Procedural Statements

The `assign` and `deassign` procedural assignment statements allow continuous assignments to be placed onto registers for controlled periods of time. The `assign` procedural statement overrides procedural assignments to a register. The `deassign` procedural statement ends a continuous assignment to a register. The `assign` and `deassign` procedural statements allow, for example, modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

The following example shows a use of the `assign` and `deassign` procedural statements in a behavioral description of a D flip-flop with preset and clear inputs.

```
module dff(q,d,clear,preset,clock);
    output q;
    input d,clear,preset,clock;
    reg q;
    always @(clear or preset)
    begin
        if(!clear)
            assign q = 0;           // assign procedural statement
        else if(!preset)
            assign q = 1;           // assign procedural statement
        else
            deassign q;             // deassign procedural statement
    end
    always @(posedge clock)
        q = d;                       // procedural assignment statement
endmodule
```

Verilog-XL Reference

Assignments

When either `clear` or `preset` is low, the output `q` is held continuously to the appropriate constant value and a positive edge on the `clock` is affect `q`. When both the `clear` and `preset` are high, then `q` is deassigned.

If the keyword `assign` is applied to a register for which there is already a procedural continuous assignment, then this new procedural continuous assignment automatically deassigns the register before making the new procedural continuous assignment.

The force and release Procedural Statements

Another form of procedural continuous assignment is provided by the `force` and `release` procedural statements. These statements have a similar effect on the `assign-deassign` pair, but a `force` can be applied to nets as well as to registers. The left-hand side of the assignment can be:

- a register
- a net
- a constant bit select of an expanded vector net
- a part select of an expanded vector net
- a concatenation

The left-hand side cannot be a memory element (array reference) or a bit-select or a part-select of a vector register or an unexpanded vector net.

A `force` procedural statement to a register overrides a procedural assignment or procedural continuous assignment that takes place on the register until a `release` procedural statement is executed on the register. After the `release` procedural statement is executed, the register does not immediately change value (as would a net that is forced). The value specified in the `force` statement is maintained in the register until the next procedural assignment takes place, except in the case where a procedural continuous assignment is active on the register.

A `force` procedural statement on a net overrides all drivers of the net—gate outputs, module outputs, and continuous assignments—until a `release` procedural statement is executed on the net.

Releasing a register that currently has an active `assign` re-establishes the `assign` statement. The reason for having a two-level override system for registers is that `assign-deassign` is meant for actual descriptions of hardware, and the `force-release` is meant for debugging purposes.

Verilog-XL Reference Assignments

The following example shows part of a log file from a simulation that included interactively entered `force` and `release` procedural statements.

At the interactive prompt on line C1, AND gate `and1` is “patched” as an OR gate by a `force` procedural statement. This “patch” forces the value of its ORed inputs onto output `e`. On line C2, an `assign` procedural statement of ANDed values to output `d` is “patched” by a `force` procedural statement of ORed values.

```
1  module test;
2      reg
2      a, // = 1'hx, x
2      b, // = 1'hx, x
2      c, // = 1'hx, x
2      d; // = 1'hx, x
3      wire
3      e; // = StX
5      and
5      and1(e, a, b, c);    // AND gate instance
7      initial
8      begin
9*         $list;
10         $monitor("d=%b,e=%b", d, e);
11         assign d = a & b & c; // assign proc. statement
12         a = 1;
13         b = 0;
14         c = 1;
15         #10
15         $stop;
16     end
17 endmodule
d=0,e=0
L15 "quasi.v": $stop at simulation time 10
Type ? for help
C1 > force e = (a | b | c); // force procedural statement
C2 > force d = (a | b | c); // force procedural statement
C3 > #10 $stop;
C4 > .
d=1,e=1
C3: $stop at simulation time 20
C4 > release e; // release procedural statement
C5 > release d; // release procedural statement
C6 > c = 0;
C7 > #10 $finish;
C8 > .
e=0,d=0
C7: $finish at simulation time 30
```

Gate and Switch Level Modeling

This chapter describes the following:

- [Overview](#) on page 98
- [Gate and Switch Declaration Syntax](#) on page 98
- [and, nand, nor, or, xor, and xnor Gates](#) on page 105
- [buf and not Gates](#) on page 106
- [bufif1, bufif0, notif1, and notif0 Gates](#) on page 107
- [MOS Switches](#) on page 108
- [Bidirectional Pass Switches](#) on page 110
- [cmos Switches](#) on page 112
- [pullup and pulldown Sources](#) on page 113
- [Implicit Net Declarations](#) on page 113
- [Logic Strength Modeling](#) on page 114
- [Strengths and Values of Combined Signals](#) on page 116
- [Strength Resolution for Continuous Assignments](#) on page 129
- [Mnemonic Format](#) on page 130
- [Strength Reduction by Non-Resistive Devices](#) on page 131
- [Strength Reduction by Resistive Devices](#) on page 131
- [Strengths of Net Types](#) on page 131
- [Gate and Net Delays](#) on page 132
- [Gate and Net Name Removal](#) on page 140

Overview

A logic network can be modeled using continuous assignments or switches and logic gates. Gates and continuous assignments serve different modeling purposes and it is important to appreciate the differences between them to achieve the right balance between accuracy and efficiency in Verilog-XL. Modeling with switches and logic gates has the following advantages:

- Gates provide a much closer one-to-one mapping between the actual circuit and the network model.
- There is no continuous assignment equivalent to the bidirectional transfer gate.
- Because gates and switches have fixed functions, Verilog-XL can optimize its data structure to reduce the amount of memory needed to simulate large circuits.
- For a random network of nets, it is likely that the use of gates and switches for modeling gives a shorter simulation run time than the use of continuous assignments.

A limitation of those nets declared with the keyword `vectored` affects gates and switches as well as continuous assignments. Individual bits of vectored nets cannot be driven; thus, gates and switches can only drive scalar output nets. If you declare a multi-bit net as `vectored` and you drive individual bits of it, Verilog-XL will display a compilation error message. If you do not declare a multi-bit net as `vectored`, Verilog-XL handles it as a vector except in the following cases. A multi-bit net is handled as a scalar if:

- Part of the vector is driven by a gate or switch.
- Part of the vector is assigned a value with a continuous assignment.

The Switch-XL algorithm, invoked with the `+switchxl` plus option, expects references to the terminals of switches to be expanded. References in the terminals of switches cannot be references to register bit-selects when the `+switchxl` option is used. See [Chapter 8, “Behavioral Modeling,”](#) of the *Verilog-XL User Guide* for more information on the Switch-XL algorithm.

Gate and Switch Declaration Syntax

A gate or switch declaration names a gate or switch type and specifies its output signal strengths and delays. It contains one or more gate instances. Gate instances include an optional instance name and a required terminal connection list. The terminal connection list specifies how the gate or switch connects to other components in the model. All the instances contained in a gate or switch declaration have the same output strengths and delays.

The gate or switch declaration syntax is as follows:

Verilog-XL Reference

Gate and Switch Level Modeling

```
<gate_declaration>
    ::= <GATETYPE> <drive_strength>? <delay>? <gate_instance>
        <, <gate_instance>*>;

<GATETYPE> is one of the following keywords:
    and nand or nor xor xnor buf bufif0 bufif1
    not notif0 notif1 pulldown pullup
    nmos rmos pmos rmos cmos rcmos
    tran rtran tranif0 rtranif0 tranif1 rtranif1

<drive_strength>
    ::= ( <STRENGTH0> , <STRENGTH1> )
    || = ( <STRENGTH1> , <STRENGTH0> )

<delay>
    ::= # <number>
    || = # <identifier>
    || = # ( <mintypmax_expression> <, <mintypmax_expression>*>?
        <, <mintypmax_expression>*>?)

<gate_instance>
    ::= <name_of_gate_instance>? ( <terminal> <, <terminal>*>* )

<name_of_gate_instance>
    ::= <IDENTIFIER> <range_spec>?

<terminal>
    ::= <IDENTIFIER>
    || = <expression>

<range_spec>
    ::= [ <lefthand_index> : <righthand_index> ]

<lefthand_index>
    ::= <constant_expression>

<righthand_index>
    ::= <constant_expression>

<constant_expression> is one of the following elements
    scalar, vector, register, or concatenations of these elements.
```

This section describes the following parts of a gate or switch declaration:

- The keyword that names the type of gate or switch primitive
- The drive strength specification
- The delay specification
- The identifier that names each gate or switch instance in gate or switch declarations
- An optional range specification for an array of instances
- The terminal connection list in primitive gate or switch instances

The Gate Type Specification

A gate declaration begins with the `<GATETYPE>` keyword. The keyword specifies the gate or switch primitive that is used by the instances that follow in the declaration.

The following keywords can begin a gate or switch declaration.

<code>and</code>	<code>nor</code>	<code>pullup</code>	<code>tran</code>
<code>buf</code>	<code>not</code>	<code>rcmos</code>	<code>tranif0</code>
<code>bufif0</code>	<code>notif0</code>	<code>rnmos</code>	<code>tranif1</code>
<code>bufif1</code>	<code>notif1</code>	<code>rpmos</code>	<code>xnor</code>
<code>cmos</code>	<code>or</code>	<code>rtran</code>	<code>xor</code>
<code>nand</code>	<code>pmos</code>	<code>rtranif0</code>	
<code>nmos</code>	<code>pulldown</code>	<code>rtranif1</code>	

The Drive Strength Specification

The drive strength specifications specify the strengths of the values on the output terminals of the instances in the gate declaration. It is possible to specify the strength of the output signals from the gate primitives in the following table.

<code>and</code>	<code>nand</code>	<code>notif1</code>	<code>xnor</code>
<code>buf</code>	<code>nor</code>	<code>or</code>	<code>xor</code>
<code>bufif0</code>	<code>not</code>	<code>pulldown</code>	
<code>bufif1</code>	<code>notif0</code>	<code>pullup</code>	

The drive strength specification in this table has two parts. A gate declaration must contain both parts or no parts, with the exception of `pullup` and `pulldown` sources. One of the parts specifies the strength of signals with a value of 1, and the other specifies the strength of signals with a value of 0.

The `STRENGTH1` specification, which specifies the strength of an output signal with a value of 1, is one of the following keywords:

`supply1` `strong1` `pull1` `weak1` `highz1`

Specifying `highz1` causes the gate to output a logic value of z in place of a 1.

The `STRENGTH0` specification, which specifies the strength of an output signal with a value of 0, is one of the following:

`supply0` `strong0` `pull0` `weak0` `highz0`

Specifying `highz0` causes the gate to output a logic value of z in place of a 0.

The strength specifications must follow the gate type keyword and precede any delay specification. The `STRENGTH0` specification can precede or follow the `STRENGTH1` specification. In the absence of a strength specification, the instances have the default strengths `strong1` and `strong0`.

Verilog-XL Reference

Gate and Switch Level Modeling

The strength specifications, `(highz0, highz1)` and `(highz1, highz0)`, are invalid and produce the following compiler error message:

```
Error!      Illegal strength specification
```

The following example shows a drive strength specification in a declaration of an open collector `nor` gate:

```
nor(highz1, strong0)(out1, in1, in2);
```

In this example, the `nor` gate outputs a `z` in place of a `1`.

The Delay Specification

The delay specifies the propagation delay through the gates and switches in a declaration. Gates and switches in declarations with no delay specification have no propagation delay. A delay specification can contain up to three delay values, depending on its gate type. The `pullup` and `pulldown` source declarations do not include delay specifications.

The Primitive Instance Identifier

The `<IDENTIFIER>` in “[Gate and Switch Declaration Syntax](#)” on page 98 is an optional name given to a gate or switch instance. An instance that is declared as an array must be named (not optional). The name is useful in tracing the operation of the circuit during debugging. Verilog-XL can generate names for unnamed gate instances in the source description. See “[Modules](#)” on page 210 for information about automatic naming. Compiler directives discussed in “[Gate and Net Name Removal](#)” on page 140 remove optional gate and net names to reduce virtual memory requirements at simulation time.

The Range Specification

Repetitive instances that are defined by a range specification (`<range_spec>` in “[Gate and Switch Declaration Syntax](#)” on page 98) differ only in the selected bits of the port expression to which they are connected. To specify an array of instances, specify an instance name followed by a range specification, which consists of two constant expressions (of local module constants or parameters) separated by a colon (`:`) inside of square brackets (`[]`). The constant expressions are the left-hand index (LHI) and the right-hand index (RHI). If these two constant expressions have the same value, only one instance is generated.

An array of instances has a continuous range. To declare an array of instances, one instance name is associated with only one range; you cannot specify the same instance name for another range.

Verilog-XL Reference

Gate and Switch Level Modeling

For example, the following specification is *illegal* because the same instance name is used for two ranges:

```
nand #2 t_nand[0:3] (...), t_nand[4:7] (...);
```

The following specifications is made legal by specifying unique instance names with separate ranges:

```
nand #2 t_nand[0:7] (...);  
nand #2 x_nand[0:3] (...), y_nand[4:7] (...);
```

Primitive Instance Connection List

The `<terminal>` in “[Gate and Switch Declaration Syntax](#)” on page 98 is the terminal list. The terminal list describes how the gate or switch connects to the rest of the model. The gate or switch type limits these expressions. The output or bidirectional terminals always come first in the terminal list, followed by the input terminals.

Rules for Using an Array of Instances

The following rules apply for arrays of instances:

Rule 1

For terminal connections to an array of instances, a comparison is made between the bit length of each port expression in a declared instance array and the bit length of each single-instance port or terminal in an instantiated model or primitive.

Port connections are as follows.

Same Bit Lengths:

For each port or terminal where the comparison shows the same bit lengths, the instance-array port expression is connected to each single-instance port. The following example shows equivalent module descriptions that illustrate connections where port sizes match.

```
module driver (in, out, en);  
  input [3:0] in;  
  output [3:0] out;  
  input en;  
  
  bufif0 ar[3:0] (out, in, en); // array of tri-state buffers  
endmodule  
  
module driver_equiv (in, out, en);  
  input [3:0] in;
```

Verilog-XL Reference

Gate and Switch Level Modeling

```
output [3:0] out;
input en;

bufif0 ar3 (out[3], in[3], en); // each buffer declared
bufif0 ar2 (out[2], in[2], en); // separately
bufif0 ar1 (out[1], in[1], en);
bufif0 ar0 (out[0], in[0], en);
endmodule
```

Different Bit Lengths:

If the bit lengths are different, then the bit length of each port expression in the declared instance array should be the product of the number of instances and the bit length of the corresponding single-instance port or terminal. In such case, each instance gets a part-select of the port expression as specified in the range.

The connections to the port of the instance start with the lowest bits of the port expressions (right-hand index) and continue to the highest bits (left-hand index).

The following example shows equivalent modules that illustrate how different instances within an array of instances are connected when the port sizes do not exactly match (but are evenly divisible).

```
module busdriver (busin, bushigh, buslow, enh, enl);
    input [15:0] busin;
    output [7:0] bushigh, buslow;
    input enh, enl;

    driver busar3 (busin[15:12], bushigh[7:4], enh);
    driver busar2 (busin[11:8], bushigh[3:0], enh);
    driver busar1 (busin[7:4], buslow[7:4], enl);
    driver busar0 (busin[3:0], buslow[3:0], enl);
endmodule

module busdriver_equiv (busin, bushigh, buslow, enh, enl);
    input [15:0] busin;
    output [7:0] bushigh, buslow;
    input enh, enl;

    driver busar[3:0] (.out({bushigh, buslow}), .in(busin),
        .en({enh, enh, enl, enl}));
endmodule
```

Too Many or Too Few Bits:

If there are too many or too few bits to connect to all the instances, an error condition is generated. For example, specifying the following bit lengths generates an error because 4 (number of instances) times 4 (bit length of the corresponding single-instance port) is not 12 bits.

```
module busdriver (busin, bushigh, buslow, enh, enl);
    input [11:0] busin;
    output [7:0] bushigh, buslow;
```

Verilog-XL Reference

Gate and Switch Level Modeling

```
.  
. .  
endmodule
```

Arrays of instances may only be referenced by referencing individual elements of the array. References to the array itself or part-selects of the array are disallowed.

Rule 2

An array of instances must have the ability to be represented as a set of single instances and their port connections. You cannot use expressions that cannot be represented as separate part-select elements (for example, $a+b$).

Rule 3

You can have an unconnected port from an array of instances only if the port is unconnected for all instances of the array.

Rule 4

Range indexes (LHI and RHI) must be local module expressions of constants or parameters that cannot exceed the range of the array. Range index constants include scalar, vector, register, or concatenations of these elements.

Rule 5

Within modules that are instantiated by arrays (or their child modules), `defparam` statements cannot alter any of the parameters that govern their instantiation. These parameters appear within the constant expressions of a range or as part of the port declarations of the instantiated module. Illegally altered parameters generate an error message.

The following code shows an example of illegally altering a parameter where the input port of instance 0 is size 32, but the input port of instance 1 is size 16.

```
module top;  
  ...  
  defparam top.kudos_array[0].p = 32;  
  kudos kudos_array[0:1] (a);  
endmodule  
  
module kudos(in)  
  parameter p = 16;  
  input [0:p]in;  
  ...  
endmodule
```


The following example also shows an illegal use of a `defparam` statement where the number of instances in the array is not able to be determined. The number of instances of `m2` is governed by `top.io.p` whose value is overridden by the `defparam` statement in the last instance of `m2`.

```
module top;
  defparam top.io.io[3].a = 2;
  m1 io();
endmodule

module m1;
  parameter p = 4;
  m2 [1:p] io();
endmodule

module m2;
  parameter a = 4;
  defparam top.io.p = a;
endmodule
```

To illustrate the incorrect logic of this example, the last instance of `m2` is `top.io.io[3]`. But the overriding parameter value is 2 which means that there are only two instances of `m2`, which means that the last instance is `top.io.io[1]` where the parameter value is 4, and so on. This rule prevents this behavior from occurring.

Rule 6

Arrays of instances for macro molecules are treated as normal modules and are not expanded.

Rule 7

System tasks can take modules or gates as arguments as long as only individual instances of an array are referenced and the indexes are constant expressions. The `$showallinstances` system task tallies the instance count for modules that have been instantiated in arrays.

and, nand, nor, or, xor, and xnor Gates

Declarations of these gates begin with one of these keywords:

```
and      nand      nor      or      xor      xnor
```

The delay specification can be zero, one, or two delays. If there is no delay, there is no delay through the gate. One delay specifies the delays for all output transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays applies to transitions to `x`.

Verilog-XL Reference

Gate and Switch Level Modeling

These six gates have one output and one or more inputs. The first terminal in the terminal list connects to the gate's output and all other terminals connect to its inputs. The truth tables for these gates, showing the result of two input values, appear in the following table.

Logic tables for and, or, and xor gates

and	0	1	x	z		or	0	1	x	z		xor	0	1	x	z
0	0	0	0	0		0	0	1	x	x		0	0	1	x	x
1	0	1	x	x		1	1	1	1	1		1	1	0	x	x
x	0	x	x	x		x	x	1	x	x		x	x	x	x	x
z	0	x	x	x		z	x	1	x	x		z	x	x	x	x

Logic tables for nand, nor, and xnor gates

nand	0	1	x	z		nor	0	1	x	z		xnor	0	1	x	z
0	1	1	1	1		0	1	0	x	x		0	1	0	x	x
1	1	0	x	x		1	0	0	0	0		1	0	1	x	x
x	1	x	x	x		x	x	0	x	x		x	x	x	x	x
z	1	x	x	x		z	x	0	x	x		z	x	x	x	x

Versions of the six gates in the previous truth tables having more than two inputs behave identically with cascaded 2-input gates in producing logic results, but the number of inputs does not alter propagation delays. The following example declares a two input and gate. The inputs are `in1` and `in2`. The output is `out`.

```
and (out,in1,in2);
```

buf and not Gates

Declarations of these gates begin with one of the following keywords:

```
buf not
```

The delay specification can be zero, one, or two delays. If there is no delay, there is no delay through the gate. One delay specifies the delays for all output transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays applies to transitions to `x`.

These two gates have one input and one or more outputs. The last terminal in the terminal list connects to the gate's input, and the other terminals connect outputs.

Verilog-XL Reference

Gate and Switch Level Modeling

Truth tables for versions of these gates with one input and one output appear in the following table.

Logic tables for buf and not gates

buf			not	
input	output		input	output
0	0		0	1
1	1		1	0
x	x		x	x
z	x		z	x

The following example declares a two output buf:

```
buf (out1,out2,in);
```

The input is `in`. The outputs are `out1` and `out2`.

bufif1, bufif0, notif1, and notif0 Gates

Declarations of these gates begin with one of the following keywords:

```
bufif0           bufif1           notif1           notif0
```

These four gates model three-state drivers. In addition to values of 1 and 0, these gates output `z`.

In order, after the keyword, you can specify a strength specification, a delay specification, an identifier, and a terminal list. For example:

```
bufif1 (weak1,weak0) #100 bfl (outw,inw,control1);
```

The keyword and terminal list are required; the strength specification, a delay specification, and identifier are optional.

The delay specification can be zero, one, two, or three delays. If there is no delay, there is no delay through the gate. One delay specifies the delay of all transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays specifies the delay of transitions to `x` and `z`. If the specification contains three delays, the first delay determines the rise delay, the second delay determines the fall delay, the third delay determines the delay of transitions to `z`, and the smallest of the three delays applies to transitions to `x`.

Verilog-XL Reference

Gate and Switch Level Modeling

Some combinations of data input values and control input values cause these gates to output either of two values, without a preference for either value. The logic tables of these gates include two symbols representing such unknown results. The symbol `L` represents a result that has a value of 0 or `z`. The symbol `H` represents a result that has a value of 1 or `z`. Delays on transitions to `H` or `L` are the same as delays on transitions to `x`.

These four gates have one output, one data input, and one control input. The first terminal in the terminal list connects to the output, the second connects to the data input, and the third connects to the control input. The following table presents the logic tables for these gates.

Logic tables for `bufif0` and `bufif1` gates

<code>bufif0</code>	CONTROL				
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	x	z	x	x

<code>bufif1</code>	CONTROL				
		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	x	x	x

Logic tables for `notif0` and `notif1` gates

<code>notif0</code>	CONTROL				
		0	1	x	z
D	0	1	z	H	H
A	1	0	z	L	L
T	x	x	z	x	x
A	z	x	z	x	x

<code>notif1</code>	CONTROL				
		0	1	x	z
D	0	z	1	H	H
A	1	z	0	L	L
T	x	z	x	x	x
A	z	z	x	x	x

MOS Switches

Models of MOS networks consist largely of the following four primitive types, which are also the keywords used to declare these gates:

`nmos` `pmos` `rnmos` `rpmos`

The `pmos` keyword stands for PMOS transistor and the `nmos` keyword stands for NMOS transistor. PMOS and NMOS transistors have relatively low impedance between their sources and drains when they conduct. The `rpmos` keyword stands for resistive PMOS transistor and

Verilog-XL Reference

Gate and Switch Level Modeling

the `rnmos` keyword stands for resistive NMOS transistor. Resistive PMOS and resistive NMOS transistors have significantly higher impedance between their sources and drains when they conduct than PMOS and NMOS transistors have. The load devices in static MOS networks are examples of `rpmos` and `rnmos` gates. These four gate types are unidirectional channels for data similar to the `bufif` gates.

A delay specification follows the keyword. The next item is the optional identifier. A terminal list completes the declaration.

The delay specification can be zero, one, two, or three delays. If there is no delay, there is no delay through the switch. A single delay determines the delay of all output transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays specifies the delay of transitions to `z` and `x`. If there are three delays, the first delay specifies the rise delay, the second delay specifies the fall delay, the third delay determines the delay of transitions to `z`, and the smallest of the three delays applies to transitions to `x`. Delays on transitions to `H` and `L` are the same as delays on transitions to `x`.

These four switches have one output, one data input, and one control input. The first terminal in the terminal list connects to the output, the second terminal connects to the data input, and the third terminal connects to the control input.

The `nmos` and `pmos` switches pass signals from their inputs and through their outputs with a change in the signals' strengths in only one case, discussed in [“Strength Reduction by Non-Resistive Devices”](#) on page 131. The `rnmos` and `rpmos` gates reduce the strength of signals that propagate through them, as discussed in [“Strength Reduction by Resistive Devices”](#) on page 131.

Some combinations of data input values and control input values cause these switches to output either of two values, without a preference for either value. The logic tables for these switches include two symbols representing such unknown results. The symbol `L` represents a result which has a value of `0` or `z`. The symbol `H` represents a result that has a value of `1` or `z`.

The following table presents the logic tables for these switches.

Logic tables for `pmos`, `rpmos`, `nmos`, and `rnmos` gates

<code>pmos</code>	CONTROL			
<code>rpmos</code>	0	1	x	z

<code>nmos</code>	CONTROL			
<code>rnmos</code>	0	1	x	z

Verilog-XL Reference

Gate and Switch Level Modeling

D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	z	z	z	z

D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	z	z	z

The following example declares a pmos switch:

```
pmos (out,data,control);
```

The output is `out`, the data input is `data`, and the control input is `control`.

Bidirectional Pass Switches

Declarations of bidirectional switches begin with one of the following keywords:

```
tran          tranif1          tranif0
rtran         rtranif1         rtranif0
```

A delay specification follows the keywords in declarations of `tranif1`, `tranif0`, `rtranif1`, and `rtranif0`; the `tran` and `rtran` devices do not take delays. The next item is the optional identifier. A terminal list completes the declaration.

The following example declares a `tranif1` and a `tran`:

```
tranif1 #100 trf1 (inout1,inout2,control);
tran tr1 (inout1,inout2,control);
```

The bidirectional terminals are `inout1` and `inout2`. The control input is `control`.

The delay specifications for `tranif1`, `tranif0`, `rtranif1`, and `rtranif0` devices can be zero, one, or two delays. If there is no delay, the device has no turn-on or turn-off delay. If the specification contains one delay, that delay determines both turn-on and turn-off delays. If there are two delays, the first delay specifies the turn-on delay, and the second delay specifies the turn-off delay.

The `tranif1`, `tranif0`, `rtranif1`, and `rtranif0` devices have three items in their terminal lists. Two are bidirectional terminals that conduct signals to and from the devices, and the other terminal connects to a control input. The terminals connected to inouts precede the terminal connected to the control input in the terminal list.

The `tranif1` and `rtranif1` devices connect the two bidirectional terminals when their control inputs are driven high and are disconnected when their control inputs are driven low. On the other hand, the `tranif0` and `rtranif0` devices connect the two bidirectional

Verilog-XL Reference

Gate and Switch Level Modeling

terminals when their control inputs are driven low and are disconnected when their control inputs are driven high.

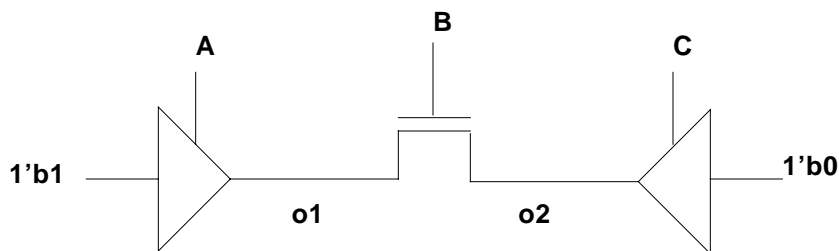
The `tran` and `rtran` devices have terminal lists that contain two bidirectional terminals. These devices connect the two bidirectional terminals.

The bidirectional terminals of all six of these devices connect only to scalar nets or bit-selects of expanded vector nets.

The `tran`, `tranif0`, and `tranif1` devices pass signals with an alteration in their strength in only one case, discussed in [“Strength Reduction by Non-Resistive Devices”](#) on page 131. The `rtran`, `rtranif0`, and `rtranif1` devices reduce the strength of signals passing through them according to rules discussed in [“Ambiguous Strength Signals and Unambiguous Signals”](#) on page 123.

To illustrate the behavior of a bidirectional pass switch, consider the example given in [Figure 6-1](#) on page 111.

Figure 6-1 Behavior of a bidirectional pass switch



The following table defines the behavior of the netlist shown in [Figure 6-1](#) on page 111.

A	B	C	o1	o2
0	0	0	z	z
0	0	1	z	0
0	1	0	z	z
0	1	1	0	0
1	0	0	1	z
1	0	1	1	0
1	1	0	1	1

1	1	1	x	x
---	---	---	---	---

cmos Switches

cmos switches are declared using one of these keywords:

```
cmos                                rcmos
```

The delay specification can be zero, one, two, or three delays. If there is no delay, there is no delay through the switch. A single delay specifies the delay for all transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays is the delay of transitions to z and x.

If the specification contains three delays, the first delay controls rise delays, the second delay controls fall delays, the third delay controls transitions to z, and the smallest of the three delays applies to transitions to x. Delays in transitions to H or L are the same as delays in transitions to x.

cmos and rcmos switches have a data input, a data output, and two control inputs. In the terminal list, the first terminal connects to the data output, the second connects to the data input, the third connects to the n-channel control input, and the last connects to the p-channel control input.

The cmos switch passes signals with an alteration in their strength in only one case, discussed in [“Strength Reduction by Non-Resistive Devices”](#) on page 131. The rcmos switch reduces the strength of signals passing through it according to rules that appear in [“Ambiguous Strength Signals and Unambiguous Signals”](#) on page 123.

The cmos switch is a combination of the pmos switch and the nmos switch whereas the rcmos switch is a combination of the rpmos switch and the rnmos switch. The combined switches in these configurations share data input and data output terminals, but they have separate control inputs. These combined configurations simulate more efficiently than the equivalent networks of two switches.

The equivalence of the cmos switch to the pairing of an nmos switch and a pmos switch is detailed in the following explanation:

```
cmos (w, datain, ncontrol, pcontrol);  
/* the cmos statement above is equivalent to the two statements below */  
nmos (w, datain, ncontrol);  
pmos (w, datain, pcontrol);
```


pullup and pulldown Sources

Declarations of these sources begin with one of the following keywords:

```
pullup    pulldown
```

A strength specification follows the keyword and an optional identifier follows the strength specification. A terminal list completes the declaration. The following example declares a `pullup` instance on net `neta`.

```
pullup pup (neta);
```

You can use a single declaration to define multiple instances. There is no limit to the number of instances you can define in a single declaration. Consider another example that declares two `pullup` instances on the nets `neta` and `netb`.

```
pullup (strong0, strong1) pup (neta), (netb);
```

A `pullup` source places a logic value of 1 on the nets listed in its terminal list. A `pulldown` source places a logic value of 0 on the nets listed in its terminal list. The signals that these sources place on nets have `pull` strength in the absence of a strength specification. There are no delay specifications for these sources because the signals they place on nets continue throughout simulation without variation.

Consider the following netlist:

```
buf buf_1 (out, in, c);  
pullup pullup_1 (out);
```

The table given below describes the behaviour of the sample netlist.

c	in	out
0	0	1
0	1	1
1	0	0
1	1	1

Implicit Net Declarations

Including a previously unused identifier in a terminal list implicitly declares a new net of the `wire` type with zero delay. If the `wire` type is unsuitable for implicitly declared nets, the compiler directive ``default_nettype` can change the type acquired by implicitly declared nets. The following is the directive's syntax:

Verilog-XL Reference

Gate and Switch Level Modeling

```
`default_nettype <type_of_net>
```

The first character in the directive is an accent grave. The `<type_of_net>` can be one of the following net types:

wire	tri	tri0
wand	triand	tri1
wor	trior	trireg

This directive must occur outside of module definitions. All the modules between any two ``default_nettype` directives are affected by the first ``default_nettype` directive. The effect of the directive crosses source file boundaries in the order in which they appear on the command line. The `'resetall` compiler directive ends the effect of a preceding ``default_nettype` directive. A source description can contain any number of these directives. Implicit nets are of type `wire` in the absence of a ``default_nettype` directive.

Each implicitly declared net must connect to one or more of the following:

- gate output
- `tranif` bidirectional terminal
- module output port

If an implicitly declared net does not connect to one of the listed items, the compiler produces an error message with this form:

```
"warning! implicit wire (<name>) has no fanin"
```

If nothing drives a net, Verilog-XL assigns a value of `z` to the net.

Logic Strength Modeling

The Verilog HDL provides for accurate modeling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing, and other technology-dependent network configurations by allowing scalar net signal values to have a full range of unknown values and different levels of strength or combinations of levels of strength. This multiple-level logic strength modeling resolves combinations of signals into known or unknown values to represent the behavior of hardware with maximum accuracy.

A strength specification has two components:

1. the strength of the 0 portion of the net value, designated `<STRENGTH0>` in [“Gate and Switch Declaration Syntax”](#) on page 98.

Verilog-XL Reference

Gate and Switch Level Modeling

- the strength of the 1 portion of the net value, designated `<STRENGTH1>` in [“Gate and Switch Declaration Syntax”](#) on page 98.

Despite this division of the strength specification, it is helpful to consider strength as a property occupying regions of a continuum in order to predict the results of combinations of signals.

The following table demonstrates the continuum of strengths. The left column lists the keywords that specify strength levels of `trireg` or gate output. The middle column shows relative strength levels correlated with the keywords. The abbreviations that Verilog-XL reports are in the right column.

Strength levels for scalar net signal values

Strength Name	Strength Level	Abbreviation
supply0	7	Su0
strong0	6	St0
pull0	5	Pu0
large0	4	La0
weak0	3	We0
medium0	2	Me0
small0	1	Sm0
highz	0	HiZ0
highz1	0	HiZ1
small1	1	Sm1
medium1	2	Me1
weak1	3	We1
large1	4	La1
pull1	5	Pu1
strong1	6	St1

In the preceding table, there are four driving strengths: `supply`, `strong`, `pull`, and `weak`. Signals with driving strengths propagate from gate outputs and continuous assignment outputs.

Verilog-XL Reference

Gate and Switch Level Modeling

There are also three charge storage strengths: `large`, `medium`, and `small`.

Signals with the charge storage strengths originate in the `triereg` net type.

It is possible to think of the strengths of signals in the preceding table as locations on the scale as shown in the following figure:

Scale of strengths

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Discussions of signal combinations later in this document employ graphics similar to this figure.

A net signal can have one or more strength levels associated with it. If a net signal value is known, its strength levels are all in either the 0 strength part of the scale represented by this figure, or they are all in its 1 strength part. If a net signal value is unknown, it has strength levels in both the 0 strength and the 1 strength parts. A signal with a value of `z` has a strength level only in the `HiZ0` or `HiZ1` subdivisions of the scale.

Strengths and Values of Combined Signals

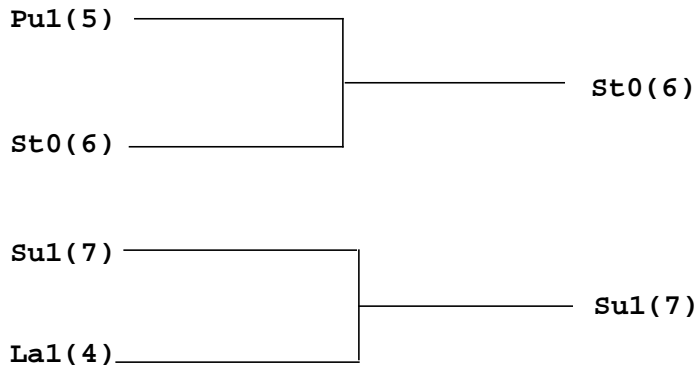
In addition to a value, a signal has either a single unambiguous strength level or it has an ambiguous strength, consisting of more than one level. When signals combine, their strengths and values determine the strength and value of the resulting signal in accord with the principles in the four sections that follow.

Combined Signals of Unambiguous Strength

This section deals with combinations of signals in which each signal has a known value and a single strength level.

If two signals of unequal strength combine in a wired net configuration, the stronger signal is the result. This case appears in the following figure.

Combining unequal strengths



In the Combining unequal strengths figure on page 117, the numbers in parentheses indicate the relative strengths of the signals. The combination of a `pull 1` and a `strong 0` results in a `strong 0`, which is the stronger of the two signals. The combination of two signals of like value results in the same value with the greater of the two strengths. The combination of signals identical in strength and value results in the same signal.

The combination of signals with unlike values and the same strength has three possible results. Two of the results occur in the presence of wired logic and the third occurs in its absence. “Wired Logic Net Types” on page 127 discusses wired logic. The result in the absence of wired logic is the subject of the figure on page 118 in the next section.

Ambiguous Strengths: Sources and Combinations

The classifications of signals possessing ambiguous strengths are the following:

- signals with known values and multiple strength levels
- signals with a value of `x`, which have strength levels consisting of subdivisions of both the strength `1` and the strength `0` parts of the scale of strengths in the Scale of strengths figure on page 116
- signals with a value of `L`, which have strength levels that consist of high impedance joined with strength levels in the `0` strength part of the scale of strengths in the Scale of strengths figure on page 116
- signals with a value of `H`, which have strength levels that consist of high impedance joined with strength levels in the `1` strength part of the scale of strengths in the Scale of strengths figure on page 116

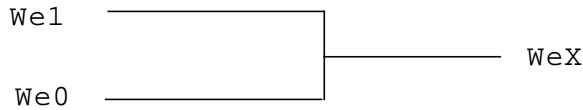
Many configurations can produce signals of ambiguous strength. When two signals of equal strength and opposite value combine, the result has a value of `x` and the strength levels of

Verilog-XL Reference

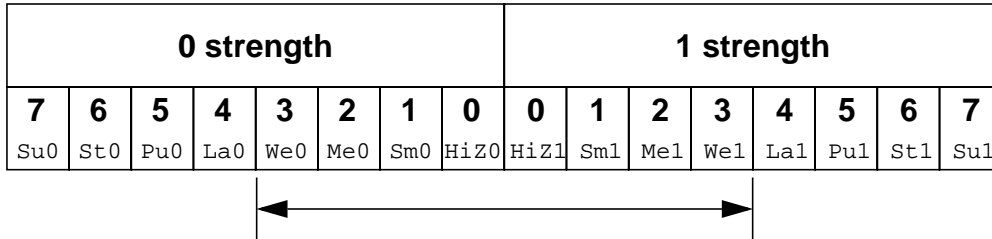
Gate and Switch Level Modeling

both signals and all the smaller strength levels. The following figure shows the combination of a weak signal with a value of 1 and a weak signal with a value of 0 yielding a signal with weak strength and a value of x. The second figure describes the signal.

Combination of signals of equal strength and opposite values

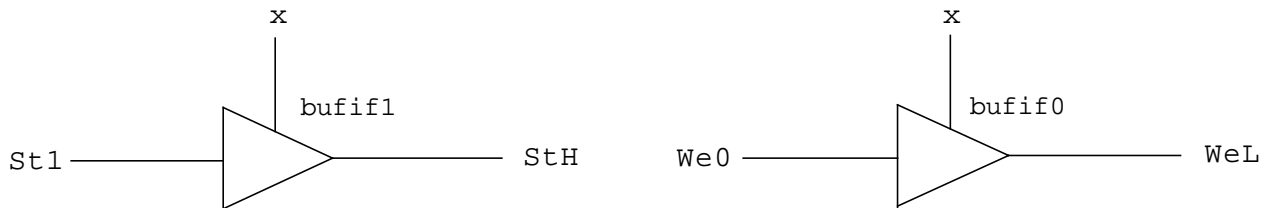


Weak x signal strength



An ambiguous signal strength can be a range of possible values. An example is the strength of the output from the tristate drivers with unknown control inputs in the following figure.

Bufifs with control inputs of x



The output of the `bufif1` in the previous figure is a strong H, composed of the range of values described in the following figure.

Verilog-XL Reference

Gate and Switch Level Modeling

Strong H range of values

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

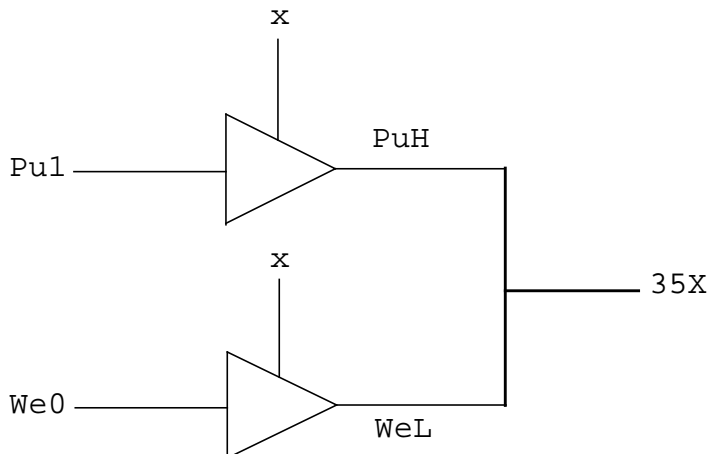
The output of the `bufif0` in is a weak L, composed of the range of values described in the following figure.

Weak L range of values

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

The combination of two signals of ambiguous strength results in a signal of ambiguous strength. The resulting signal has a range of strength levels that includes the strength levels in its component signals. The combination of outputs from two tristate drivers with unknown control inputs, shown in the following figure, is an example.

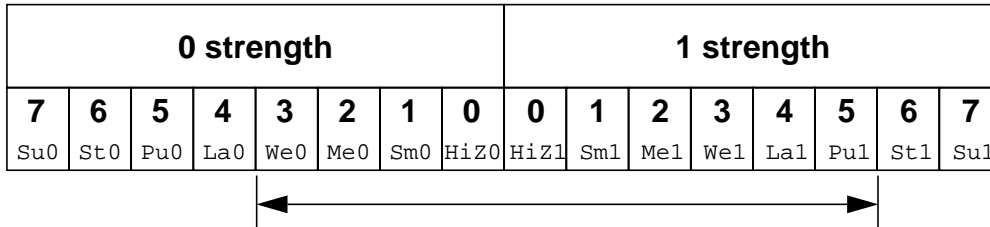
Combined signals of ambiguous strength



Verilog-XL Reference

Gate and Switch Level Modeling

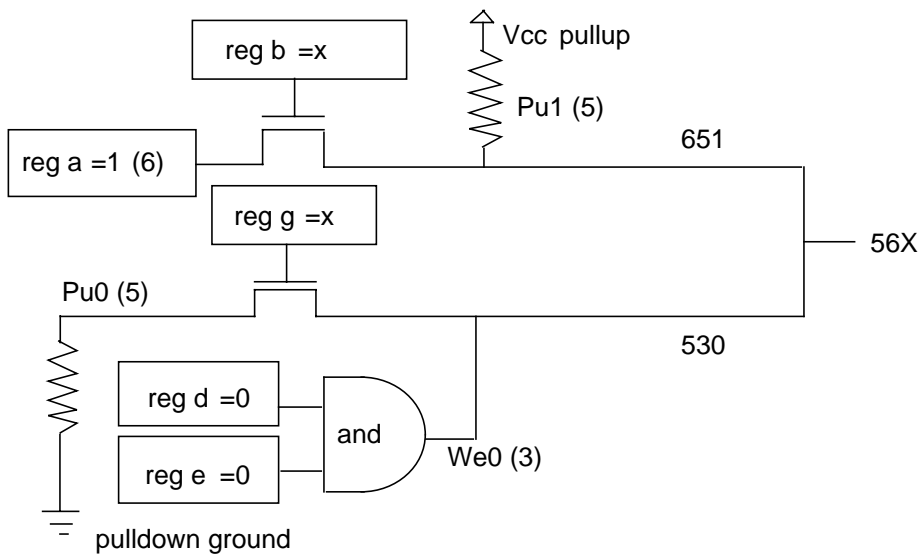
In the Combined signals of ambiguous strength figure on page 119, the combination of signals of ambiguous strengths produces a range which includes the extremes of the signals and all the strengths between them, as described in the following figure



The result is an x because values of both H and L are being driven onto the output net with ambiguous strengths. The number 35, which precedes the x, is a concatenation of two digits. The first is the digit 3, which corresponds to the highest strength level for the result's value of 0. The second digit, 5, corresponds to the highest strength level for the result's value of 1.

Switch networks can produce a range of strengths of the same value, such as the signals from the upper and lower configurations in the following figure.

Ambiguous strengths from switch networks

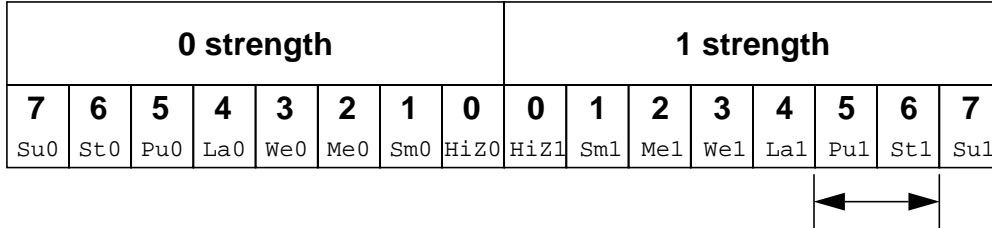


In the Ambiguous strengths from switch networks figure on page 120, the upper combination of a register, a gate controlled by a register of unspecified value, and a pullup produces a signal with a value of 1 and a range of strengths (651) described in the following figure.

Verilog-XL Reference

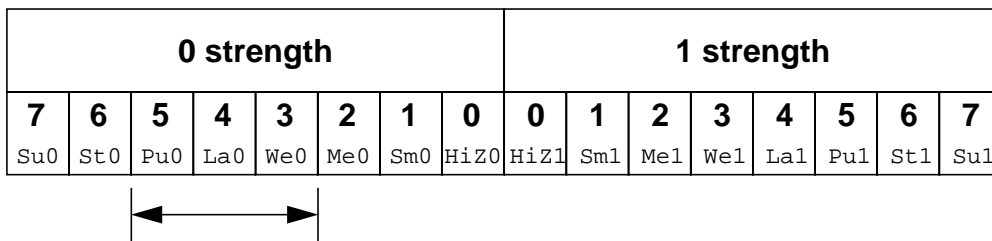
Gate and Switch Level Modeling

Range of two strengths of a defined value



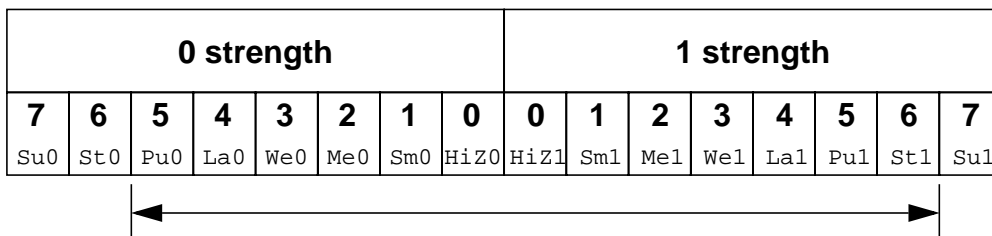
In the Ambiguous strengths from switch networks figure on page 120, the lower combination of a pulldown, a gate controlled by a register of unspecified value, and an and gate produces a signal with a value of 0 and a range of strengths (530) described in the following figure.

Range of three strengths of a defined value



When the signals from the upper and lower configurations in the figure on page 120 combine, the result is an unknown with a range (56X) determined by the extremes of the two signals shown in the following figure.

Unknown value with a range of strengths



In the Ambiguous strengths from switch networks figure on page 120, replacing the pulldown in the lower configuration with a supply0 changes the range of the result to the range (StX) described in the following figure.

Verilog-XL Reference

Gate and Switch Level Modeling

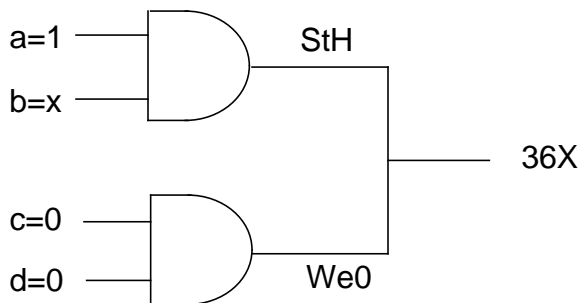
Strong x range

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

The range in the [Strong x range](#) figure on page 122 is strong x, because it is unknown and both of its components' extremes are strong. The extreme of the output of the lower configuration is strong because the lower pmos reduces the strength of the supply0 signal. [“Strength Reduction by Non-Resistive Devices”](#) on page 131 discusses this modeling feature.

Logic gates produce results with ambiguous strengths as well as tristate drivers. Such a case appears in the following figure.

Ambiguous strength from gates



In the previous figure, register b has an unspecified value, so its input to the upper and gate is strong x. The upper and gate has a strength specification including highz0. The signal from the upper and gate is a strong H composed of the values described in the following figure.

Ambiguous strength signal from a gate

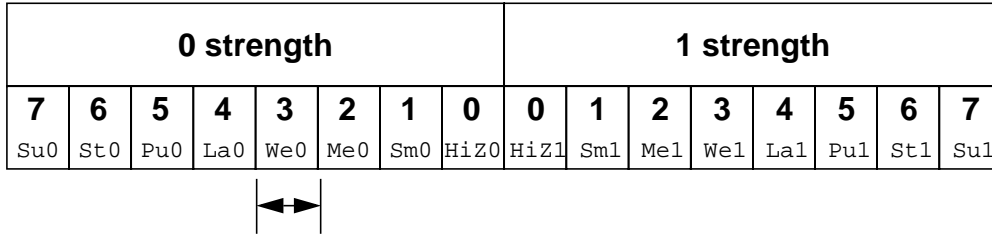
0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Verilog-XL Reference

Gate and Switch Level Modeling

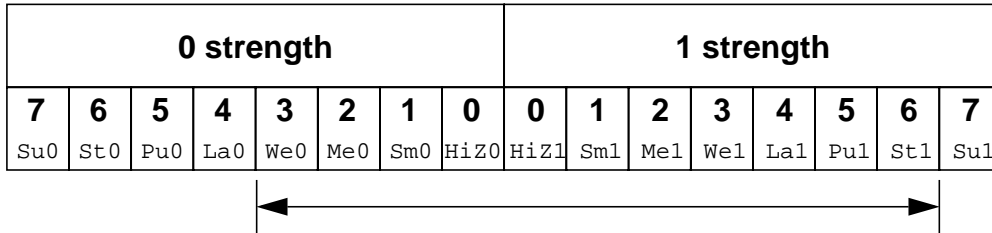
HiZ0 is part of the result because the strength specification for the gate in question specified that strength for an output with a value of 0. A strength specification other than high impedance for the 0 value output results in a gate output of x. The output of the lower and gate is a weak 0 described in the following figure.

Weak 0



When the signals combine, the result is the range (36X) described in the following figure.

Ambiguous strength in combined gate signals



This figure presents the combination of an ambiguous signal and an unambiguous signal. Such combinations are the topic of the following section.

Ambiguous Strength Signals and Unambiguous Signals

The combination of a signal with unambiguous strength and known value with another signal of ambiguous strength presents several possible cases. To understand a set of rules governing this type of combination, it is necessary to consider the strength levels of the ambiguous strength signal separately from each other and relative to the unambiguous strength signal. When a signal of known value and unambiguous strength combines with a component of a signal of ambiguous strength, these are the effects:

Verilog-XL Reference

Gate and Switch Level Modeling

Rule 1

The strength levels of the ambiguous strength signal that are greater than the strength level of the unambiguous signal remain in the result.

Rule 2

The strength levels of the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous signal disappear from the result, subject to [Rule 3](#).

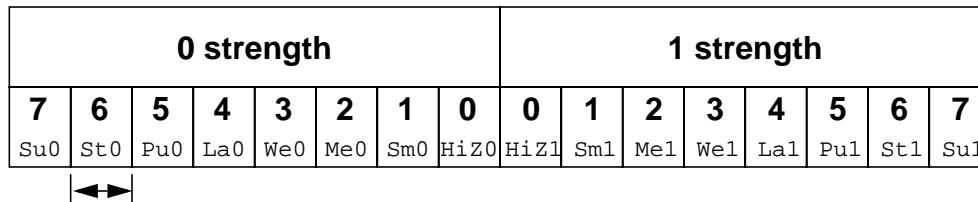
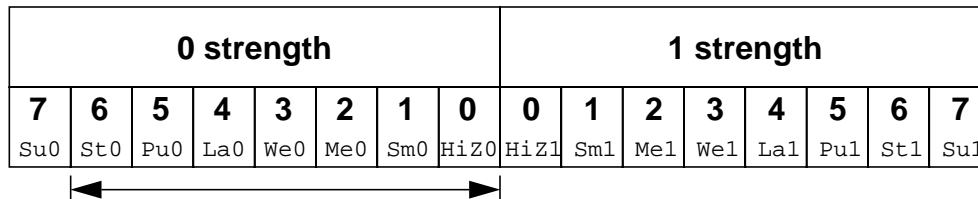
Rule 3

If the operation of Rule 1 and Rule 2 results in a gap in strength levels because the signals are of opposite value, the signals in the gap are part of the result.

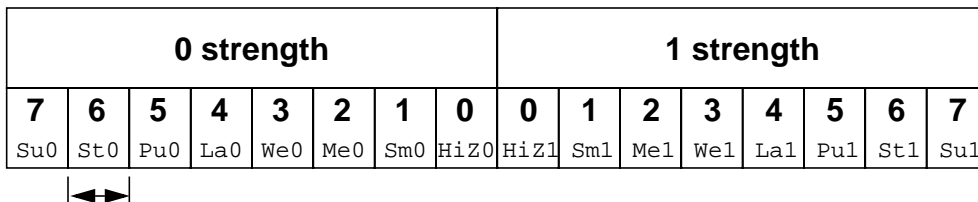
The following figures show some applications of the rules.

In the following figure, the strength levels in the ambiguous strength signal that are smaller than or equal to the strength level of the unambiguous strength signal disappear from the result, demonstrating [Rule 2](#).

Elimination of strength levels



Combining the two signals above results in the following signal:

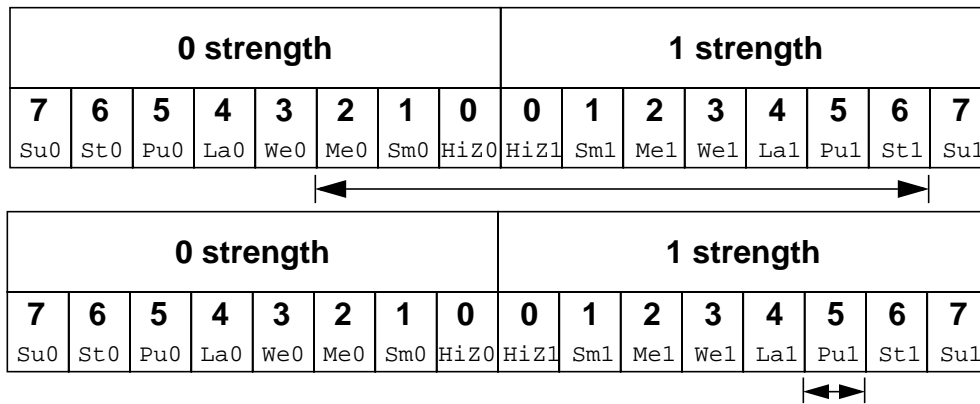


Verilog-XL Reference

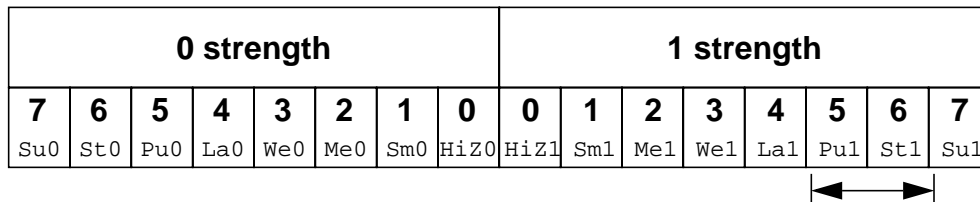
Gate and Switch Level Modeling

In the following figure, Rule 1, Rule 2, and Rule 3 apply. The strength levels of the ambiguous strength signal that are of opposite value and lesser strength than the unambiguous strength signal disappear from the result. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal, and of the same value, disappear from the result. The strength level of the unambiguous strength signal and the greater extreme of the ambiguous strength signal define a range in the result.

Result demonstrating a range and the elimination of strength levels of two values



Combining the two signals above results in the following signal:



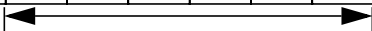
In the following figure, Rule 1 and Rule 2 apply. The strength levels in the ambiguous strength signal that are less than the strength level of the unambiguous strength signal disappear from the result. The strength level of the unambiguous strength signal and the strength level at the greater extreme of the ambiguous strength signal define a range in the result.

Verilog-XL Reference


Gate and Switch Level Modeling

Result demonstrating a range and the elimination of strength levels of one value

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

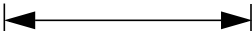


0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1



Combining the two signals above results in the following signal:

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

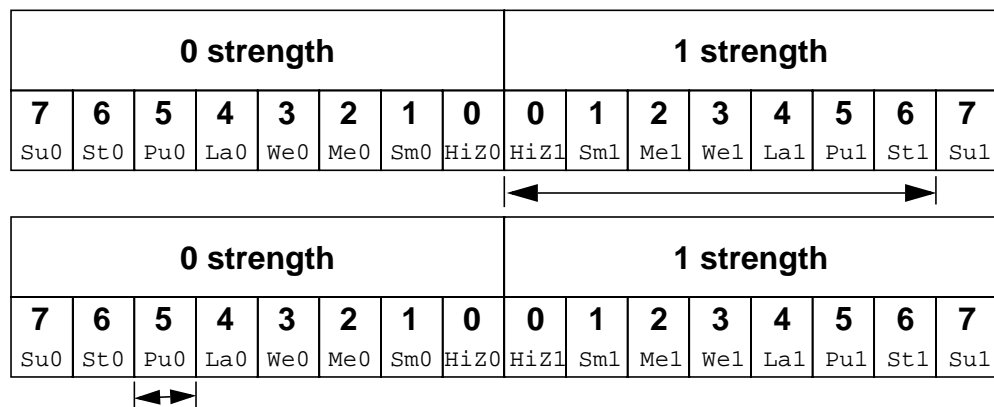


In the next figure, [Rule 1](#), [Rule 2](#), and [Rule 3](#) apply. The greater extreme of the range of strengths for the ambiguous strength signal is larger than the strength level of the unambiguous strength signal. The result is a range defined by the greatest strength in the range of the ambiguous strength signal and by the strength level of the unambiguous strength signal.

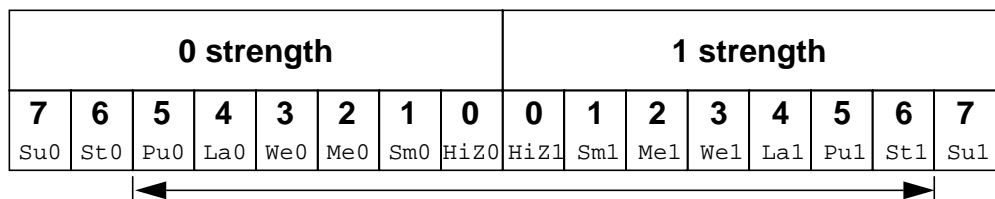
Verilog-XL Reference

Gate and Switch Level Modeling

A range of both values



The combination of the two signals above produces the following result:



Wired Logic Net Types

The net types `triand`, `wand`, `trior`, and `wor` resolve conflicts when multiple drivers are at the same level of strength. These net types resolve signal values by treating signals as inputs of logic functions.

For example, consider the combination of two signals of unambiguous strength in the following figure.

Verilog-XL Reference

Gate and Switch Level Modeling

Wired logic with unambiguous strength signals

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

wired AND logic value result : 0
wired OR logic value result : 1

The combination of the signals in this figure, using wired AND logic, produces a result with the same value as the result produced by an AND gate with the two signals' values as its inputs. The combination of signals using wired OR logic produces a result with the same value as the result produced by an OR gate with the two signals' values as its inputs. The strength of the result is the same as the strength of the combined signals in both cases. If the value of the upper signal changes so that both signals possess a value of 1, then the results of both types of logic have a value of 1.

When ambiguous strength signals combine in wired logic, it is necessary to consider the results of all combinations of each of the strength levels in the first signal with each of the strength levels in the second signal, as shown in the following figure.

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Signal 1

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Signal 2

Verilog-XL Reference

Gate and Switch Level Modeling

The combinations of strength levels for AND logic appear in the following table:

Signal 1		Signal 2		Signal 3	
Strength	Value	Strength	Value	Strength	Value
5	0	5	1	5	0
6	0	5	1	6	0

The result is the following signal:

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

|<—>|

The combinations of strength levels for OR logic appear in the following table:

Signal 1		Signal 2		Signal 3	
Strength	Value	Strength	Value	Strength	Value
5	0	5	1	5	1
6	0	5	1	6	0

The result is the following signal:

0 strength								1 strength							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

|<—————>|

Strength Resolution for Continuous Assignments

In continuous assignment statements, if the strength value is not specified, the logic value of the LHS (Left Hand Side) is the resolution of the logic values of the RHS (Right Hand Side), with default strength. However, if the strength value is specified, then the strength value of the LHS will depend on the logic value of the RHS, as explained in the following example:

Verilog-XL Reference

Gate and Switch Level Modeling

Example

```
//test.v

module top;
  wire a,b;
  wire c;

  assign (pull1,strong0) a = 1'b0;////////(1)
  assign (pull1,supply0) b =1'b1;////////(2)

  assign #2 c = b;////////(3)
  assign #1 c = a;////////(4)

  initial
  begin
    $monitor($time,"value of c is %v",c);
  end

endmodule
```

In the above example, *a* has a logic value of ZERO (0) with strength *St0* and *b* has the logic value of ONE (1) with strength *St1*. Refer assignment statements (1) and (2) above. The strength information will not pass to the assignment statement. That is, the strength values specified in *a* and *b* in statements (3) and (4), will not get carried over to *c* after the assignment. As *c* is driven by both *a* and *b*, the resolved value of *c* is the resolution between the logic values of *a* and *b*. In the resolution, the strengths of both *a* and *b* will not be considered.

Therefore, as the strength value is not specified in statements (3) and (4), *c* will have the resolved logic value of *a* and *b* with default strength - *St0*, *St1*, or *StX* depending upon its logic value.

However, if we replace statements (3) and (4) with the following:

```
assign (pull1, supply0) #2 c = b;
assign (pull1, strong0) #1 c = a;
```

Then, the strength value of *c* will depend on the logic value of the RHS, which in this example will be *St0*.

Mnemonic Format

Trace messages giving signal strength information are compatible with the %v format option in the \$display, \$write, \$strobe, \$monitor system tasks. See “[Strength Format](#)” on page 338 of the Verilog-XL User Guide for more information on this mnemonic strength notation.

Strength Reduction by Non-Resistive Devices

The `nmos`, `pmos`, and `cmos` gates pass through the strength from the data input to the output, except that a supply strength is reduced to a strong strength.

The `tran`, `tranif0`, and `tranif1` gates do not affect signal strength across the bidirectional terminals, except that a supply strength is reduced to a strong strength.

Strength Reduction by Resistive Devices

The `rnmos`, `rpmos`, `rcmos`, `rtran`, `rtranif1`, and `rtranif0` devices reduce the strength of signals that pass through them according to the following table:

input strength	reduced strength
supply drive	pull drive
strong drive	pull drive
pull drive	weak drive
weak drive	medium capacitor
large capacitor	medium capacitor
medium capacitor	small capacitor
small capacitor	small capacitor
high impedance	high impedance

Strengths of Net Types

The `tri0`, `tri1`, `supply0`, and `supply1` net types generate signals with specific strength levels. The `triereg` declaration can specify either of two signal strength levels other than a default strength level.

tri0 and tri1 Net Strengths

The `tri0` net type models a net connected to a resistive pulldown device. Its signal has a value of 0 and a pull strength in the absence of an overriding source. The `tri1` net type models a net connected to a resistive pullup device: its signal has a value of 1 and a pull strength in the absence of an overriding source.

triereg Strength

The `triereg` net type models charge storage nodes. The strength of the drive resulting from a `triereg` net that is in the charge storage state (that is, a driver charged the net and then went to high impedance) is one of these three strengths: `large`, `medium`, or `small`. The specific strength associated with a particular `triereg` net is specified by the user in the net declaration. The default is `medium`. The syntax of this specification is described in [“Charge Strength”](#) on page 39.

supply0 and supply1 Net Strengths

The `supply0` net type models ground connections. The `supply1` net type models connections to power supplies. The `supply0` and `supply1` net types have `supply` driving strengths.

Gate and Net Delays

Gate and net delays provide a means of accurately describing delays through a circuit. The gate delays specify the signal propagation delay from any gate input to the gate output. Up to three values per output can be specified. The descriptions in this chapter of each gate type give the rules for determining how many delays gates can take.

Net delays refer to the time it takes from any driver on the net changing value to the time when the net value is updated and propagated further. Up to three delay values per net can be specified.

Note: Verilog-XL treats two nets connected by a bidirectional switch as one net and simulates the delays on both nets in parallel.

For both gates and nets, the default delay is zero when no delay specification is given. When one delay value is given, then this value is used for all propagation delays associated with the gate or net. The following is an example of a delay specification with one delay:

```
and #(10) (out, in1, in2);
```

The following is an example of a delay specification with two delays:

```
and #(10, 12) (out, in1, in2);
```

When two delays are given, the first specifies the rise delay and the second specifies the fall delay. The delay when the signal changes to high impedance or to unknown is the lesser of the two delay values.

The following is an example of a delay specification with three delays:

Verilog-XL Reference

Gate and Switch Level Modeling

```
bufif1 #(10, 12, 11) (out, in1, in2);
```

For a three delay specification:

- the first delay refers to the transition to the 1 value (rise delay)
- the second delay refers to the transition to the 0 value (fall delay)
- the third delay refers to the transition to the high-impedance value

When a value changes to the unknown (x) value, the delay is the smallest of the three delays.

When an entire vector net changes value the selection of which of its net delays controls the transition depends on whether Verilog-XL treats the net as a one entity or as a group of scalar nets. The default is that the net is one entity and the selection of net delays depends on the state or transition of its least significant bit. A net with this status is an unexpanded net.

Declaring the net with the keyword `scalared` or requiring Verilog-XL to access its individual bits changes the default behavior by making the net act like a group of scalars. A net with this status is an expanded net. Expanding a net results in a separate delay selection for each transition of each bit. These separate delay selections follow the rules given previously in this section.

Selection of the net delay for an unexpanded net follows these rules:

- If the right-hand side LSB remains 0 or becomes 0, then the falling (second) delay is used.
- If the right-hand side LSB remains z or becomes z , then the turn-off (third) delay is used.
- If the right-hand side LSB remains a 1 or becomes 1, then the rising (first) delay is used.
- If the right-hand side LSB is an x or becomes an x , then the smallest of the delay values is used.

The following examples show the effects on a simulation of a continuous assignment causing transitions on a vector net with net delays, with and without declaring the `scalared` keyword.

Selection of rise and fall delay on a vector net declared without `scalared`

```
module top;
  reg [3:0] a;
  wire [3:0] #(5,20) b;
  assign b=a;

  initial
  begin
    a = 'b0000;
    #100 a = 'b1101;
    #100 a = 'b0111;
  end
endmodule
```

Verilog-XL Reference

Gate and Switch Level Modeling

```
        #100 a = 'b1110;
    end
    initial
    begin
        $monitor($time, , "a=%b, b=%b", a, b);
        #1000 $finish;
    end
endmodule
```

The results of the preceding simulation follow:

```
0 a=0000, b=xxxx
20 a=0000, b=0000
100 a=1101, b=0000
105 a=1101, b=1101
200 a=0111, b=1101
205 a=0111, b=0111
300 a=1110, b=0111
320 a=1110, b=1110
```

The following example (with `scalared`) shows a value for `b` at time 205 that differs from the results in the previous example (without `scalared`) because it shows only the rising transition of bit [1].

A falling transition in bit [3] occurs at time 220 in the following example, but occurs at time 205 in the previous example, where the value of 1 on bit [0] controls the delay.

At time 305, the following example shows a separate rising transition on bit [3] that is delayed until time 320 in previous example, where its delay is controlled by a falling transition on bit [0].

Selection of rise and fall delay on a vector net declared with `scalared`

```
module top;
    reg [3:0] a;
    wire scalared [3:0] #(5,20) b;
    assign b=a;
    initial
    begin
        a = 'b0000;
        #100 a = 'b1101;
        #100 a = 'b0111;
        #100 a = 'b1110;
    end
    initial
    begin
        $monitor($time, , "a=%b, b=%b", a, b);
        #1000 $finish;
    end
endmodule
```

The results of the preceding simulation follow:

```
0 a=0000, b=xxxx
20 a=0000, b=0000
```

Verilog-XL Reference

Gate and Switch Level Modeling

```
100 a=1101, b=0000
105 a=1101, b=1101
200 a=0111, b=1101
205 a=0111, b=1111
220 a=0111, b=0111
300 a=1110, b=0111
305 a=1110, b=1111
320 a=1110, b=1110
```

The following table summarizes the from-to propagation delay choice for the two and three delay specifications.

Rules for propagation delays

from value	to value	delay used if there are	
		2 delays	3 delays
0	1	d1	d1
0	x	min(d1,d2)	min(d1,d2,d3)
0	z	min(d1,d2)	d3
1	0	d2	d2
1	x	min(d1,d2)	min(d1,d2,d3)
1	z	min(d1,d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1,d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1,d2)	min(d1,d2,d3)

The following example specifies a simple latch module with tri-state outputs, where individual delays are given to the gates. The propagation delay from the primary inputs to the outputs of the module will be cumulative, and depends on the signal path through the network.

```
module tri_latch(qout, nqout, clock, data, enable);
    output qout, nqout;
    input clock, data, enable;

    tri qout, nqout;
    not #5
        (ndata, data);
    nand #(3, 5)
```

Verilog-XL Reference

Gate and Switch Level Modeling

```
        (wa, data, clock),
        (wb, ndata, clock);
nand #(12, 15)
    (q, nq, wa),
    (nq, q, wb);
bufif1 #(3, 7, 13)
    q_drive (qout, q, enable),
    nq_drive (nqout, nq, enable);
endmodule
```

min/typ/max Delays

The syntax for delays on gate primitives (including user-defined primitives), nets, and continuous assignments allows three values each for the rising, falling, and turn-off delays. The minimum, typical, and maximum values for each are specified as constant expressions separated by colons.

The following example shows the minimum, typical, and maximum values for rising, falling, and turn-off delays:

```
module iobuf(io1, io2, dir);
    .
    .
    .
    bufif0 #(5:7:9, 8:10:12, 15:18:21) (io1, io2, dir);
    bufif1 #(6:8:10, 5:7:9, 13:17:19) (io2, io1, dir);
    .
    .
    .
endmodule
```

The syntax for delay controls in procedural statements also allows minimum, typical, and maximum values. These are specified by expressions separated by colons. The following example illustrates this concept.

```
parameter
    min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;
always
    begin
        #(95:100:105) clk = 1;
        #(min_hi:typ_hi:max_hi) clk = 0;
    end
```

The delay used during simulation will be one of the three—either minimum, typical, or maximum. One delay choice is used throughout a simulation run; it cannot be changed dynamically.

Selection of which delays will be used is done using one of three command options. The `+maxdelays` option selects all of the maximum delays; the `+typdelays` option selects all of the typical delays; the `+mindelays` option selects all of the minimum delays. For example,

the following command line runs Verilog-XL with only the values specified for the maximum delay:

```
verilog source1.v +maxdelays
```

Note: If only one delay is specified, then Verilog-XL uses it regardless of whether minimum, typical, or maximum delays are selected. If more than one delay is desired, then all three delays must be specified; for example, it is not possible to specify minimum and maximum without typical.



Caution

There is currently no syntax checking on plus command options. Be very careful in specifying them to avoid confusing results. If you misspell “maxdelays”, “mindelays” or “typdelays”, the option will be ignored.

triereg Net Charge Decay

Like all nets, a `triereg` declaration's delay specification can contain up to three delays. The first two delays specify the simulation time that elapses in a transition to the 1 and 0 logic states when the `triereg` is driven to these states by a driver. The third delay specifies the charge decay time instead of the time that elapses in a transition to the z logic state. The charge decay time specifies the simulation time that elapses between when a `triereg`'s drivers turn off and when its stored charge can no longer be determined.

A `triereg` needs no turn-off delay specification because a `triereg` never makes a transition to the z logic state. When a `triereg`'s drivers make transitions from the 1, 0, or x logic states to off, the `triereg` retains the previous 1, 0, or x logic state that was on its drivers. The z value does not propagate from a `triereg`'s drivers to a `triereg`. A `triereg` can only hold a z logic state when z is the `triereg`'s initial logic state or when it is forced to the z state with a `force` statement.

A delay specification for charge decay models a charge storage node that is not ideal; a charge storage node whose charge leaks out through its surrounding devices and connections.

This section describes the charge decay process and the delay specification for charge decay.

The charge decay process

Charge decay is the cause of transition of a 1 or 0 that is stored in a `triereg` to an unknown value (x) after a specified number of time units. The charge decay time is that specified number of time units.

The charge decay process begins when the `triereg`'s drivers turn off and the `triereg` starts to hold charge. The charge decay process ends under the following two conditions:

1. The specified number of time units elapse and the `triereg` makes a transition from 1 or 0 to x.
2. The `triereg`'s drivers turn on and propagate a 1, 0 or x into the `triereg`.

When charge decay causes a `triereg`'s value to change to x, Verilog-XL issues a warning message such as the following:

```
Warning! Time = simulation_time:
  Charge on node hierarchical_name_of_triereg has
  decayed [Verilog-DECAY]
  "source_file_name", line_number: triereg_identifier
```

You can tell Verilog-XL not to issue this warning with the `$disable_warnings` system task.

The delay specification for charge decay time

The third delay in a `triereg` declaration specifies the charge decay time. A three-valued delay specification in a `triereg` declaration has the following form:

```
 #(d1, d2, d3)
// three delays -
// (rising_delay, falling_delay, charge_decay_time)
```

The specification in a `triereg` declaration of the charge decay time must be preceded by a rise and fall delay specification. The following example shows a specification of the charge decay time in a `triereg` declaration:

```
triereg (large) #(0,0,50) cap1;
```

This example declares a `triereg` with the identifier `cap1`. This `triereg` stores a large charge. The delay specifications for the rise delay is 0, the fall delay is 0, and the charge decay time specification is 50 time units.

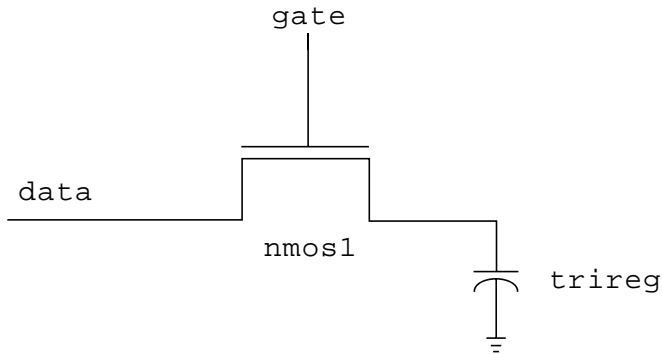
Note: A charge decay time is not a propagation delay like a rising delay or a falling delay. A charge decay time greater than 0 does not prevent the acceleration of the `triereg`.

The following figure and source description file contains a `triereg` declaration with a charge decay time specification.

Verilog-XL Reference

Gate and Switch Level Modeling

trireg with a charge decay



```
module capacitor;
reg data,gate;
trireg (large) #(0,0,50) cap1; // trireg declaration with a
                               // charge decay time of
                               // 50 time units
nmos nmos1 (cap1,data,gate); // nmos switch that drives the trireg
initial
begin
    $monitor("%0d data = %v gate = %v cap1 = %v",
            $time,data,gate,cap1);
    data = 1;
    gate = 1;
    #10 gate = 0; // <-- toggles the driver of the
    #30 gate = 1; // <-- control input to the
    #10 gate = 0; // <-- nmos switch
    #100 $finish;
end
endmodule
```

The following shows the simulation results of the model in the [trireg with a charge decay](#) figure on page 139.

```
0 data = St1 gate = St1 cap1 = St1
10 data = St1 gate = St0 cap1 = La1
40 data = St1 gate = St1 cap1 = St1
50 data = St1 gate = St0 cap1 = La1
```

```
Warning! Time = 100: Charge on node capacitor.cap1 has
           decayed [Verilog-DECAY]
           "triregl.v", 4: cap1
100 data = St1 gate = St0 cap1 = LaX
```

The last line shows that the `trireg cap1` changes value to `LaX` at simulation time 100.

The results show the following sequence of events:

1. At simulation time 0, data drives a strong 1 into trireg cap1.
2. At simulation time 10, gate's value changes to 0, disconnecting trireg cap1 from data; trireg cap1 enters the capacitive state, storing its value of 1 with a large strength.

The charge decay process begins for `triereg cap1`; its value is scheduled to change to `x` at simulation time 60.

3. At simulation time 40, gate's value changes to 1, connecting `triereg cap1` to `data`; `triereg cap1` enters the driven state, and `data` drives a strong 1 into `triereg cap1`. The charge decay process stops for `triereg cap1` because it is no longer in the capacitive state.
4. At simulation time 50, `reg gate`'s value changes to 0, disconnecting `triereg cap1` from `reg data` again; `triereg cap1` enters the capacitive state, storing its value of 1 with a large strength. The charge decay process begins again for `triereg cap1`; its value is scheduled to change to `x` at simulation time 100.
5. At simulation time 100, the charge decay process changes the stored value in `triereg cap1` from 1 to `x`.

Note: Specifying a charge decay time can affect performance. You may see a performance degradation caused by specifying `triereg` charge decay time in a design—such as a dynamic circuit, whose `triereg`s frequently enter the capacitive state.

Gate and Net Name Removal

Four compiler directives have been provided that control the removal of gate and/or net names to reduce the virtual memory requirements at the gate and switch level. The names are removed from the second module and all subsequent module instances so that removing gate and net names saves the most memory in designs containing gate-level modules that are instantiated many times.

The compiler directives are the following:

```
`remove_gatenames
`noremove_gatenames
`remove_netnames
`noremove_netnames
```

The first two directives control the removal of gate names, and the latter two control the removal of net names. For both controls, the default is to NOT remove the names.

These directives can only be specified outside modules. The control applies to all modules following a directive until the end of the source description (going across source files if necessary) or until another of these directives is given or until a ``resetall` directive is given. Any number of these compiler directives can be given in a source description.

The removal of gate names is more useful than the removal of net names because gate names at the present are used only for the tracing of value changes across the gates.

Verilog-XL Reference

Gate and Switch Level Modeling

Net names cannot be removed if they have been referenced in a hierarchical name. An example of a hierarchical referencing is a monitoring task, or nets that need to be referenced interactively. Another example of hierarchical referencing is named port connections.

As shown in the following partial description, all gate names from modules `a` and `b`, and net names from all the instances of module `b` are removed.

```
...
`remove_gatenames
module a;
    ...
    b b1(), b2(), b3();
    ...
endmodule
`remove_netnames
module b;
    ...
    c c1(), c2();
    ...
endmodule
`noremove_gatenames
`noremove_netnames
module c;
    ...
endmodule
```

Note that it is not possible to selectively remove the gate and/or net names from particular instances of a module.

Verilog-XL Reference
Gate and Switch Level Modeling

User-Defined Primitives (UDPs)

This chapter describes the following:

- [Overview](#) on page 143
- [UDP Syntax](#) on page 144
- [UDP Definition](#) on page 145
- [Summary of UDP Symbols](#) on page 148
- [Combinational UDPs](#) on page 148
- [Level-Sensitive Sequential UDPs](#) on page 150
- [Edge-Sensitive UDPs](#) on page 150
- [Sequential UDP Initialization](#) on page 151
- [Mixing Level-Sensitive and Edge-Sensitive Descriptions](#) on page 154
- [Level-Sensitive Dominance](#) on page 155
- [UDP Instances](#) on page 156
- [Compilation](#) on page 157
- [Reducing Pessimism](#) on page 158
- [Processing of Simultaneous Input Changes](#) on page 159
- [Memory Usage and Performance Considerations](#) on page 160
- [UDP Examples](#) on page 161

Overview

This chapter describes how to extend the set of gate-level primitives provided with Verilog-XL by designing and specifying new primitive elements called user-defined primitives (UDPs).

You can write both combinational and sequential UDPs in a way that is similar to a truth table enumeration of a logic function. You can then instantiate UDP definitions in the same way as gate primitives. This technique can reduce the amount of memory that a description needs and can improve simulation performance. Evaluation of UDPs is accelerated by the Verilog-XL algorithm.

UDP Syntax

The formal syntax of the UDP definition is as follows.

Syntax for user-defined primitives

```
<UDP>
 ::= primitive <name_of_UDP> ( <output_terminal_name> ,
   <input_terminal_name> <,<input_terminal_name>>* ) ;
   <UDP_declaration>+
   <UDP_initial_statement>?
   <table_definition>
   endprimitive

<name_of_UDP>
 ::= <IDENTIFIER>

<UDP_declaration>
 ::= <UDP_output_declaration>
   || = <reg_declaration>
   || = <UDP_input_declaration>

<UDP_output_declaration>
 ::= output <output_terminal_name>;

<reg_declaration>
 ::= reg <output_terminal_name> ;

<UDP_input_declaration>
 ::= input <input_terminal_name> <,<input_terminal_name>>* ;

<UDP_initial_statement>
 ::= initial <output_terminal_name> = <init_val> ;

<init_val>
 ::= 1'b0
   || = 1'b1
   || = 1'bx
   || = 1
   || = 0

<table_definition>
 ::= table
   <table_entries>
   endtable

<table_entries>
 ::= <combinational_entry>+
   || = <sequential_entry>+

<combinational_entry>
 ::= <level_input_list> : <OUTPUT_SYMBOL> ;

<sequential_entry>
 ::= <input_list> : <state> : <next_state> ;
```


Verilog-XL Reference

User-Defined Primitives (UDPs)

```
<input_list>
  ::= <level_input_list>
     || = <edge_input_list>
<level_input_list>
  ::= <LEVEL_SYMBOL>+
<edge_input_list>
  ::= <LEVEL_SYMBOL>* <edge> <LEVEL_SYMBOL>*
<edge>
  ::= ( <LEVEL_SYMBOL> <LEVEL_SYMBOL> )
     || = <EDGE_SYMBOL>
<state>
  ::= <LEVEL_SYMBOL>
<next_state>
  ::= <OUTPUT_SYMBOL>
     || = -
```

(This is a literal hyphen — see [“Memory Usage and Performance Considerations”](#) on page 160 for more details.)

Lexical tokens:

```
<OUTPUT_SYMBOL> is one of the following:
  0  1  x  X
<LEVEL_SYMBOL> is one of the following:
  0  1  x  X  ?  b  B
<EDGE_SYMBOL> is one of the following:
  r  R  f  F  p  P  n  N  *
```

UDP Definition

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are used inside a module. They *cannot* appear between the keywords `module` and `endmodule`. The maximum number of UDP definitions that you can use in a simulation is 240.

A UDP definition begins with the keyword `primitive` followed by the name of the UDP. After the UDP name, specify a comma-separated list of terminals enclosed in parentheses. Follow this header with the terminal declarations and a state table. Terminate the UDP definition with the keyword `endprimitive`.

For example:

```
primitive and_or(out, a1,a2,a3, b1,b2);
  output out;
  input a1,a2,a3, b1,b2;
  table
//state table information goes here
...
```

```
    endtable  
endprimitive
```

UDP Terminals

UDPs can have multiple input terminals, but only one output terminal. They cannot have bidirectional inout terminals. All UDP terminals are scalar. No vector terminals are allowed.

The maximum number of inputs to a combinational UDP is ten. The maximum number of inputs to a sequential UDP is limited to nine because the internal state counts as an input.

Only logic values of 0, 1, or x are allowed on input and output. The tri-state value z is not supported.

The output terminal must be the first terminal in the terminal list.

The output terminal of a sequential UDP requires an additional declaration as type `reg`. It is illegal to declare a `reg` for the output terminal of a combinational UDP.

UDP Declarations

UDPs must contain input and output terminal declarations. The output terminal declaration begins with the keyword `output`, followed by one output terminal name. The input terminal declaration begins with the keyword `input`, followed by one or more input terminal names.

Sequential UDPs must contain a `reg` declaration for the output terminal. Combinational UDPs cannot contain a `reg` declaration. You can specify the initial value of the output terminal `reg` in an `initial` statement in a sequential UDP.

Sequential UDP initial Statement

The sequential UDP `initial` statement specifies the value of the output terminal when simulation begins. This statement begins with the keyword `initial`. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal `reg`. See [“Sequential UDP Initialization”](#) on page 151 for more information.

UDP State Table

The state table that defines the behavior of a UDP begins with the keyword `table` and ends with the keyword `endtable`.

Verilog-XL Reference

User-Defined Primitives (UDPs)

Each row of the table is created using a variety of characters that indicate input and output states. Three states—0, 1, and *x*—are supported. The *z* state is not supported. There are a number of special characters you can use to represent certain combinations of state possibilities. These are listed in [“Summary of UDP Symbols”](#) on page 148.

Combinational UDPs have one field per input and one field for the output. Use a colon to separate the input fields from the output field. Each row of the table is terminated by a semicolon. For example, the following state table entry specifies that when the three inputs are all 0, the output is 0.

```
table
0 0 0 : 0;
...
endtable
```

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons. For example:

```
table
0 0 0 : 0 : 0;
...
endtable
```

The order of the inputs in the state table description must correspond to the order of the inputs in the port list in the UDP definition header. It is not related to the order of the input declarations.

Each row in the table defines the output for a particular combination of input states. If all inputs are specified as *x*, then the output must be specified as *x*. All combinations that are not explicitly specified result in a default output state of *x*.

Consider the following entry from a UDP state table:

```
0 1 : ? : 1 ;
```

In this entry, the *?* represents a don't-care condition. This symbol indicates iterative substitution of 1, 0, and *x*. The table entry specifies that when the inputs are 0 and 1, the output is 1 no matter what the value of the current state is.

You do not have to explicitly specify every possible input combination. All combinations that are not explicitly specified result in a default output state of *x*.

It is illegal to have the same combination of inputs, including edges, specified for different outputs.

Summary of UDP Symbols

Like the ? symbol described in the preceding section, there are several symbols that you can use in UDP definitions to make the description more readable. The following table summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Table 7-1 UDP Table Symbols

Symbol	Interpretation	Notes
0	Logic 0	
1	Logic 1	
x	Unknown	
?	Iteration of 0, 1, and x	Cannot be used in output field
b	Iteration of 0 and 1	Like ?, except x is excluded Cannot be used in output field
-	No change	Can only be used in output field of a sequential UDP
(vw)	Value change from v to w	v and w can be any one of: 0, 1, x, ?, or b
*	Same as ??	Any value change on input
r	Same as 01	Rising edge on input
f	Same as 10	Falling edge on input
p	Iteration of (01), (0x), and (x1)	Positive edge on input
n	Iteration of (10), (1x), and (x0)	Negative edge on input

Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input changes state, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row.

The following example defines a multiplexer with two data inputs and a control input. Remember, there can only be a single output.

Combinational user-defined primitive

```
primitive multiplexer(mux, control, dataA, dataB ) ;
  output mux ;
  input control, dataA, dataB ;
  table
  // control dataA dataB mux
    0      1      0  :  1  ;
    0      1      1  :  1  ;
    0      1      x  :  1  ;
    0      0      0  :  0  ;
    0      0      1  :  0  ;
    0      0      x  :  0  ;
    1      0      1  :  1  ;
    1      1      1  :  1  ;
    1      x      1  :  1  ;
    1      0      0  :  0  ;
    1      1      0  :  0  ;
    1      x      0  :  0  ;
    x      0      0  :  0  ;
    x      1      1  :  1  ;
  endtable
endprimitive
```

The first entry in the previous table specifies the following: when `control` equals 0 and `dataA` equals 1 and `dataB` equals 0, then output `mux` equals 1.

All combinations of the inputs that are not explicitly specified drive the output to the unknown value `x`. For example, in the table for `multiplexer`, the input combination `0xx(control=0, dataA=x, dataB=x)` is not specified. If this combination occurs during simulation, the value of output `mux` will be `x`.

To improve readability and to make writing the tables easier, several special symbols are provided. A `?` represents iteration of the table entry over the values 0, 1, and `x`. That is, `?` generates cases of that entry where the `?` is replaced by a 0, 1, or `x`. It represents a don't-care condition on that input. Using `?`, the description of the multiplexer given in [“Combinational user-defined primitive”](#) on page 149 can be abbreviated as shown in the following example.

Using the ? symbol in a user-defined primitive

```
primitive multiplexer(mux,control,dataA,dataB ) ;
  output mux ;
  input control, dataA, dataB ;
  table
  // control dataA dataB mux
    0      1      ?  :  1  ;    // ? = 0,1,x
    0      0      ?  :  0  ;
    1      ?      1  :  1  ;
    1      ?      0  :  0  ;

    x      0      0  :  0  ;
    x      1      1  :  1  ;
```

```
    endtable  
endprimitive
```

Level-Sensitive Sequential UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type `reg`, and there is an additional field in each table entry. This new field represents the current state of the UDP.

The output field in a sequential UDP represents the next state.

The following is an example of a latch.

UDP for a latch

```
primitive latch(q, clock, data) ;  
    output q;    reg q;  
    input clock, data;  
    table  
        // clock data  q    q+  
        0      1  : ? :   1  ;  
        0      0  : ? :   0  ;  
        1      ?  : ? :   -  ; // - = no change  
    endtable  
endprimitive
```

This description differs from a combinational UDP in two ways:

- The output identifier `q` has an additional `reg` declaration to indicate that there is an internal state `q`. The output value of the UDP is always the same as the internal state.
- A field for the current state has been added. This field is separated by colons from the inputs and the output.

Edge-Sensitive UDPs

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table as illustrated in the following example.

UDP for an edge-sensitive D-type flip-flop

```
primitive d_edge_ff(q, clock, data);  
output q; reg q;  
input clock, data;
```

Verilog-XL Reference

User-Defined Primitives (UDPs)

```
    table
// obtain output on rising edge of clock
// clock data    q    q+
(01)            0    :    ?    :    0    ;
(01)            1    :    ?    :    1    ;
(0?)            1    :    1    :    1    ;
(0?)            0    :    0    :    0    ;
// ignore negative edge of clock
(?0)           ?    :    ?    :    -    ;
// ignore data changes on steady clock
?              (??) :    ?    :    -    ;
    endtable
endprimitive
```

The previous example has terms like (01) in the input fields. These terms represent transitions of the input values. Specifically, (01) represents a transition from 0 to 1. The first line in the table can be interpreted as follows: when clock changes value from 0 to 1 and data equals 0, the output goes to 0 no matter what the current state is.

Note: Each table entry can have a transition specification on only one input. Entries such as the one shown below are illegal:

```
(01)(01)0 : 0 : 1;
```

As in the combinational and the level-sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (-) in the output column indicates no value change.

All unspecified transitions default to the output value x. Thus, in the previous example, transition of clock from 0 to x with data equal to 0 and current state equal to 1 result in the output q going to x.

All transitions that should not affect the output *must* be explicitly specified. Otherwise, they will cause the value of the output to change to x. If the UDP is sensitive to edges of any input, the desired output state must be specified for *all* edges of *all* inputs.

Sequential UDP Initialization

To specify the value on the output terminal of a sequential UDP, use an `initial` statement that contains a procedural assignment statement. The `initial` statement is optional.

Like `initial` statements in modules, `initial` statements in UDPs begin with the keyword `initial`. However, the contents of `initial` statements in UDPs and the valid left- and right-hand sides of their procedural assignment statements differ from `initial` statements

Verilog-XL Reference

User-Defined Primitives (UDPs)

in modules. The difference between these two types of `initial` statements is described in the following table:

initial statements in UDPs	initial statements in modules
The content is limited to one procedural assignment statement.	The contents can be one procedural statement of any type or a block statement that contains more than one procedural statement.
The procedural assignment statement must assign a value to a <code>reg</code> whose identifier matches the identifier of an output terminal.	Procedural assignment statements can assign values to a <code>reg</code> whose identifier does not match the identifier of an output terminal.
The procedural assignment statement must assign one of the following values: <code>1'b1</code> , <code>1'b0</code> , <code>1'bx</code> , <code>1</code> , <code>0</code>	Procedural assignment statements can assign values of any size, radix, and value.

The following example shows a sequential UDP that contains an `initial` statement.

Sequential UDP with initial statement

```
primitive srff (q,s,r);
output q;
input s,r;
reg q;
initial q = 1'b1;           // initial statement specifies that output
                          // terminal q has a value of 1 at the start
                          // of the simulation

table
// s r q q+
  1 0 : ? : 1 ;
  f 0 : 1 : - ;
  0 r : ? : 0 ;
  0 f : 0 : - ;
  1 1 : ? : 0 ;
endtable
endprimitive
```

In the previous example, the output `q` has an initial value of `1` at the start of the simulation; a delay specification in the UDP instance does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table.

Note: Verilog-XL does not have an initialization or power-up phase. The initial value on the output to a sequential UDP does not propagate to the design output before simulation starts. All nets in the fanout of the output of a sequential UDP begin with a value of `x` even when that output has an initial value of `1` or `0`.

Verilog-XL Reference

User-Defined Primitives (UDPs)

The following example and figure show how values are applied in a module that instantiates a sequential UDP with an `initial` statement. The following example shows the source description for the module and UDP.

Figure 7-1 Instance of a sequential UDP with an initial statement

```
primitive dff1 (q,clk,d);
input clk,d;
output q;
reg q;
initial
    q = 1'b1;          // initial statement
table
//   clk   d       q       q+
    p   0   :   ?   :   0   ;
    p   1   :   ?   :   1   ;
    n   ?   :   ?   :   -   ;
    ?   *   :   ?   :   -   ;
endtable
endprimitive

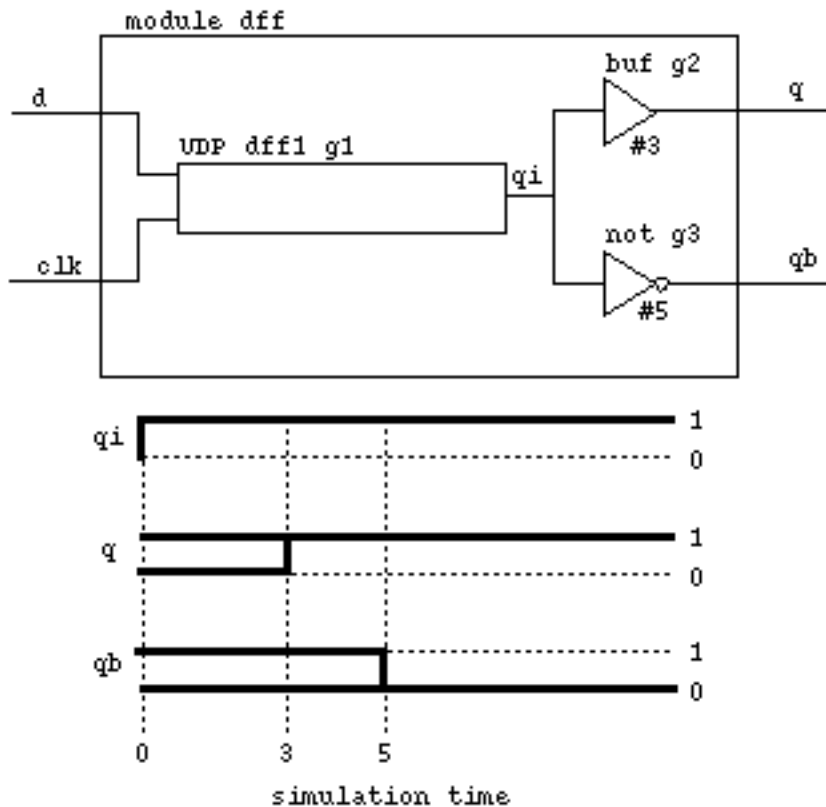
module dff (q,qb,clk,d);
input clk,d;
output q,qb;
    dff1  g1 (qi,clk,d);          // UDP instance output is qi
    buf #3 g2 (q,qi);
    not #5 g3 (qb,qi);           // q and qb are in the fanout of qi
endmodule
```

In this example, UDP `dff1` contains an `initial` statement that sets the initial value of its output to 1. Module `dff` contains an instance of UDP `dff1`. In this instance, the UDP output is `qi`; the output's fanout includes nets `q` and `qb`.

Verilog-XL Reference

User-Defined Primitives (UDPs)

The following figure shows the schematic of the module in [Figure 7-1](#) on page 153 and the simulation times of the propagation of the initial value of the output of the UDP



In this figure, the fanout from the UDP output q_i includes nets q and q_b . At simulation time 0, q_i changes value to 1. That initial value of q_i does not propagate to net q until simulation time 3, and does not propagate to net q_b until simulation time 5.

Mixing Level-Sensitive and Edge-Sensitive Descriptions

UDP definitions allow a mixing of level-sensitive and edge-sensitive constructs in the same description. The following example, which shows an edge-triggered JK flip-flop with asynchronous preset and clear, illustrates this concept.

Sequential UDP for level-sensitive and edge-sensitive behavior

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
  output q; reg q;
  input clock, j, k, preset, clear;
```

Verilog-XL Reference

User-Defined Primitives (UDPs)

```

table
//clock jk pc state output/next state
    ?  ?? 01 : ? : 1 ; //preset logic
    ?  ?? *1 : 1 : 1 ;
    ?  ?? 10 : ? : 0 ; //clear logic
    ?  ?? 1* : 0 : 0 ;

    r  00 00 : 0 : 1 ; //normal clocking cases
    r  00 11 : ? : - ;
    r  01 11 : ? : 0 ;
    r  10 11 : ? : 1 ;
    r  11 11 : 0 : 1 ;
    r  11 11 : 1 : 0 ;
    f  ?? ?? : ? : - ;

    b  *? ?? : ? : - ; //j and k transition cases
    b  ?* ?? : ? : - ;
endtable
endprimitive

```

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is 01, the output has value 1. Similarly, whenever the preset and clear combination has value 10, the output has value 0.

The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge as indicated by an r in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see [Table](#) on page 148) for the entry with an f as the value of clock. Remember, you must specify the desired output for this input transition to avoid unwanted x values at the output. The last two entries show that the transitions in j and k inputs do not change the output on a steady low or high clock.

Level-Sensitive Dominance

In some cases, an edge-sensitive and a level-sensitive table entry may conflict with each other. In these cases, the general rule is that when the input and current state conditions of a level-sensitive table entry and an edge-sensitive table entry specify conflicting next states, the level-sensitive entry dominates the edge-sensitive entry.

The following table shows a level-sensitive table entry and an edge-sensitive entry from the [Sequential UDP for level-sensitive and edge-sensitive behavior](#) figure on page 154. The column on the right shows a case that is included by the table entry.

Conflicting level-sensitive and edge-sensitive entries

Behavior	Table entry	Included case
Level-sensitive	? ?? 01 : ? : 1 ;	0 00 01 : 0 : 1 ;

Verilog-XL Reference

User-Defined Primitives (UDPs)

Conflicting level-sensitive and edge-sensitive entries

Behavior	Table entry	Included case
Edge-sensitive	f ?? ?? : ? : - ; f	00 01 : 0 : 0 ;

The included cases specify opposite next state values for the same input and current state combination. The level-sensitive case specifies that when the inputs `clock`, `jk` and `pc` are 0 00 01, and the current state is 0, the output changes to 1. The edge-sensitive case specifies that when `clock` falls from 1 to 0, and the other inputs `jk` and `pc` are 00 01, and the current state is 0, the output changes to 0.

In this example, the level-sensitive entry dominates, and the output changes to 1.

UDP Instances

You specify instances of user-defined primitives inside modules in the same manner as gate and switch primitives. See [“Gate and Switch Declaration Syntax”](#) on page 98 for information about declaring gates and switches. The instance name is optional except when the instance is declared as an array. The system can automatically generate names for unnamed instances of UDPs. See [“Automatic Naming”](#) on page 233 for more information on automatic naming.

The following syntax shows how to create a UDP instance:

Syntax for UDP Instances

```
<udp_instantiation>
  ::= udp_identifier [drive_strength] [delay2] udp_instance
     {,udp_instance};

<udp_instance>
  ::= [name_of_udp_instance] (output_port_connection,
     input_port_connection {, input_port_connection})

<name_of_udp_instance>
  ::= udp_instance_identifier [range]
```

The port order is as specified in this syntax definition. Only two delays (rising and falling) can be specified, because `z` is not supported for UDPs. An optional range may be specified for an array of UDP instances. The port connection rules are the same as outlined in [“Rules for Using an Array of Instances”](#) on page 102.

The following example creates an instance of the D-type flip-flop `d_edge_ff` (defined in the [UDP for an edge-sensitive D-type flip-flop](#) figure on page 150).

```
module flip;
  reg clock , data ;
```

Verilog-XL Reference

User-Defined Primitives (UDPs)

```
parameter p1 = 10 ;
parameter p2 = 33;
d_edge_ff #(5,7) d_inst( q, clock, data);
initial
begin
  data = 1; clock = 1;
  #100 $finish;
end
always #p1 clock = ~clock;
always #p2 data = ~data;
endmodule
```

Compilation

When UDPs are compiled, the table entries are checked for:

- **Consistency** — If two entries specify different outputs for the same combination of inputs, including edges, Verilog-XL issues an error message. Take special care when using the `?`, `b`, `*`, `p`, and `n` symbols.
- **Redundancy** — If two or more table entries specify the same output for the same combination of inputs, including edges, Verilog-XL issues a warning message. The message indicates the entry that duplicates what is specified in previous lines.

For example, line 2 in the following example will be flagged as redundant because the CLK input for line 1 (`?`) includes the CLK input for line 2 (`1`).

```
//          D  CLK  RB  SB  notifier:  Qt  :  Qt+1
(line 1)   1   ?   1   ?       ?   :   ?   :   1;
(line 2)   1   1   1   ?       ?   :   ?   :   1;
```

However, the following lines 3 and 4 are not redundant because the `?` entry for CLK in line 4 also represents the 0 and 1 cases where an `x` entry in line 3 does not.

```
//          D  CLK  RB  SB  notifier:  Qt  :  Qt+1
(line 3)   1   x   1   ?       ?   :   ?   :   1;
(line 4)   1   ?   1   ?       ?   :   ?   :   1;
```

The following lines 5 and 6 will be flagged as redundant because level-sensitive behavior takes precedence over edge-sensitive behavior. The `(01)` entry for CLK in line 5 will never be selected, and is, therefore, redundant to the `?` entry for CLK in line 6.

```
//          D  CLK  RB  SB  notifier:  Qt  :  Qt+1
(line 5)   1  (01)  1   ?       ?   :   ?   :   1;
(line 6)   1   ?   1   ?       ?   :   ?   :   1;
```

Reducing Pessimism

Three-valued logic tends to make pessimistic estimates of the output when one or more inputs are unknown. You can use UDPs to reduce this pessimism. The following is an extension of the example “[UDP for a latch](#)” on page 150 illustrating reduction of pessimism.

Latch UDP illustrating pessimism

```
primitive latch(q, clock, data);
  output q; reg q ;
  input clock, data ;
  table
//      clock data state output/next state
      0      1  : ? :   1   ;
      0      0  : ? :   0   ;
      1      ?  : ? :   -   ; // - = no change
//      ignore x on clock when data equals state
      x      0  : 0 :   -   ;
      x      1  : 1 :   -   ;
  endtable
endprimitive
```

The last two entries specify what happens when the clock input has value x. If these are omitted, the output will go to x whenever the clock is x. This is a pessimistic model, as the latch should not change its output if it is already 0 and the data input is 0. This is also true when the data input is 1 and the current output is 1.

Consider the jk flip-flop with preset and clear in the following example. This example has additional entries for the positive clock (p) edges, the negative clock edges (?0 and 1x), and with the clock value x. In all of these situations, the output remains unchanged rather than going to x. Thus, this model is less pessimistic than the previous example.

UDP for a JK flip-flop with preset and clear

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
  output q; reg q;
  input clock, j, k, preset, clear;
  table
// clock jk pc state output/next state
// preset logic
  ?  ?? 01 : ? :   1   ;
  ?  ?? *1 : 1 :   1   ;
// clear logic
  ?  ?? 10 : ? :   0   ;
  ?  ?? 1* : 0 :   0   ;
// normal clocking cases
  r  00 00 : 0 :   1   ;
  r  00 11 : ? :   -   ;
  r  01 11 : ? :   0   ;
  r  10 11 : ? :   1   ;
  r  11 11 : 0 :   1   ;
  r  11 11 : 1 :   0   ;
  f  ?? ?? : ? :   -   ;
```

Verilog-XL Reference

User-Defined Primitives (UDPs)

```
// j and k cases
b  *? ?? : ? : - ;
b  ?* ?? : ? : - ;
// cases reducing pessimism
p  00 11 : ? : - ;
p  0? 1? : 0 : - ;
p  ?0 ?1 : 1 : - ;
(?0)?? ?? : ? : - ;
(1x)00 11 : ? : - ;
(1x)0? 1? : 0 : - ;
(1x)?0 ?1 : 1 : - ;
x  *0 ?1 : 1 : - ;
x  0* 1? : 0 : - ;
endtable
endprimitive
```

Processing of Simultaneous Input Changes

When multiple UDP inputs change at the same simulation time, the UDP will be evaluated multiple times, once per input value change. This situation cannot be detected by any form of table entry. This fact has important implications for modeling sequential circuits where the order of input changes and subsequent UDP evaluations can have a profound effect on the results of the simulation.

Consider the D-type flip-flop in the following example.

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;

table
// obtain output on rising edge of clock
// clock  data  q    q+
(01)      0    :    ?    :    0    ;
(01)      1    :    ?    :    1    ;
(0?)      1    :    1    :    1    ;
(0?)      0    :    0    :    0    ;
// ignore negative edge of clock
(?0)      ?    :    ?    :    -    ;
// ignore data changes on steady clock
?         (??) :    ?    :    -    ;
endtable
endprimitive
```

If the current state of the flip-flop is 0 and the clock and data inputs make transitions from 0 to 1 at the same simulation time, then the state of the output at the next simulation time is unpredictable because it cannot predict which of these transitions is processed first.

If the clock input transition is processed first and the data input transition is processed second, then the next state of the output is 0. However, if the data input transition is processed first and the clock transition is processed second, then the next state of the output will be 1.

Take this fact into consideration when constructing models. Keep in mind that gate-level models have the same sort of unpredictable behavior given particular input transition sequences; event-driven simulation is subject to idiosyncratic dependence on the order in which events are processed.

Use timing checks to detect simultaneous input transitions and to provide a warning. See Chapter 12, “[Using Specify Blocks and Path Delays](#)” for more information.

Memory Usage and Performance Considerations

You should be aware of the amount of memory required for the internal tables created for the evaluation of UDPs during simulation. Although only one such table is required per UDP definition, and not one for each instance, the UDPs with 8, 9, and 10 inputs do consume a large amount of memory, especially if you are using the path delay accuracy enhancement algorithm (see “[Enhancing Path Delay Accuracy](#)” on page 279 for details). The trade-off here is speed versus memory. If you need many instances of a large UDP, it is easily possible to gain back the memory used by the definition, because each UDP instance can take less memory than that required for the group of gates it replaces.

The memory required for a UDP definition is shown below. The figures in this table are worst case, and can vary depending on the actual number of terms specified in the definition. Note that the number of variables is the number of inputs for combinational UDPs and the number of inputs plus one for sequential UDPs.

Table 7-2 UDP memory requirements

Number of Variables	Memory Required (K bytes)	With Path Delay Accuracy
1 - 4	<1	<1
5	<1	3
6	5	10
7	17	36
8	56	138
9	187	538
10	623	2125

UDP Examples

The following examples show UDP modeling for an and-or gate, a majority function for carry, and a 2-channel multiplexor with storage.

UDP for an and-or gate

```
// Description of an AND-OR gate.
// out = (a1 & a2 & a3) | (b1 & b2).
primitive and_or(out, a1,a2,a3, b1,b2);
  output out;
  input a1,a2,a3, b1,b2;
  table
  //   a  b  : out ;
    111 ?? : 1 ;
    ??? 11 : 1 ;
    0?? 0? : 0 ;
    0?? ?0 : 0 ;
    ?0? 0? : 0 ;
    ?0? ?0 : 0 ;
    ??0 0? : 0 ;
    ??0 ?0 : 0 ;
  endtable
endprimitive
```

UDP for a majority function for carry

```
// Majority function for carry
// carryout = (a & b) | (a & carryin) | (b & carryin)
primitive carry(carryout, carryin, a, b);
  output carryout;
  input carryin, a, b;
  table
    0 00 : 0;
    0 01 : 0;
    0 10 : 0;
    0 11 : 1;
    1 00 : 0;
    1 01 : 1;
    1 10 : 1;
    1 11 : 1;
    // the following cases reduce pessimism
    0 0x : 0;
    0 x0 : 0;
    x 00 : 0;
    1 1x : 1;
    1 x1 : 1;
    x 11 : 1;
  endtable
endprimitive
```

UDP for a 2-channel multiplexor with storage

```
// Description of a 2-channel multiplexor with storage.
// The storage is level sensitive.
```

Verilog-XL Reference

User-Defined Primitives (UDPs)

```
primitive mux_with_storage(out,clk,control,dataA,dataB);
  output out;
  reg out;
  input clk, control, dataA, dataB;
  table
  //clk control dataA dataB : current-state : next state ;
    1     0     1     ?   :     ?     :     1     ;
    1     0     0     ?   :     ?     :     0     ;
    1     1     ?     1   :     ?     :     1     ;
    1     1     ?     0   :     ?     :     0     ;
    1     x     0     0   :     ?     :     0     ;
    1     x     1     1   :     ?     :     1     ;
    0     ?     ?     ?   :     ?     :     -     ;
    x     0     1     ?   :     1     :     -     ;
    x     0     0     ?   :     0     :     -     ;
    x     1     ?     1   :     1     :     -     ;
    x     1     ?     0   :     0     :     -     ;
  endtable
endprimitive
```

Behavioral Modeling

This chapter describes the following:

- [Overview](#) on page 163
- [Structured Procedures](#) on page 164
- [Procedural Assignments](#) on page 166
- [Conditional Statements](#) on page 174
- [Multi-Way Decision Statements](#) on page 175
- [Looping Statements](#) on page 179
- [Procedural Timing Controls](#) on page 182
- [Block Statements](#) on page 189
- [Behavior Model Examples](#) on page 193

Overview

The language constructs introduced so far allow hardware to be described at a relatively detailed level. Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing the complex high-level aspects of a system. The procedural constructs described in this chapter are well suited to tackling such problems as describing a microprocessor and implementing complex timing checks.

Verilog behavioral models contain procedural statements that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it.

Each activity flow starts at the control constructs `initial` and `always`. Each `initial` statement and each `always` statement starts a separate activity flow, and all of the activity flows are concurrent, allowing you to model the inherent concurrence of the hardware.

Verilog-XL Reference

Behavioral Modeling

The following example is a complete Verilog behavioral model.

```
module behave;
  reg [1:0]a,b;
  initial
    begin
      a = 'b1;
      b = 'b0;
    end
  always
    begin
      #50 a = ~a;
    end
  always
    begin
      #100 b = ~b;
    end
endmodule
```

During the simulation of this model, all of the flows defined by the `initial` and `always` statements start together at simulation time zero. The `initial` statements execute once, and the `always` statements execute repetitively.

In this model, the register variables `a` and `b` initialize to binary 1 and 0 respectively at simulation time zero. The `initial` statement is then complete and does not execute again during this simulation run. This `initial` statement contains a `begin-end` block (also called a sequential block) of statements. In this `begin-end` block, `a` is initialized first, followed by `b`.

The `always` statements also start at time zero, but the values of the variables do not change until the times specified by the delay controls (introduced by `#`) have gone by. Thus, register `a` inverts after 50 time units, and register `b` inverts after 100 time units. Since the `always` statements repeat, this model produces two square waves. Register `a` toggles with a period of 100 time units, and register `b` toggles with a period of 200 time units. The two `always` statements proceed concurrently throughout the entire simulation run.

Structured Procedures

All procedures in Verilog are specified within one of the following four statements:

- `always` statement
- `initial` statement
- `task`
- `function`

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are covered in detail in [Chapter 9, “Tasks and Functions.”](#)

The `initial` and `always` statements are enabled at the beginning of simulation. The `initial` statement executes only once and its activity dies when the statement has finished. The `always` statement executes repeatedly. Its activity dies only when the simulation is terminated. There is no limit to the number of `initial` and `always` blocks that can be defined in a module.

always Statement

Each `always` statement repeats continuously throughout the whole simulation run. The syntax for the `always` statement is as follows:

```
<always_statement>  
  ::= always <statement>
```

The `always` statement, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an `always` statement provides no means for time to advance, the `always` statement creates a simulation deadlock condition. The following code, for example, creates an infinite zero-delay loop:

```
always areg = ~areg;
```

Providing a timing control to this code creates a potentially useful description, as in the following example:

```
always #half_period areg = ~areg;
```

initial Statement

An `initial` statement is similar to an `always` statement, except that it is executed only once. The syntax for an `initial` statement is as follows:

```
<initial_statement>  
  ::= initial <statement>
```

The following example illustrates the use of an `initial` statement for the initialization of variables at the start of simulation.

```
initial  
  begin  
    areg = 0; // initialize a register  
    for (index = 0; index < size; index = index + 1)  
      memory[index] = 0; //initialize a memory word  
  end
```

A typical use of the `initial` statement is the specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated. The following example illustrates this usage:

```
initial  
  begin
```

```
inputs = 'b000000;           // initialize at time zero
#10 inputs = 'b011001;       // first pattern
#10 inputs = 'b011011;       // second pattern
#10 inputs = 'b011000;       // third pattern
#10 inputs = 'b001000;       // last pattern
end
```

Procedural Assignments

As described in [Chapter 5, “Assignments.”](#), procedural assignments are for updating `reg`, `integer`, `time`, and `memory` variables.

There is a significant difference between procedural assignments and continuous assignments, as described below:

- Continuous assignments drive net variables and are evaluated and updated whenever an input operand changes value.
- Procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

The Verilog HDL contains two types of procedural assignment statements:

- blocking procedural assignment statements
- non-blocking procedural assignment statements

[“Blocking Procedural Assignments”](#) on page 167 and [“Non-Blocking Procedural Assignments”](#) on page 167 specify different procedural flow in sequential blocks.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. However, part-selects on the right-hand side must have constant indexes. The left-hand side of the procedural assignment indicates the variable that receives the assignment from the right-hand side.

The left-hand side of a procedural assignment can take one of the following forms:

- Register, integer, real, or time variable; that is, an assignment to the name reference of one of these data types.
- Bit-select of a register, integer, real, or time variable; that is, an assignment to a single bit that leaves the other bits untouched.
- Part-select of a register, integer, real, or time variable; that is, a part-select of two or more contiguous bits that leaves the rest of the bits untouched. For the part-select form, only constant expressions are legal.

- Memory element; that is, a single word of a memory.

Note: Bit-selects and part-selects are illegal on memory element references.

- concatenation of any of the above:

A concatenation of any of the previous four forms, which effectively partitions the result of the right-hand side expression and assigns the partition parts to the various parts of the concatenation.

Note: Assignment to a register or time variable does not sign-extend. Assignment to a register differs from assignment to a `time` or `integer` variable when the right-hand side evaluates to fewer bits than the left-hand side. Registers are unsigned; if you assign a register to an integer, the variable does not sign-extend.

Blocking Procedural Assignments

A blocking procedural assignment statement must be executed before executing the statements that follow it in a sequential block (see “[Sequential Blocks](#)” on page 189). A blocking procedural assignment statement does not have to be executed before statements that follow in a parallel block (see “[Parallel Blocks](#)” on page 191).

The syntax for a blocking procedural assignment is as follows:

```
<value> = <timing_control> <expression>
```

Where `<value>` is a data type that is valid for a procedural assignment statement, `=` is the assignment operator, and `timing_control` is the optional intra-assignment delay. The `timing_control` delay can be either a delay control (for example, `#6`) or an event control (for example, `@(posedge clk)`). The `expression` is the right-hand side value that the simulator assigns to the left-hand side.

The following example shows blocking procedural assignments:

```
rega = 0; // a register assignment
rega[3] = 1; // a bit-select assignment
rega[3:5] = 7; // a part-select assignment
mema[address] = 8'hff; // a memory element assignment
{carry, acc} = rega + regb; // a concatenation
```

Non-Blocking Procedural Assignments

The non-blocking procedural assignment allows you to schedule assignments without blocking the procedural flow. You can use the non-blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependence upon each other.

The syntax for a non-blocking procedural assignment is as follows:

```
<value> <= <timing_control> <expression>
```

Where `<value>` is a data type that is valid for a procedural assignment statement, `<=` is the non-blocking assignment operator, and `timing_control` is the optional intra-assignment timing control. The `timing_control` delay can be either a delay control (for example, `#6`) or an event control (for example, `@(posedge clk)`). The `expression` is the right-hand side value that the simulator assigns to the left-hand side.

The non-blocking assignment operator is the same operator that the simulator uses for the less-than-or-equal relational operator. The simulator interprets the `<=` operator to be a relational operator when you use it in an expression, and it interprets the `<=` operator to be an assignment operator when you use it in a non-blocking procedural assignment construct.

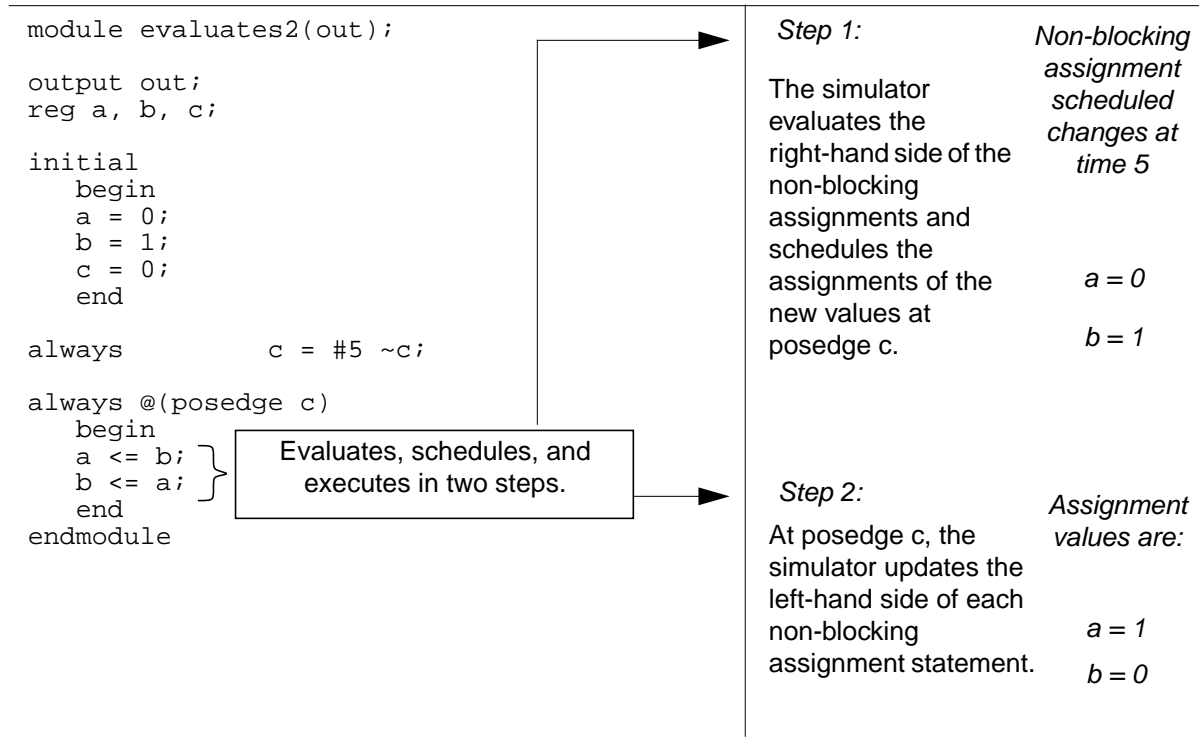
Evaluating Non-Blocking Procedural Assignments

When the simulator encounters a non-blocking procedural assignment, the simulator evaluates and executes the non-blocking procedural assignment in two steps as follows:

1. The simulator evaluates the right-hand side and schedules the assignment of the new value to take place at a time specified by a procedural timing control.
2. At the end of the time step, in which the given delay has expired or the appropriate event has taken place, the simulator executes the assignment by assigning the value to the left-hand side.

Verilog-XL Reference Behavioral Modeling

These two steps are shown in the following figure.



At the end of the time step in Step 2 means that the non-blocking assignments are the last assignments executed in a time step—with one exception. Non-blocking assignment events can create blocking assignment events. The simulator processes these blocking assignment events after the scheduled non-blocking events.

Unlike a regular event or delay control, the non-blocking assignment does not block the procedural flow. The non-blocking assignment evaluates and schedules the assignment, but

Verilog-XL Reference Behavioral Modeling

does not block the execution of subsequent statements in a begin-end block, as shown in the following example.

```
//non_block1.v
module non_block1(out);
//input
output out;
reg a, b, c, d, e, f;
//blocking assignments
initial begin
  a = #10 1;
  b = #2 0;
  c = #4 1;
end
```

The simulator assigns 1 to register a at simulation time 10, assigns 0 to register b at simulation time 12, and assigns 1 to register c at simulation time 16.

```
//non-blocking assignments
initial begin
  d <= #10 1;
  e <= #2 0;
  f <= #4 1;
end
initial begin
  $monitor ($time, , "a = %b b = %b c = %b", a, b, c);
  #100 $finish;
end
endmodule // non_block1
```

The simulator assigns 1 to register d at simulation time 10, assigns 0 to register e at simulation time 2, and assigns 1 to register f at simulation time 4.

*non-blocking
assignment lists*

*Scheduled
changes at
time 2*

e = 0

*Scheduled
changes at
time 4*

f = 1

*Scheduled
changes at
time 10*

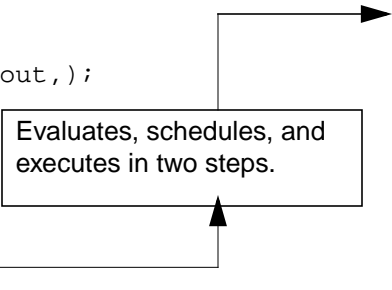
d = 1

//Size the windows correctly. In the previous page, the example is not fully visible because window size is small

Verilog-XL Reference Behavioral Modeling

Note: As shown in the next example, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with non-blocking procedural assignments.

```
//non_block1.v
module non_block1(out,);
output out;
reg a, b;
initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;
end
    initial begin
$monitor ($time, , "a = %b b = %b", a,b);
#100 $finish;
    end
endmodule
```



Step 1:

The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments for the end of the current time step.

Step 2:

At the end of the current time step, the simulator updates the left-hand side of each non-blocking assignment statement.

Assignment values are:

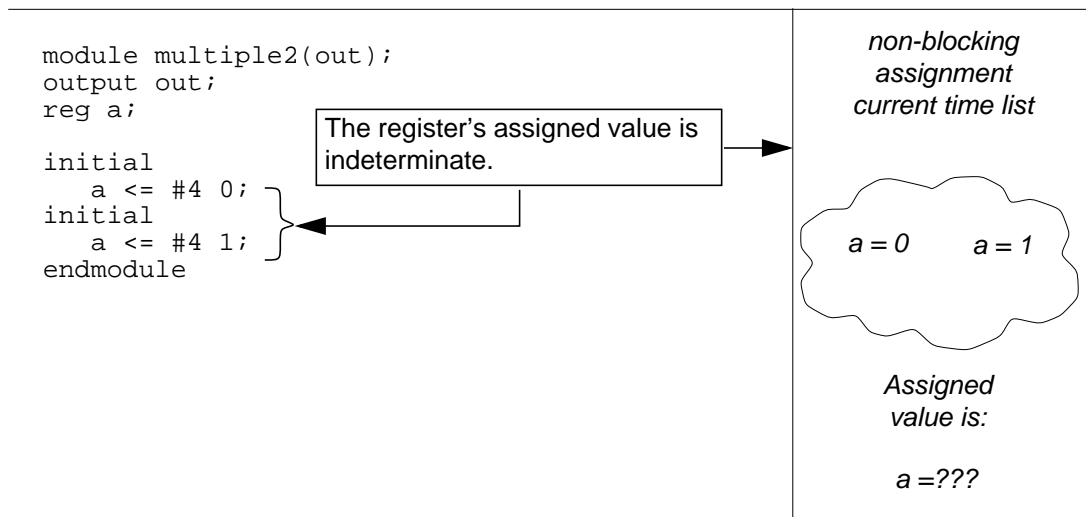
a = 1

b = 0

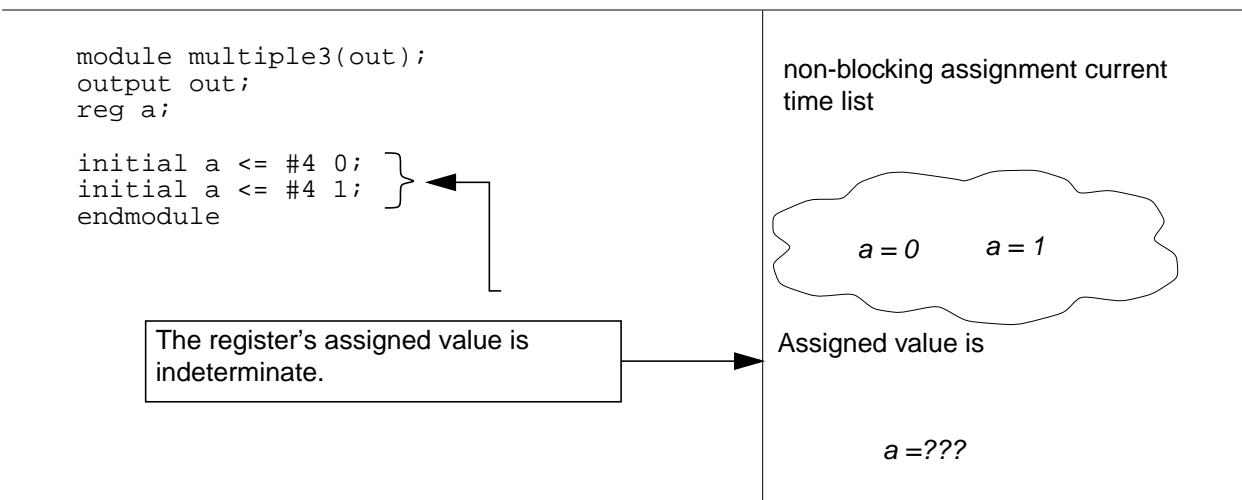
When you schedule multiple non-blocking assignments to occur in the same register in a particular time slot, the simulator cannot guarantee the order in which it processes the

Verilog-XL Reference Behavioral Modeling

assignments—the final value of the register is indeterminate. As shown in the following example, the value of register *a* is not known until the end of time step 4



If the simulator executes two procedural blocks concurrently, and these procedural blocks contain non-blocking assignment operators, the final value of the register is indeterminate as in the following example.



When multiple non-blocking assignments with timing controls are made to the same register, the assignments can be made without cancelling previous non-blocking assignments. In the

Verilog-XL Reference Behavioral Modeling

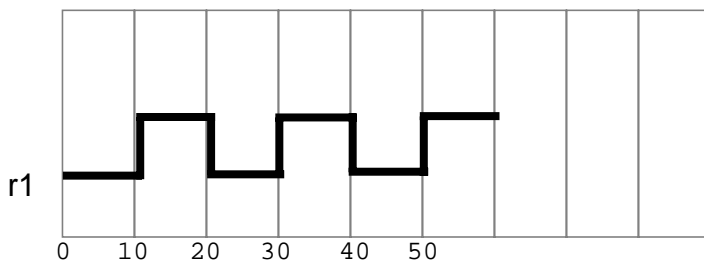
following example, the simulator evaluates the value of `i[0]` to `r1` and schedules the assignments to occur after each time delay.

:

```
module multiple;
  reg r1;
  reg [2:0] i;

  initial
  begin
    // starts at time 0 doesn't hold the block
    for (i = 0; i <= 5; i = i+1)
      r1 <= # (i*10) i[0];
    end
  endmodule
```

Make the assignments to `r1` without cancelling previous non-blocking assignments.



S changes at time 0

r1 = 0

*Scheduled changes at
time 10*

r1 = 1

*Scheduled changes at
time 20*

r1 = 0

*Scheduled changes at
time 30*

r1 = 1

*Scheduled changes at
time 40*

r1 = 0

*Scheduled changes at
time 50*

r1 = 1

Processing Blocking and Non-Blocking Procedural Assignments

For each time slot during simulation, blocking and non-blocking procedural assignments are processed in the following way:

1. Evaluate the right-hand side of all assignment statements in the current time slot.
2. Execute all blocking procedural assignments and non-blocking procedural assignments that have no timing controls. At the same time, set aside for processing non-blocking procedural assignments with timing controls.
3. Check for procedures that have timing controls and execute a procedure if its timing control is set for the current time unit.
4. Advance the simulation clock.

Conditional Statements

The conditional statement (or `if-else` statement) is used to decide whether to execute a statement. The syntax is as follows:

```
<statement>
 ::= if ( <expression> ) <statement_or_null>
    ||= if ( <expression> ) <statement_or_null>
       else <statement_or_null>

<statement_or_null>
 ::= <statement>
    ||= ;
```

The `<expression>` is evaluated; if it is true (that is, it has a non-zero known value), the first statement executes. If it is false (that is, it has a zero value or the value is `x` or `z`), the first statement does not execute. If there is an `else` statement and `<expression>` is false, the `else` statement executes.

Since the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
if (expression != 0)
```

Because the `else` part of an `if-else` is optional, there can be confusion when an `else` is omitted from a nested `if` sequence. This is resolved by always associating the `else` with the closest previous `if` that lacks an `else`. In the following example, the `else` goes with the inner `if`, as we have shown by indentation:

```
if (index > 0)
  if (rega > regb)
    result = rega;
  else // else applies to preceding if
    result = regb;
```

If that association is not what you want, use a `begin-end` block statement to force the proper association, as shown in the following example:

```
if (index > 0)
  begin
    if (rega > regb)
      result = rega;
  end
else
  result = regb;
```

`Begin-end` blocks left out inadvertently can change the logic behavior being expressed, as shown in the following example:

```
if (index > 0)
  for (scani = 0; scani < index; scani = scani + 1)
    if (memory[scani] > 0)
      begin
```

```
        $display("...");
        memory[scani] = 0;
    end
else /* WRONG */
    $display("error - index is zero");
```

The indentation in the previous example shows unequivocally what you want, but the compiler does not get the message and associates the `else` with the inner `if`. This kind of bug can be very hard to find.

Note: One way to find this kind of bug is to use the `$list` system task, which indents according to the logic of the description.

Notice that in the next example, there is a semicolon after `result = rega`. This is because a `<statement>` follows the `if`, and a semicolon is an essential part of the syntax of a `<statement>`.

```
if (rega > regb)
    result = rega;
else
    result = regb;
```

For Verilog-XL to behave predictably in interactive mode, each conditional statement must conform 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.

Multi-Way Decision Statements

There are two statements that you can use to specify one or more actions to be taken based on specified conditions: `if-else-if` and `case`.

if-else-if Statements

The sequence of `if` statements, known as an `if-else-if` construct, is the most general way to write a multi-way decision. The syntax of an `if-else-if` construct is shown as follows:

```
if (<expression>)
    <statement>
else if (<expression>)
    <statement>
else if (<expression>)
    <statement>
else
    <statement>
```

Verilog-XL Reference

Behavioral Modeling

The expressions are evaluated in order. If any expression is true, the statement associated with it is executed, and the whole conditional chain is terminated. Each statement is either a single statement or a block of statements.

The last `else` part of the `if-else-if` construct handles the default case in which none of the other conditions were satisfied. Sometimes there is no explicit action for the default; in that case, the trailing `else` can be either omitted or used for error checking to catch an impossible condition.

The following example uses the `if-else` statement to test the variable `index` to decide whether one of three `modify_segn` registers must be added to the memory address, and to decide which increment is to be added to the `index` register. The first ten lines declare the registers and parameters.

```
// Declare registers and parameters
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0, inc_seg1 = 1,
    segment2 = 20, inc_seg2 = 2,
    segment3 = 64, inc_seg3 = 4,
    data = 128;

initial
begin
// Test the index variable
if (index < segment2)
    begin
        instruction = segment_area [index + modify_seg1];
        index = index + inc_seg1;
    end
else if (index < segment3)
    begin
        instruction = segment_area [index + modify_seg2];
        index = index + inc_seg2;
    end
else if (index < data)
    begin
        instruction = segment_area [index + modify_seg3];
        index = index + inc_seg3;
    end
else
    instruction = segment_area [index];
end // initial block
```

case Statements

The `case` statement is a special multi-way decision statement that tests whether an expression matches one of several other expressions, and branches accordingly. For example, the `case` statement is useful for describing the decoding of a microprocessor

Verilog-XL Reference Behavioral Modeling

instruction. The syntax of the `case` statement is as follows. The default statement is optional. Using multiple default statements in one `case` statement is illegal syntax.

```
<statement>
 ::= case ( <expression> ) <case_item>+ endcase
    || = casez ( <expression> ) <case_item>+ endcase
    || = casex ( <expression> ) <case_item>+ endcase
<case_item>
 ::= <expression> <,<expression>>* : <statement_or_null>
    || = default : <statement_or_null>
    || = default <statement_or_null>
```

A simple example of the `case` statement is the decoding of register `rega` to produce a value for `result` as shown in the following example:

```
reg [15:0] rega;
reg [9:0] result;
...
case (rega)
  16'd0: result = 10'b0111111111;
  16'd1: result = 10'b1011111111;
  16'd2: result = 10'b1101111111;
  16'd3: result = 10'b1110111111;
  16'd4: result = 10'b1111011111;
  16'd5: result = 10'b1111101111;
  16'd6: result = 10'b1111110111;
  16'd7: result = 10'b1111111011;
  16'd8: result = 10'b1111111101;
  16'd9: result = 10'b1111111110;
  default result = 'bx;
endcase
```

The case expressions are evaluated and compared in the exact order in which they are given. During the linear search, if one of the case item expressions matches the expression in parentheses, then the statement associated with that case item is executed. If all comparisons fail, and the default item is given, then the default item statement is executed. If the default statement is not given, and all of the comparisons fail, then none of the `case` item statements is executed.

Apart from syntax, the `case` statement differs from the multi-way `if-else-if` construct in two important ways:

- The conditional expressions in the `if-else-if` construct are more general than comparing one expression with several others, as in the `case` statement.
- The `case` statement provides a definitive result when there are `x` and `z` values in an expression.

In a case comparison, the comparison only succeeds when each bit matches exactly with respect to the values 0, 1, `x`, and `z`. Consequently, care is needed in specifying the expressions in the `case` statement. The bit length of all the expressions must be equal so that exact bit-wise matching can be performed. The length of all the case item expressions, as

well as the controlling expression in the parentheses, is made equal to the maximum width of any of the `<case_item>` expressions and the control expression. The most common mistake made here is to specify `'bx` or `'bz` instead of `n'bx` or `n'bz`, where `n` is the bit length of the expression in parentheses. The default length of `x` and `z` is the word size of the host machine, usually 32 bits.

The reason for providing a `case` comparison that handles the `x` and `z` values is that it provides a mechanism for detecting those values and reducing the pessimism that can be generated by their presence. The following example illustrates the use of a `case` statement to properly handle `x` and `z` values.

```
case (select[1:2])
  2'b00: result = 0;
  2'b01: result = flaga;
  2'b0x,
  2'b0z: result = flaga ? 'bx : 0;
  2'b10: result = flagb;
  2'bx0,
  2'bz0: result = flagb ? 'bx : 0;
  default: result = 'bx;
endcase
```

This example contains a `case` statement used to trap `x` and `z` values. Notice that if `select[1]` is 0 and `flaga` is 0, then no matter what the value of `select[2]` is, the result is set to 0. The first, second, and third `case` items cause this assignment.

The following example shows another way to use a `case` statement to detect `x` and `z` values:

```
case(sig)
  1'bz:
    $display("signal is floating");
  1'bx:
    $display("signal is unknown");
  default:
    $display("signal is %b", sig);
endcase
```

Using case Statements with Inconsequential Conditions

Two other types of `case` statements are provided to handle inconsequential conditions in comparisons. One type treats high-impedance values (`z`) as inconsequential. The other type treats both high-impedance and unknown (`x`) values as inconsequential.

You use these types of `case` statements in the same way as traditional `case` statement, but they begin with new keywords—`casez` and `casex`.

Inconsequential values (`z` values for `casez`, `z` and `x` values for `casex`) in any bit of either the case expression or the case items are treated as inconsequential conditions during the comparison; bit position is not considered.

Note: Allowing inconsequential values in the case items means that you can dynamically control which bits of the case expression are compared during simulation.

The syntax of literal numbers allows the use of the question mark (?) in place of z in `casez` and `casex` statements. This provides a convenient format for specification of inconsequential bits in `case` statements.

The following is an example of the `casez` statement. It demonstrates an instruction decode, in which values of the most significant bits select the task to call. If the most significant bit of `ir` is a 1, then the task `instruction1` is called, regardless of the values of the other bits of `ir`.

```
reg [7:0] ir;
...
casez (ir)
  8'b1??????: instruction1(ir);
  8'b01?????: instruction2(ir);
  8'b00010???: instruction3(ir);
  8'b000001??: instruction4(ir);
endcase
```

The following is an example of the `casex` statement. It demonstrates an extreme case of the dynamic control of inconsequential conditions during simulation. In this case, if `r = 8'b01100110`, then the task `stat2` is called.

```
reg [7:0] r, mask;
...
mask = 8'bx0x0x0x0;
casex (r ^ mask)
  8'b001100xx: stat1;
  8'b1100xx00: stat2;
  8'b00xx0011: stat3;
  8'bxx001100: stat4;
endcase
```

Looping Statements

There are four types of looping statements. They provide a means of controlling the execution of a statement, either zero, one, or more times.

- `forever` continuously executes a statement.
- `repeat` executes a statement a fixed number of times.
- `while` executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.
- `for` controls execution of its associated statement(s) by a three-step process, as follows:

- a. Executes an assignment, normally used to initialize a variable, that controls the number of loops executed
- b. Evaluates an expression—if the result is zero, the `for` loop exits. If it is not zero, the `for` loop executes its associated statement(s) and then performs step 3
- c. Executes an assignment, normally used to modify the value of the loop-control variable, then repeats step 2

The following are the syntax rules for looping statements:

```
<statement>
 ::= forever <statement>
 ||= forever
    begin
        <statement>+
    end

<statement>
 ::= repeat ( <expression> ) <statement>
 ||= repeat ( <expression> )
    begin
        <statement>+
    end

<statement>
 ::= while ( <expression> ) <statement>
 ||= while ( <expression> )
    begin
        <statement>+
    end

<statement>
 ::= for ( <assignment> ; <expression> ; <assignment> )
    <statement>
 ||= for ( <assignment> ; <expression> ; <assignment> )
    begin
        <statement>+
    end
```

The rest of this section presents examples for three of the looping statements.

forever Loop

Use the `forever` loop only conjunction with the timing controls, or the `disable` statement. See [“Event Control”](#) on page 184 for an example of a `forever` loop.

repeat Loop

In the following example of a `repeat` loop, add and shift operators implement a multiplier.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
```

Verilog-XL Reference Behavioral Modeling

```
begin :mult    ...
    reg [longsize:1] shift_opa, shift_opb;

    shift_opa = opa;
    shift_opb = opb;
    result = 0;

    repeat (size)
        begin
            if (shift_opb[1])
                result = result + shift_opa;
            shift_opa = shift_opa << 1;
            shift_opb = shift_opb >> 1;
        end
    end
end
```

while Loop

An example of the `while` loop follows. It counts up the number of logic 1 values in `rega`.

```
begin :count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while(tempreg)
        begin
            if (tempreg[0]) count = count + 1;
            tempreg = tempreg >> 1;
        end
    end
end
```

for Loop

The `for` loop construct accomplishes the same results as the following pseudocode that is based on the `while` loop:

```
begin
    initial_assignment;
    while (condition)
        begin
            statement
            step_assignment;
        end
    end
end
```

The `for` loop implements the logic in the preceding 8 lines while using only two lines, as shown in the pseudocode in the following example.

```
for (initial_assignment; condition; step_assignment)
    statement
```

The following example uses a `for` loop to initialize a memory:

Verilog-XL Reference

Behavioral Modeling

```
begin :init_mem
  reg [7:0] tempi;
  for (tempi = 0; tempi < memsize; tempi = tempi + 1)
    memory[tempi] = 0;
end
```

The next example shows another `for` loop statement. It is the same multiplier that was described in “[repeat Loop](#)” on page 180.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

...

begin :mult
  integer bindex;
  result = 0;
  for (bindex = 1; bindex <= size; bindex = bindex + 1)
    if (opb[bindex])
      result = result + (opa << (bindex - 1));
end
```

Note: You can use the `for` loop statement more generally than the normal arithmetic progression of an index variable, as in the following example. This is another way of counting the number of logic 1 values in `rega` (see “[while Loop](#)” on page 181):

```
begin :count1s
  reg [7:0] tempreg;
  count = 0;
  for (tempreg = rega; tempreg; tempreg = tempreg >> 1)
    if (tempreg[0]) count = count + 1;
end
```

Procedural Timing Controls

In Verilog, actions are scheduled in the future through the use of delay controls. A general principle of the Verilog language is that where you do not see a timing control, simulation time does not advance—if you specify no timing delays, the simulation completes at time zero.

The Verilog language provides two types of explicit timing control over when in simulation time procedural statements are to occur. The first type of timing controls is a delay control, in which an expression specifies the time duration between initially encountering the statement and executing the statement. This delay expression can be a dynamic function of the state of the circuit, but is usually a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is more fully described in “[Delay Control](#)” on page 183, “[Zero-Delay Control](#)” on page 183, and “[Intra-Assignment Timing Controls](#)” on page 186.

The second type of timing control is the event expression, which allows a statement execution to wait for the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or a

register (an implicit event), or the occurrence of an explicitly named event that is triggered from other procedures (an explicit event). Most often, an event control is a positive or negative edge on a clock signal.

To schedule activity for the future, use one of the following methods of timing control:

- a `delay` control, which is introduced by the number symbol (`#`)
- an `event` control, which is introduced by the at symbol (`@`)
- a `wait` statement, which operates like a combination of an event control and a `while` loop

The next several sections discuss these three methods.

Delay Control

The execution of a procedural statement can be delay-controlled by using the following syntax:

```
<statement>
  ::= <delay_control> <statement_or_null>

<delay_control>
  ::= # <NUMBER>
     || = # <identifier>
     || = # ( <mintypmax_expression> )
```

The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

The next three examples provide an expression following the number sign (`#`). The execution of the assignment is delayed by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;           // d is defined as a parameter
#((d+e)/2) rega = regb;  // delay is the average of d and e
#regr regr = regr + 1;   // delay is the value in regr
```

Zero-Delay Control

A special case of the delay control is the zero-delay control, as in the following example:

```
forever
  #0 a = ~a;
```

This type of delay control has the effect of moving the assignment statement to the end of the list of statements to be evaluated at the current simulation time unit. Note that if there are

several such delay controls encountered at the same simulation time, the order of evaluation of the statements which they control cannot be predicted.

Event Control

The execution of a procedural statement can be synchronized with a value change on a net or register or with the occurrence of a declared event by using the following event control syntax:

```
<statement>
  ::= <event_control> <statement_or_null>
<event_control>
  ::= @ <identifier>
      || = @ ( <event_expression> )
<event_expression>
  ::= <expression>
      || = posedge <SCALAR_EVENT_EXPRESSION>
      || = negedge <SCALAR_EVENT_EXPRESSION>
      || = <event_expression> <or <event_expression>*>
<SCALAR_EVENT_EXPRESSION>
  an expression that resolves to a one bit value.
```

You can use value changes on nets and registers as events to trigger the execution of a statement. This is known as detecting an implicit event.

See item 1 in the following example for a syntax example of a `wait` for an implicit event. Verilog syntax also allows you to detect change based on the direction of the change—that is, toward the value 1 (`posedge`) or toward the value 0 (`negedge`). The behavior of `posedge` and `negedge` for unknown expression values is as follows:

- A `negedge` is detected on the transition from 1 to unknown and from unknown to 0.
- A `posedge` is detected on the transition from 0 to unknown and from unknown to 1.

Items 2 and 3 in the following example show illustrations of edge-controlled statements.

```
@r rega = regb; // Item 1: controlled by any value changes in the register r

@(posedge clock) rega = regb; // Item 2: controlled by positive
                             // edge on clock

forever @(negedge clock) rega = regb; // Item 3: controlled by negative edge
```

Named Events

Verilog also provides syntax to name an event and then to trigger the occurrence of that event. A model can use an event expression to wait for the triggering of this explicit event.

Verilog-XL Reference

Behavioral Modeling

Named events can be made to occur from a procedure. This allows control over the enabling of multiple actions in other procedures. Named events and event control provide a powerful and efficient means of describing the communication between, and the synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes the control of a synchronous circuit by signalling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

An event name must be declared explicitly before it is used. The following is the syntax for declaring events:

```
<event_declaration>
 ::= event <name_of_event> <,<name_of_event>>* ;
<name_of_event>
 ::= <IDENTIFIER>
      the name of an explicit event
```

Note: An event does not hold any data. Therefore, you cannot use edge-triggering expressions like `posedge` or `negedge` with named events because named events carry no timing duration information.

The following are the characteristics of a Verilog event:

- It can be made to occur at any particular time.
- It has no time duration.
- Its occurrence can be recognized by using the `<event_control>` syntax described in “[Event Control](#)” on page 184.

The power of the explicit event is that it can represent any general happening. For example, it can represent a positive edge of a clock signal, or it can represent a microprocessor transferring data down a serial communications channel. A declared event is made to occur by the activation of an event-triggering statement of the following syntax:

```
-> <name_of_event> ;
```

An event-controlled statement (for example, `@trig rega = regb;`) causes the simulation of its containing procedure to wait until some other procedure executes the appropriate event-triggering statement (for example, `->trig;`).

Event OR Construct

The ORing of any number of events can be expressed such that the occurrence of any one event will trigger the execution of the statement. The next two examples show the ORing of two and three events respectively.

```
@(trig or enable) rega = regb; // controlled by trig or enable
@(posedge clock_a or posedge clock_b or trig) rega = regb;
```

Level-Sensitive Event Control

The execution of a statement can be delayed until a condition becomes true. This is accomplished using the `wait` statement, which is a special form of event control. The nature of the `wait` statement is level-sensitive, as opposed to basic event control (specified by the `@` character), which is edge-sensitive. The `wait` statement checks a condition. If the condition is false, the `wait` statement causes the procedure to pause until the condition becomes true before continuing.

The `wait` statement has the following form:

```
wait(<condition_expression>) <statement>
```

The following example shows the use of the `wait` statement to accomplish level-sensitive event control:

```
begin
    wait(!enable) #10 a = b;
    #10 c = d;
end
```

If the value of `enable` is one when the block is entered, the `wait` statement delays the evaluation of the next statement (`#10 a = b;`) until the value of `enable` changes to zero. If `enable` is already zero when the begin-end block is entered, then the next statement is evaluated immediately and no delay occurs.

Intra-Assignment Timing Controls

The delay and event control constructs previously described precede a statement and delay its execution. The intra-assignment delay and event controls are contained within an assignment statement and modify the flow of activity in a slightly different way.

Encountering an intra-assignment delay or event control delays the assignment just as a regular delay or event control does, but the right-hand side expression is evaluated before, not after the delay. This allows data swap and data shift operations to be described without the need for temporary variables. This section describes the purpose of intra-assignment timing controls and the `repeat` timing control that can be used in intra-assignment delays.

Verilog-XL Reference

Behavioral Modeling

The following table illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effects without using intra-assignments.

Intra-assignment timing control with intra-assignment construct	Intra-assignment timing control without intra-assignment construct
<pre>a = #5 b;</pre>	<pre>begin temp = b; #5 a = temp; end</pre>
<pre>a = @(posedge clk) b;</pre>	<pre>begin temp = b; @(posedge clk) a = temp;</pre>
<pre>a = repeat(3)@(posedge clk) b;</pre>	<pre>begin temp = b; @(posedge clk; @(posedge clk; @(posedge clk) a = temp;</pre>

The next three examples use the `fork-join` behavioral construct. All statements between the keywords `fork` and `join` execute concurrently. [“Parallel Blocks”](#) on page 191 describes this construct in more detail.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
    #5 a = b;
    #5 b = a;
join
```

The code in the previous example samples the values of both `a` and `b` at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the following example prevents this race condition:

```
fork                                // data swap
    a = #5 b;
    b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of `a` and `b` to be evaluated *before* the delay, and the assignments to be made *after* the delay. Verilog-XL and other tools that implement intra-assignment timing control use temporary storage in evaluating each expression on the right-hand side.

Intra-assignment waiting for events is also effective. In the example below, the right-hand-side expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal.

Verilog-XL Reference Behavioral Modeling

```
fork                                     // data shift
  a = @(posedge clk) b;
  b = @(posedge clk) c;
join
```

The repeat event control

The `repeat` event control specifies an intra-assignment delay of a specified number of occurrences of an event. This construct is convenient when events must be synchronized with counts of clock signals.

The `repeat` event control syntax is as follows:

```
<repeat_event_controlled_assignment>
  ::= <value> = <repeat_event_control><expression>;
  || <value> <= <repeat_event_control><expression>;

<repeat_event_control>
  ::= repeat (<expression>)@(<identifier>)
  || repeat (<expression>)@(<event_expression>)

<event_expression>
  ::= <expression>
  || posedge<SCALAR_EVENT_EXPRESSION>
  || negedge<SCALAR_EVENT_EXPRESSION>
  || <event_expression>or<event_expression>
```

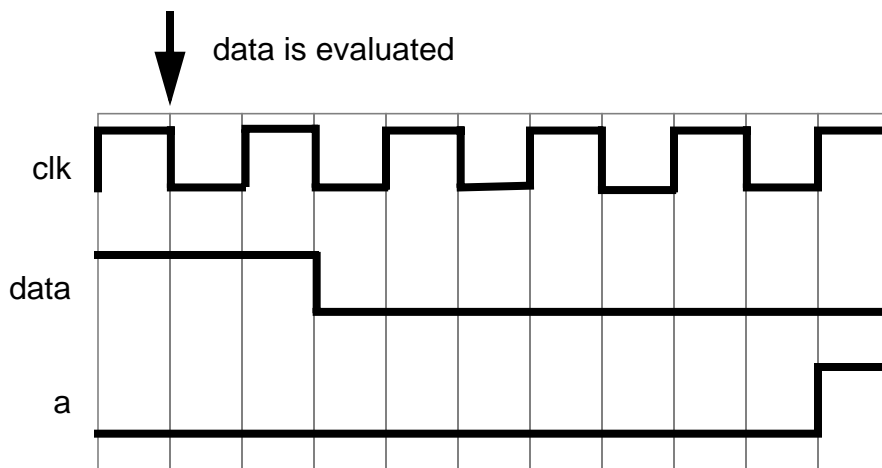
The event expression must resolve to a one bit value.

The following is an example of a `repeat` event control as the intra-assignment delay of a non-blocking assignment:

```
a<=repeat(5)@(posedge clk)data;
```

The following figure illustrates the activities that result from the previous `repeat` event control.

:



In this example, the value of `data` is evaluated when the assignment is encountered. After five occurrences of `posedge clk`, `a` is assigned the previously evaluated value of `data`.

The following is an example of a `repeat` event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num)@(clk)data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the number of transitions of `clk` equals the value of `num`, `a` is assigned the previously evaluated value of `data`.

The following is an example of a `repeat` event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b)@(posedge phi1 or negedge phi2)data;
```

In the example above, the value of `data` is evaluated when the assignment is encountered. After the positive edges of `phi1`, the negative edges of `phi2`, or the combination of these two events occurs a total of $(a+b)$ times, `a` is assigned the previously evaluated value of `data`.

Block Statements

The block statements are a means of grouping two or more statements together so that they act syntactically like a single statement. We have already introduced and used the sequential block statement which is delimited by the keywords `begin` and `end`. The following section discusses sequential blocks in more detail.

A second type of block, delimited by the keywords `fork` and `join`, is used for executing statements in parallel. A `fork-join` block is known as a parallel block, and enables procedures to execute concurrently through time. "[Parallel Blocks](#)" on page 191 discusses parallel blocks.

Sequential Blocks

A sequential block has the following characteristics:

- Statements execute in sequence, one after another.
- The delays are cumulative; each statement executes after all the delays preceding it have elapsed.
- Control passes out of the block after the last statement executes.

Verilog-XL Reference

Behavioral Modeling

The following is the formal syntax for a sequential block:

```
<seq_block>
 ::= begin <statement>* end
 || = begin : <name_of_block>
       <block_declaration>*
       <statement>*
       end

<name_of_block>
 ::= <IDENTIFIER>

<block_declaration>
 ::= <parameter_declaration>
 || = <reg_declaration>
 || = <integer_declaration>
 || = <real_declaration>
 || = <time_declaration>
 || = <event_declaration>
```

A sequential block enables the following two assignments to have a deterministic result:

```
begin
  areg = breg;
  creg = areg; // creg becomes the value of breg
end
```

In the previous example, the first assignment is performed and `areg` is updated before control passes to the second assignment.

Delay control can be used in a sequential block to separate two assignments in time.

```
begin
  areg = breg;
  #10 creg = areg; // this gives a delay of 10 time
end // units between assignments
```

The following example shows how the combination of the sequential block and the delay control can be used to specify a time-sequenced waveform:

Time-Sequenced Waveform

```
parameter d = 50; // d declared as a parameter
reg [7:0] r; // and r declared as an 8-bit register

begin // a waveform controlled by sequential
  // delay
  #d r = 'h35;
  #d r = 'hE2;
  #d r = 'h00;
  #d r = 'hF7;
  #d -> end_wave; // trigger the event called end_wave
end
```

The following code shows three more examples of sequential blocks.:

```
begin
  @trig r = 1;
```

Verilog-XL Reference Behavioral Modeling

```
#250 r = 0; // a 250 delay monostable
end
begin
  @(posedge clock) q = 0;
  @(posedge clock) q = 1;
end
begin // a waveform synchronized by the event c
  @c r = 'h35;
  @c r = 'hE2;
  @c r = 'h00;
  @c r = 'hF7;
  @c -> end_wave;
end
```

Parallel Blocks

A parallel block has the following characteristics:

- Statements execute concurrently.
- Delay values for each statement are relative to the simulation time when the control enters the block.
- Delay control is used to provide time-ordering for assignments.
- Control passes out of the block when the last time-ordered statement executes or when a `disable` statement executes.

The formal syntax for a parallel block is as follows.

```
<par_block>
 ::= fork <statement>* join
 || = fork : <name_of_block>
      <block_declaration>*
      <statement>*
      join
<name_of_block>
 ::= <IDENTIFIER>
<block_declaration>
 ::= <parameter_declaration>
 || = <reg_declaration>
 || = <integer_declaration>
 || = <real_declaration>
 || = <time_declaration>
 || = <event_declaration>
```

The following example codes the waveform description shown in the [Time-Sequenced Waveform](#) figure on page 190 by using a parallel block instead of a sequential block. The waveform produced on the register is exactly the same for both implementations.

```
fork
  #50 r = 'h35;
  #100 r = 'hE2;
```

```
#150 r = 'h00;  
#200 r = 'hF7;  
#250 -> end_wave;  
join
```

Block Names

Blocks can be named by adding `<name_of_block>` after the keywords `begin` or `fork`. The naming of blocks serves several purposes:

- It allows local variables to be declared for the block.
- It allows the block to be referenced in statements like the `disable` statement (as discussed in [Chapter 10, “Disabling of Named Blocks and Tasks”](#)).
- In the Verilog language, all variables are static—that is, a unique location exists for all variables and leaving or entering blocks does not affect the values stored in them. Thus, block names give a means of uniquely identifying all variables at any simulation time. This is very important for debugging purposes because it is necessary to be able to reference a local variable inside a block from outside the body of the block.

Start and Finish Times

Both parallel and sequential blocks have the notion of start and finish times. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement is finished. For parallel blocks, the start time is the same for all the statements, and the finish time is when the last time-ordered statement is finished executing. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution does not continue with the statement following a block until the block’s finish time is reached—that is, until the block is completely finished executing.

Moreover, the timing controls in a `fork-join` block do not have to be given sequentially in time. The following example shows that the statements from the example in [“Parallel Blocks”](#) on page 191 written in the reverse order still produce the same waveform.

```
fork  
  #250 -> end_wave;  
  #200 r = 'hF7;  
  #150 r = 'h00;  
  #100 r = 'hE2;  
  #50 r = 'h35;  
join
```

Sequential and parallel blocks can be embedded within each other allowing complex control structures to be expressed easily with a high degree of structure.

Verilog-XL Reference

Behavioral Modeling

One simple example of this advantage occurs when an assignment is to be made after two separate events have transpired. This is known as the “joining” of events.

```
begin
  fork
    @Aevent;
    @Bevent;
  join
  areg = breg;
end
```

Note: The two events can occur in any order (or even at the same time). The `fork-join` block will complete, and the assignment will be made. In contrast to this, if the `fork-join` block was a `begin-end` block, and the `Bevent` occurred before the `Aevent`, then the block would be deadlocked waiting for the `Bevent`.

The following example shows two sequential blocks, each of which executes when its controlling event occurs. Because the `wait` statements are within a `fork-join` block, they execute in parallel and the sequential blocks can therefore also execute in parallel.

```
fork
  @enable_a
  begin
    #ta wa = 0;
    #ta wa = 1;
    #ta wa = 0;
  end
  @enable_b
  begin
    #tb wb = 1;
    #tb wb = 0;
    #tb wb = 1;
  end
join
```

Behavior Model Examples

This section contains two behavioral model examples. These examples are given as complete descriptions enclosed in modules—such that they can be put directly through the Verilog-XL compiler and simulated, and the results can be observed.

The following example shows a simple traffic light sequencer described with its own clock generator:

```
module traffic_lights;
  reg clock, red, amber, green;
  parameter
    on = 1,
    off = 0,
    red_ticks = 350,
    amber_ticks = 30,
    green_ticks = 200;
endmodule
```

Verilog-XL Reference Behavioral Modeling

```
always          // the sequence to control the lights
begin
    red = on;
    amber = off;
    green = off;
    repeat (red_tics) @(posedge clock);
    red = off;
    green = on;
    repeat (green_tics) @(posedge clock);
    green = off;
    amber = on;
    repeat (amber_tics) @(posedge clock);
end

always          // waveform for the clock
begin
    #100 clock = 0;
    #100 clock = 1;
end

initial        // simulate for 10 changes on the red light
begin
    repeat (10) @red;
    $finish;
end

always          // display the time and changes made to the lights
@(red or amber or green)
$display("%d red=%b amber=%b green=%b",
         $time, red, amber, green);
endmodule
```

The following example shows a use of variable delays. The module has a clock input and produces two synchronized clock outputs. Each output clock has equal mark and space times, is out of phase from the other by 45 degrees, and has a period half that of the input clock.

Note: The clock generation is independent of the simulation time unit, except as it affects the accuracy of the divide operation on the input clock period.

```
module synch_clocks;
    reg
        clock,
        phase1,
        phase2;
    time clock_time;
    initial clock_time = 0;
    always @(posedge clock)
        begin :phase_gen
            time d; // a local declaration is possible
                    // because the block is named
            d = ($time - clock_time) / 8;
            clock_time = $time;
            phase1 = 0;
            #d phase2 = 1;
            #d phase1 = 1;
            #d phase2 = 0;
            #d phase1 = 0;
            #d phase2 = 1;
            #d phase1 = 1;
        end
endmodule
```

Verilog-XL Reference Behavioral Modeling

```
        #d phase2 = 0;
    end
    // set up a clock waveform, finish time,
    // and display
    always
        begin
            #100 clock = 0;
            #100 clock = 1;
        end
    initial #1000 $finish; //end simulation at time 1000
    always
        @(phase1 or phase2)
            $display($time,,
                "clock=%b phase1=%b phase2=%b",
                clock, phase1, phase2);
endmodule
```

Verilog-XL Reference

Behavioral Modeling

Tasks and Functions

This chapter describes the following:

- [Overview](#) on page 197
- [Distinctions Between Tasks and Functions](#) on page 197
- [Tasks and Task Enabling](#) on page 198
- [Functions and Function Calling](#) on page 201

Overview

Tasks and functions provide the ability to execute common procedures at several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make the code easier to read and to debug the source descriptions. `input`, `output`, and `inout` argument values can be passed into and out of both tasks and functions.

Distinctions Between Tasks and Functions

The following rules distinguish tasks from functions:

- A function must execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function must have at least one input argument; a task can have zero or more arguments of any type.
- A function returns a single value; a task does not return a value.

The purpose of a function is to respond to an input value by returning a single value. A task can support multiple goals and can calculate multiple result values. However, only the `output` or `inout` arguments can pass result values back from the invocation of a task. A

Verilog-XL Reference

Tasks and Functions

Verilog model uses a function as an operand in an expression; the value of that operand is the value returned by the function.

For example, you could define either a task or a function to switch bytes in a 16-bit word. The task would return the switched word in an `output` argument, so the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`. A word-switching function would return the switched word directly. Thus, the function call for the function `switch_bytes` might look like the following example:

```
new_word = switch_bytes (old_word);
```

Tasks and Task Enabling

A task is enabled by the statement that defines the argument values to be passed to the task, and by the variables that will receive the results. Control is passed back to the enabling process after the task is complete. Thus, if a task has timing controls inside it, then the time of enabling can be different from the time at which control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control does not return until all enabled tasks are complete.

Defining a Task

The following is the syntax for defining tasks:

```
<task>
 ::= task <name_of_task> ;
      <tf_declaration>*
      <statement_or_null>
      endtask
<name_of_task>
 ::= <IDENTIFIER>
<tf_declaration>
 ::= <parameter_declaration>
    | <input_declaration>
    | <output_declaration>
    | <inout_declaration>
    | <reg_declaration>
    | <time_declaration>
    | <integer_declaration>
    | <real_declaration>
    | <event_declaration>
```

Task and function declarations specify the following:

local variables
I/O ports
registers
times
integers
real
events

These declarations all have the same syntax as the corresponding declarations in a module definition. If there is more than one output, input, and inout port declared in a task, these must be enclosed within a block.

Task Enabling and Argument Passing

The statement that enables a task passes the I/O arguments as a comma-separated list of expressions enclosed in parentheses. The following is the formal syntax of the task-enabling statement:

```
<task_enable>  
  ::= <name_of_task> ;  
  || = <name_of_task> ( <expression> <,<expression>>* ) ;
```

The first form of a task enabling statement applies when there are no I/O arguments declared in the task body. In the second form, the list of *<expression>* items is an ordered list that must match the order of the list of I/O arguments in the task definition.

If an I/O argument is an input, then the corresponding *<expression>* can be any expression. If the I/O argument is an *output* or an *inout*, then Verilog restricts it to an expression that is valid on the left-hand side of a procedural assignment. The following items satisfy this requirement:

- *reg*, *integer*, *real*, and *time variables*
- *memory references*
- concatenations of *reg*, *integer*, *real*, and *time variables*
- concatenations of *memory references*
- *bit-selects* and *part-selects* of *reg*, *integer*, *real*, and *time variables*

The execution of the task-enabling statement passes *input* values from the variables listed in the enabling statement to the variables specified within the task. Execution of the return from the task passes values from the task *output* and *inout* variables to the corresponding variables in the task-enabling statement. Verilog passes all arguments by value—that is, Verilog passes the *value* rather than a *pointer* to the value.

Verilog-XL Reference

Tasks and Functions

The following example illustrates the basic structure of a task definition with five arguments:

```
module this_task;
  task my_task;
    input a, b;
    inout c;
    output d, e;
    reg foo1, foo2, foo3;
    begin
      <statements>    // the set of statements that performs the work of the task
        c = foo1;    // the assignments that initialize
        d = foo2;    // the results variables
        e = foo3;
    end
  endtask
endmodule
```

The following statement enables the task in the previous example:

```
my_task (v, w, x, y, z);
```

The calling arguments (v, w, x, y, z) correspond to the I/O arguments (a, b, c, d, e) defined by the task. At the task-enabling time, the input and inout arguments (a, b, and c) receive the values passed in v, w, and x. Thus, the execution of the task-enabling call effectively causes the following assignments:

```
a = v; b = w; c = x;
```

As part of the processing of the task, the task definition for my_task must place the computed results values into c, d, and e. When the task completes, the processing software performs the following assignments to return the computed values to the calling process:

```
x = c; y = d; z = e;
```

Task Example

The following example illustrates the use of tasks by redescribing the traffic light sequencer that was introduced in [Chapter 8, “Behavioral Modeling.”](#)

```
module traffic_lights;
  reg clock, red, amber, green;
  parameter on = 1, off = 0, red_tics = 350,
            amber_tics = 30, green_tics = 200;
  // initialize colors
  initial
    red = off;
  initial
    amber = off;
  initial
    green = off;
  // sequence to control the lights
  always begin
    red = on; // turn red light on
    light(red, red_tics); // and wait.
    green = on; // turn green light on
```


Verilog-XL Reference Tasks and Functions

```
        light(green, green_tics); // and wait.
        amber = on; // turn amber light on
        light(amber, amber_tics); // and wait.
    end
// task to wait for 'tics' positive edge clocks
// before turning 'color' light off
    task light;
        output color;
        input [31:0] tics;
        begin
            repeat (tics)
                @(posedge clock);
                color = off; // turn light off
        end
    endtask
// waveform for the clock
always begin
    #100 clock = 0;
    #100 clock = 1;
end
endmodule // traffic_lights
```

Effect of Enabling an Already Active Task

Because Verilog supports concurrent procedures, and tasks can have non-zero time duration, you can write a model that invokes a task when that task is already executing (a special case of invoking a task that is already active is when a task recursively calls itself). Verilog-XL allows multiple copies of a task to execute concurrently, but it does not copy or otherwise preserve the task arguments or local variables. Verilog-XL uses the same storage for each invocation of the task. This means that when the simulator interrupts a task to process another instance of the same task, it overwrites the argument values from the first call with the values from the second call. The user must manage what happens to the variables of a task that is invoked while it is already active.

Functions and Function Calling

The purpose of a function is to return a value that is to be used in an expression. The rest of this chapter explains how to define and use functions.

Defining a Function

To define functions, use the following syntax:

```
<function>
 ::= function <range_or_type>? <name_of_function> ;
    <tf_declaration>+
    <statement_or_null>
endfunction
```

Verilog-XL Reference

Tasks and Functions

```
<range_or_type>
 ::= <range>
    | = integer
    | = real

<name_of_function>
 ::= <IDENTIFIER>

<tf_declaration>
 ::= <parameter_declaration>
    | = <input_declaration>
    | = <reg_declaration>
    | = <time_declaration>
    | = <integer_declaration>
    | = <real_declaration>
    | = <event_declaration>
```

A function returns a value by assigning the value to the function's name. The *<range_or_type>* item, which specifies the data type of the function's return, is optional.

The following example defines a function called `getbyte`, using a *<range>* specification.

```
module fact;
  function [7:0] getbyte;
    input [15:0] address;
    reg [3:0] result_expression;
    begin
      //<statements>                               code to extract low-order
      // byte from addressed word
      getbyte = result_expression;
    end
  endfunction
endmodule
```

Returning a Value from a Function

The function definition implicitly declares a register, internal to the function, with the same name as the function. This register either defaults to one bit or is the type that *<range_or_type>* specifies. The *<range_or_type>* can specify that the function's return value is a *real*, an *integer*, or a value with a range of $[n:m]$ bits. The function assigns its return value to the internal variable bearing the function's name. The following line from the previous example illustrates this concept:

```
getbyte = result_expression;
```

Calling a Function

A function call is an operand within an expression. The operand has the following syntax:

```
<function_call>
 ::= <name_of_function> ( <expression> <, <expression>>* )
    <name_of_function>
 ::= <identifier>
```

Verilog-XL Reference

Tasks and Functions

The following example creates a word by concatenating the results of two calls to the function `getbyte` (defined in the example in “[Defining a Function](#)” on page 201).

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)} : 0;
```

Function Rules

Functions are more limited than tasks. The following five rules govern their usage:

- A function definition cannot contain any time controlled statements—that is, any statements introduced with `#`, `@`, or `wait`.
- Functions cannot enable tasks.
- A function definition must contain at least one `input` argument.
- A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.
- A function definition can not contain an `inout` declaration or an `output` declaration.

Function Example

The following example defines a function called `factorial` that returns a 32-bit register. A loop repeatedly executes the `factorial` function and prints the results until `n=10`.

```
module tryfact;
  // define function
  function [31:0] factorial;
    input [3:0] operand;
    reg [3:0] index;

    begin
      factorial = operand ? 1 : 0;
      for(index = 2; index <= operand; index = index + 1)
        factorial = index * factorial;
    end
  endfunction

  // Test the function
  reg [31:0] result;
  reg [3:0] n;

  initial
  begin
    result = 1;
    for(n = 2; n <= 9; n = n+1)
    begin
      $display("Partial result  n=%d result=%d",
              n, result);
      result = n * factorial(n) / ((n * 2) + 1);
    end
  end
endmodule
```

Verilog-XL Reference

Tasks and Functions

```
        $display("Final result=%d", result);
    end
endmodule // tryfact
```

Disabling of Named Blocks and Tasks

This chapter describes the following:

- [Overview](#) on page 205
- [Syntax](#) on page 205
- [disable Statement Examples](#) on page 206

Overview

The `disable` statement allows you to perform the following:

- Terminate the activity associated with concurrently active procedures while maintaining the structured nature of Verilog HDL procedural descriptions.
- Disable activity in the particular block or task containing the `disable` statement.
- Handle exception conditions such as hardware interrupts and global resets.

The `disable` statement can also:

- Return from a task before executing all the statements in the task.
- Break from a looping statement.
- Skip statements to continue with another iteration of a looping statement.

Syntax

The `disable` statement has one of the following two syntax forms:

```
<disable_statement>  
 ::= disable <name_of_task> ;  
 ||= disable <name_of_block> ;
```

Verilog-XL Reference

Disabling of Named Blocks and Tasks

The `disable` statement removes the evaluated and scheduled nonblocking procedural assignments from the schedule of events. Execution resumes at the statement following either the named block or the task enabling statement.

Termination of activity also applies to all activity enabled within the named block or task. If task `enable` statements are nested—that is, if one task enables another, and that one enables yet another—then disabling a task within the chain disables all tasks downward on the chain. The following is a simple example showing a `disable` statement that disables the block that contains the statement.

```
module disable_block;
reg rega, regb, regc;
initial
begin :block_name
    rega = regb;
    disable block_name;
    regc = rega; // this assignment will never execute
end
endmodule
```

disable Statement Examples

The following example shows the `disable` statement being used within a named block in a manner similar to a forward `goto`. If `a` equals 0, the `disable` statement is executed, and the next statement that is executed is the one following the named block.

```
module disable_block;
reg a;
initial
begin : block_name
    ...
    if(a == 0) disable block_name;
    ...
end // end of named block
// continue with code following named block
    ...
endmodule
```

The following example shows the `disable` statement being used as an early return from a task:

```
module task;
reg a;
task proc_a;
begin
    ...
    if(a == 0) disable proc_a; // return if true
    ...
end
endtask
endmodule
```

Verilog-XL Reference

Disabling of Named Blocks and Tasks

The following example shows the `disable` statement being used in a way that is analogous to the `continue` and `break` statements in the C language.

```
module disable2;
integer i;
reg a, b, n, clk;
initial
begin :break
  for(i = 0; i < n; i = i+1)
    begin :continue
      @clk
        if(a == 0)          // "continue" loop
          disable continue;
        ...<statements>...
      @clk
        if(a == b)         // "break" from loop
          disable break;
        ...<statements>...
    end
end
initial #10 clk = ~ clk;
endmodule
```

The previous example illustrates control code that allows a named block to execute until a loop counter reaches `n` iterations, or until the variable `a` gets set to a value of `b`. The named block `break` contains the code that executes until `a == b`, at which point the `disable break` statement terminates the execution of that block. The named block `continue` contains the code that executes for each iteration of the `for` loop. Each time this code executes the `disable continue` statement, the `continue` block terminates and execution passes to the next iteration of the `for` loop. For each iteration of the `continue` block, a set of `<statements>` executes if (`a != 0`). Another set of `<statements>` executes if (`a!=b`).

The following example shows the `disable` statement being used to concurrently disable a sequence of timing controls and the task `action`, when the `reset` event occurs:

```
fork
  begin :event_expr
    @ev1;
    repeat (3) @trig;
    #d action(areg, breg);
  end
  @reset disable event_expr;
join
```

This example shows a `fork/join` block within which is a named sequential block (`event_expr`) and a `disable` statement that waits for the occurrence of the event `reset`. The sequential block and the wait for `reset` execute in parallel. The `event_expr` block waits for one occurrence of the event `ev1` and three occurrences of the event `trig`. When these four events have happened, plus a delay of `d` time units, the task `action` executes. When the event `reset` occurs, regardless of events within the sequential block, the `fork/join` block terminates—including the task `action`.

Verilog-XL Reference

Disabling of Named Blocks and Tasks

The following example is a behavioral description of a monostable that can be triggered again. The named event `retrig` restarts the monostable time period. If `retrig` continues to occur within 250 time units, then `q` will remain at 1.

```
always
  begin :monostable
    #250 q = 0;
  end

always @retrig
  begin
    disable monostable;
    q = 1;
  end
```

Hierarchical Structures

This chapter describes the following:

- [Overview](#) on page 209
- [Modules](#) on page 210
- [Overriding Module Parameter Values](#) on page 213
- [Macro Modules](#) on page 216
- [Ports](#) on page 219
- [Hierarchical Names](#) on page 228
- [Automatic Naming](#) on page 233
- [Scope Rules](#) on page 234

Overview

The Verilog HDL supports a hierarchical hardware description structure by allowing modules to be embedded within other modules. Higher-level modules create instances of lower-level modules and communicate with them through input, output, and bidirectional ports. These module input/output ports can be scalar or vector.

As an example of a module hierarchy, consider a system consisting of printed circuit boards. The system would be represented as the top-level module and would create instances of modules that represent the boards. The board modules would, in turn, create instances of modules that represent ICs, and the ICs could, in turn, create instances of modules that represent predefined cells such as flip-flops, mux's, and alu's.

To describe a hierarchy of modules, the user provides textual definitions of the various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

Modules

This section gives the formal syntax for a module definition and then gives the syntax for module instantiation, along with an example of a module definition and a module instantiation.

A module definition is enclosed between the keywords `module` and `endmodule`, where the `<IDENTIFIER>` after `module` gives the name of the module. The optional `<list_of_ports>` specifies an ordered list of the module's I/O ports. The order used can be significant when instantiating the module (see [“Connecting Module Ports by Ordered List”](#) on page 220). The identifiers in this list must be declared in `input`, `output`, and `inout` statements within the module definition. The `<module_items>` define what constitutes a module, and include many different types of declarations and definitions; many of them have already been introduced.

```
<module>
 ::= module <name_of_module><list_of_ports>? ;
    <module_item>*
    endmodule

<name_of_module>
 ::= <IDENTIFIER>

<list_of_ports>
 ::= ( <port> <,><port>)* )

<module_item>
 ::= <parameter_declaration>
    = <input_declaration>
    = <output_declaration>
    = <inout_declaration>
    = <net_declaration>
    = <reg_declaration>
    = <time_declaration>
    = <integer_declaration>
    = <real_declaration>
    = <event_declaration>
    = <gate_instantiation>
    = <primitive_instantiation>
    = <module_instantiation>
    = <parameter_override>
    = <continuous_assign>
    = <specify_block>
    = <initial_statement>
    = <always_statement>
    = <task>
    = <function>
```

See [“Ports”](#) on page 219 for the definitions of the syntax item `<port>`. See [“Module Definition and Instance Example”](#) on page 211 for module definition and instantiation examples.

Top-Level Modules

Top-level modules are modules that are included in the source text supplied as input to a particular simulation run, but are not instantiated, as described in the following section.

Module Instantiation

Instantiation allows one module to incorporate a copy of another module into itself. Module definitions do not nest. That is, one module definition cannot contain the text of another module definition within its `module/endmodule` keyword pair. A module definition nests another module by *instantiating* it. The `<module_instantiation>` statement creates one or more named *instances* of a defined module. For example, a counter module might instantiate a D flip-flop module to create eight instances of the flip-flop.

The following is the syntax for specifying instantiations of modules:

```
<module_instantiation>
  ::= <name_of_module> <parameter_value_assignment>?
     <module_instance> <,<module_instance>>* ;
<name_of_module>
  ::= <IDENTIFIER>
<parameter_value_assignment>
  ::= # ( <expression> <,<expression>>* )
<module_instance>
  ::= <name_of_instance> ( <list_of_module_connections>? )
<name_of_instance>
  ::= <IDENTIFIER>
<list_of_module_connections>
  ::= <module_port_connection> <,<module_port_connection>>*
     || <named_port_connection> <,<named_port_connection>>*
<module_port_connection>
  ::= <expression>
     || <NULL>
<named_port_connection>
  ::= .<IDENTIFIER> ( <expression>? )
```

The definition for `<named_port_connection>` includes an `<IDENTIFIER>` token that can be satisfied only with a port name from the definition of the module being instantiated. See [“Connecting Module Ports by Name”](#) on page 221 for more details.

Module Definition and Instance Example

The code in the following example illustrates a circuit (the lower-level module) being driven by a simple waveform description (the higher-level module) where the circuit module is instantiated inside the waveform module.

Verilog-XL Reference

Hierarchical Structures

```
// THE LOWER-LEVEL MODULE:
//module description of a nand flip-flop circuit
module ffnand (q, qbar, preset, clear);
    output q, qbar;           //declares 2 circuit output nets
    input preset, clear;     //declares 2 circuit input nets

    nand
        //declaration of two nand gates and
        //their interconnections
        g1 (q, qbar, preset),
        g2 (qbar, q, clear);
endmodule

// THE HIGHER-LEVEL MODULE:
//a waveform description for the nand flip-flop
module ffnand_wave;
    wire out1, out2;        //outputs from the circuit
    reg in1, in2;           //variables to drive the circuit

    //instantiate the circuit ffnand, name it "ff",
    //and specify the IO port interconnections
    ffnand ff(out1, out2, in1, in2);

    //define the waveform to stimulate the circuit
    parameter d = 10;
    initial
        begin
            #d in1 = 0; in2 = 1;
            #d in1 = 1;
            #d in2 = 0;
            #d in2 = 1;
        end
endmodule
```

You can specify one or more module instances (identical copies of a module) in a single module instantiation statement.

The list of module terminals is provided only for modules defined with terminals. The parentheses, however, are always required. When a list of module terminals is given, the first element in the list connects to the first port, the second to the second port, and so on. See [“Ports”](#) on page 219 for a more detailed discussion of ports and port connection lists.

A terminal can be a simple reference to a variable, an expression, or a blank. You can use an expression for supplying a value to a module input port.

A blank module terminal represents a situation in which the I/O port is not to be connected. For example, the following instantiation of the three-port module definition `qq` has an unconnected port:

```
qq qq_num1 (a,,c);
```

The following is another instantiation of `qq` with connections by name that has an unconnected port indicated by empty parentheses:

```
qq qq_num2 (.a(x),.b(),.c(y));
```

Verilog-XL Reference

Hierarchical Structures

As shown in “[Module Instantiation](#)” on page 211, a module instantiation can have either order-based module connections or module connections based on names in its module definition. The two types of module port connections cannot be mixed. If an instantiation with either type of module connection has fewer items in its list of module connections than there are items in the list of ports in its module definition, the compiler generates a warning. Depending on whether order-based or name-based connections generate the warning, the warning’s error code varies in the letter that follows TF.

The following warning applies to order-based lists:

```
Warning! Too few module port connections      [Verilog-TFMPC]
        "a.v", 8:t1(In)
```

The following warning applies to name-based lists:

```
Warning! Too few module port connections      [Verilog-TFNPC]
        "a.v", 7:t1(.in(In))
```

The code in the following example creates two instances (`ff1` and `ff2`) of the flip-flop module `ffnand`, which is defined in the example at the beginning of this section, and connects only to the `q` output in one instance (`out1`) and only to the `qbar` output in the other instance (`out2`).

```
//a waveform description for testing the nand flip-flop
//without the outputs
module ffnand_wave;
    reg in1, in2; //variables to drive the circuit
    //make two copies of the circuit ffnand
    //and connect to one output for each
    ffnand
        ff1(out1, , in1, in2),
        ff2( , out2, in1, in2);
    //define the waveform to stimulate the circuit
    parameter d = 10;
    initial
        begin
            #d in1 = 0; in2 = 1;
            #d in1 = 1;
            #d in2 = 0;
            #d in2 = 1;
        end
endmodule
```

Overriding Module Parameter Values

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module. There are two ways to alter parameter values: the `defparam` statement, which allows assignment to parameters using their hierarchical

names, and the module instance parameter value assignment, which allows values to be assigned in-line during module instantiation. The next two sections describe these two methods.

Using the defparam Statement

Using the `defparam` statement, you can change parameter values in any module instance throughout the design using the hierarchical name of the parameter. The `defparam` statement is particularly useful for grouping all of the parameter value override assignments together in one module.

The code in the following example illustrates the use of a `defparam`:

```
module top;
  parameter tenten=10;
  reg clk;
  reg [0:4] in1;
  reg [0:9] in2;
  wire [0:4] o1;
  wire [0:9] o2;
  vdff m1 (o1, in1, clk);
  vdff m2 (o2, in2, clk);
endmodule

module vdff (out, in, clk);
  parameter size = 1, delay = 1;
  input [0:size-1] in;
  input clk;
  output [0:size-1] out;
  reg [0:size-1] out;

  always @(posedge clk)
    # delay out = in;
endmodule

module annotate;
  parameter ten = 10;
  defparam
    top.m1.size = 5,           // <--- referenced parameter must
    top.m1.delay = ten,       // be declared in the same module
    top.m2.size = 10,         // as the defparam statement
    top.m2.delay = 20;
endmodule
```

The expressions on the right-hand side of the `defparam` assignments must be constant, involving only numbers and references to parameters. The referenced parameters (on the right-hand side of the `defparam`) must be declared in the same module as the `defparam` statement.

Using Module Instance Parameter Value Assignment

An alternative method for assigning values to parameters within module instances is similar in appearance to the assignment of delay values to gate instances. It uses the syntax # (*<expression>* <,<expression>*) to supply values for particular instances of a module to any parameters that have been specified in the definition of that module.

Consider the following example, in which the parameters within module instance `mod_a` are changed during instantiation. The name of the module being instantiated is `vdff`. The construct `#(10,15)` assigns values to parameters used in the `mod_a` instance of `vdff`.

```
module m;
  reg clk;
  wire[1:10] out_a, in_a;
  wire[1:5] out_b, in_b;
  // create an instance and set parameters
  vdff #(10,15)
    mod_a(out_a, in_a, clk);
  // create an instance leaving default values
  vdff
    mod_b(out_b, in_b, clk);
endmodule

module vdff (out, in, clk);
  parameter size = 1, delay = 1;
  input [0:size-1] in;
  input clk;
  output [0:size-1] out;
  reg [0:size-1] out;

  always @(posedge clk)
    # delay out = in;
endmodule
```

The order of the assignments in a module instance parameter value assignment follows the declaration order of the parameters within the module. In the example above, `size` is assigned the value 10 and `delay` is assigned the value 15 for the instance of module `vdff` called `mod_a`. Notice that the default size defined in `vdff` for `in` and `out` is one bit and, therefore, the compilation of module `m` with module `vdff` results in the following Verilog-XL warning message: `Port sizes differ in port connection.`

It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter. This means that if you want to assign values to a subset of the parameters declared within a module, then you must declare the parameters that make up this subset prior to declaring the parameters to which you do *not* want to assign values. An alternative is to assign values to all of the parameters, but use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that you do not want to affect.

The assignment of parameter values using a module instance parameter value assignment uses less memory during simulation than the equivalent assignment using the `defparam` statement. The use of this method therefore improves compilation times.

Parameter Dependence

You can define a parameter (for example, `memory_size`) with an expression containing another parameter (for example, `word_size`). Since `memory_size` depends on the value of `word_size`, a modification of `word_size` changes the value of `memory_size`.

For example, in the following parameter declaration, an update of `word_size`, whether by `defparam` or in an instantiation statement for the module that defined these parameters, automatically updates `memory_size`.

```
parameter
    word_size = 32,
    memory_size = word_size * 4096;
```

Macro Modules

The Verilog language includes a construct called a macro module. A macro module serves the same functions as a standard module, but because it conforms to certain limitations, it can simulate much faster in Verilog-XL.

The way that module instances are created in the Verilog-XL internal data structure carries a fairly high overhead in memory usage during compilation. This can have a severe impact on the runtime for designs that contain many instances of simple modules, such as a gate array or a standard cell design. Macro modules help to reduce this overhead.

When the simulator compiles an instance of a macro module, it merges the macro module definition with the definition of the module that contains the macro instance. It creates no name scope and makes no port connections. Instead, it places the macro definition at the same hierarchical level as the containing module. This process is called *macro module expansion*. A compiled macro module instance is said to be *expanded*.

Constructs Allowed in Macro Modules

The contents of macro modules are limited to the following constructs:

- gate and switch instances
- user-defined primitive instances
- nets
- parameter declarations (used for specifying delays)

The following restrictions apply to the constructs used in macro modules:

- The terminal lists in gate instances and the port lists in UDP instances cannot contain expressions with variable operands (such as those used in dynamic bit selects).
- Procedural statements and register declarations are illegal in macro modules. If these are present, then the module is not expanded; it is treated as a normal module.
- The only valid use for parameters in macro modules is to specify delays.

Specifying Macro Modules

You can specify macro modules by using the keyword `macromodule` in place of the keyword `module` in the module definition. The following example defines a macro module called `NAND2`.

```
macromodule NAND2(q, a, b);
output q;
input a, b;
    nand (q,a,b);
endmodule
```

Instances of Macro Modules

Instances of macro modules are specified in exactly the same way as instances of normal modules.

If there are part-selects or concatenations in the port connections, then the macro module instance simulates as a normal module, and no memory savings occur.

If macro module instances are expanded, they will not be tagged as cells even if they appear between the ``celldefine` and ``endcelldefine` compiler directives. PLI access routines that recognize cells, such as `acc_next_cell`, do not select expanded macro modules.

Using Parameters with Macro Modules

Observe the following restrictions when using parameters with macro modules:

- The value expression given in the parameter definition must be a constant expression that does not depend on other parameters. If the definition contains references to other parameters, the macro module is not expanded.
- The `defparam` statement cannot be used to redefine parameter values within macro module instances. Therefore, module instance parameter value assignment is the only method available for changing the value of parameters within macro module instances.

Verilog-XL Reference

Hierarchical Structures

If you redefine parameter values within macro module instances using the `defparam` statement you will get an error message indicating that the `defparam` statement is illegal.

Effect on Decompile and Tracing

When you decompile a macro module instance using `$list` or the `-d` command option, or when you trace the statements within a macro module using the single step or `$settrace` commands, the statements appear as part of the module containing the macro module instance.

The code in the following example uses `$list` to illustrate this point:

```
module topmod;
  wire [4:0] v;
  wire a,b,c,w;

  modB b1 (v[0], v[3], w, v[4]);
  initial $list;
endmodule

macromodule modB(wa, wb, c, d);
  inout wa, wb;
  input c, d;

  parameter d1=6, d2=5;
  parameter d3=2, d4=6;

  tranif1 g1(wa, wb, cinvert);
  not #(d3, d4) (cinvert, int);
  and #(d1, d2) g2(int, c, d);
endmodule
```

Running Verilog on the above description yields the results shown in the following example. Note that the module boundary for `modB` has disappeared, and that the gates inside `modB` are now inside `topmod`.

Compiling source file "test.v"
Highest level modules:
topmod

```
1      module topmod;
2          wire [4:0]
2              v; // = 5'hz, z (scalared)
3          wire
3              a, // = HiZ
3              b, // = HiZ
3              c, // = HiZ
3              w; // = HiZ
17         tranif1
17             \b1^g1 (v[0], v[3], \b1^cinvert );
18         not #(2, 6)
18             (\b1^cinvert , \b1^int );
19         and #(6, 5)
19             \b1^g2 (\b1^int , w, v[4]);
6         initial
6*             $list;
8     endmodule
```

Verilog-XL Reference

Hierarchical Structures

```
3 simulation events
CPU time: 0 secs to compile + 0 secs to link + 0 secs in simulation
```

In the above screen display, notice the text `\b1^` is prefixed to all identifiers in the macro module. See [“Macro Modules and Hierarchical Names”](#) on page 231 for an explanation of this convention.

Ports

Ports provide a means of interconnecting a hardware description consisting of modules, primitives, and macro modules. For example, module `A` can instantiate module `B`, using port connections appropriate to module `A`. These port names can differ from the names of the internal nets and registers specified in the definition of module `B`, but the connection is still made.

Port Definition

The syntax for a port is given below (this is the completion of the syntax presented in [“Modules”](#) on page 210):

```
<port>
    ::= <port_expression>?
       || = .<name_of_port>( <port_expression>? )

<port_expression>
    ::= <port_reference>
       || = { <port_reference> <, <port_reference>>* }

<port_reference>
    ::= <name_of_variable>
       || = <name_of_variable> [ <constant_expression> ]
       || = <name_of_variable>
           [ <constant_expression> : <constant_expression> ]

<name_of_port>
    ::= <IDENTIFIER>
```

The `<port_expression>` syntax item in the `<port>` definition can be one of the following:

- a simple identifier
- a bit-select of a vector declared within the module
- a part-select of a vector declared within the module
- a concatenation of any of the above

Bit-selects and part-selects result in the automatic expansion of the vector nets they reference. Note that the *<port_expression>* is optional because ports can be defined that do not connect to anything internal to the module.

Port Declarations

Each port listed in the module definition's *<list_of_ports>* must be declared in the body of the module as an *input*, *output*, or bidirectional *inout*. This is in addition to any other declaration for a particular port—for example, a *net*, a *reg*, or a *wire*. The syntax for port declarations is as follows:

```
<input_declaration>
    ::=input <range>? <list_of_variables> ;

<output_declaration>
    ::= output <range>? <list_of_variables> ;

<inout_declaration>
    ::= inout <range>? <list_of_variables> ;
```

Connecting Module Ports by Ordered List

One method of making the connection between the ports listed in a module instantiation and the ports defined by the instantiated module is the ordered list—that is, the ports listed for the module instance are in the same order as the ports listed in the module definition.

The following example illustrates a top-level module (*topmod*) that instantiates a second module (*modB*). Module *modB* has ports that are connected by an ordered list. The connections made are as follows:

- Port *wa* in the *modB* definition connects to the bit-select *v[0]* in the *topmod* module.
- Port *wb* connects to *v[3]*.
- Port *c* connects to *w*.
- Port *d* connects to *v[4]*.

```
module topmod;
wire [4:0] v;
wire a,b,c,w;
...
modB b1 (v[0], v[3], w, v[4]);
...
endmodule

module modB(wa, wb, c, d);
inout wa, wb;
input c, d;
```

Verilog-XL Reference

Hierarchical Structures

```
    tranif1 g1(wa, wb, cinvert);
    not #(2, 6) (cinvert, int);
    and #(6, 5) g2(int, c, d);
endmodule
```

In the `modB` definition, ports `wa` and `wb` are declared as `inouts` while ports `c` and `d` are declared as `input`.

During simulation of the `b1` instance of `modb`, the `and` gate activates first to produce a value on `int`. This value triggers the `not` gate to produce output on `cinvert`, which then activates the `tranif1` gate `g1`.

Connecting Module Ports by Name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection—the name used in the module definition, followed by the terminal used in the instantiating module. The terminal used in the instantiating module can be a reference to a variable, an expression, or a blank. This compound name is then placed in the list of module connections. The following is the syntax for connection by name:

```
.<name_of_port>(<expression>?)
```

.<name_of_port>

The `.<name_of_port>` is the name specified in the module definition for the module of which you are making an instance. The `.<name_of_port>` must meet the following conditions:

- `.<name_of_port>` must be identical in both the module definition port list and the list of port connections for the instances of the module.
- `.<name_of_port>` cannot be a bit-select, a part-select, or a concatenation of ports, with the exception of a method shown in the following explanation.

You cannot connect module ports by name for an instance of a module definition with a list of ports that contains a range, such as a definition that begins with the following lines:

```
module modA (p1[7:0], p2[3:0]);
input [7:0] p1;
output [3:0] p2;
```

The following declarations are functionally identical to the preceding and differ only in the absence of an explicit range in the port list. You can connect module ports by name for an instance of a module definition that begins as shown in the following example:

```
module modA (p1, p2);
input [7:0] p1;
output [3:0] p2;
```

Verilog-XL Reference

Hierarchical Structures

To connect module ports by name for an instance of a module definition that has a bit-select, part-select, or concatenation in its list of ports, you can define the module with a port list as shown in the following example:

```
module modB (q, .p1({q[7:4], q[3:0]}), .p2({q, q[5]}), .p3(q[4:0]));
output [7:0] q;
```

The ports in the preceding definition's list of ports have names, and as a result they can serve as *<port_name>*s in instances. For example, the following instance uses the port names:

```
modB B1 (.p1({wirea[11:6], wirea[1:0]}), .p3(wireb[63:59]), .p2(wirec));
```

The *wirec* identifier in the last connection in the module instance above names a nine-bit wire.

<expression>

The *<expression>* is the element in the instantiating module that connects to the port of the instantiated module, and is one of the following:

- a simple identifier
- a bit-select of a vector declared within the module
- a part-select of a vector declared within the module
- a concatenation of any of the above
- an expression
- a blank ()— empty parentheses document the existence of the port without connecting it to anything. The parentheses are required.

In the following example, the instantiating module connects its signals *topA* and *topB* to the ports *In1* and *Out* defined by the module *ALPHA*. At least one port provided by *ALPHA* is unused; it is named *In2*. There may be other unused ports not mentioned in the instantiation.

```
ALPHA instance1 (.Out(topB), .In1(topA), .In2());
```

The following example defines the modules *modB* and *topmod*. and then *topmod* instantiates *modB* using ports connected by name.

```
module topmod;
wire [4:0] v;
wire a,b,c,w;
    modB b1 (.wb(v[3]), .wa(v[0]), .d(v[4]), .c(w));
endmodule

module modB(wa, wb, c, d);
inout wa, wb;
input c, d;
```

Verilog-XL Reference

Hierarchical Structures

```
    tranif1 g1(wa, wb, cinvert);
    not #(6, 2) (cinvert, int);
    and #(5, 6) g2(int, c, d);
endmodule
```

Note: Because these connections are made by name, the order in which they appear is irrelevant. Note also that the two types of module port connections cannot be mixed; connections to the ports of a particular module instance must be made either all by position or all by name. Syntax errors result from attempts to mix the two.

Real Numbers in Port Connections

The `real` data type cannot be directly connected to a port. Rather it must be connected indirectly, as shown in the following example. The system functions `$realtobits` and `$bitstoreal` are used for passing the bit patterns across the module ports.

```
module driver (net_r);
    output net_r;
    real r;
    wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
    input net_r;
    wire [64:1] net_r;
    real r;
    initial assign r = $bitstoreal(net_r);
endmodule
```

Port Collapsing

A port of a module can be viewed as providing a link or connection between two items (nets, registers, expressions, and so on)—one internal to the module instance and one external to the module instance. Wherever it is possible, the Verilog-XL simulator collapses port connections during processing—that is, the two items become one entity. Both names continue to exist for reference purposes, but internally, the simulator eliminates one of the items. This corresponds to the physical case in which a net described at two levels of a Verilog HDL hierarchy is actually just one wire.

An examination of the port connection rules described below will show that the item receiving the value of the port (the inside item for inputs, the outside item for outputs) must be a net. The item that provides the value can be any expression, but port collapsing is only possible if both items are nets. Using an expression such as $(a+b)$ as the outside item in a module port connection precludes the collapsing of that port.

The Verilog-XL simulator sometimes expands vector nets. This makes simulation more efficient by allowing ports to be collapsed that otherwise could not be collapsed. This happens when a bit-select or part-select of a vector net is connected to a module output or inout port.

The expansion has no effect on simulation results and will only be observed when you execute the `$showvars` system task. When the simulator expands a vector net, `$showvars` lists each bit of the net separately, along with its strength.

A user will rarely need to know the details of port collapsing or vector net expansion. For certain cases, such as when errors or warnings are issued by the compiler, the rules in the following section will explain what has happened.

Port Connection Rules

The following rules govern the ways module ports are declared and the ways they are interconnected:

Rule 1

An input or inout port must be declared as a net type.

Rule 2

Each port connection is a continuous assignment of source to sink—that is, where one connected item is a signal source and the other is a signal sink. Only nets are permitted to be the sinks in an assignment.

Both scalar and vector nets are permitted. The output ports of a module are by definition connected to signal source items internal to the module. The following external items cannot be connected to the output or inout ports of modules:

- registers
- expressions other than:
 - a scalar net
 - a vector net
 - a constant bit-select of a vector net
 - a part-select of a vector net
 - a concatenation of the expressions listed above

Rule 3

A constant bit- or part-select of a vector net that is specified as the external item connected to an output or inout port of a module causes the expansion of the vector net.

It is legal to connect items of any size when making inter-module port connections. However, a warning is issued whenever the sizes of the connected items are not the same.

In port collapsing, the two items that are connected through a module port—one being external to the module, the other being internal to the module—are merged into a single item with accompanying reduction in simulation events. Not every port can be collapsed. The following rule defines when port collapsing occurs:

Rule 4

A module port is collapsed only if:

- the port connects two nets, *and*
- the connected nets are either both scalars or have the same vector size.

Vector nets must be expanded before they can be collapsed. Verilog-XL automatically expands vector nets so that port collapsing can occur in a circuit. Splitting causes a vector net to be internally represented as a collection of scalars, thus allowing this rule to be applied. This occurs whenever the items on both sides of the port are nets, and either following condition:

- at least one of them is a bit-select or part-select of a vector net
- the net is specified with the keyword `scalared`

Given [Rule 4](#), it is clear that only ports that connect nets can be collapsed. But what happens if the nets on either side of the port are of different net types—for example, when one is a `triand` and the other is a `tri`? When different net types are connected through a module port and the port can be collapsed, the resulting net type is determined based on [Rule 6](#).

In [Rule 6](#) for two net types with identical signal sources, net type A “dominates” net type B if either of the following are true:

- The state on B is the same as that on A.
- The state on B is not completely known but does not conflict with the state on A (for example, X does not conflict with 1 or 0; H does not conflict with Z or 1; and so on).

Rule 5

Connections to inout ports must be collapsible onto lower-level nets.

Rule 6

When the two nets connected by a collapsed port are of different net type, the resulting single net is assigned one of the following:

- The dominating net type, if one of the two nets is “dominating”
- The net type external to the module

When Verilog-XL applies this rule, it issues a warning message whenever a dominating net type does not exist. When a dominating net type does not exist, the external net type is used.

The following table shows the net type dictated by [Rule 6](#) as a result of collapsing a module port that connects two nets.

Verilog-XL Reference

Hierarchical Structures

The simulated net takes the net type specified in the table plus the delay specified for that net. If the simulated net selected is a `triereg`, any strength value specified for the `triereg` applies to the simulated net.

	<code>wire & tri</code>	<code>wand & triand</code>	<code>wor & trior</code>	<code>triereg</code>	<code>tri0</code>	<code>tri1</code>	<code>supply 0</code>	<code>supply 1</code>
I N T E R N A L N E T	<code>wire & tri</code>	<code>wand & triand</code>	<code>wor & trior</code>	<code>triereg</code>	<code>tri0</code>	<code>tri1</code>	<code>supply 0</code>	<code>supply 1</code>
	ext	ext	ext	ext	ext	ext	ext	ext
	int	ext	warn	warn	warn	warn	ext	ext
	int	warn	ext	warn	warn	warn	ext	ext
	int	warn	warn	ext	ext	ext	ext	ext
	int	warn	warn	int	ext	warn	ext	ext
	int	warn	warn	int	warn	ext	ext	ext
	int	int	int	int	int	int	ext	warn
	int	int	int	int	int	int	warn	ext

KEY

ext	The external net is used for merging
int	The internal net is used for merging
warn	A warning is issued and the external net type is used

Port Connections in Macro Modules

Combinations of net types across the module port connections of normal modules affect whether the module port connections can be collapsed, as described in [“Port Connection](#)

[Rules](#)” on page 224. Similarly, these combinations affect whether macro modules are expanded, and whether a warning message is printed during compilation.

Using the table in [Rule 6](#) of the previous section, you can determine the effect of various net type combinations on macro module expansion. A macro module can be expanded only for the net combinations that select the external type according to the table in [Rule 6](#). Those combinations for which the table dictates the internal net type do not allow the macro module to be expanded; the instance is treated just as it would be if the module were not a macro module. Those combinations that trigger a warning message for a normal module instance also trigger a warning for a macro module instance. In cases where a warning is issued for a normal module instance, the macro module is expanded.

Hierarchical Names

Every identifier in a Verilog description has a unique hierarchical path name. The hierarchy of modules and the definition of items such as tasks and named blocks within modules define these path names. The hierarchy of names can be viewed as a tree structure, in which each module instance, task, function, or named `begin-end` or `fork-join` block defines a new hierarchical level, (also known as a scope).

At the top of the scope are the names of modules for which no instances have been created. The top of the scope is the root of the hierarchy. Inside any module, each module instance, task definition, function definition, and named `begin-end` or `fork-join` block defines a new branch of the hierarchy. Named blocks within named blocks and within tasks and functions also create new branches.

Each node in the hierarchical name tree is a separate scope with respect to identifiers. A particular identifier can be declared, at most, once in any scope. See [“Scope Rules”](#) on page 234 for a discussion of scope rules.

You can reference any named Verilog object uniquely in its full form by concatenating the names of the modules, tasks, functions, or blocks that contain it. Use the period character to separate each of the names in the hierarchy. For example, `wave.a.bmod.keep.hold` shows five levels. The complete path name to any object starts at a top-level module. You can use this path name from any level in the description. The first node name in this path name (`wave`, in the example) can also be the top of a hierarchy that starts at the level in which the path is being used.

Note: Because the Verilog-XL system automatically generates names for unnamed instances of primitives, you can reference those instances interactively by using the system-generated names. See [“Automatic Naming”](#) on page 233 for information about system-generated names.

Verilog-XL Reference

Hierarchical Structures

The code in the following example defines a hierarchy of module instances and named blocks. [“Hierarchy in a model”](#) on page 230 illustrates the hierarchy implicit in this Verilog code. Following the example is a list of the hierarchical forms of the names of all the objects defined in the code.

```
module mod(in);
    input in;
    always @(posedge in)
    begin :keep
        reg hold;
        hold = in;
    end
endmodule

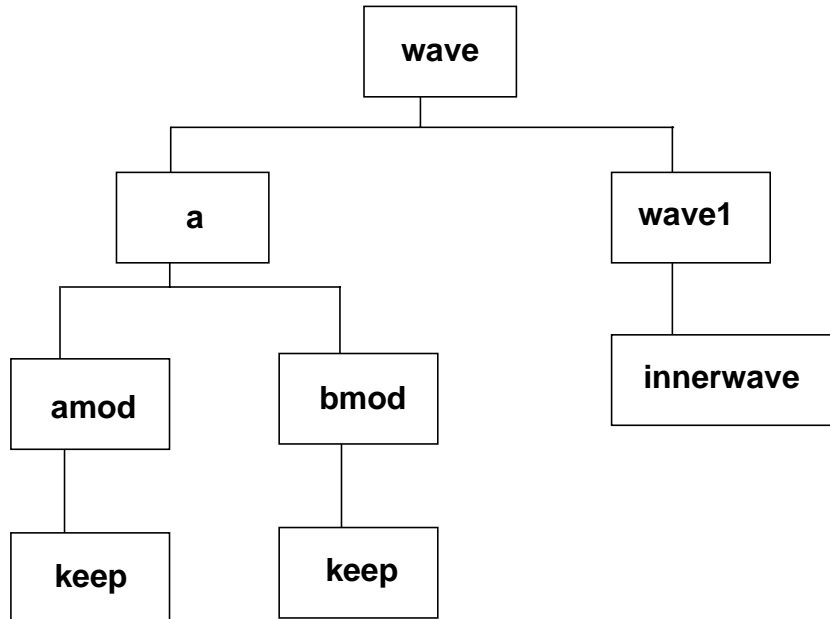
module cct(stim1, stim2);
    input stim1, stim2;
    // instantiate mod
    mod amod(stim1), bmod(stim2);
endmodule

module wave;
    reg stim1, stim2;
    // instantiate cct
    cct a(stim1, stim2);
    initial
    begin :wave1
        #100
        fork :innerwave
            reg hold;
            join
            #150
            begin
                stim1 = 0;
            end
        end
    end
endmodule
```

The following list gives the hierarchical path names for all the objects in the previous example:

```
wave
wave.stim1
wave.stim2
wave.a
wave.a.stim1
wave.a.stim2
wave.a.amod
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
wave.a.bmod
wave.a.bmod.in
wave.a.bmod.keep
wave.a.bmod.keep.hold
wave.wave1
wave.wave1.innerwave
wave.wave1.innerwave.hold
```

Hierarchy in a model



Hierarchical name referencing allows free data access to any object from any level in the hierarchy. If the unique hierarchical path name of an item is known, its value can be sampled or changed from anywhere within the description. This feature is particularly powerful for symbolic debugging.

The following example shows how a pair of named blocks can refer to items declared within one another:

```
begin
  fork :mod_1
    reg x;
    mod_2.x = 1;
    ...
  join
  fork :mod_2
    reg x;
    mod_1.x = 0;
    ...
  join
end
```

Data Structures

Hierarchical path names enable you to define modules that logically group connected variables together and separate them from other variables, which provides a way to define abstract data structures.

The following example describes a data structure template and makes four copies of this template:

```
module memory_control;
    ...
    memory_template block1(), block2(), block3(),
                    block4();
    ...
    initial
        begin
            block1.read_flag = 0;
            block1.write_flag = 1;
            block2.read_flag = 0;
            block2.write_flag = 1;
        end
endmodule // memory_control

module memory_template;
    parameter memsize = 1024;
    reg [7:0] abyte, bbyte, data, memory[1:memsize];
    reg [15:0] aword, bword;
    reg write_flag, read_flag;
endmodule // memory_template
```

Macro Modules and Hierarchical Names

Using a macro module moves all of the internal elements up one level in the hierarchy; this has a significant effect on the hierarchical path names of these elements. The names of the module ports within macro modules disappear due to port collapsing. The internal names of the ports simply cease to exist during simulation. These ports become unified with the nets or registers connected to them in the instantiation.

To maintain the uniqueness of the names of gates and nets internal to the macro module, the name of the scope that contains the instance of the macro module is appended to the internal name. Thus, the hierarchical path names of these elements will be identical to what they would be in a normal module instance, except for the following two differences:

- A backslash (\) precedes the instance name.
- The separator between the instance name and the net name or primitive instance name is a caret (^) character instead of a period (.).

Verilog-XL Reference

Hierarchical Structures

For example, if module `top` creates a module instance called `A`, and the definition of `A` contains a net called `N`, then in a normal module instance, the hierarchical name of `N` would be as follows:

```
top.A.N
```

In a macro module instance, however, the hierarchical name of `N` would be as follows:

```
top.\A^N
```

Note: The backslash is an escape character. Also, you must terminate any escape sequence with a space character. It is expected that the internal names in macro modules will be referenced only in exceptional cases.

Upwards Name Referencing

The name of a module is sufficient to identify the module and its location in the hierarchy. A lower-level module can reference items in a module above it in the hierarchy if the name of the higher-level module is known. The syntax for an upward reference is as follows:

```
<name_of_module>.<name_of_item>
```

There can be no spaces within the reference. The following example demonstrates upward referencing. In this example, there are four modules, `mod_a`, `mod_b`, `mod_c`, and `mod_d`. Each module contains an integer `x`. The highest-level modules in this segment of the model hierarchy are `mod_a` and `mod_d`. There are two copies of module `mod_b.x` because `mod_a` and `mod_d` each instantiate a copy of `mod_b.x`. There are four copies of `mod_c.x` because each of the two copies of `mod_b.x` instantiates `mod_c.x` twice.

```
module mod_a;
integer x;
    mod_b inst_b1();
endmodule

module mod_b;
integer x;
    mod_c inst_c1(), inst_c2();
    initial #10 inst_c1.x = 2; // downward path - references 2
endmodule // copies of x: mod_a.inst_b1.inst_c1.x
// mod_d.inst_b1.inst_c1.x

module mod_c;
integer x;
initial begin
    x = 1; // local name reference -each of the 4 instances
        // of mod_c will modify its own x:
        // mod_a.inst_b1.inst_c1.x mod_a.inst_b1.inst_c2.x
        // mod_d.inst_b1.inst_c1.x mod_d.inst_b1.inst_c2.x

    mod_b.x = 1; // upward path references 2 copies of x:
        // mod_a.inst_b1.x mod_d.inst_b1.x
    end
endmodule
```


Verilog-XL Reference

Hierarchical Structures

```
module mod_d;
integer x;
mod_b inst_b1();
initial begin
    mod_a.x = 1;           // full path name references each copy of x
    mod_a.inst_b1.x = 2;
    mod_a.inst_b1.inst_c1.x = 3;
    mod_a.inst_b1.inst_c2.x = 4;
    mod_d.x = 5;
    mod_d.inst_b1.x = 6;
    mod_d.inst_b1.inst_c1.x = 7;
    mod_d.inst_b1.inst_c2.x = 8;
end
endmodule
```

Automatic Naming

Verilog-XL can automatically generate system names for instances of standard primitives and user-defined primitives that you have not named. You can specify that the system generate names during debugging by using the `+autonaming` option. You can turn off automatic naming for specific parts of your design by using the `'remove_gatenames` compiler directive (see [“Gate and Net Name Removal”](#) on page 140).

System-generated names do not affect simulation results, but they do affect memory usage and compile times. You may want to leave certain primitives unnamed to improve simulator performance.

System generated names have the following format:

```
<primitive_type>${<sequence_number>}
```

The `<primitive_type>` variable can be any of the Verilog-XL-supplied primitives or UDPs. The `<sequence_number>` variable is a decimal number that Verilog-XL assigns to the unnamed instance. Numbers start at 1 for each type within a module description and increase sequentially for each instance.

If the system generates a name that is identical to a user-defined name, then Verilog-XL assigns the next available sequence number that does not conflict.

Note: To avoid conflicts with system-generated names, Cadence recommends that you do not use the dollar sign (\$) within user-defined names.

The following are several ways to view system-generated names:

- Call `$list` to decompile a source description.
- Call `$showvars` to see the names of gates that drive nets.

Verilog-XL Reference

Hierarchical Structures

- Call `$settrace` to see the names of gates that are contained in the source you are tracing.
- Use the `-d` command line option to decompile the entire description, including system-generated names.

The latch description in the following example contains instances of `nand` and `udpfunc` that are unnamed:

```
module latch (q, in1, in2, in3);
output q;
input in1, in2, in3;
    nand #1 (q, nq, w), (nq, q, in3); // <-- unnamed gate instances
    udpfunc (w, in1, in2);           // <-- unnamed UDP instance
endmodule

primitive udpfunc (q,clk,d);
input clk,d;
output q;
reg q;
initial
    q = 1'b1;

table
//clk d      :   q      :   q+
  r  0      :   ?      :   0   ;
  r  1      :   ?      :   1   ;
  f  ?      :   ?      :   -   ;
  ?  *      :   ?      :   -   ;
endtable
endprimitive
```

Decompiling this description with `$list` produces the following output. The second and third “20” lines, and the second “21” line, contain system generated names.

```
15 module latch(q, in1, in2, in3);
16 output
16 q; // = StX
17 input
17 in1, // = StX
17 in2, // = StX
17 in3; // = StX
20 nand #(1)
20 nand$1(q, nq, w),
20 nand$2(nq, q, in3);
21 udpfunc
21 udpfunc$1(w, in1, in2);
23 endmodule
```

Scope Rules

The following four elements define a new scope in Verilog:

- modules
- tasks

Verilog-XL Reference

Hierarchical Structures

- functions
- named blocks

Rule 1

You can use an identifier to declare only one item within a scope. For example, it is illegal to do the following:

- declare two variables that have the same name
- name a task the same as a variable within the same module
- give a gate the same instance name as the name of the net connected to its output

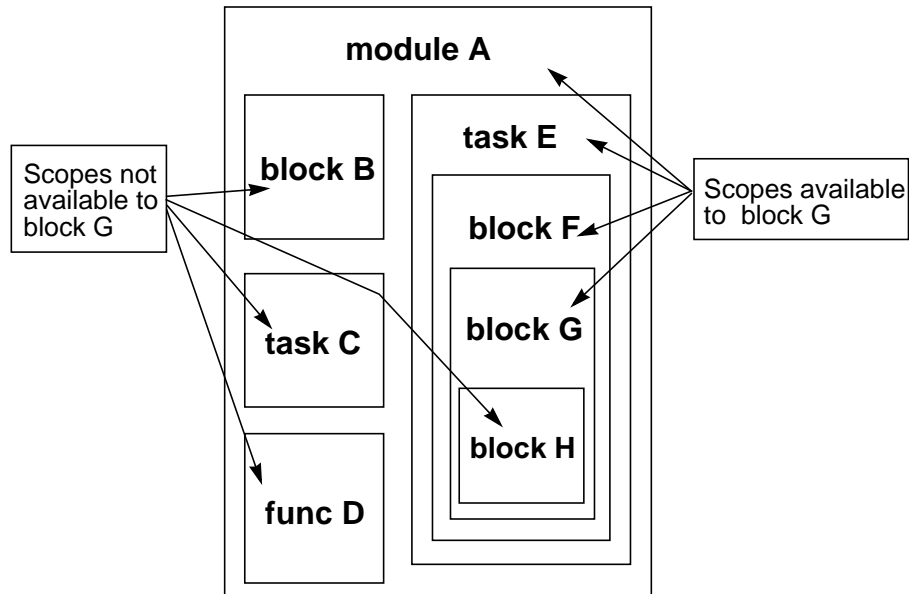
Rule 2

If an identifier is referenced directly (without a hierarchical path) within a task, function, or named block, it must be declared either locally (within the task, function, or named block), or within a module, task or named block that is higher in the same branch of the name tree that contains the task, function, or named block that is referenced. If an identifier is declared locally, then the local item is used; if it is not declared locally, then Verilog-XL searches upward until it finds an item by that name or until it finds a module boundary. Searching crosses named-block, task, and function boundaries, but *not* module boundaries. This fact means that tasks and functions can use and modify the variables within the containing module by name, without going through their ports. If the identifier is not found in the upward search path, the simulator generates an error at compile time.

In the following figure, each rectangle represents a local scope. The scope available to upward searching extends outward to all containing rectangles—with the boundary of the

Verilog-XL Reference Hierarchical Structures

module A as the outer limit. Thus, block G can directly reference identifiers in F, E, and A; it cannot directly reference identifiers in H, B, C, and D.



Because of the upward searching, path names that are not strictly downward can be used and will work. However, these should be avoided as they are confusing, and the compiler generates a warning when it encounters one. This warning can be disabled with the `-w` command-line option.

The following example shows an incompletely defined downward reference that compiles correctly, but generates a compiler warning:

```
task t;
reg r;
begin :b
    // redundant assignments to reg r
    t.b.r = 0; // this is poorly defined but can find r by an upward search
    t.r = 0; // this is a fully defined downward reference
end
endtask
```

Using Specify Blocks and Path Delays

This chapter describes the following:

- [Understanding Specify Blocks](#) on page 237
- [Understanding Path Delays](#) on page 239
- [Describing Module Paths](#) on page 247
- [Using State-Dependent Path Delays \(SDPDs\)](#) on page 269
- [Working with Multiple Path Delays](#) on page 275
- [Enhancing Path Delay Accuracy](#) on page 279

Understanding Specify Blocks

In addition to gate-level or other distributed delays specified inside a module, you can assign delays to paths across a module. A *specify block* adds timing specifications to paths across a module and performs the following modeling tasks:

- Describes various paths across the module
- Assigns delays to those paths
- Performs timing checks to ensure that events occurring at the module inputs satisfy the timing constraints of the device described in the module
- Defines pulse filtering limits

Specify block syntax is as follows:

```
<specify_block>  
  ::= specify  
     <specify_item>*  
  endspecify
```

Verilog-XL Reference

Using Specify Blocks and Path Delays

```
<specify_item>
  ::= <specparam_declaration>
  ||= <path_declaration>
  ||= <level_sensitive_path_declaration>
  ||= <edge_sensitive_path_declaration>
  ||= <system_timing_check>
```

The following example illustrates a specify block:

```
specify
  specparam tRise_clk_q=150, tFall_clk_q=200;
  specparam tSetup=70; //two specparam_declarations
  (clk=>q)=(tRise_clk_q, tFall_clk_q); //path_assignment
  $setup(d, posedge clk, tSetup); //system_timing_check
endspecify
```

Specparam Declarations

The keyword `specparam` declares parameters within specify blocks—called specify parameters or *specparams*, to distinguish them from module parameters. Unlike specify parameters, module parameters are declared outside the specify block with the keyword `parameter`. You cannot use module parameters in specify blocks.

The following demonstrates the syntax for declaring specify parameters:

```
<specparam_declaration>
  ::= specparam <list_of_param_assignments> ;

<list_of_param_assignments>
  ::= <param_assignment><,<param_assignment>> *

<param_assignment>
  ::= <<identifier> = <constant_expression>>
```

The following example illustrates `specparam` declarations:

```
specify
  specparam tRise_clk_q=150, tFall_clk_q=200;
  specparam tRise_control=40, tFall_control=50;
endspecify
```

Verilog-XL Reference

Using Specify Blocks and Path Delays

It is important not to confuse a `specparam` statement with a module `parameter` statement; they are *not* interchangeable. The following table summarizes the differences between the two types of parameter declarations:

SPECPARAM (Specify parameter)	PARAMETER (Module parameter)
■ must be declared <i>inside</i> specify blocks	■ must be declared <i>outside</i> specify blocks
■ cannot use <code>defparam</code> to override values	■ use <code>defparam</code> to override values
■ save memory because they are not replicated with each module	■ use memory because they are replicated with each module instance

Verilog-XL supports specify block constructs with the XL algorithm expressly disabled or enabled (the default). Module path destination signals must always qualify as accelerated nets by meeting the rules outlined in [“Describing Module Paths”](#) on page 247 and in [“Differences of parallel and full connections between equal-sized vectors”](#) on page 251.

Specify blocks may not appear in macro modules in sources that are to be read by Verilog-XL or Verifault-XL. Macro modules that contain specify blocks will not be expanded. See [“Macro Modules”](#) on page 216 for more information about macro modules.

Switch-XL simulation and module path delays are incompatible. The Switch-RC algorithm turns all switches subject to it into non-accelerated bidirectionals (`inouts`) which are incompatible with module path delays.

Understanding Path Delays

In the Verilog hardware description language (HDL), a *module path* is the connection between a source signal and a destination signal that is defined inside a specify block.

A module source signal can be either an `input` (unidirectional path) or an `inout` (bidirectional path). A module destination signal can be either an `output` (unidirectional path) or an `inout`.

Module path delay assignments can apply in all conditions, or only under specified conditions.

The Verilog HDL can describe two types of delays:

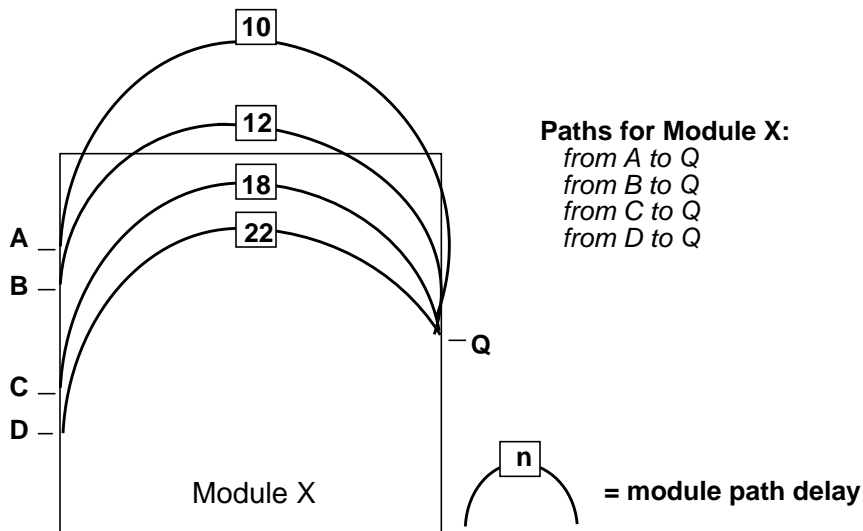
Verilog-XL Reference

Using Specify Blocks and Path Delays

- A *module path delay*, which occurs for the whole module, specifies the time it takes an event at a source (`input` or `inout`) to transmit to a destination (`output` or `inout`). See “[Describing Module Paths](#)” on page 247 for more information about module path delays.
- A *distributed delay*, which occurs on primitive instances within the module, specifies the time it takes an event to transmit through gates and nets. See “[Simulating Distributed Delays as Inertial and Transport Delays](#)” on page 244 for information about simulating distributed delays. See “[Working with Distributed Delays and SDPDs](#)” on page 274 for information about working with distributed delays and SDPDs.

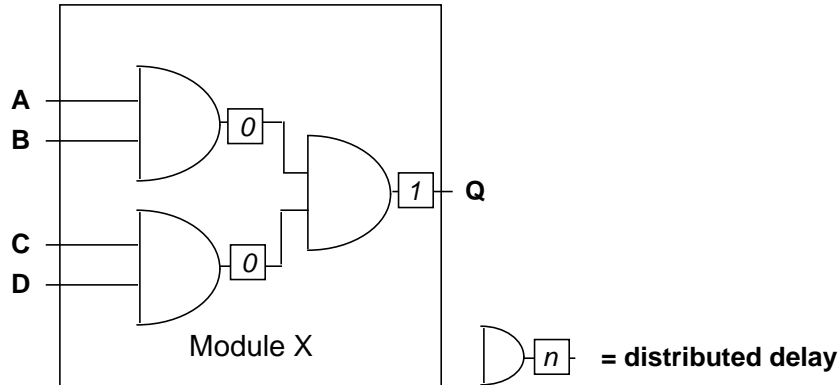
The following figure illustrates module path delays. Note that more than one source (**A**, **B**, **C**, and **D**) may have a module path to the same destination (**Q**).

Module path delays



The following figure shows distributed delays:

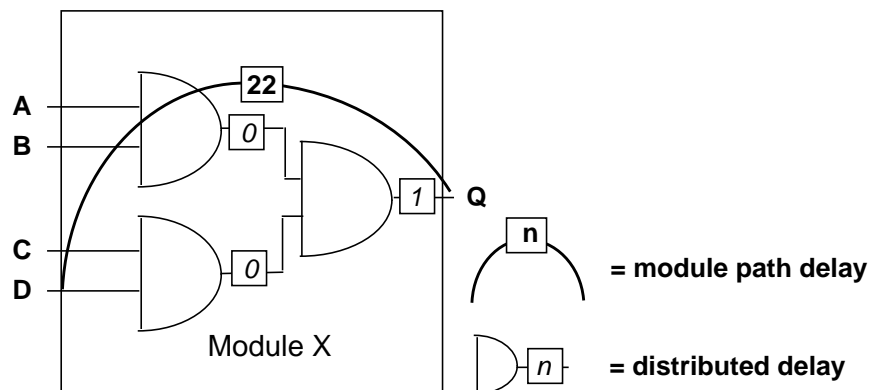
Distributed delays



The figure and the figure contrast module path delays and distributed delays on separate modules to emphasize that you rarely mix module path delays and distributed delays. However, there are situations when you do need both types of delays—for example, to set up a feedback delay to prevent zero delay oscillation.

When one module requires both module path delays and distributed delays, the larger delay prevails. If the distributed delays exceed the module path delay, the distributed delays are used.

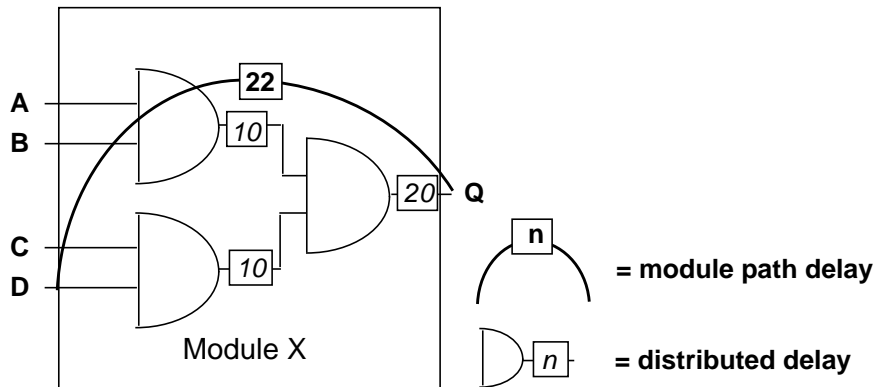
The following two figures illustrate mixed delay-type modules. In the following figure, the delay on the module path from input D to output Q is 22, while the sum of the distributed delays is 1 (0+1=1). Therefore, it takes 22 time units for an event on D to cause an event on Q.



Verilog-XL Reference

Using Specify Blocks and Path Delays

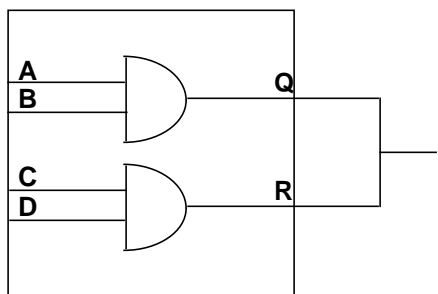
In the next figure, the delay on the module path from D to Q is 22, but the distributed delays along that module path now add up to 30 (10+20=30). Therefore, it takes 30 time units for an event on D to cause an event on Q.



Driving Wired Logic Outputs

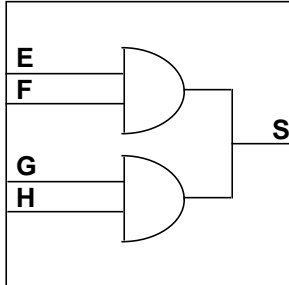
Module path output nets cannot have more than one driver within the module. Therefore, wired logic is not allowed at module path outputs. The following two figures show illegal module paths that illustrate violations of this rule. In the first figure, any module path to Q or R is *illegal*.

Illegal module paths: Two module path outputs with multiple output drivers



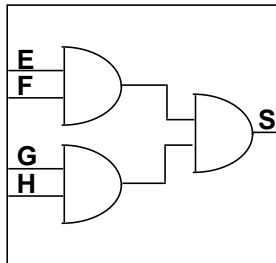
In the following figure, any module path to S is *illegal*.

Illegal module paths: One module path output with multiple output drivers



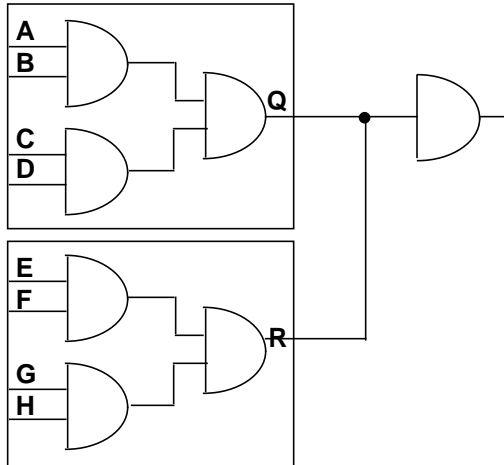
Assuming signal *S* in this figure is a wired AND, you can circumvent this limitation by replacing wired logic with gated logic to create a single driver to the output. The following figure shows how adding a third AND gate solves the problem for the module in previous figure.

Legal module paths: One output driver



Note: Although multiple output drivers are prohibited *inside* the module, they are allowed *outside* the module, as in the next figure (where all module paths to *R* and all paths to *Q* are *legal*).

Legal module paths: Multiple output drivers outside the module



Simulating Distributed Delays as Inertial and Transport Delays

Verilog-XL simulates distributed delays as inertial delays, which means that the delaying element does not pass a pulse of shorter duration than the element's delay.

An element with transport delay functionality may pass pulses of shorter duration than the element's delay. However:

- In Verilog-XL version 2.0 and earlier versions, module path delays might have limited transport delay functionality. [“Pulse Handling in Verilog-XL 2.0 and Earlier Versions”](#) of the *Verilog-XL User Guide* discusses this limited implementation of transport delay in path delays.
- In Verilog-XL version 2.1 and later versions, module path delays have unlimited transport delay functionality when you invoke with the `+transport_path_delays` command-line plus option. See [“Pulse Filtering and Cancelled Schedule Dilemmas”](#) on page 266 for the impact of the `+transport_path_delays` option on path pulse control.

Simulating Path Delays

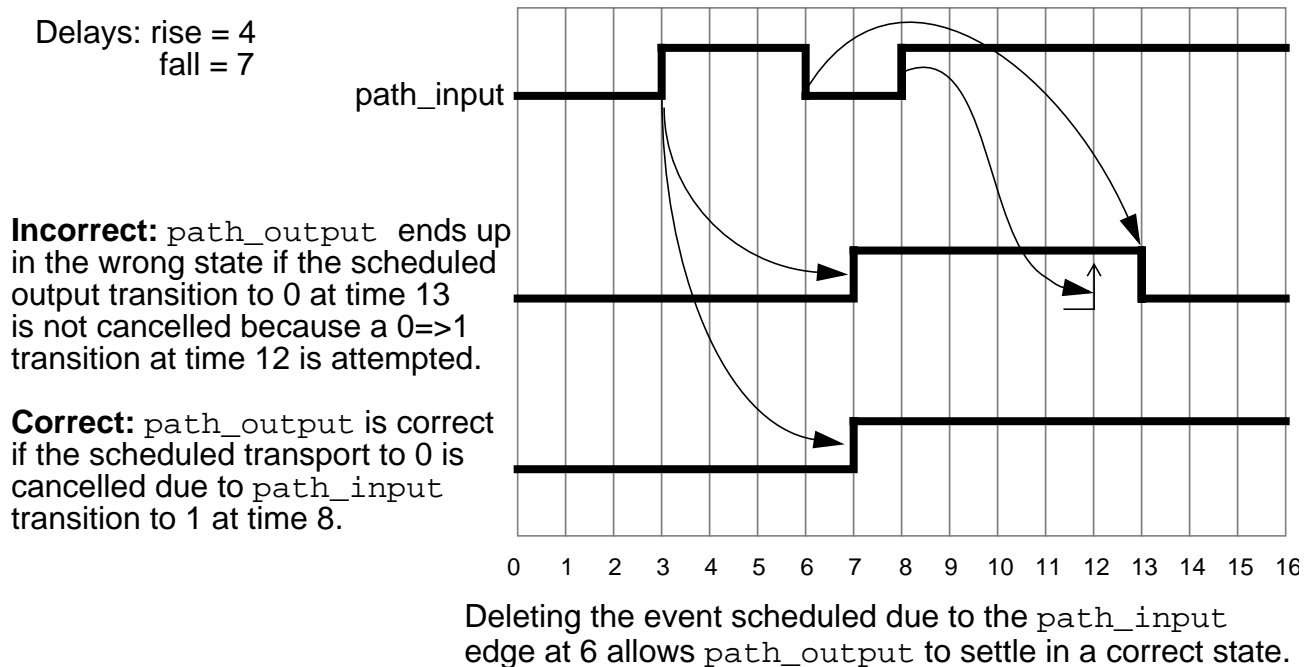
The fact that path delays implement transport delay functionality does not mean that they pass every transition. There are two cases in which transitions can be lost or filtered. One case occurs when pulse control applies to the simulation. The other case is an event cancellation due to an event being scheduled earlier than events already scheduled.

Verilog-XL Reference

Using Specify Blocks and Path Delays

The following figure shows why an event cancellation policy that can lose transitions is necessary. The module path delay has different delays specified for two types of output transitions: a delay of 4 for rising transitions, and a delay of 7 for falling transitions. The waveform named `path_input` represents the signal at the path input, and `path_output` is the signal propagating from the end of the module path. The two versions of the `path_output` signal show the signal propagating from the module with and without event cancellation.

Transport delay cannot pass all transitions



As this figure shows, passing all transitions in transport delay makes an output transmit an incorrect signal, so Verilog-XL deletes scheduled events that lead to such a result.

The following figure illustrates two plus options related to using transport delays on a pass gate model:

■ `+x_transport_pessimism`

This plus option causes an X state to appear on the output when timing dilemmas are caused by event cancellations that occur when using transport delays, or when using the `accu_path` delay selection algorithm (see [“Enhancing Path Delay Accuracy”](#) on page 279).

■ `+alt_path_delays`

Verilog-XL Reference

Using Specify Blocks and Path Delays

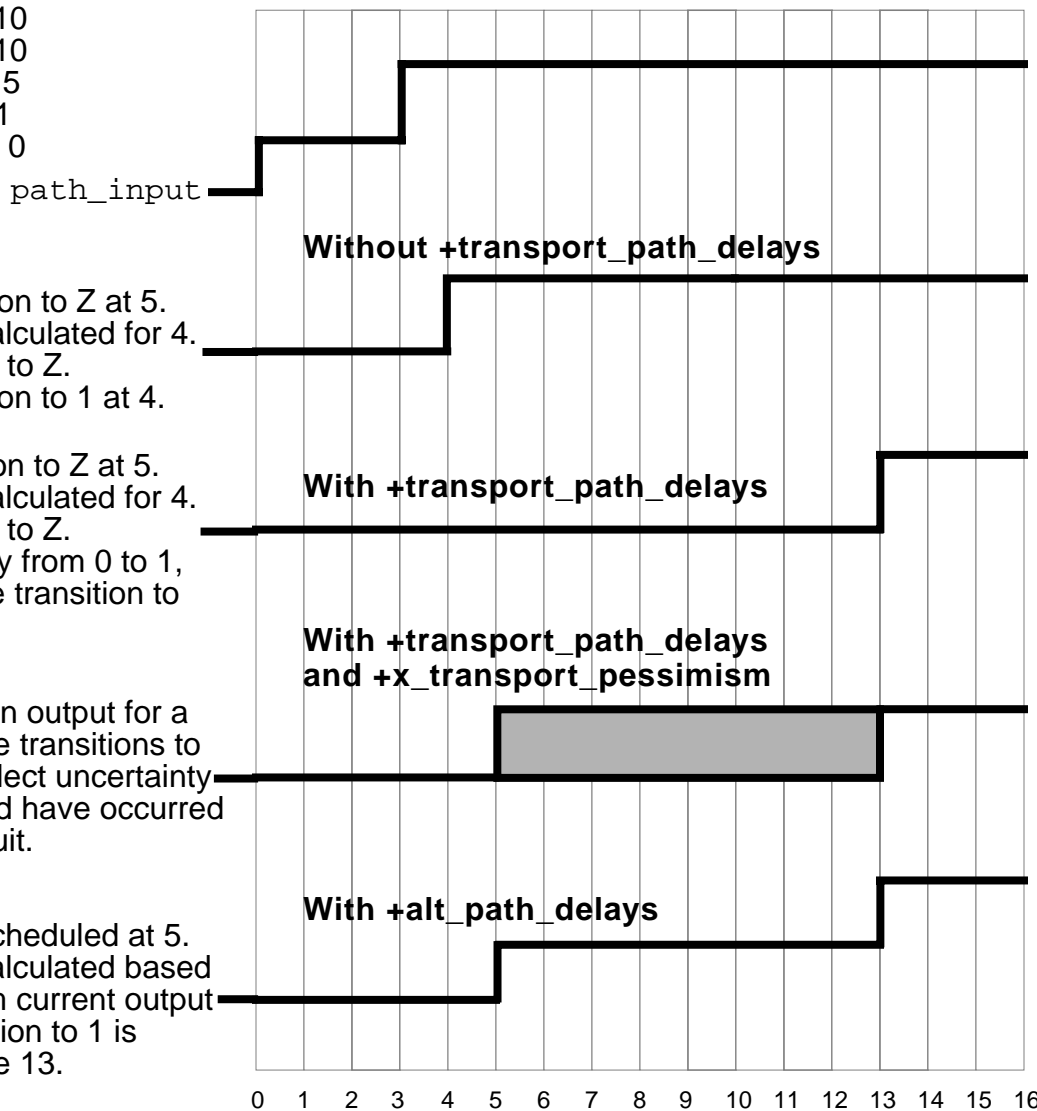
This plus option changes the way in which certain path delays are calculated. If a new transition on an output is scheduled while a scheduled transition is pending, then the new schedule time is based on the transition from the current output value, rather than on the transition from the most future schedule value.

Note: See “[Specifying Global Pulse Control on Module Paths](#)” on page 257 for information about pulse reject limits.

Effects of plus options on a pass gate mode

Delays: 0 -> 1 = 10
 1 -> 0 = 10
 0 -> Z = 5
 Z -> 1 = 1

Pulse reject limit: 0



Schedule transition to Z at 5.
 Transition to 1 calculated for 4.
 Cancel transition to Z.
 Schedule transition to 1 at 4.

Schedule transition to Z at 5.
 Transition to 1 calculated for 4.
 Cancel transition to Z.
 Recalculate delay from 0 to 1,
 yielding schedule transition to
 1 at time 13.

Display X state on output for a
 period of possible transitions to
 and from Z to reflect uncertainty
 that Z state would have occurred
 in the actual circuit.

Transition to Z scheduled at 5.
 Transition to 1 calculated based
 on transition from current output
 value (0). Transition to 1 is
 scheduled at time 13.

Describing Module Paths

To specify the delays that occur at the module outputs where paths terminate, you assign delay values to the module path descriptions. Delay values can be constant expressions that contain literals or specparams. The syntax of the module path declaration is as follows:

```
<path_declaration>
    ::= (<path_description>) = (<path_delay_value>);

<path_description>
    ::= ( <specify_input_terminal_descriptor> =>
        <specify_output_terminal_descriptor> )
        ||= ( <list_of_path_inputs> * > <list_of_path_outputs> )
            <path_delay_value>
    ::= <path_delay_expression>
    ||= ( <path_delay_expression>, <path_delay_expression> )
    ||= ( <path_delay_expression>, <path_delay_expression>,
        <path_delay_expression> )
    ||= ( <path_delay_expression>, <path_delay_expression>,
        <path_delay_expression>, <path_delay_expression>,
        <path_delay_expression> )
    ::= <constant_expression>
```

Note: Each delay defines either a single delay value or a triplet of minimum, typical, and maximum (*min:typ:max*) delay values. For more information about transition delays on module paths, see [“Specifying Transition Delays on Module Paths”](#) on page 251.

The following example shows module path delay assignments. Each `specparam` keyword specifies one set of delays for the rising transitions and another set of delays for the falling transitions. Each delay triplet specifies the minimum, typical, and maximum delay values.

```
specify
    // the following are specify parameters
    specparam tRise_clk_q=45:150:270, tFall_clk_q=60:200:350;
    specparam tRise_control=35:40:45, tFall_control=40:50:65;

    // the following are module path assignments
    (clk=>q)=(tRise_clk_q,tFall_clk_q);
    (clr,pre*>q)=(tRise_control,tFall_control);
endspecify
```

When you compile the code in the previous example, you assign the minimum delay to the `tRise_clk_q` and `tFall_clk_q` identifiers by specifying the `+mindelays` plus option on the command line; you specify `+maxdelays` for maximum delays and `+typdelays` (the default) for typical delays. For example, if you compiled the code with the `+maxdelays` option, the value for `tRise_clk_q` would be 270, and the value for `tFall_clk_q` would be 350; `tRise_control` would be 45, and `tFall_control` would be 65.

Module paths can connect any combination of vectors and scalars. However, there are two restrictions:

Verilog-XL Reference

Using Specify Blocks and Path Delays

- The module path source must be a net that is declared as a module `input` or `inout` net, either of which must be scalar or vector.
- The module path destination must be a net that is declared as a module `output` or `inout` net, either of which must be scalar or vector and is driven only by a gate-level primitive that is not a bidirectional transfer gate. A module path destination must qualify for acceleration by the XL algorithm, even if the XL algorithm is disabled.

Signals that do not qualify as accelerated nets are as follows:

- forced nets
- nets with non-zero delays
- vector nets or scalar nets that receive continuous assignments
- signals driven by gates with an expression involving any operator on an input
- signals driven by gates that have dynamic delay expressions
- signals driven by *buf* and *not* gates with more than one output
- signals driven by the following bidirectional primitives:
 - `tran`
 - `tranif1`
 - `tranif0`
 - `rtran`
 - `rtranif1`
 - `rtranif0`

During compilation, Verilog-XL flags as errors any module path destination signals that do not qualify as accelerated nets.

Establishing Parallel or Full Connections

This section illustrates two ways to describe module paths, using the `=>` and `*>` operators. The following example shows the module path syntax for parallel and full connections.

```
<path_description>
 ::= ( <specify_input_terminal_descriptor> =>
      <specify_output_terminal_descriptor> )
 || = ( <list_of_path_inputs> *> <list_of_path_outputs> )
      <specify_input_terminal_descriptor>
```


Verilog-XL Reference

Using Specify Blocks and Path Delays

```
 ::= <input_identifier>
 || = <input_identifier> [ <constant_expression> ]
 || = <input_identifier> [ <constant_expression> :
   <constant_expression> ]

<specify_output_terminal_descriptor>
 ::= <output_identifier>
 || = <output_identifier> [ <constant_expression> ]
 || = <output_identifier> [ <constant_expression> :
   <constant_expression> ]

<input_identifier>
 ::= the <IDENTIFIER> of a module input or inout terminal

<output_identifier>
 ::= the <IDENTIFIER> of a module input or inout terminal
```

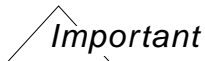
In the following example, the module path from *s* to *q* uses **>* because it connects a scalar source—the 1-bit select line—to a vector destination—the 8-bit output bus. The module paths from both input lines *In1* and *In2* to *q* use *=>* because they set up parallel connections between two 8-bit buses.

```
module MUX8 (In1,In2,s,q);
input [0:7] In1,In2;
input s;
output [0:7] q;
...
specify
  specparam In_to_q=40, s_to_q=45;
  (In1 => q) = In_to_q;      //parallel connection
  (In2 => q) = In_to_q;      //parallel connection
  (s *> q) = s_to_q;        //full connection
endspecify
endmodule
```

Establishing a Parallel Connection

The *=>* operator establishes a parallel connection between source and destination. In a parallel connection, each bit in the source connects to a corresponding bit in the destination. You can create parallel module paths only between a source and destination that contain the same number of bits.

Note: You must use the *=>* operator for bit-to-bit connections to describe a module path between two vectors of the same size.

 **Important**

You will not receive a compiler error if you use the operator `=>` to establish a full connection between one scalar and one vector, or between one scalar and multiple sources or destinations. This practice is not recommended and may cause a compiler error in releases beyond Verilog-XL 2.3.

Establishing a Full Connection

The `*>` operator establishes a full connection between source and destination. You can define multiple module paths in a single statement by using the `*>` operator to connect a list of sources, as shown in the following example:

```
(a, b, c *> q1, q2) = 10;
```

This statement is equivalent to the following six individual module path assignments:

```
(a *> q1) = 10;  
(b *> q1) = 10;  
(c *> q1) = 10;  
(a *> q2) = 10;  
(b *> q2) = 10;  
(c *> q2) = 10;
```

When describing multiple module paths in one statement, the lists of sources and destinations can contain a mix of scalars and vectors of any size. However, all sources must be net inputs or inouts; and all destinations must be net outputs or inouts.

In a full connection, each bit in the source connects to every bit in the destination. The module path source does not need to have the same number of bits as the module path destination.

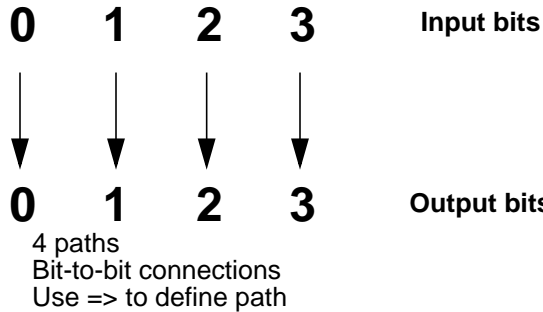
The full connection will handle most types of module paths, since it does not restrict the size or number of source signals and destination signals. However, you must use the full connection operator (`*>`) to set up full connections in the following situations:

- Describing a module path between one vector and one scalar
- Describing a module path between vectors of different sizes
- Describing a module path with multiple sources or multiple destinations in a single statement

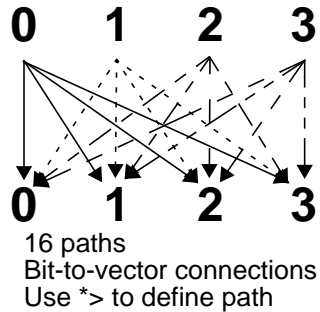
The following figure illustrates how a parallel connection differs from a full connection between two 4-bit vectors:

Differences of parallel and full connections between equal-sized vectors

Parallel module path



Full module path



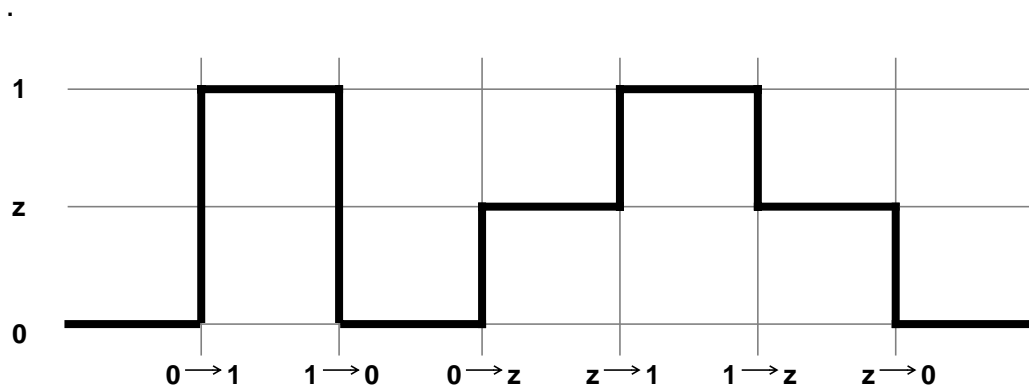
Specifying Transition Delays on Module Paths

The following conditions apply to delays:

- You can assign delay values independently for each of the six output transitions (0→1, 0→Z, 1→0, 1→Z, Z→0, and Z→1).
- You specify delays as a list of one, two, three, or six path delay expressions separated by commas.
- You can specify a single delay value for all three delays, or a colon-separated list of values for minimum, typical, and maximum delays.

Note: You can specify only one delay or a delay triplet. The format `delay1:delay2` is illegal in a module path delay assignment.

The left-to-right order in which you specify delays for all six transitions in a `specify` statement is based on the following figure



Verilog-XL Reference

Using Specify Blocks and Path Delays

The following example shows the syntax and code for assigning one delay value for all transitions:

```
// Syntax: (module_path)=delay;
// one delay value is assigned to all transitions:
//      0->1, 1->0, 0->Z, Z->1, 1->Z, z->0
//
// Examples:
(C=>Q)=20;           // assigns a delay of 20 for all
                    // transitions from C to Q
(C=>Q)=10:14:20;    // assigns min:typ:max delays to all
                    // transitions from C to Q
```

The following example shows the syntax and code for assigning different delays for rising and falling transitions:

```
// Syntax: (module_path)=(rise_delay,fall_delay);
//      transitions:  0->1      1->0
//                  //      0->z      1->z
//                  //      z->1      z->0
// Examples:
specparam tPLH=12,tPHL=25;
(C=>Q)=(tPLH,tPHL);
specparam tPLH=12:16:22,tPHL=16:22:25;
(C=>Q)=(tPLH,tPHL);
```

Any transition delay associated with a module path can be triggered at run time by the appropriate state change at the module path destination net. For instance, the previous example assigns one set of *minimum:typical:maximum* delays for the rising transitions and another set of *minimum:typical:maximum* delays for the falling transitions.

The following example shows the syntax and code for assigning different delays for rising, falling, and z transitions:

```
// Syntax: (module_path)=(rise_delay, fall_delay, z_delay);
//      0->1      1->0      0->z
//      z->1      z->0      1->z
// Examples:
specparam tPLH = 12, tPHL = 22, tPz = 34;
(C => Q) = (tPLH, tPHL, tPz);
specparam tPLH=12:14:30, tPHL=16:22:40, tPz=22:30:34;
(C => Q) = (tPLH, tPHL, tPz);
```

The following example shows the syntax and code for assigning six different transition delays:

```
// Syntax: (module_path)=(delay,delay,delay,delay,delay,delay);
//      0->1  1->0  0->z  z->1  1->z  z->0
// Examples:
specparam t0l=12, t10=16, t0z=13, tzl=10, tlz=14, tz0=34;
(C => Q) = ( t0l, t10, t0z, tzl, tlz, tz0);
specparam t0l=12:14:24, t10=16:18:20, t0z=13:16:30;
specparam tzl=10:12:16, tlz=14:23:36, tz0=15:19:34;
(C => Q) = ( t0l, t10, t0z, tzl, tlz, tz0) ;
```

Calculating Delay Values for X Transitions

Calculating delay values for x transitions is based on the following two pessimistic rules:

- Transitions from a known state (S) to x ($S \rightarrow X$) occur as quickly as possible—that is, they receive the shortest possible delay.
- Transitions from x to a known state (S) ($X \rightarrow S$) take as long as possible—that is, they receive the longest possible delay.

The following table presents the general algorithm for calculating delay values for x transitions, along with specific examples.

X TRANSITION	DELAY VALUE
General Algorithm	
$S \rightarrow X$	Minimum of (S → S)
$X \rightarrow S$	Maximum of (S → S)
Specific Transitions	
$0 \rightarrow X$	Minimum of (0 → Z delay, 0 → 1 delay)
$1 \rightarrow X$	Minimum of (1 → Z delay, 1 → 0 delay)
$Z \rightarrow X$	Minimum of (Z → 1 delay, Z → 0 delay)
$X \rightarrow 0$	Maximum of (Z → 0 delay, 1 → 0 delay)
$X \rightarrow 1$	Maximum of (Z → 1 delay, 0 → 1 delay)
$X \rightarrow Z$	Maximum of (1 → Z delay, 0 → Z delay)
Usage: (C=>Q) = (5, 12, 17, 10, 6, 22)	
$0 \rightarrow X$	Minimum of (17, 5) = 5
$1 \rightarrow X$	Minimum of (6, 12) = 6
$Z \rightarrow X$	Minimum of (10, 22) = 10
$X \rightarrow 0$	Maximum of (22, 12) = 22
$X \rightarrow 1$	Maximum of (10, 5) = 10
$X \rightarrow Z$	Maximum of (6, 17) = 17

Specifying Module Path Polarity

The polarity of a module path determines how a signal transition (at its source) passes to its destination when there are no logic simulation events. A module path can exhibit unknown, positive, or negative polarity. The polarities are described as follows:

- Unknown polarity
 - A rise at the source causes either a rise or a fall at the destination.
 - A fall at the source causes either a rise or a fall at the destination.
- Positive polarity
 - A rise at the source always causes a rise at the destination.
 - A fall at the source always causes a fall at the destination.
- Negative polarity
 - A rise at the source always causes a fall at the destination.
 - A fall at the source always causes a rise at the destination.

By default, module paths have unknown polarity—that is, a transition at the path source transmits to the destination in an unpredictable way.

Note: Polarity has no effect on the scheduling of simulation events; a timing analysis tool can use polarity when performing path tracing. The Veritime timing analyzer uses polarity to calculate module path delays.

Whether a rise or a fall transmits to the destination depends on the states of the module's other inputs and internal logic.

To set up module paths with positive polarity, add the plus sign (+) prefix to the connection operators `*>` and `=>`. For negative polarity, add the minus sign (-) prefix. For unknown polarity, add no prefix. The following example shows each type of path polarity.

```
(In1 +=> q) = In_to_q; // Positive Polarity
(s +=> q) = s_to_q;   // Positive Polarity

(In1 -=> q) = In_to_q; // Negative Polarity
(s -=> q) = s_to_q;   // Negative Polarity

(In1 => q) = In_to_q; // Unknown Polarity
(s => q) = s_to_q;   // Unknown Polarity
```

In addition, you can assign the same polarity to multiple module paths in a single statement, as follows:

Verilog-XL Reference

Using Specify Blocks and Path Delays

```
(a, b, c +*> q1, q2) = 10;    // Positive Polarity
(a, b, c -*> q1, q2) = 10;    // Negative Polarity
```

In the previous example, (providing there are no vectors), the first line assigns positive polarity to six different paths. The second line assigns negative polarity to six different paths.

Verilog-XL treats all module paths as if they contain no polarity operators. It chooses the delay based only on the output transition and without regard to the input transition that initiates the delay. For more information about how to use module path polarity for timing analysis, refer to the *Veritime Reference Manual* and *Veritime User Guide*.

Using Path Delays in Behavioral Descriptions

To use module path delays on behavioral descriptions, a path destination signal must be a net that is driven only by a gate-level primitive, qualifying it as an accelerated net. The primitive **must not** be a bidirectional transfer gate. Whenever an error occurs because a module path destination does not qualify as an accelerated net, you can recover from the error condition by placing a zero delay `buf` gate between the module boundary and the desired signal.

Consider the following module, `adder`. The signals `sum` and `carry` do not qualify as accelerated nets because they are scalar nets that receive continuous assignments.

```
module adder (A, B, sum, carry);
  input  A, B;
  output sum, carry ;

  wire  sum  = A + B ; //continuous assignment
  wire  carry = A & B ; //continuous assignment

  specify
    //module path delays
    (A, B *> sum)  = 10 ;
    (A, B *> carry) = 5 ;
  endspecify
endmodule
```

The following example shows that by adding zero delay `buf` gates between `adder` and the signals `sum` and `carry`, you can create two new path destinations, `sum_sig` and `carry_sig` that do follow acceleration guidelines.

```
module adder (A, B, sum_sig, carry_sig);
  input  A, B;
  output sum_sig, carry_sig ;

  wire  sum = A + B ; //continuous assignment
  wire  carry = A & B ; //continuous assignment
  buf g1 (sum_sig, sum) ; //zero delay buf
  buf g2 (carry_sig, carry) ; //zero delay buf

  specify
    //module path delays
    (A, B *> sum_sig) = 10 ;
    (A, B *> carry_sig) = 5 ;
  endspecify
endmodule
```

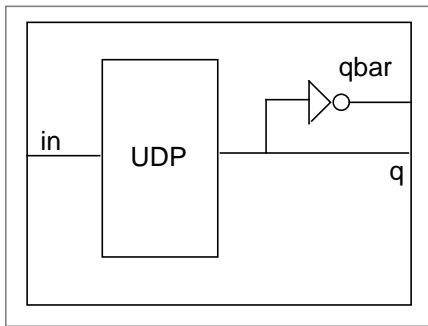
Verilog-XL Reference

Using Specify Blocks and Path Delays

```
endspecify
endmodule
```

Simulating Path Outputs that Drive Other Path Outputs

If one module path output drives another module path output, the delay on the driving path must be less than the delay on the driven path. Otherwise, Verilog-XL schedules an event on the driven path output later than expected—at the time when the driving path output occurs. Consider the following figure which shows module path outputs driving other module path outputs.

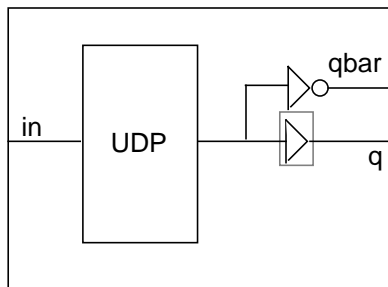


Pin-to-pin delays:

```
( in => q ) = 12 ; ← DRIVING MODULE PATH
( in => qbar ) = 10 ; ← DRIVEN MODULE PATH
```

In the previous figure, the output of module path ($in \Rightarrow q$) drives the output of module path ($in \Rightarrow qbar$). Assuming the last in input occurred at time 0, Verilog-XL would schedule a q output event at time 12 and a $qbar$ output event at time 12—even though the desired result is to schedule the $qbar$ output at time 10.

The solution to problem is to place a buffer on the driving output, as shown in the following figure, which creates an internal net to drive $qbar$ so that any event on $qbar$ caused by an event on in occurs after 10 time units.



Pin-to-pin delays:

```
( in => qbar ) = 10 ; ← Driven module path
( in => q ) = 12 ; ← Driving module path
```


Understanding Strength Changes on Path Inputs

The strength is an implementation function of the internal module. When scheduling module path output events, Verilog-XL does not consider the time of the strength change at the input. Strength changes always propagate through a circuit using the gate and net delays, not the module path delays.

Specifying Global Pulse Control on Module Paths

When you set global pulse control using the delay value and pulse limits, Verilog-XL determines which of the following actions to take on all module path output and interconnect pulses.

- Reject the pulse (the output is unaffected by the pulse).
- Flag the pulse as an error state (e).
- Let the pulse pass through.

Note: Pulse widths are measured at the output and not at the input.

The following equations show how Verilog-XL calculates the level of acceptance from the error and reject values that you supply as percentages of the module path delay:

```
error_limit = (error% / 100) * (module_path_delay)
reject_limit = (reject% / 100) * (module_path_delay)
```

Note: Calculated limits are truncated, not rounded.

After calculating the limits, Verilog-XL acts on the pulse according to the following rules:

```
REJECT if 0 < pulse < (reject_limit)
SET TO E if reject_limit <= pulse <(error_limit)
PASS if pulse >= error_limit
```

You can specify pulse limits for module paths and interconnect delays separately in the same simulation by entering two pairs of plus options on the command line. The `+pulse_e/n` and `+pulse_r/m` plus options set global module path pulse control in Verilog-XL.

For module path delays, use the following plus options on the command line:

```
+pulse_e/n
+pulse_r/m
```

For interconnect delays, use the following plus options on the command line:

```
+pulse_int_e/n
+pulse_int_r/m
+transport_int_delays
```

For example, the following command sets `reject%` to 50 and `error%` to 80:

Verilog-XL Reference

Using Specify Blocks and Path Delays

```
verilog source.v +pulse_r/50 +pulse_e/80
```

This command specifies the following:

- A module path delay of 50 time units has a `reject_limit` of up to 25 time units (50% of a delay of 50).
- The `error_limit` is set at 40 time units (80% of a delay of 50).
- Pulses smaller than 25 time units are rejected.
- At 25 through 39 time units, the module path delays are set to `e` (error state).
- At 40 time units and above, the module path delays pass through.

To generate an error whenever a module path pulse is less than a module path delay, use the following command line:

```
verilog source.v +pulse_r/0 +pulse_e/100
```

The default values for `reject%` and `error%` are 100. However, Verilog-XL modifies the default `error%` under the following conditions:

- If you omit both the `reject%` and `error%`, both specifications are set to 100.
- If you omit one of the pulse limits, the omitted specification is set to 100, but the next rule can reset this value.
- If the `reject%` exceeds the `error%` or if the `reject%` specification is not accompanied by an `error%` specification, Verilog-XL issues a warning and resets the `error%` equal to the `reject%`. For example, the `error%` in the following command line is reset to 100 because the `reject%` has the default value of 100.

```
verilog source.v +pulse_e/80
```

The following command line sets the `error%` equal to the `reject%` limit of 50, because it does not include an `error%` specification.

```
verilog source.v +pulse_r/50
```

If you omit the `+pulse_e/n`, the `+pulse_r/m` and the `+transport_path_delays` plus options, module path delays work the same way as gate delays because Verilog-XL rejects all module path output pulses that are shorter than the module path delay. Note, however, that gate delays are not affected by `reject%` and `error%`.

Signals with the error state (`e`) value generate warnings as follows:

```
Warning! Time = 180: Pulse flagged as an error at node
top.nol.output1, value = StE
Path: top.nol.input3 ---> top.nol.output1
[Verilog-PLSERR]
```

Verilog-XL Reference

Using Specify Blocks and Path Delays

The description `Path: top.no1.input3 ---> top.no1.output1` identifies the path that generated the pulse error.

These warning messages can be suppressed by specifying the `+no_pulse_msg` plus option. Verilog-XL then treats the module output nets transmitting signals with the `e` value as if their signals had the `x` value.

Specifying Local Pulse Control for Module Paths

You can provide individual control over path pulse limits, effectively overriding global pulse control, by declaring specialized specparams that use the prefix `PATHPULSE$`. The `PATHPULSE$` specparam narrows the scope of module path pulse control to a specific module or to particular paths within modules. The command line must include the `+pathpulse` option for the `PATHPULSE$` specparams to be effective.

Standard Delay Format (SDF) annotation provides new values for pulse limits of both specify path delays and interconnect delays. This annotation method operates independently of the `PATHPULSE$` specparam construct, and the `+pathpulse` option is not needed when pulse control values are provided by SDF annotation.

Note: The `+pathpulse` command-line option adds significant compilation overhead.

`PATHPULSE$` syntax is as follows:

```
<pulse_control_specparam>
 ::=PATHPULSE$(<reject>,<error>);
 ||=PATHPULSE$<module_path_source>$
   <module_path_destination>=(<reject>,<error>);
```

If you supply the `source` and `destination` variables, Verilog-XL applies the indicated pulse handling to the specific module path that you define between `path` and `destination`. Otherwise, Verilog-XL applies the specified pulse handling characteristics to all paths declared within the module. The sources and destinations in `source` and `destination` can be scalar nets or vector nets, but cannot be bit-selects or part-selects. The pulse handling characteristics you specify for paths beginning in a vector or list and ending in a vector or list automatically apply to all module paths connecting the two vectors or lists.

The `<reject>` and `<error>` limit values assigned to the `PATHPULSE$` specparam define the pulse handling windows in time units—not percentages as in the global command line. If you supply a value only for the `<reject>` variable, `<reject>` and `<error>` are set to the same value. The following example shows how to use `PATHPULSE$`:

```
specify
  (clk => q) = 12;
  (data => q) = 10;
  (clr, pre *> q) = 4;
specparam
```

Verilog-XL Reference

Using Specify Blocks and Path Delays

```

PATHPULSE$ = 3,
PATHPULSE$clk$q = ( 2, 9 ),
PATHPULSE$clr$q = 1;

```

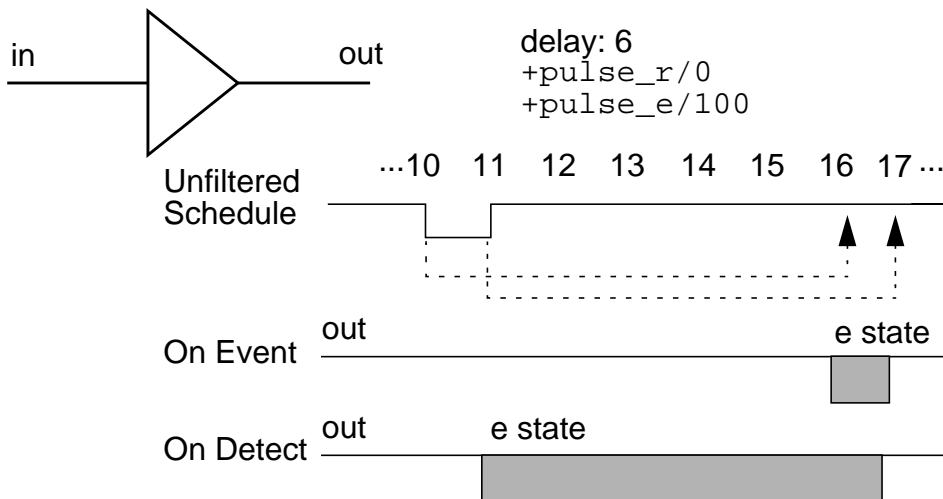
endspecify

The path (`clk=>q`) acquires a *reject* value of 2 and an *error* value of 9, due to the second `PATHPULSE$` declaration. The path (`data=>q`) acquires *reject* and *error* values of 3 in compliance with the first `PATHPULSE$` declaration. The paths (`clr*>q`) and (`pre*>q`) receive *reject* and *error* values of 1, following the last `PATHPULSE$` declaration.

The previous example specifies a pulse control limit for the first input signal `clr` in module path (`clr,pre => q`), but does not explicitly specify a pulse control limit for `pre`, the second signal in the path. The reason is because all signals in module paths with multiple inputs or outputs must have the same delays and pulse handling. Therefore, the pulse handling characteristics for one module path apply to all paths defined in the same declaration.

Pulse Filtering for Module Path Delays

Verilog-XL provides two kinds of pulse filtering called *on event* and *on detect*. The following figure shows the *e* states that each method of pulse filtering produces.



With a delay of 6, Verilog-XL schedules the delay for time 16 and 17 based on the pulse edges. If you use the *on event* pulse filter, the *e* state region exists from time 16 to time 17. If you use the *on detect* pulse filter, the *e* state region is extended from the ending edge of the pulse at time 11 for the length of the entire delay factor (6) to time 17.

Either style of pulse filtering can work on a module or path output. You can specify the style in `specify blocks` using the `$pulsestyle_oneevent` and/or `$pulsestyle_ondetect`

Verilog-XL Reference

Using Specify Blocks and Path Delays

tasks. You can also specify the style from the command line using the `+pulse_e_style_onevent` and/or `+pulse_e_style_ondetect` plus options. When you use a pulse filtering plus option, Verilog-XL globally uses the specified style on all paths, overriding any specify block tasks in the description. The syntax for the pulse-filtering styles follows:

```
$pulsestyle_onevent[(<path_output>+)];  
$pulsestyle_ondetect[(<path_output>+)];
```

Note: All paths that terminate at a particular output must use the same style of pulse filtering.

Examples of determining pulse-filtering styles follow:

```
specify  
  (in => outbar) = (2, 3); // on event (by default)  
  $pulsestyle_ondetect; // affects out  
  (in => out) = (5, 6); // on detect style  
  (clk => out) = (4); // on detect style  
  $pulsestyle_onevent; // affects synch and output  
  (in => sync) = (20, 30); // on event style  
  (in => output) = (7, 9); // on event style  
endspecify  
  
specify  
  $pulsestyle_ondetect(out); // affects out only  
  (in=>out)=(15,25); // on detect style  
  (clk=>q)=5; // on event style (by default)  
endspecify
```

The pulse-filtering specifications in the following example produce an error because they are *incorrect* pulse style specifications.

```
specify  
  $pulsestyle_ondetect(out); // affects out  
  $pulsestyle_onevent(out); // error by changing  
  // styles on out  
endspecify  
  
specify  
  $pulsestyle_ondetect; // sets on detect style  
  (in=>out)=(15,25); // on detect style for out  
  $pulsestyle_onevent(out); // error by changing  
  // styles on out  
endspecify
```

The following warning message is displayed when an `e` state appears on a path output due to the cancellation of a schedule:

```
Warning! Time = <simulation time>: Schedule cancel flagged as an error at node  
  <output>, value = StE  
  Path: <path> [Verilog-CANERR]
```

These warning messages do not appear by default. To display warning messages, you need to use the `+show_cancelled_e` plus option. This option displays the path that caused the schedule cancellation to occur. To disable these warning messages, use the `+no_show_cancelled_e` plus option. Using this option, however, suppresses the `e` state

Verilog-XL Reference

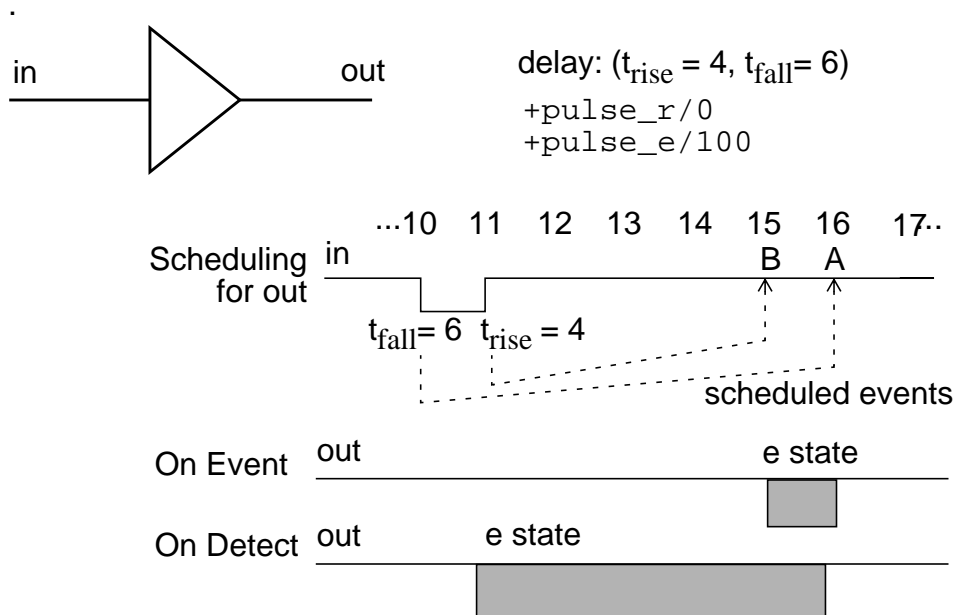
Using Specify Blocks and Path Delays

too. To disable warning messages without suppressing the `e` state, use the `+no_cancelled_e_msg` plus option.

Note: The `+no_show_cancelled_e` and `+no_cancelled_e_msg` plus options work only if warning messages are enabled using the `+show_cancelled_e` plus option.

Pulse Filtering and Cancelled Schedules

A schedule is cancelled when a delay schedules a transition to occur before a previously scheduled transition. The following figure shows the `e` state that occurs with a cancelled schedule for each method of pulse filtering.



The events in this figure occur as follows:

1. At time 10, a 1->0 transition on the input causes Verilog-XL to schedule event A at time 16 (based on adding the fall delay of 6 to time 10).
2. At time 11, a 0->1 transition on the input causes Verilog-XL to schedule event B at time 15 (based on adding the rise delay value of 4 to time 11).
3. Because event B is scheduled to occur before event A, the schedule for A is cancelled and produces an `e` state region that is based on the pulse filtering method you use.
 - ❑ The on event pulse filtering produces an `e` state region on `out` that begins at the time of the second scheduled event B and ends at the time of the cancelled scheduled event A, which is replaced with a scheduled event to the new logic state (in this case, 1).

Verilog-XL Reference

Using Specify Blocks and Path Delays

- The on detect pulse filtering produces an e state region on `out` that begins at the time of the trailing edge of the input and ends at the time of the cancelled scheduled event A, which is replaced with a scheduled event to the new logic state (in this case, 1).

The figure “[Module path delay pulse filtering](#)” on page 264 shows a more complex example of cancelled schedules using two inputs to `out`. The various combinations of plus options affect the e state region differently for module path delays.

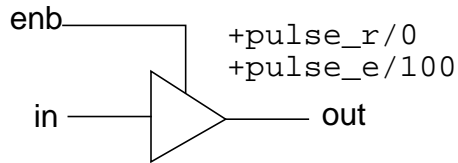
The following figure shows wave forms produced using various combinations of plus options.

Note: The tasks that correspond to the plus options also produce the same wave forms.

Verilog-XL Reference

Using Specify Blocks and Path Delays

Module path delay pulse filtering



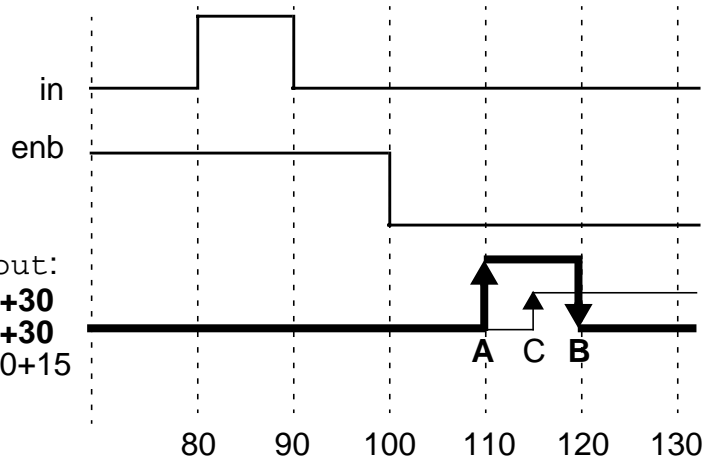
```

in => out = (30, 30);
enable => out = 0->1 = 0
           1->0 = 0
           0->z = 15
           z->1 = 0
           1->z = 25
           z->0 = 0
    
```

Scheduled events in and enb to out:

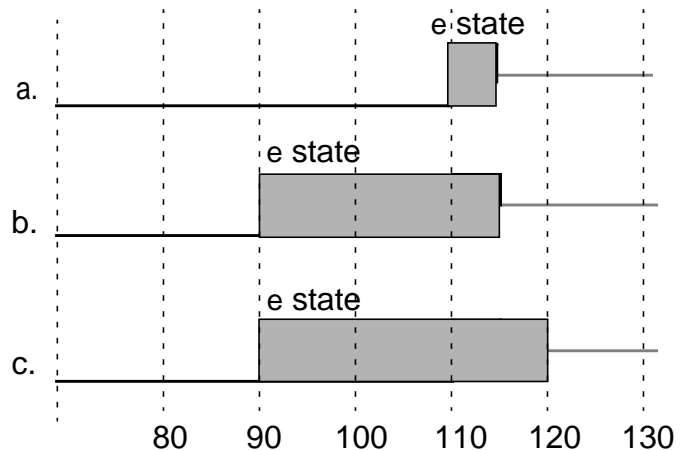
```

A: 0->1 80+30
B: 1->0 90+30
C: 0->Z 100+15
    
```



Plus Option Waveform Effect on Module Path Delays

- a. +pulse_e_style_onevent,
+show_cancelled_e
 or
 +pulse_e_style_onevent,
 +show_cancelled_e,
 +no_show_cancelled_e
- b. +pulse_e_style_ondetect,
+show_cancelled_e,
+no_show_cancelled_e
- c. +pulse_e_style_ondetect,
+show_cancelled_e



The waveforms in the previous figure are explained as follows:

- Wave a occurs because of pulse filtering. It indicates an inaccurate delay because Verilog-XL is choosing a 0->Z delay (115) to schedule a 1->Z transition.
- Wave b shows an e state from time 90 to time 115, which is the time from the final schedule to the time of the cancelled schedule. After time 115, the output is in the Z state.

Verilog-XL Reference

Using Specify Blocks and Path Delays

- Wave `c` shows an `e` state from time 90 to time 120, which is the time from the pulse that caused the `e` state to the last output transition. After time 120, the output is in the Z state. The wave at time 90 to 100 is due to pulse filtering; time 100 to 120 is due to schedule cancellation.

You can display cancelled schedules in specify blocks using the `$showcancelled` task. The `$noshowcancelled` task disables the display of cancelled schedules.

You can also display cancelled schedules or disable the display of cancelled schedules from the command line using the `+show_cancelled_e` or the `+no_show_cancelled_e` plus option. When you use the plus option, Verilog-XL displays all or none of the cancelled schedules, overriding any specify block tasks in the description.

The syntax to display or hide cancelled schedules is as follows:

```
$showcancelled[(<path_output>+)];  
$noshowcancelled[(<path_output>+)];
```

The following example shows how to display or hide cancelled schedules:

```
specify  
    (in => outbar) = (2, 3); // not shown by default  
    $showcancelled; // $showcancelled affects out  
    (in => out) = (5, 6);  
    (clk => out) = (4);  
    $noshowcancelled; // affects sync and output  
    (in => sync) = (20, 30);  
    (in => output) = (7, 9);  
endspecify  
specify  
    $showcancelled(out); // affects out only  
    (in=>out)=(15,25);  
    (clk=>q)=5;  
endspecify
```

The cancelled schedule specifications in the following example produces an error.

```
specify  
    $noshowcancelled(out); // sets no display on out  
    $showcancelled(out); // error by changing out state  
endspecify  
specify  
    $noshowcancelled; // sets no display  
    (in=>out)=(15,25);  
    $showcancelled(out); // error by changing out state  
endspecify
```

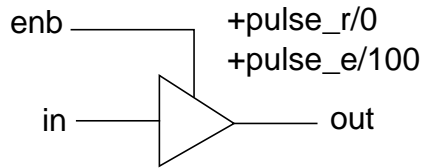
Pulse Filtering and Cancelled Schedule Dilemmas

Some cancelled schedules create a dilemma because Verilog-XL can recalculate a delay. This may result in the original schedule not having to be cancelled. The following figures show how cancelled schedule dilemmas occur and how various combinations of plus options affect the e state region for transport path delays. The same waveforms are produced when you use the tasks that correspond to the plus option (`$pulsestyle_ondetect`, `$pulsestyle_onevent`, `$showcancelled`, and `$noshowcancelled`).

Verilog-XL Reference

Using Specify Blocks and Path Delays

Note: The following figures show waveforms that apply to transport path delays (using the `+transport_path_delays` plus option) and transport interconnect delays (using the `+transport_int_delays` plus option).



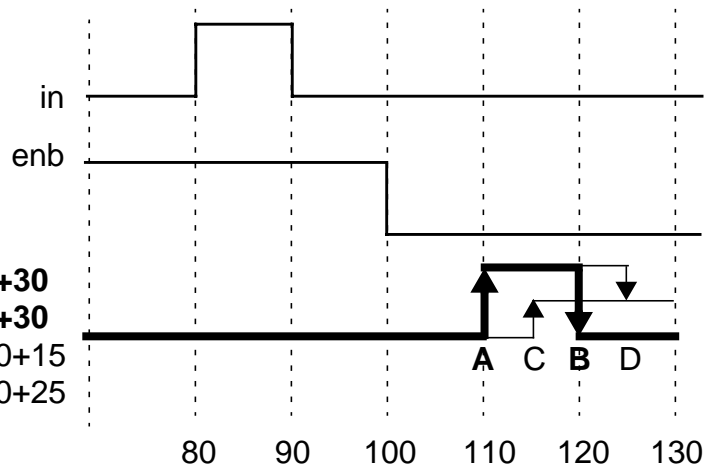
```

in => out = (30, 30);
enable => out = 0->1 = 0
              1->0 = 0
              0->z = 15
              z->1 = 0
              1->z = 25
              z->0 = 0
    
```

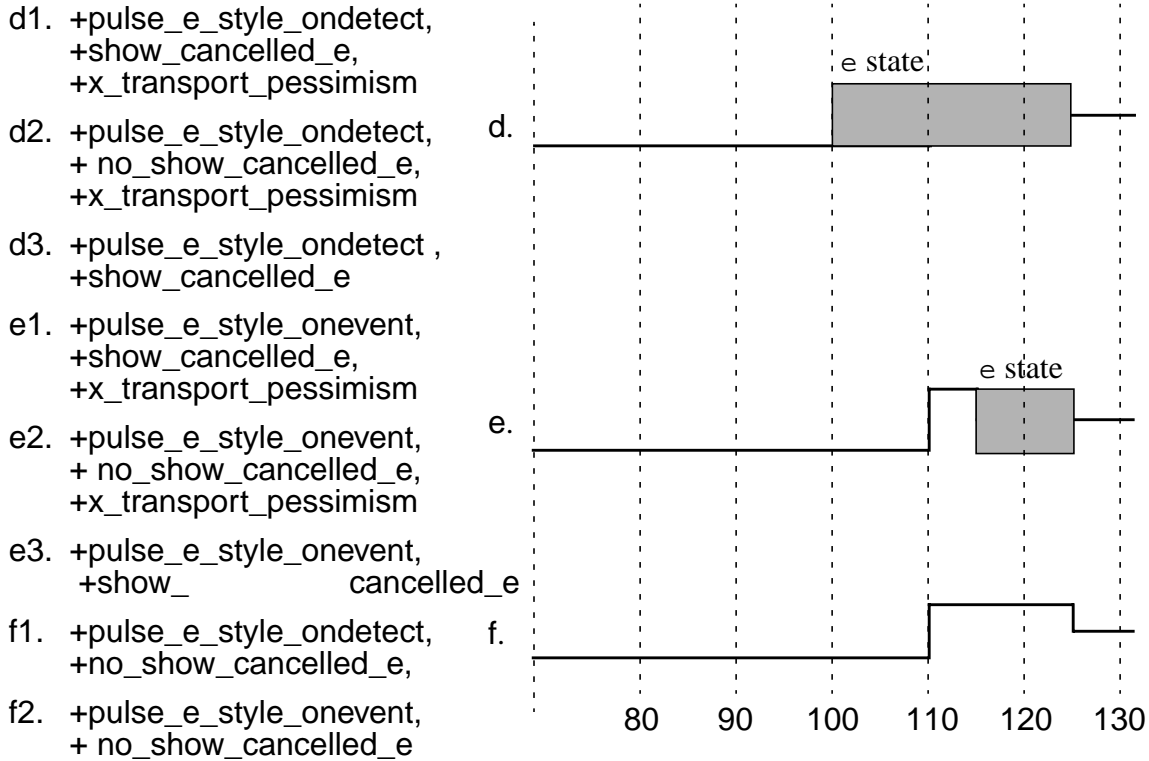
Scheduled events in and enb to out:

```

A: 0->1 80+30
B: 1->0 90+30
C: 0->Z 100+15
D: 1->Z 100+25
    
```



Plus Option Waveform Effects on Transport Path Delays



The wave forms in the previous figures are explained as follows:

- Wave d occurs on transport path delays for the following reasons:
 - Case d1 produces wave d because the on-detect style schedules an e state at the edge of the event that caused the pulse to occur, which is the transition on enb at time 100. (Note that the outcome of setting +x_transport_pessimism makes the +show_cancelled_e plus option redundant.)
 - Case d2 produces the same wave as d1 because the outcome of setting +x_transport_pessimism makes the +no_show_cancelled_e plus option irrelevant.
 - Case d3 produces wave d because the display of cancelled events causes the x state to appear on the output, beginning at time 100, because the pulse filtering style is set to on-detect, which is the time of the input transition that caused the cancelled schedule.

- Wave e occurs on transport path delays for the following reasons:
 - Case e1 produces wave e because the transport delay algorithm is being used, and the 0->Z delay is recalculated based on a 1->Z transition. This changes the time of

Verilog-XL Reference

Using Specify Blocks and Path Delays

the transition to Z on the output to time 125. However, if that is done, then the 1->0 transition at time 120 no longer needs to be cancelled, producing a cancelled schedule dilemma (CSD). Because `+x_transport_pessimism` is set, the CSD causes an x to appear on the output from time 115 to 125. (Note that the outcome of setting `+x_transport_pessimism` makes the `+show_cancelled_e` plus option redundant.)

- ❑ Case e2 produces the same wave as e1 because the outcome of setting `+x_transport_pessimism` makes the `+no_show_cancelled_e` plus option irrelevant.
- ❑ Case e3 produces wave e because even though there is no `+x_transport_pessimism`, the display of cancelled schedules causes an x state to appear on the output due to the 1->0 schedule at time 120 being cancelled. (The only difference from case d1 is the time of the final transition to Z, which is due to the delay recalculation that is part of the transport delay algorithm.)
- Both cases f1 and f2 produce wave f because neither `+x_transport_pessimism` nor `+show_cancelled_e` is set. Therefore, no x state appears in the output regardless of the setting of on-event or on-detect.

Using State-Dependent Path Delays (SDPDs)

An SDPD is a conditional module path delay; it assigns a delay to a module path when specific conditions are true. An SDPD includes the following items:

- a conditional expression
- a module path description
- a delay expression that applies to the module path

The syntax for an SDPD is as follows:

```
<sdpd>
 ::= if(<sdpd_conditional_expression>)(<path_description>)=
    (<path_delay_value>)
    ||= <ifnone_path>

<sdpd_conditional_expression>
 ::= <expression>

<path_description>
 ::= (<path_input> => <path_output>)
    ||= (<list_of_path_inputs> * > <list_of_path_outputs>)
    ||= (<edge_identifier> <path_input> => (<path_output>
```

Verilog-XL Reference

Using Specify Blocks and Path Delays

```
    <polarity_operator>?:<data_source_expression>))
  ||= (<edge_identifier> <path_input> * (<list_of_path_outputs>
    <polarity_operator>?:<data_source_expression>))

<path_delay_value>
  ::= (<path_delay_expression>)
  ||= (<path_delay_expression>, <path_delay_expression>)
  ||= (<path_delay_expression>, <path_delay_expression>,
    <path_delay_expression>)
  ||= (<path_delay_expression>, <path_delay_expression>,
    <path_delay_expression>, <path_delay_expression>,
    <path_delay_expression>)

(Parentheses in <path_delay_value> are optional.)

<ifnone_path>
  ::= ifnone(<path_description>)=(<path_delay_value>)
```

Evaluating SDPD Expressions

An SDPD expression must evaluate to one bit. In Verilog-XL, SDPD expressions that evaluate to 0 are false, and SDPD expressions that evaluate to 1, X, or Z are true.

Note: Evaluating SDPD expressions is different from evaluating other Verilog HDL constructs. For example, in the behavioral language, `if` statements that evaluate to `x` or `z` are false. The SDPD expression is consistent with Veritime™ path selection.

If multiple SDPDs are specified for a path, Verilog-XL looks at the delays for all statements whose conditions are true, determines which source has had the most recent transition, and selects the smallest delay. See [“Working with Multiple Path Delays”](#) on page 275 for details on how Verilog-XL selects a delay when multiple delays are specified for a path.

Unconditional path delays are always considered true. The `ifnone` construct allows you to specify a delay for cases in which all of the SDPD expressions are false.

The operands in an SDPD expression must be one of the following:

- A scalar or vector module input, output, or inout port—in its entirety or in bit-select or part-select form.
- A compile time constant—an expression whose value can be computed at compile time. Its value does not change during simulation.
- A parameter—an expression that can be changed after compile time, even though the updated value is *not* used. The parameter value that Verilog-XL uses is only the compile-time value.

Verilog-XL Reference

Using Specify Blocks and Path Delays

- A net or register (subject to restrictions)—declared within the module containing the SDPD description.

The nets or registers in an SDPD conditional expression are subject to the same limitations as the right-hand sides of accelerated continuous assignments.

The *prohibited* nets and registers in an SDPD expression are the following:

- expanded vector nets that contain more than 127 bits
- unexpanded vector nets
- bit-selects of unexpanded vector nets
- part-selects of unexpanded vector nets
- vector registers
- bit-selects of vector registers
- part-selects of vector registers
- `specparams`, the parameters declared in specify blocks
- integers or real numbers

The SDPD expression can have any number of operators. The following table shows the valid operators in SDPD expressions. The *valid* operators in SDPD expressions are the same ones supported by accelerated continuous assignments.

Bit-wise	& and, or, ^ xor, ~^ xnor, ~ negation
Reduction	& and, or, ^ xor, ~^ xnor, ~& nand, ~ nor
Logical	&& and, or, == equality, != inequality, ! not
Other	{} concatenation, {{}} duplicate concatenation, === case equality, !== case inequality, ?: conditional

The following table shows the operators that are *invalid* in SDPD expressions:

Arithmetic	+ - * /
Relational	> >= < <=
Left shift	<<
Right shift	>>

Verilog-XL Reference

Using Specify Blocks and Path Delays

Modulus	%
---------	---

In the following example, you use SDPDs to describe a pair of output rise and fall delay times when the XOR inverts a changing input. When the XOR buffers a changing input, SDPDs allow you to describe another pair of output rise and fall delay times.

```
module sdpdexample (a,b,out);
input a,b;
output out;
xor (out,a,b);
    specify
    specparam noninvrise = 1, noninvfall = 2;
    specparam invertrise = 3, invertfall = 4;
    if(a) (b=>out)=(invertrise,invertfall); // SDPD
    if(~a) (b=>out)=(noninvrise,noninvfall); // SDPD
    if(b) (a=>out)=(invertrise,invertfall); // SDPD
    if(~b) (a=>out)=(noninvrise,noninvfall); // SDPD
    endspecify
endmodule
```

In the next example, SDPDs specify different sets of path delays for different ALU operations. The first three path declarations declare paths extending from the operand inputs to the o1 output. The delays on these paths are assigned to operations based on the operation specified by the inputs on opcode. The last path declaration declares a path from the opcode input to the o1 output.

```
`timescale 1ns/100ps
module ALU(o1,i1,i2,opcode);
input [7:0] i1,i2;
input [2:1] opcode;
output [7:0]o1;
    ...
    specify
    // add operation
    if (opcode == 2'b00)
        (i1,i2 *> o1) = (25.0,25.0);

    // pass-through i1 operation
    if (opcode == 2'b01)
        (i1 => o1) = (5.6,8.0);

    // pass-through i2 operation
    if (opcode == 2'b10)
        (i2 => o1) = (5.6,8.0);

    // delays on opcode changes
    (opcode *> o1) = (6.1,6.5);
    endspecify
endmodule
```


Using Edge Keywords in SDPDs

SDPDs in Verilog-XL permit edge keywords (posedge and negedge) in module path descriptions, but ignore their meaning. See [“Using Edge-Control Specifiers”](#) on page 291 for more information about edge-control specifiers.

The following example shows how Verilog-XL interprets the edge keywords. The upper specify block contains module path descriptions that Verilog-XL interprets as if they were the module path descriptions in the lower specify block.

```
specify
  specparam trise=2, tfall=3;
  if(in1&&in2)(posedge clock=>(out1-:in3))=(trise,tfall);
  if(~in1)(negedge clock=>(out2+:in3))=(trise,tfall);
endspecify

specify
  specparam trise=2, tfall=3;
  if(in1&&in2)(clock=>out1)=(trise,tfall);
  if(~in1)(clock=>out2)=(trise,tfall);
endspecify
```

You can implement edge conditions, however, using the ``ifdef` compiler directive.

The following examples illustrate a problem in using edge keywords in SDPD expressions, and show how you can have the functionality of edge keywords in SDPDs.

```
/* The following lines are an unsuccessful attempt to impose
   different edge-conditioned delays on a path, because Verilog-XL
   does not use the edge information. */
(posedge clk => (q_out +: d_in)) = 15;
(negedge clk => (q_out +: d_in)) = 9;

/* Verilog-XL interprets the preceding lines to have the meaning
   of the following lines, and it chooses 9, the lesser of the delays,
   in all conditions. */
( clk => q_out ) = 15;
( clk => q_out ) = 9;

/* The following lines implement the intended edge conditions.
   Their syntax makes it possible for both Verilog-XL and Veritime
   to use the same library. */
`ifdef verilog if(clk == 1) `endif
  (posedge clk => (q_out +: d_in)) = 15;
`ifdef verilog if(clk == 0) `endif
  (negedge clk => (q_out +: d_in)) = 9;
```

Both lines in the previous example contain conditions that ensure that the SDPDs apply only to a specific edge transition. The first line results in a path delay of 15 for signals with a positive edge propagating from `clk` to `q_out`. The second line results in a path delay of 9 for signals with a negative edge propagating from `clk` to `q_out`.

Making SDPDs Function as Unconditional Delays

To simulate SDPDs as unconditional delays while performing Verilog-XL simulations, use the `+pre_16a_paths` plus option on the command line.

You can also enable and disable the conditional path functionality with the ``pre_16a_paths` and the ``end_pre_16a_paths` compiler directives, which are described as follows:

- ``pre_16a_paths`
Treats conditional paths as if their conditional expressions are always true, as in Verilog-XL versions prior to 1.6a.
- ``end_pre_16a_paths`
Simulates SDPDs as conditional delays.

Note: The ``resetall` compiler directive also turns off the functionality of conditional paths.

The ``pre_16a_paths` compiler directive remains in effect, even across multiple Verilog files, until the ``end_pre_16a_paths` or ``resetall` compiler directive is specified. If you specify the `+pre_16a_paths` plus option on the command line, the ``pre_16a_paths` and the ``end_pre_16a_paths` compiler directives are disabled.

Simulating 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

Working with Distributed Delays and SDPDs

When a distributed delay and a path delay apply to a path, the larger of the two delays schedules an output change.

For realistic modeling, larger modules require gate and net delays, and behavioral models require procedural delays. These delays can have an undesirable impact on the choice of path delays. In larger cells and modules that require distributed delays and SDPDs, inputs should not change before an edge has passed to the outputs so that Verilog-XL can choose an appropriate path delay based on the input state that generates the output change.

Working with Multiple Path Delays

The following table summarizes how Verilog-XL selects a delay from multiple path delay specifications.

Delay Types and Examples	Verilog-XL Delay Choice
<p>Two edge-sensitive delays with different edges</p> <pre>(posedge in => (out:d1)) = 10; (negedge in => (out:d2)) = 9;</pre>	<p>Select min delay</p> <pre>delay = min(9, 10)</pre>
<p>Multiple SDPDs</p> <pre>if (c1) (in => out) = 10; if (c2) (in => out) = 9;</pre>	<p>Select min of all true conditions</p> <pre>if (c1 && c2) delay = min(10, 9) else if (c1) delay=10 else if (c2) delay=9 else delay=0;</pre>
<p>Multiple SDPDs and an unconditional path delay</p> <pre>if (c1) (in => out) = 10; if (c2) (in => out) = 9; (in => out) = 8;</pre>	<p>Unconditional path delay is always true. Select min of all true conditions</p> <pre>if (c1 && c2) delay = min(10,9,8) else if (c1 && !c2) delay=min(10,8) else if (!c1 && c2) delay=min(9,8) else delay=8;</pre>
<p>Multiple SDPDs and ifnone delay</p> <pre>if (c1) (in => out) = 10; if (c2) (in => out) = 9; ifnone (in => out) = 8;</pre>	<p>Select min of all true conditions. Select ifnone delay if no true conditions</p> <pre>if (c1 && c2) delay=min(10,9) else if (c1 && !c2) delay=10 else if (!c1 && C2) delay=9) else delay = 8;</pre>
<p>Multiple unconditional path delays</p> <pre>(in => out) = 10; (in => out) = 9;</pre>	<p>Error</p>
<p>Multiple unconditional path delays, one of which redefines an input or output as a bit- or part-select</p> <pre>(in => out) = 10; (in[1] => out[1]) = 9;</pre>	<p>Error</p>
<p>Unconditional edge-sensitive delay and unconditional level-sensitive delay</p> <pre>(posedge in => (out:d)) = 10; (in => out) = 9;</pre>	<p>Error</p>

Effects of Unknowns on SDPDs

With the typical implementation of level-sensitive qualifiers, Verilog-XL handles unknowns properly. The following example shows a level-sensitive path delay.

```
if (flag == 1)    ( in => out ) = 7,9;
if (flag == 0)    ( in => out ) = 10,5;
```

When `flag` is 1, the `out` signal rises 7 time units and falls 9 time units after the `in` signal changes. When `flag` is 0, the `out` signal rises 10 time units or falls 5 time units after the `in` signal changes. But when `flag` is unknown, the output rises in $\min(7,10)$ time units and falls in $\min(9,5)$ time units.

When an SDPD expression has an unknown value as an operand, Verilog-XL treats the resulting delay as an SDPD whose condition is true. The following table shows all possible conditional expressions, using `flag`, for a path from `in` to `out`. The table also shows the delays that Verilog-XL selects when `flag` is 1, 0, X, or Z.

SDPD expression: path selected when...	flag is 1	flag is 0	flag is X or Z
<code>if (flag == 1) (in => out) = a;</code>	Yes	No	Yes
<code>if (flag == 0) (in => out) = b;</code>	No	Yes	Yes
<code>if (flag == X) (in => out) = c;</code>	Yes	Yes	Yes
<code>(in => out) = d;</code>	Yes	Yes	Yes
Delay selected for path from <code>in</code> to <code>out</code>	$\min(a,c,d)$	$\min(b,c,d)$	$\min(a,b,c,d)$

Note: The examples in the previous table are used only to help you understand how each statement is handled by Verilog-XL. They are not recommended modeling practice.

The condition `(flag == X)` has no effect because this path delay will always be selected. If you do not care about the value of `flag`, specify an unconditioned path using the delay by itself (`d`) or a complete set of conditional path delays (`a` and `b`).

The case equality operator (`===`) and the case inequality operator (`!==`) have different effects than the logical equality operator. The condition `if (flag === 1)` is true if `flag` is 1, but false if `flag` is X.

Verilog-XL uses the minimum delays because the only time when multiple paths should be selected is when unknowns are introduced into conditional expressions. When unknowns are in the conditional expression, then it is likely that the output value will be corrupted by the unknown signal. This results in the output signal going to an unknown value after the minimal delay.

Effects of Unknowns on Edge-Sensitive Delays

To specify edge-sensitive delays with unknowns, you can specify a conditional expression similar to the D-type flip-flop shown in the following example

```
if (clk == 1) (posedge clk => (q_out +: d_in)) = 15;
```

When `clk` makes a 1→X or X→0 transition, Verilog-XL evaluates the model and determines that the output does not change. Therefore, it will not use the path delay for these transitions. The output changes only when `clk` makes a 0→X or X→1 transition. In this case, Verilog-XL uses the path delay for positive transitions of the `clk` signal.

When a model has different edge qualifiers for the same path, as in the following example, Verilog-XL ignores the edge information and selects both path delays.

```
`ifdef verilog if(clk==1) `endif (posedge clk=>(q_out +: d_in)) = 9;  
`ifdef verilog if(clk==0) `endif (negedge clk=>(q_out -: d_in)) = 15;
```

To specify edge-sensitive conditional expressions in Verilog-XL, see [“Using Edge Keywords in SDPDs”](#) on page 273.

The following table shows the effects in Verilog-XL when `clk` from the previous example makes various transitions:

clk	Effect in Verilog-XL
0→X	Ignores the edge information, uses level-sensitive qualifiers, and selects both path delays: min(9,15).
X→1	Selects the first path delay.
1→X	Selects both paths: min(9,15).
X→0	Selects the second path delay.

Possible Effects of Internal Logic

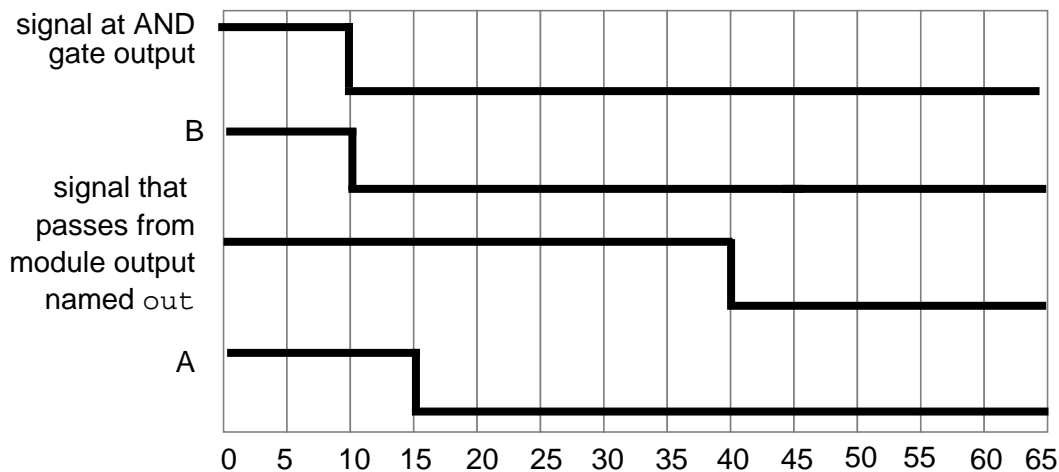
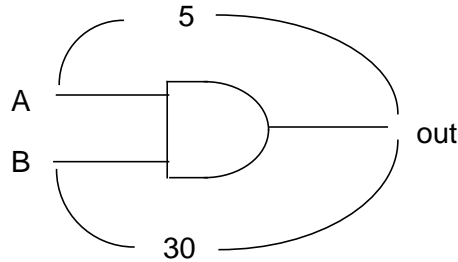
When the same output terminates multiple paths, some combinations of module path declarations that include that output can cause unexpected modeling results. The following figure shows this with a module that has one output port designated `out` and two input ports designated `A` and `B`.

The module contains zero delay logic. Input `A` has a delay of 5 to the output. Input `B` has a delay of 30 to the output. At simulation time 10, Verilog-XL evaluates the gate and determines

Verilog-XL Reference

Using Specify Blocks and Path Delays

a change in `out` to 0. It schedules the change to appear at time 40, based on the path delay from `B` to `out`. When input `A` changes to 0 at time 15, Verilog-XL does not reschedule the change in `out` to time 20, because Verilog-XL schedules output changes when edges transmit to module outputs. The change in `A` to 0 at time 15 does not transmit an edge to `out` because the net named `out`, which is internal to the module, already has the value 0, due to the change in `B` at time 10.



The 25 time unit difference between the two path delays is significant for the following reasons:

- It is the length of the period that follows the change on the input of the longer delay path during which a change on the input of the shorter delay path can introduce the unexpected behavior.
- It is the maximum possible deviation from the expected timing for the change in the module output signal.

Enhancing Path Delay Accuracy

The default module path delay algorithm in Verilog-XL selects path delays without considering circuit logic. Therefore, Verilog-XL may select a delay that cannot cause a transition.

You can use an alternative `accu_path` delay algorithm to affect the choice of delay paths. You can also use SDPDs to affect the choice of delay paths, but `accu_path` generally produces the least complicated source code. See [“Using State-Dependent Path Delays \(SDPDs\)”](#) on page 269 for information about SDPDs.

Invoking the `accu_path` Algorithm

You can enable the `accu_path` algorithm in two ways:

- Specify the `+accu_path_delay` plus option on the command line to apply the `accu_path` algorithm to any module output for which there is a path delay specification.
- Specify the `$eventcond` system task in a specify block to apply either the `accu_path` algorithm to all paths in a module or to selected paths in a module. (Specify the `$noeventcond` system task to re-apply the default delay selection algorithm.)

The following example shows how to use the `$eventcond` system task:

```
specify
  $eventcond;      // Use accu_path algorithm
  (a => out) = 5;
  (b => q) = 10;
  $noeventcond;   // Use default algorithm
  (c => out) = 5;
  (d => q) = 10;
endspecify
```

You should use `$eventcond` and `$noeventcond` for the following reasons:

- Model vendors can enable `accu_path` in specific cells or for paths within a cell where necessary or appropriate.
- Using the `accu_path` algorithm selectively usually results in better performance than using it all the time with the `+accu_path_delay` plus option.

If you specify the name of a path output as a parameter to `$eventcond`, Verilog-XL applies the `accu_path` algorithm to paths terminating with that output and uses the default algorithm for other paths.

However, when you specify the `+accu_path_delay` plus option, the `accu_path` algorithm is in effect for all paths. Then, if you specify the name of a path output as a parameter to

Verilog-XL Reference

Using Specify Blocks and Path Delays

`$noeventcond`, Verilog-XL applies the default algorithm only to paths terminating with that output.

The following examples show how to use a parameter with `$eventcond` and with `$noeventcond`. The specified algorithm is the `accu_path` algorithm if you specified the `+accu_path_delay` plus option on the command line. Otherwise, the specified algorithm is the default.

```
specify
  $eventcond (out);
  (a => out) = 5;      // Use accu_path algorithm
  (b => q) = 10;      // Use specified algorithm
endspecify
specify
  (a => o) = 3;      // Use specified algorithm
  $noeventcond (out);
  (i => out) = 5;    // Use default algorithm
  (i => q) = 10;    // Use specified algorithm
  (b => out) = 5;    // Use default algorithm
endspecify
```

All paths to the same output must use the same algorithm. If you set paths to the same output to two different settings, a warning is issued and the first setting is used. For instance, in the following example, Verilog-XL uses the `accu_path` algorithm.

```
specify
  $eventcond;
  (i => out) = 5;    // Use accu_path
  $noeventcond;
  (a => out) = 10;   // Use older
endspecify
```

To use the `accu_path` algorithm, Verilog-XL determines how the logic of the circuit affects the choice of delay paths. If Verilog-XL finds a condition that prevents it from making this determination, Verilog-XL uses the default path delay algorithm.

The following conditions prevent Verilog-XL from using the `accu_path` algorithm:

- A circuit loop between the path input and the path output
- A distributed delay on a net or gate in the module on the path between the path input and the path output
- Bidirectional switches on the path between the path input and the path output
- An expression driving a net on the path between the path input and the path output
- A register driving a net on the path between the path input and the path output
- A net on the path between the path input and the path output that has no driver in the module

- A net on the path between the path input and the path output that is driven by a sequential user defined primitive
- A net on the path between the path input and the path output that is driven by an internal primitive used for SWITCH-XL, CAXL, or TURBO

Comparing the Default and `accu_path` Delay Selection Algorithms

This section compares the default delay selection algorithm with the `accu_path` algorithm. It includes several examples to help you decide when to use the performance optimizations of the default algorithm or the greater accuracy of the `accu_path` algorithm.

Summary of the default delay selection algorithm

To simulate module path delays, Verilog-XL schedules events at module path outputs. When Verilog-XL detects an event at a module path output, the default algorithm:

1. Examines each input that has a module path to the output when its event occurred.
2. Determines the time of the most recent input event (T1).
3. Determines the delay of the module path connecting the output to the input with the most recent event (T2).
4. Adds T1 and T2 to determine the time for scheduling the output event. If two or more of the most recent events are simultaneous, Verilog-XL selects the shortest of the path delays. If internal delays cause the sum of T1 and T2 to be less than the current time, then Verilog-XL schedules the output event at the current time.
5. Schedules the output event.

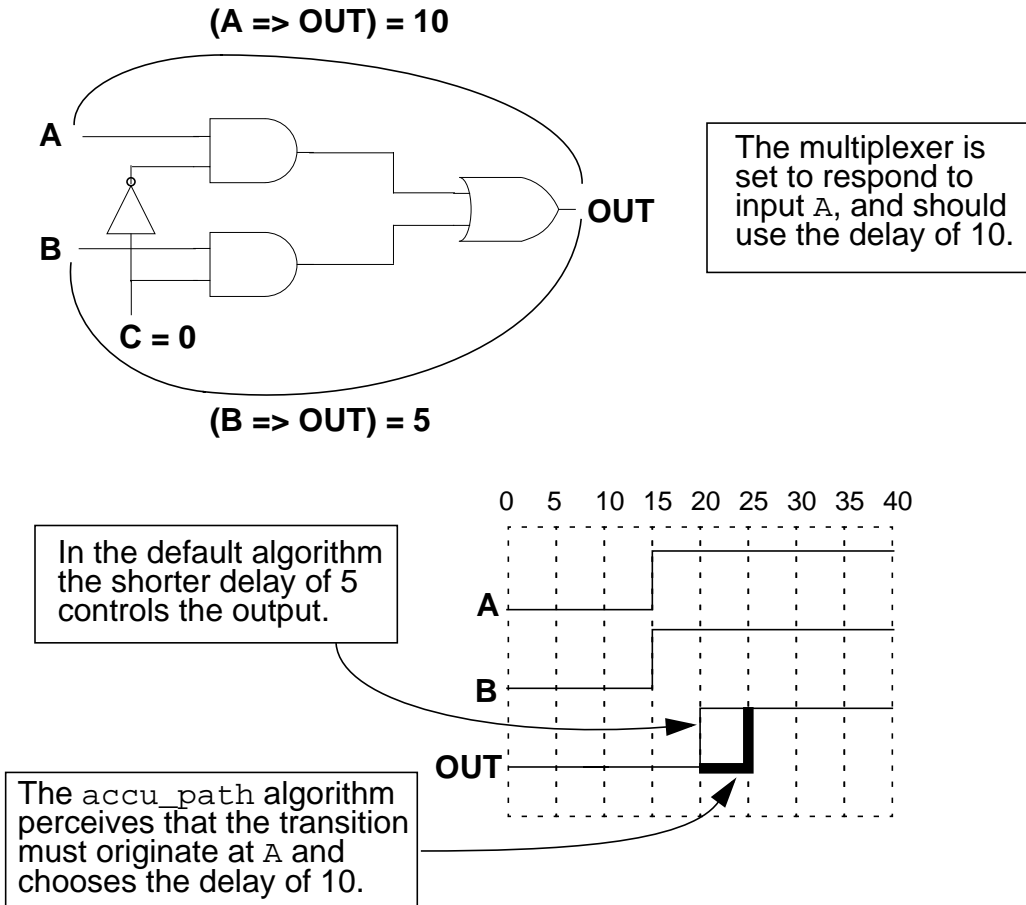
Examples of choices by the two algorithms

The following figure shows how the default delay selection algorithm causes the selection of the nonlogical shorter path when a multiplexer's inputs experience simultaneous transitions. In this example, the default algorithm groups all inputs connected to the output by unconditional paths, and the path in that group with the shortest delay (`B=>OUT`) cannot

Verilog-XL Reference

Using Specify Blocks and Path Delays

logically control any transition at `OUT`. By contrast, the `accu_path` algorithm evaluates only the input that can affect `OUT`.

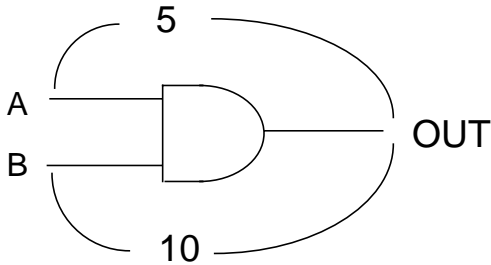


The next figure shows how the default algorithm does not correctly evaluate a path when an earlier input event should control the output event delay. The zero-delay logic of the `AND` gate delivers a change to `OUT` at time 10, and the default algorithm determines a time for scheduling the event on `OUT` by evaluating only the path with the latest input event (`A=>OUT`).

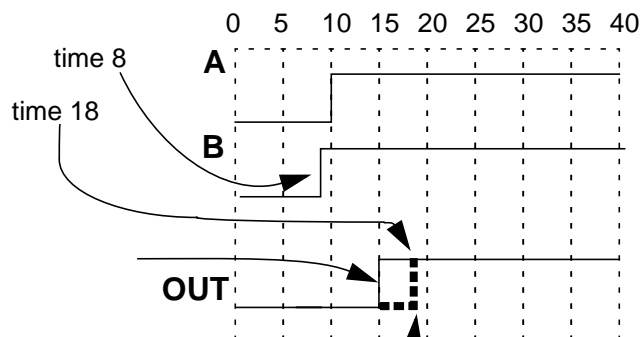
Verilog-XL Reference

Using Specify Blocks and Path Delays

By contrast, the *accu_path* algorithm schedules the event on `OUT` at the earliest time that `OUT` can change, which is 10 units after the transition on `B` in this case.



The default algorithm schedules the transition on `OUT` too early because it does not evaluate the delay on the path from `B` to `OUT`. It evaluates only the delay on the path from `A` to `OUT`, because that path has had the most recent input event.



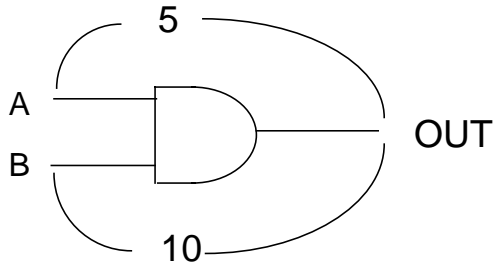
The *accu_path* algorithm correctly selects the time at which both inputs can affect `OUT`.

The following figure shows how a selection does not model hardware in the manner suggested by the path delay specifications because the default algorithm selects the shorter path delay when two inputs transition simultaneously. When both inputs `A` and `B` transition at time 10, the zero-delay logic immediately delivers an output change to `OUT`. The default algorithm schedules the transition at `OUT` by selecting the shorter delay of 5 on the path from

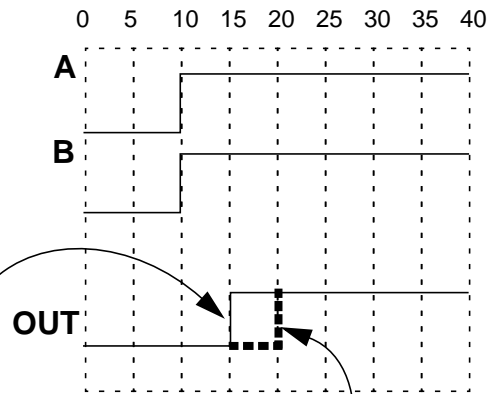
Verilog-XL Reference

Using Specify Blocks and Path Delays

A to OUT. By contrast, the *accu_path* algorithm schedules the change on OUT because the transition at input B transmits to OUT at time 20.



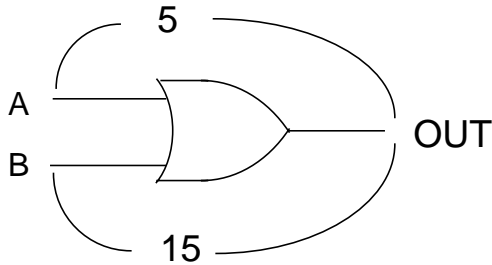
The default algorithm schedules the transition on OUT earlier than it should appear, because it selects the shorter of the two delays on paths with simultaneous input events.



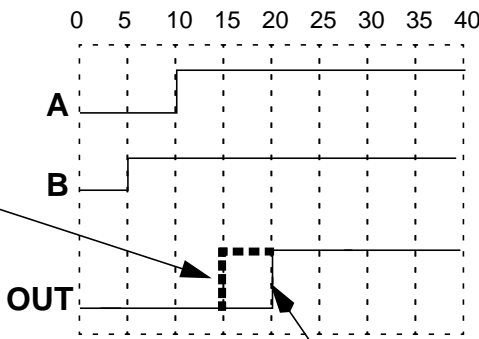
The *accu_path* algorithm correctly selects the time at which both inputs can affect OUT.

The following figure shows how an input transition, that occurs after a previous input transition has already scheduled an output event, can cause the *accu_path* algorithm to reschedule

the output event. If the later input event indicates that the output is to occur earlier than previously scheduled, then the output event is rescheduled.



The `accu_path` algorithm reschedules the output event to an earlier time, correctly reflecting its origin in a later input event on a path that has a short delay.



The default algorithm cannot reschedule an output event to an earlier time.

Limits of the `accu_path` Algorithm

The `accu_path` algorithm can exhibit unexpected behavior when the following conditions exist:

- Inputs change simultaneously or before the time difference between the two inputs.
- User-defined primitives (UDPs) are written in such a way that not all inputs have explicit table entries

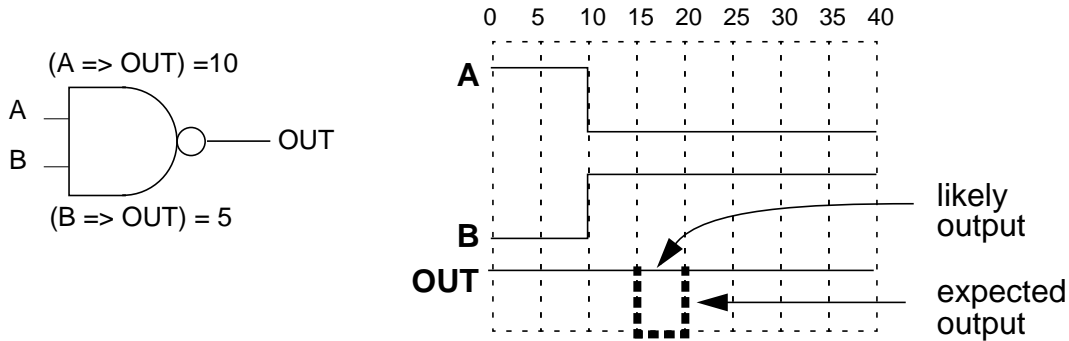
Examples of the Limits of the `accu_path` Algorithm

Consider the `nand` gate with the delays on the left side of the following figure. The right side shows potential waveforms at the inputs and output of the gate. The `accu_path` algorithm selects the shorter delay and produces the expected output *only* if input B makes the transition first, because only that situation can cause a change at the output. If the inputs make the transition simultaneously, or if A makes the transitions first, no change is scheduled

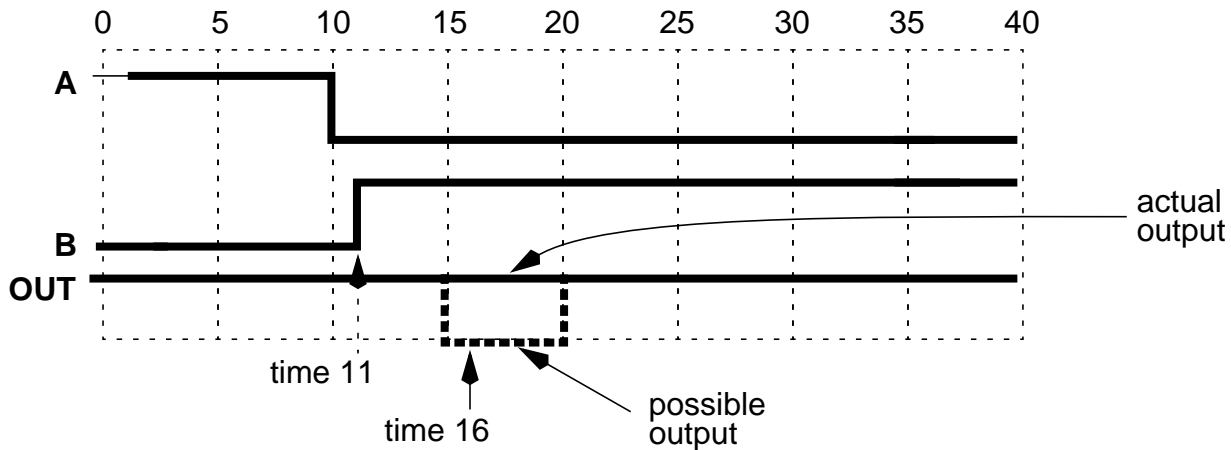
Verilog-XL Reference

Using Specify Blocks and Path Delays

on the NAND gate output, and so no transition is delivered to `OUT` for delay selection and scheduling.



The following figure shows the input of the longer path making a transition first. Because the inputs never simultaneously have values of 1, no output event on the NAND gate is delivered to `OUT` so that the `accu_path` algorithm can select a delay.

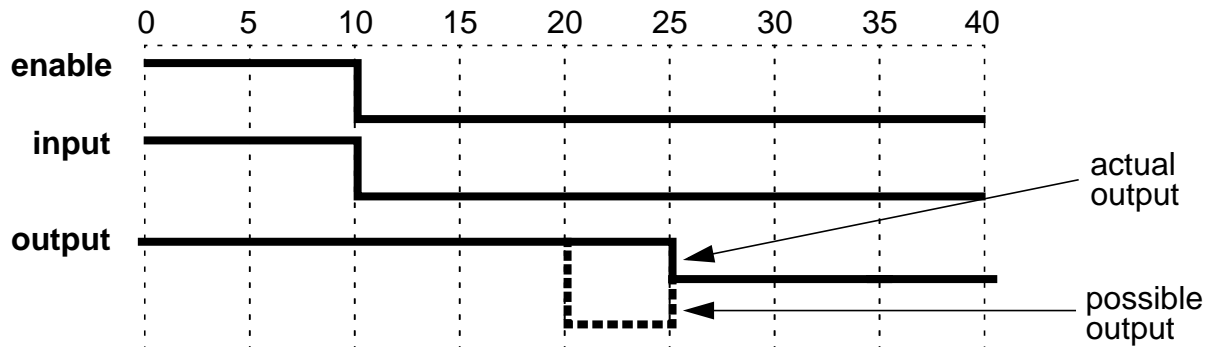


Verilog-XL Reference

Using Specify Blocks and Path Delays

A `bufif1` or `bufif0` with an `enable` to output delay that exceeds its input to output delay may not show an expected pulse when the `enable` and the `input` transition is simultaneous, as the following figure shows for a `bufif1`.

```
:  
( enable => out ) =15;      ( input => out ) = 10;
```



The `accu_path` algorithm can behave differently for logically equivalent but differently written UDPs. Listing all the possible input patterns results in the most pessimistic, or latest, scheduling of output changes. A logically equivalent, but shorter, description including the `?` and `b` symbols can result in less pessimistic or earlier, appearances of values at outputs. The reduction in pessimism occurs because inputs with non-explicit UDP table entries do not affect the output delay for all input vectors described by the table entries.

Using the `accu_path` algorithm can result in timing dilemmas caused by event cancellation. Use the `+x_transport_pessimism` plus option to cause an X state to appear on the output. See [“Understanding Path Delays”](#) on page 239 for more information and an example.

Verilog-XL Reference

Using Specify Blocks and Path Delays

Timing Checks

This chapter describes the following:

- [Overview](#) on page 289
- [Using Timing Checks](#) on page 289
- [Using the Timing Check System Tasks](#) on page 295
- [Using Negative Timing Check Limits in \\$setuphold and \\$recrem](#) on page 313

Overview

A timing check is a system task that performs the following steps:

1. Determines the time between two events.
2. Compares the elapsed time to specified minimum or maximum time limits.
3. Reports a timing violation whenever the elapsed time occurs outside the specified time limits.

Timing checks can contain edge-control specifiers (see [“Using Edge-Control Specifiers”](#) on page 291), notifiers (see [“Using Notifiers for Timing Violations”](#) on page 292), and conditions (see [“Enabling Timing Checks with Conditioned Events”](#) on page 293).

Note: If you do not want to check timing data, you can improve processing performance by disabling timing checks by compiling with the `+notimingchecks` plus option on the command line, as follows. Module path delays remain active.

```
verilog source.v +notimingchecks
```

Using Timing Checks

To verify the timing characteristics of your design, you can invoke the following timing check system tasks in specify blocks:

Verilog-XL Reference

Timing Checks

```
$hold(<clk_event>, <data_event>, <hold_limit>{, <notifier>});  
$nochange(<clk_event>, <data_event>, <start_offset>, <end_offset>);  
$period(<clk_event>, <period_limit> {, <notifier>});  
$recovery(<control_event>, <clk_event>,  
    <recovery_limit>, {<notifier>});  
$creem(<control_event>, <clk_event>,  
    <recovery_limit>, <removal_limit>,  
    {<notifier>}, {<tstamp_cond>}, {<tcheck_cond>},  
    {<delayed_ctrl>}, {<delayed_clk>});  
$removal(<control_event>, <clk_event>,  
    <removal_limit>, {<notifier>});  
$setup(<data_event>, <clk_event>, <setup_limit>{, <notifier>});  
$setuphold(<clk_event>, <data_event>, <setup_limit>, <hold_limit>,  
    {<notifier>}, {<tstamp_cond>}, {<tcheck_cond>},  
    {<delayed_clk>}, {<delayed_data>});  
$skew(<clk_1>, <clk_2>, <skew_limit> {, <notifier>});  
$width(<edge_clk>, <min_limit> {,<threshold>{, <notifier>}});
```

Understanding Timing Violation Messages

When a system timing check encounters a timing violation, Verilog-XL reports the following information:

- File, line number, and instance name of the module in which the violation occurred
- Time of the second event (which is the violation)
- Time of the first event
- Value of the timing check limit

Timing check violation messages have one of two different formats depending upon whether the ``timescale` compiler directives control the modules containing them. In both of the examples, a timing violation occurred on line 16 of the Verilog source description file "source.v" in module `top.ff`.

The following message shows that without the ``timescale` directive the `$setup` system task reports the violation that has occurred at time 405 with a `clk` time of 410 and a timing check limit of 6.

```
"source.v", 16: Timing violation in top.ff  
    $setup( data:405, posedge clk:410, 6 );
```

Verilog-XL Reference

Timing Checks

The following example shows that with the ``timescale` directive, the `$setup` system task reports the violation that has occurred at time 4050 with a `clk` time of 4100 and a timing check limit of 60. The 6.0 in the violation message is the unscaled value that appears in the timing check code; the 60 is the scaled value that the timing check tests.

```
"source.v", 16: Timing violation in top.ff
  $setup( data:4050, posedge clk:4100, 6.0 : 60 );
```

The values of time limits in timing violation messages are always current, reflecting any changes made by PLI and SDF annotation.

Note: When a timing violation occurs due to a vector, system timing checks report one violation for each bit that changed.

Using Edge-Control Specifiers

You can control timing check events using specific edge transitions between 0, 1, and x. Verilog-XL treats edge transitions involving z the same way as edge transitions involving x.

To use edge-control specifiers, type the `edge` keyword followed by a square-bracketed list of from one to six edge specifiers separated by commas (01, 10, 0x, x1, 1x, x0), as shown in the following syntax:

```
edge[<edge_specifier_list>]
```

You can also specify the `posedge` and `negedge` keywords for edge transitions, as follows:

- The `posedge` keyword is equivalent to `edge[01,0x,x1]`.
- The `negedge` keyword is equivalent to `edge[10,x0,1x]`.

The following example shows how to use edge-control specifiers using the `$setup`, `$hold`, and `$width` system tasks. Timing checks for the `$setup` and `$hold` system tasks occur only when the `clk` transitions 0->1 or x->1. Timing checks for the first `$width` system task occurs when `clk` transitions 0->1 or x->1. Timing checks for the second `$width` system task occur when `clk` transitions 1->0 or x->0.

```
module DFF2(clk, d, q, qb);
input clk, d;
output q,qb;
...
specify
  specparam tSetup=60:70:75, tHold=45:50:55;
  specparam tWpos=180:600:1050,tWneg=150:500:880;
  $setup(d,edge[01,x1]clk,tSetup); // edge
  $hold(edge[01,x1]clk,d,tHold); // control
  $width(edge[01,x1]clk,tWpos); // specifiers:
  $width(edge[10,x0]clk,tWneg); // edge[...,...]
endspecify
endmodule
```

Verilog-XL Reference

Timing Checks

Using Notifiers for Timing Violations

A notifier is a register that you specify as an optional argument to all system timing checks (except \$nochange, which is ignored by Verilog-XL). A timing violation toggles the value of the notifier as shown in the following table:

Time	Notifier Value			
Before Violation	X	0	1	Z
After Violation	1	1	0	Z

Timing check notifiers let you detect timing check violations behaviorally, and take an action that you specify as soon as they occur. For example, you may print an informative error message describing the violation, or you may propagate an x value at the output of the device that reported the violation.

Note: Do not initialize notifier registers because this could affect the behavior of the circuit. For example, initializing a notifier could cause a sequential UDP to go to the X state depending on the order in which the UDP received its inputs at time 0.

The following examples show timing checks with notifier arguments:

```
$setup( data, posedge clk, 10, notify_reg ) ;
$width( posedge clk, 16, notify_reg ) ;
```

The following is a more complex example of how to use notifiers in a behavioral model. A notifier is used to set the D flip-flop output to x when a timing violation occurs in an edge-sensitive user-defined primitive (UDP). This model applies to edge-sensitive UDPs only; for level-sensitive models, you must generate an additional UDP for x propagation.

```
primitive posdff_udp(q, clock, data, preset, clear, notifier);
```

```
    output q; reg q;
    input clock, data, preset, clear, notifier;
    table
    //      clock data  p c notifier state  q
    //-----
        r      0      1 1      ?      : ?      : 0 ;
        r      1      1 1      ?      : ?      : 1 ;

        p      1      ? 1      ?      : 1      : 1 ;
        p      0      1 ?      ?      : 0      : 0 ;

        n      ?      ? ?      ?      : ?      : - ;
        ?      *      ? ?      ?      : ?      : - ;

        ?      ?      0 1      ?      : ?      : 1 ;
        ?      ?      * 1      ?      : 1      : 1 ;
```

Verilog-XL Reference Timing Checks

```
        ?      ?      1 0      ?      : ?      : 0 ;
        ?      ?      1 *      ?      : 0      : 0 ;
        ?      ?      ? ?      *      : ?      : x ;
// At any notifier event, output to x
    endtable
endprimitive
module dff(q, qbar, clock, data, preset, clear);
    output q, qbar;
    input clock, data, preset, clear;
    reg notifier;
    and (enable, preset, clear);
    not (qbar, ffout);
    buf (q, ffout);
    posdff_udp (ffout, clock, data, preset, clear, notifier);
    specify
// Define timing check specparam values
    specparam tSU = 10, tHD = 1, tPW = 25, tWPC = 10, tREC = 5;
// Define module path delay rise and fall specparam
//   min:typ:max values
    specparam tPLHc = 4:6:9 , tPHLc = 5:8:11;
    specparam tPLHpc = 3:5:6 , tPHLpc = 4:7:9;
// Specify module path delays
    (clock *> q,qbar) = (tPLHc, tPHLc);
    (preset,clear *> q,qbar) = (tPLHpc, tPHLpc);
// Setup time : data to clock, only when
//   preset and clear are 1
    $setup(data, posedge clock &&& enable, tSU, notifier);
// Hold time : clock to data, only when preset and clear are 1
    $hold(posedge clock, data &&& enable, tHD, notifier);
// Clock period check
    $period(posedge clock, tPW, notifier);
// Pulse width : preset, clear
    $width(negedge preset, tWPC, 0, notifier);
    $width(negedge clear, tWPC, 0, notifier);
// Recovery time: clear or preset to clock
    $recovery(posedge preset, posedge clock, tREC, notifier);
    $recovery(posedge clear, posedge clock, tREC, notifier);
    endspecify
endmodule
```

Enabling Timing Checks with Conditioned Events

A conditioned event allows a timing check to occur only when a signal with a specific value exists, instead of whenever a clock event occurs to trigger a timing check. A conditioned event is a scalar expression of one of the following forms:

```
<controlled_timing_check_event>
 ::= <timing_check_event_control> <specify_terminal_descriptor>
    < &&& <timing_check_condition>?
```

Verilog-XL Reference

Timing Checks

```
<timing_check_condition>
 ::= <scalar_expression>
    | ~<scalar_expression>
    | <scalar_expression>==<scalar_constant>
    | <scalar_expression>===<scalar_constant>
    | <scalar_expression>!=<scalar_constant>
    | <scalar_expression>!==<scalar_constant>
```

The comparisons used in the condition may be deterministic, as in `===`, `!==`, `~`, or no operation; or non-deterministic as in `==`, or `!=`. When comparisons are deterministic, an `x` value on the conditioning signal will not enable the timing check except (`signal===`1bx`). When comparisons are non-deterministic, an `x` on the conditioning signal enables the timing check.

The following constraints apply when using conditioned events:

- The conditioning signal must be a scalar net; the conditioning signal cannot be a vector or an expression.
- Because conditioning signals cannot be expressions, you may use only one conditioning signal per event.

The following example shows unconditional and conditional timing checks. In the conditional timing checks, a timing check occurs only when a positive edge of `clk` occurs *and* the `clr` signal is high (in the second timing check) or low (in the third and fourth timing checks).

```
// Unconditional timing check
$setup( data, posedge clk, 10 );

// Conditional timing check where clr is high
$setup( data, posedge clk &&& clr, 10 );

// Two conditional timing checks where clr is low
$setup( data, posedge clk &&& (~clr), 10 );
$setup( data, posedge clk &&& (clr==0), 10 );
```

Multiple conditioning signals

You can allow a timing check to occur using values of multiple signals that serve as fanin to a gate whose output is the conditioning signal. For example, to invoke `$setup` on the positive `clk` edge only when `clr` and `set` are high, perform the following steps:

1. Specify the following declaration outside the specify block:

```
and( clr_and_set, clr, set );
```

2. Specify the condition to the timing check using the signal `clr_and_set` as follows:

```
$setup( data, posedge clk &&& clr_and_set, 10 );
```

Using the Timing Check System Tasks

This section describes the following timing check system tasks.

- [“\\$hold”](#) on page 295
- [“\\$nochange”](#) on page 297
- [“\\$period”](#) on page 298
- [“\\$recovery”](#) on page 299
- [“\\$recrem”](#) on page 301
- [“\\$setup”](#) on page 305
- [“\\$setuphold”](#) on page 307
- [“\\$skew”](#) on page 310
- [“\\$width”](#) on page 311

Event signals must be module inputs or module inouts, and they must be scalar or expanded vector nets. When event signals are vectors, Verilog-XL generates a timing check for each bit of the vector, although each bit's timing check has the same limits

The `$setuphold` and `$recovery` system tasks allow negative time specifications to generate delayed signals as inputs to other devices. For more information about negative timing checks, see [“Using Negative Timing Check Limits in \\$setuphold and \\$recrem”](#) on page 313.

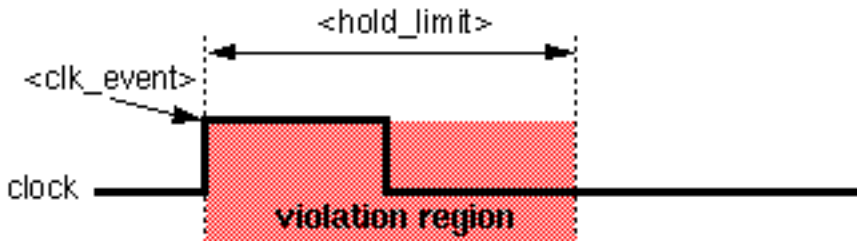
\$hold

The `$hold` system task determines whether a data signal remains stable for a minimum specified time after a transition in an enabling signal, such as a clock signal that latches data

Verilog-XL Reference

Timing Checks

in a memory. The following figure shows the violation region specified by the `$hold` system task:



The `$hold` system task has the following syntax:

```
$hold(<clk_event>, <data_event>, <hold_limit> {, <notifier>});
```

The `$hold` system task arguments are as follows:

<code><clk_event></code>	Module input or inout transition at a control signal that establishes the reference time
<code><data_event></code>	Module input or inout transition at a data signal that initiates a timing check against the value in <code><hold_limit></code>
<code><hold_limit></code>	Positive constant expression or <code>specparam</code> that specifies the interval between the clock and data events (that is, <i>after</i> clock transition). Any change to the data signal within this interval results in a timing violation. If <code><hold_limit></code> is 0, a timing check does not occur.
<code><notifier></code> (optional)	Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See “Using Notifiers for Timing Violations” on page 292 for details.

Note: The `$hold` timing check reports a violation when the `<clk_event>` and `<data_event>` occur simultaneously.

The following example illustrates how to use the `$hold` system task:

```
specify
  specparam hold_param=11;
  $hold( posedge clk, data, hold_param );
  $hold( posedge clk, data, hold_param, flag ) ;
endspecify
```


Verilog-XL Reference

Timing Checks

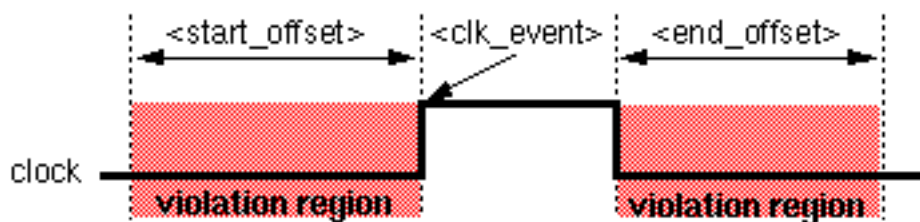
In this example, `$hold` reports a violation if the time that elapses from `posedge clk` to a change in `data` is smaller than `hold_param` (which is 11). The optional register, `flag`, toggles to report a violation in the second `$hold` system task.

`$nochange`

The `$nochange` system task is supported by Veritime. The `$nochange` system task reports a timing violation if an event occurs during the specified time of the control signal.

Note: Verilog-XL does not support the `$nochange` system task, but will compile source descriptions containing calls to `$nochange` inside `specify` blocks.

The following figure shows the violation regions for the `$nochange` system task:



The `$nochange` system task has the following syntax:

```
$nochange(<clk_event>, <data_event>, <start_offset>, <end_offset>);
```

The `$nochange` system task arguments are as follows:

<code><clk_event></code>	Module input or inout transition at a control signal that establishes the start time
<code><data_event></code>	Module input or inout transition at a control signal that establishes the end time
<code><start_offset></code>	Any constant expression or <code>specparam</code> that defines the violation region
<code><end_offset></code>	Any constant expression or <code>specparam</code> that defines the violation region. A positive value in <code><start_offset></code> starts the region earlier; a negative value starts it later. A positive value in <code><end_offset></code> ends the region later; a negative value ends it earlier.

Verilog-XL Reference

Timing Checks

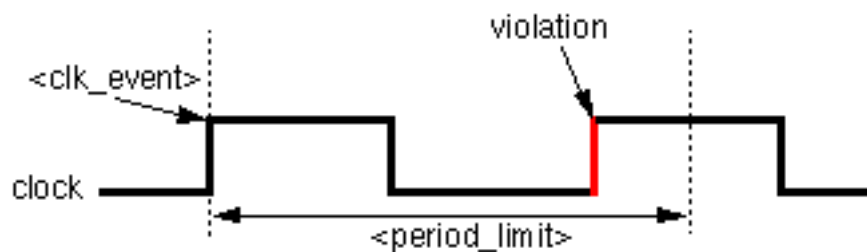
The following example shows how to use the `$nochange` system task to report a timing violation if the `<data_event>` occurs while `clk` is high:

```
$nochange(posedge clk, data, startoff, endoff);
```

Note: You can specify the edge event with `posedge` or `negedge`, but you cannot use edge-control specifiers, which are described in “[Using Edge-Control Specifiers](#)” on page 291. For more information about using the `$nochange` system task for timing analysis, see the Timing Checks chapter of the Veritime User Guide.

\$period

The `$period` system task issues a violation when a clock event of the same edge occurs within a specified time. The following figure shows how a violation occurs with the `$period` system task:



The `$period` system task has the following format:

```
$period(<clk_event>, <period_limit> {, <notifier>});
```

The `$period` system task arguments are as follows:

<code><clk_event></code>	Edge-triggered event
<code><period_limit></code>	Positive constant expression or <code>specparam</code> that specifies the minimum period for complete signal cycle
<code><notifier></code> (optional)	Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See “ Using Notifiers for Timing Violations ” on page 292 for details.

Because of the way Verilog-XL derives the `<data_event>` for `$period`, you must pass an edge-triggered event as the `<clk_event>`. A compilation error occurs if the `<clk_event>` is not an edge specification.

Verilog-XL Reference

Timing Checks

If you use an edge specifier, the edge must be either all positive (01, 0X, X1) or all negative (10,1X, X0).

The following example shows how to use the `$period` system task:

```
specify
  specparam period_param=13;
  $period( negedge clk, period_param ) ;
  $period( negedge clk, period_param, flag ) ;
endspecify
```

In this example, the `<data_event>` for both `$period` specifications is `negedge clk`. The `$period` system task reports a violation if the time between a `negedge clk` and the next `negedge clk` is less than `period_param` (which is 13). The optional register, `flag`, toggles to report a violation in the second `$period` system task.

\$recovery

The `$recovery` system task specifies a time constraint between an asynchronous control signal and a clock signal (for example, between the clearbar signal and the clock signal for a flip-flop). A violation occurs when either signal changes within the specified time constraint.

The `$recovery` system task has the following two syntax formats:

```
$recovery(<reference_event>, <data_event>, <recovery_limit>
         {, <notifier>});
```

or

```
$recovery(<reference_event>, <data_event>,
         <removal_limit>, <recovery_limit>,
         {<notifier>}, {<tstamp_cond>}, {<tcheck_cond>},
         {<delayed_clk>}, {<delayed_data>});
```

The `$recovery` system task arguments are as follows:

`<reference_event>` Asynchronous control signal, which normally has an edge identifier associated with it to indicate the transition that corresponds to the release from the active state

`<data_event>` A clock (flip-flops) or gate (latches) signal, which normally has an edge identifier to indicate the active edge of the clock or the closing edge of the gate.

`<removal_limit>` Minimum interval between the active edge of the clock event and the release of the asynchronous control signal. Any change to a signal within this interval results in a timing violation.

Verilog-XL Reference

Timing Checks

- <recovery_limit>* Positive minimum interval between the release of the asynchronous control signal and the next active edge of the clock or gate event. The simulator uses the recovery limit for deterministic comparisons and does not admit *x* values.
- <notifier>* (optional) Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See [“Using Notifiers for Timing Violations”](#) on page 292 for details.
- <tstamp_cond>* (optional) Places a condition on the *<reference_event>* and the *<data_event>*, if both *<removal_limit>* and *<recovery_limit>* are positive values. Places a condition only on the *<reference_event>* if the *<removal_limit>* is negative. Places a condition only on the *<data_event>* if the *<recovery_limit>* is negative.
- <tcheck_cond>* (optional) Places a condition on the *<reference_event>* and the *<data_event>* if both *<removal_limit>* and *<recovery_limit>* are positive values. Places a condition only on the *<data_event>* if the *<removal_limit>* is negative. Places a condition only on the *<reference_event>* if the *<recovery_limit>* is negative.
- <delayed_clk>* (optional) Delayed signal value for *<reference_event>* when one of the limits is negative.
- <delayed_data>* (optional) Delayed signal value for *<data_event>* when one of the limits is negative.

Note: `$recovery` records the new reference event time before performing the timing check, so if a data event and a reference event occur at the same simulation time, a violation occurs.

The following example shows the `$recovery` system task:

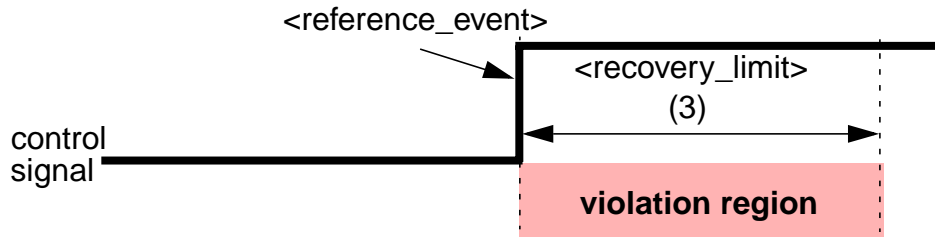
```
specify
  specparam recovery_param=3;
  $recovery( posedge set, posedge clk, recovery_param );
  $recovery( posedge set, posedge clk, recovery_param, flag );
endspecify
```

Verilog-XL Reference

Timing Checks

In this example, `$recovery` specifies a positive value (3) for `<recovery_param>`. The second `$recovery` specification shows the optional notifier, `flag`, which toggles to report a violation.

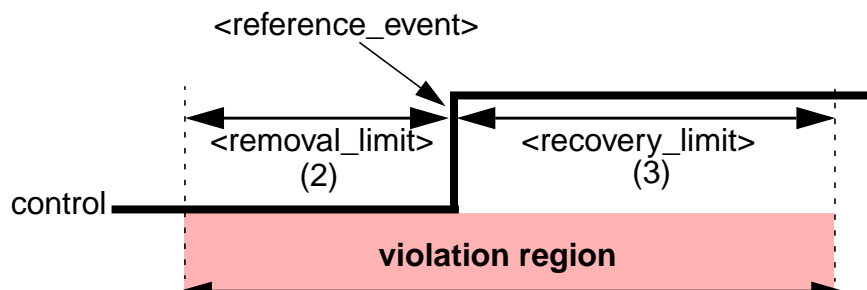
The following figure shows the violation region of 3 time units created when `<recovery_limit>` is specified alone with a value of 3:



`$recrem`

The `$recrem` system task combines the functionality of `$removal` and `$recovery` into one system task. It defines a time period relative to an asynchronous control signal during which another control signal (often a clock) must be stable. A violation occurs when either signal changes during this time constraint.

The following figure shows the violation region when you specify two positive time limits with the `$recrem` system task:



The `$recrem` system task has the following syntax:

```
$recrem(<reference_event>, <data_event>,
        <recovery_limit>, <removal_limit>,
        {<notifier>}, {<tstamp_cond>}, {<tcheck_cond>},
        {<delayed_clk>}, {<delayed_data>});
```

Note: You must indicate absent optional parameters as null parameters by using commas. Do not add one or more commas after the `$recrem` system task's last argument because you can truncate the syntax after any argument.

The `$recrem` system task arguments are as follows:

Verilog-XL Reference

Timing Checks

- <reference_event>* Asynchronous control signal, which normally has an edge identifier to indicate which transition corresponds to the release from the active state.
- <data_event>* Data signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state.
- <recovery_limit>* Minimum interval between the release of the asynchronous control signal and the active edge of the clock event. Any change to a signal within this interval results in a timing violation.
- <removal_limit>* Minimum interval between the active edge of the clock event and the release of the asynchronous control signal. Any change to a signal within this interval results in a timing violation.
- <notifier>* (optional) Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See [“Using Notifiers for Timing Violations”](#) on page 292 for details.
- <tstamp_cond>* (optional) Places a condition on the *<reference_event>* and the *<data_event>*, if both *<removal_limit>* and *<recovery_limit>* are positive values. Places a condition only on the *<reference_event>* if the *<removal_limit>* is negative. Places a condition only on the *<data_event>* if the *<recovery_limit>* is negative.
- <tcheck_cond>* (optional) Places a condition on the *<reference_event>* and the *<data_event>* if both *<removal_limit>* and *<recovery_limit>* are positive values. Places a condition only on the *<data_event>* if the *<removal_limit>* is negative. Places a condition only on the *<reference_event>* if the *<recovery_limit>* is negative.
- <delayed_clk>* (optional) Delayed signal value for *<reference_event>* when one of the limits is negative. See [“Using Negative Timing Check Limits in \\$setuphold and \\$recrem”](#) on page 313 for more information.

Verilog-XL Reference

Timing Checks

`<delayed_data>` (optional)

Delayed signal value for `<data_event>` when one of the limits is negative. See [“Using Negative Timing Check Limits in \\$setuphold and \\$recrem”](#) on page 313 for more information.

The following example shows how the `$recrem` system task includes the functions of a `$removal` system task with a positive `<removal_limit>`, and the functions of a `$recovery` system task with a positive `<recovery_limit>`. The first system task in the example is equivalent to both of the subsequent system tasks if `<recovery_limit>` and `<removal_limit>` are positive values.

```
$recrem(posedge ctrl, clk, <recovery_limit>, <removal_limit>);
$removal( posedge ctrl, clk, <removal_limit> );
$recovery( posedge ctrl, clk, <recovery_limit> );
```

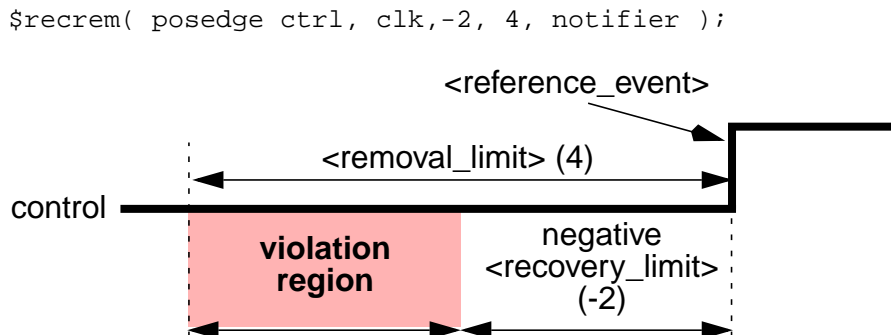
The next example illustrates `$recrem` with positive specifications:

```
specify
  specparam tSU=16, tHLD=17;
  $recrem( posedge ctrl, clk, tSU, tHLD );
  $recrem( posedge ctrl, clk, tSU, tHLD, flag );
endspecify
```

In this example, `$recrem` reports a violation if the interval between `posedge ctrl` and `clk` is less than the value of `tSU` (which is 16), enacting its `$removal` component. It also reports a violation if the interval between `posedge ctrl` and `clk` is less than the value of `tHLD` (which is 17), enacting its `$recovery` component. The second statement in the example shows an optional notifier, `flag`, which toggles to report a violation.

You can specify negative times for either the `<recovery_limit>` or `<removal_limit>` arguments. The sum of the two arguments must be greater than 0.

A negative `<recovery_limit>` value specifies a time period preceding a change in the control signal as shown in the following example:

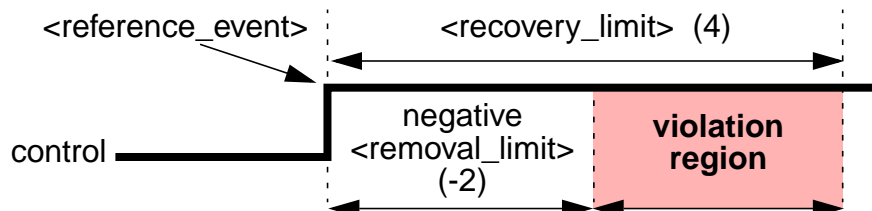


A negative `<removal_limit>` value specifies a time period following a change in the control event signal as shown in the following example:

Verilog-XL Reference

Timing Checks

```
$recrem( posedge ctrl, clk, 4, -2, notifier);
```



Note: You must specify the `+neg_tchk` plus option on the command line to make Verilog-XL accept negative timing check arguments. If you do not specify the `+neg_tchk` plus option, negative limits are set to 0 in the description or annotation, and a warning is issued.

A violation of `$recrem` in signals passing from a vector port to a vector port generates an identical message for each bit that experiences a violation.

Note: You cannot condition a `$recrem` timing check with both the `&&&` conditioned event symbol and the inclusion of the `<tstamp_cond>` or `<tcheck_cond>` in the syntax. If you attempt to use both methods, only the parameters in the `<tstamp_cond>` and `<tcheck_cond>` positions in the syntax are effective, and the attempt generates a warning similar to that shown in the following example.

```
Warning! Conditions for timecheck input specified both on
argument and as explicit condition, argument condition ignored [Verilog-SPAMCN]
"/net/machine/home/willy/1.8/code/cond8.18.525",
31: $recrem(ckin &&& cond2in, datin, su, hl, flag, , condlin);
```

\$removal

The `$removal` system task specifies a time constraint between an asynchronous control signal and a clock signal (for example, between the clearbar signal and the clock signal for a flip-flop). A violation occurs when either signal changes within the specified time constraint.

The `$removal` system task has the following syntax format:

```
$removal(<reference_event>, <data_event>, <removal_limit> {, <notifier>});
```

The `$removal` system task arguments are as follows:

`<reference_event>` Asynchronous control signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state

`<data_event>` A clock (flip-flops) or gate (latches) signal, which normally has an edge identifier to indicate the active edge of the clock or the closing edge of the gate

Verilog-XL Reference

Timing Checks

- `<removal_limit>` Positive minimum interval between the release of the asynchronous control signal and the next active edge of the clock or gate event. The simulator uses the removal limit for deterministic comparisons and does not admit x values.
- `<notifier>` (optional) Register that informs Verilog-XL when a timing violation occurs. You can use notifiers to define responses to timing violations. See [“Using Notifiers for Timing Violations”](#) on page 292 for details.

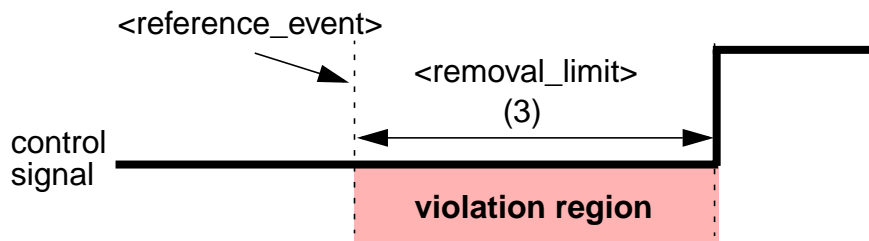
Note: If the `<data_event>` and `<reference_event>` occur simultaneously, `$removal` performs the timing check before it records the new `<reference_event>` time. Therefore, no violation is reported.

The following example shows the `$removal` system task:

```
specify
  specparam recovery_param=3;
  $removal( posedge set, posedge clk, removal_param );
  $removal( posedge set, posedge clk, removal_param, flag );
endspecify
```

In this example, `$removal` specifies a positive value (3) for `<removal_param>`. The second `$removal` specification shows the optional notifier, `flag`, which toggles to report a violation.

The following figure shows the violation region of 3 time units when `<removal_limit>` is specified alone with a value of 3:



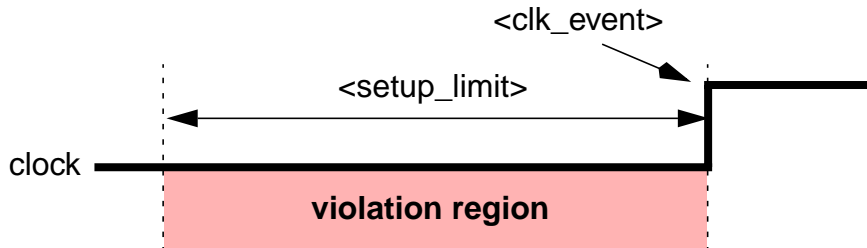
`$setup`

The `$setup` system task determines whether a data signal remains stable before a transition in an enabling signal, such as a clock signal that latches data in memory. A violation occurs

Verilog-XL Reference

Timing Checks

when a change in the signal occurs within a specified time limit before the clock event. The following figure shows the violation region specified by the `$setup` system task.



The `$setup` system task has the following format:

```
$setup(<data_event>, <clk_event>, <setup_limit> {, <notifier>});
```

Transitions in the data signal that occur at the beginning or end of the period that the task evaluates do not generate a timing violation.

The `$setup` system task arguments are as follows:

`<data_event>` Module input or inout transition at a data signal that initiates a timing check against the value in `<setup_limit>`.

`<clk_event>` Module input or inout transition at a control signal that establishes the reference time.

`<setup_limit>` Positive constant expression or `specparam` that specifies the minimum interval between the data and the clock event (that is, *before* clock transition). Any change to the data signal within these intervals results in a timing violation.

`<notifier>` (optional) Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See [“Using Notifiers for Timing Violations”](#) on page 292 for details.

Note: If the `<clk_event>` and `<data_event>` occur simultaneously, `$setup` performs the timing check before it records the new `<data_event>` value. Therefore, no violation is reported.

In the following example, the `$setup` system task reports a violation if the interval from `<data_event>` to `<clk_event>` is less than `<setup_limit>` (10). The second specification of the `$setup` system task uses an option notifier, `flag`, to indicate a timing violation.

```
specify
  specparam setup_param=10;
  $setup( data, posedge clock, setup_param ) ;
```

Verilog-XL Reference

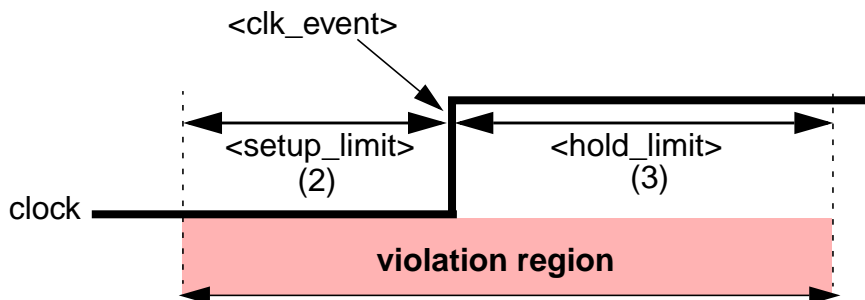
Timing Checks

```
$setup( data, posedge clock, setup_param, flag ) ;  
endspecify
```

\$setuphold

The `$setuphold` system task combines the functionality of `$setup` and `$hold` into one system task. It also offers additional functionality in the form of negative time specifications. A violation occurs when a change in one of the signals causes a violation of this constraint.

The following figure shows the violation region when you specify two positive time limits with the `$setuphold` system task.:



The `$setuphold` system task has the following format:

```
$setuphold(<clk_event>, <data_event>, <setup_limit>, <hold_limit>,  
          {<notifier>}, {<tstamp_cond>}, {<tcheck_cond>},  
          {<delayed_clk>}, {<delayed_data>});
```

Note: Absent optional parameters must be indicated as null parameters by using commas. Do not add one or more commas after the `$setuphold` system task's last argument because the syntax can be truncated after any argument.

The `$setuphold` system task arguments are as follows:

- | | |
|----------------------------------|--|
| <code><clk_event></code> | Clock (flip-flops) or gate (latches) signal, which normally has an edge identifier to indicate the active edge of the clock or the closing edge of the gate. |
| <code><data_event></code> | Data signal, which normally has an edge identifier associated with it to indicate which transition corresponds to the release from the active state. |
| <code><setup_limit></code> | Minimum interval between the next active edge of the clock or gate event and the release of the asynchronous control signal. |

Verilog-XL Reference

Timing Checks

Any change to the clock signal within these intervals results in a timing violation.

`<hold_limit>` Minimum interval between the release of the asynchronous control signal and the next active edge of the clock or gate event.

`<notifier>` (optional) Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See [“Using Notifiers for Timing Violations”](#) on page 292 for details.

`<tstamp_cond>` (optional) Places a condition on the `<control_event>` and the `<clk_event>`, if both `<setup_limit>` and `<hold_limit>` are positive values. Places a condition only on the `<control_event>` if the `<setup_limit>` is negative. Places a condition only on the `<clk_event>` if the `<hold_limit>` is negative.

`<tcheck_cond>` (optional) Places a condition on the `<control_event>` and the `<clk_event>` if both `<setup_limit>` and `<hold_limit>` are positive values. Places a condition only on the `<clk_event>` if the `<setup_limit>` is negative. Places a condition only on the `<control_event>` if the `<hold_limit>` is negative.

`<delayed_clk>` (optional) Delayed signal value for `<clk_event>` when one of the limits is negative. See [“Using Negative Timing Check Limits in \\$setuphold and \\$recrem”](#) on page 313 for more information.

`<delayed_data>` (optional) Delayed signal value for `<data_event>` when one of the limits is negative. See [“Using Negative Timing Check Limits in \\$setuphold and \\$recrem”](#) on page 313 for more information.

The following example shows how the `$setuphold` system task includes the functions of a `$setup` system task with a positive `<setup_limit>`, and the functions of a `$hold` system task with a positive `<hold_limit>`. The first system task in the example is equivalent to both of the subsequent system tasks if `<setup_limit>` and `<hold_limit>` are positive values.

```
$setuphold(posedge clk, data, <setup_limit>, <hold_limit>);  
$setup( data, posedge clk, <setup_limit> );  
$hold( posedge clk, data, <hold_limit> );
```

Verilog-XL Reference

Timing Checks

The next example illustrates `$setuphold` with positive specifications:

```
specify
  specparam tSU=16, tHLD=17;
  $setuphold( posedge clk, data, tSU, tHLD ) ;
  $setuphold( posedge clk, data, tSU, tHLD, flag ) ;
endspecify
```

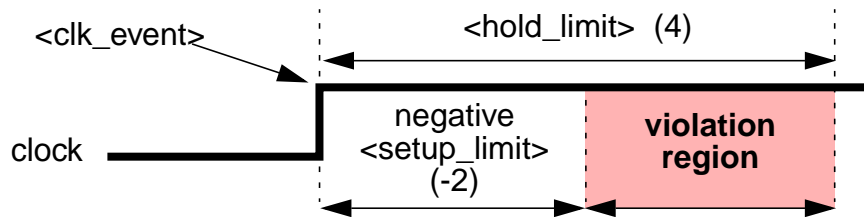
In this example, `$setuphold` reports a violation if the interval from `<data_event>` to `<clk_event>` is less than the value of `tSU` (which is 16), enacting its `$setup` component. It also reports a violation if the interval from `<clk_event>` to `<data_event>` is less than the value of `tHLD` (which is 17), enacting its `$hold` component.

The second statement in the example shows an optional notifier, `flag`, which toggles to report a violation.

You can specify negative times for either the `<setup_limit>` or `<hold_limit>` arguments. The sum of the two arguments must be greater than 0.

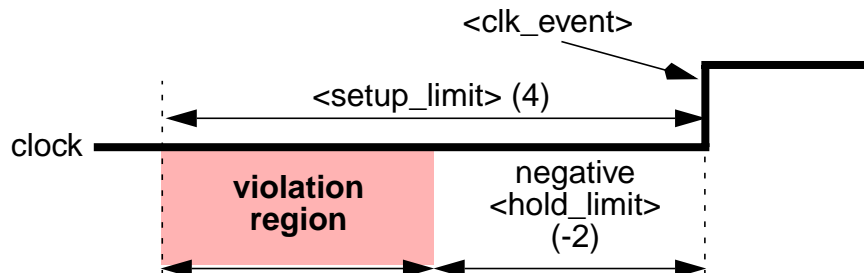
A negative `<setup_limit>` value specifies a period following a change in the clock signal as shown in the following example:

```
$setuphold( posedge clk, data, -2, 4, notifier );
```



A negative `<hold_limit>` value specifies a period preceding a change in the clock event signal as shown in the following example:

```
$setuphold( posedge clk, data, 4, -2, notifier );
```



Note: You must specify the `+neg_tchk` plus option on the command line to make Verilog-XL accept negative timing check arguments. If you do not specify the `+neg_tchk` plus option, negative limits are set to 0 in the description or annotation, and a warning is issued.

Verilog-XL Reference

Timing Checks

The `$skew` system task arguments are as follows:

<code><clk_1></code>	Module input or inout transition at a control signal that establishes the reference time
<code><clk_2></code>	Module input or inout transition at a data signal that initiates a timing check against the value in <code><skew_limit></code>
<code><skew_limit></code>	Positive constant expression or <code>specparam</code> that specifies the delay allowed between the two signals
<code><notifier></code> (optional)	Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See “Using Notifiers for Timing Violations” on page 292 for details.

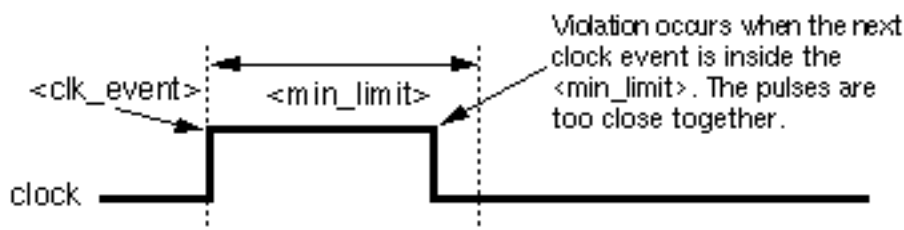
Note: The `$skew` system task records the new time of `<clk_1>` before it performs the timing check. If the `<clk_1>` and `<clk_2>` events occur at the same time, `$skew` does not report a timing violation.

The following example shows how to use the `$skew` system task to report a violation if the interval from `<clk_1>` to `<clk_2>` exceeds `<skew_limit>` (which is 14). The second `$skew` statement uses the notifier, `flag`, to report a violation.

```
specify
  specparam <skew_limit>=14;
  $skew(posedge clk_1, negedge clk_2, <skew_limit>);
  $skew(posedge clk_1, negedge clk_2, <skew_limit>, flag);
endspecify
```

\$width

The `$width` system task specifies the duration of signal levels from one clock edge to the opposite clock edge. A violation occurs when signals are too close together. If you use edge specifiers, all edges must be of the same direction. The following figure shows how a violation occurs with the `$width` system task.



Verilog-XL Reference

Timing Checks

The syntax for the `$width` system task is as follows:

```
$width(<edge_clk>, <min_limit> {, <threshold>{, <notifier>}});
```

The `$width` system task arguments are as follows:

<code><edge_clk></code>	Edge-triggered event. A compilation error occurs if the <code><edge_clk></code> is not an edge specification.
<code><min_limit></code>	Positive constant expression or <code>specparam</code> that specifies the maximum time for the positive or negative phase of each cycle
<code><threshold></code> (optional)	Largest ignored pulse width, used for timing analysis by Veritime. Verilog-XL ignores <code><threshold></code> , but compiles system calls to <code>\$width</code> that contain this argument.
<code><notifier></code> (optional)	Register that changes value when a timing violation occurs. You can use notifiers to define responses to timing violations. See “Using Notifiers for Timing Violations” on page 292 for details.

The following example shows how to use the `$width` system task to report a violation if the interval from `<edge_clk>` (`negedge clr`) to the implicit `<data_event>` (`posedge clr`) is less than `<min_limit>` (which is 12). Note that the `<data_event>` and the `<clk_event>` will never occur simultaneously because they are triggered by opposite transitions. The optional notifier, `flag`, in the second `$width` statement toggles to report a violation.

```
specify
  specparam width_param=12;
  $width( negedge clr, width_param );
  $width( negedge clr, width_param, 0, flag );
endspecify
```

Note: Verilog-XL does not accept null arguments for `$width`. Therefore, if you pass a `<notifier>` to `$width`, you must also supply the `<threshold>` argument. However, you do not have to specify the `<threshold>` and `<notifier>` arguments when invoking `$width` in Verilog-XL. The following example shows legal and illegal calls:

```
$width( negedge clr, lim ); // legal
$width( negedge clr, lim, thresh, notif ); // legal
$width( negedge clr, lim, 0, notif ); // legal
$width( negedge clr, lim, , notif ); // illegal call
$width( negedge clr, lim, notif ); // illegal call
```


Using Negative Timing Check Limits in `$setuphold` and `$recrem`

If you are using a negative limit in a `$setuphold` or `$recrem` timing check, you need to use the `+neg_tchk` option when invoking the simulator. This will ensure that the negative limit values are used. If you ignore this option, all negative values are converted to 0.

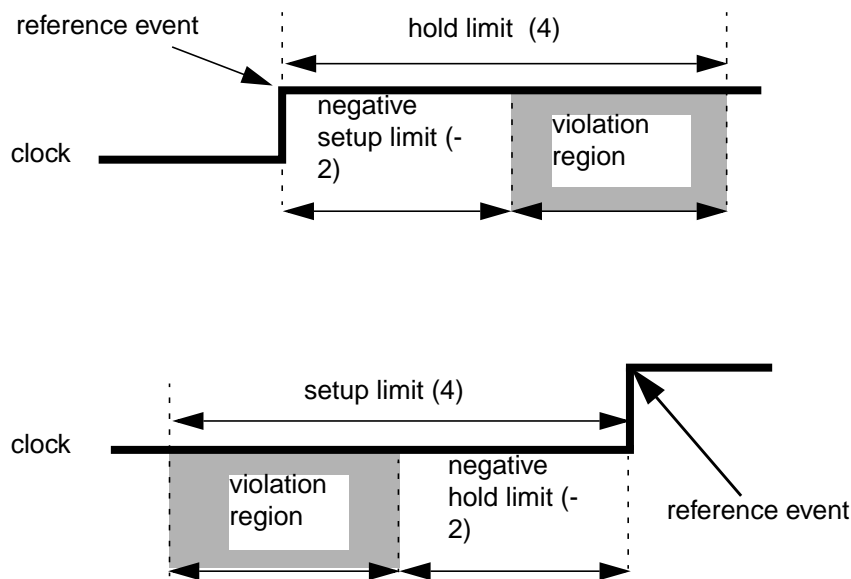
Using negative limits in `$setuphold` or `$recrem` timing checks can affect the evaluation of timing checks. For each timing check with a negative limit, the reference or data event may be delayed, thereby delaying the execution of the timing check. When either the reference or data signal of a check is delayed, the limits of the check are appropriately modified to verify the same constraint using the delayed signals. See [“Effects of Delayed Signals on Timing Checks”](#) on page 314 for more information on delayed signals and on how timing check limits are modified.

The delayed version of a signal generated by a `$setuphold` or `$recrem` timing check with a negative limit does not only apply to that specific `$setuphold` or `$recrem` check. Once a delayed version of a signal is calculated, it is also used when evaluating other checks, such as `$setup`, `$hold`, `$recovery`, `$width`, and `$period`. All timing checks are considered together. For example, if multiple timing checks are driven with the signal `CLK`, then one delay is calculated for `CLK`, and each timing check is evaluated using this single delayed version of `CLK`. See [“Calculation of Delayed Signals and Limit Modification”](#) on page 316 for details on how delay values are calculated and on how limits are adjusted.

In some cases, you may want to drive your functional model using the delayed version of signals. To do this, you can explicitly define the delayed versions of signals in the `$setuphold` and `$recrem` timing checks using the `delayed_reference` and `delayed_data` arguments. See [“Explicitly Defining Delayed Signals”](#) on page 318 for details.

Effects of Delayed Signals on Timing Checks

When a negative limit value is specified in a `$setuphold` or `$recrem` timing check, the violation region is offset from the reference signal. The following figures illustrate the violation region that is offset from the reference signal for the `$setuphold` task.



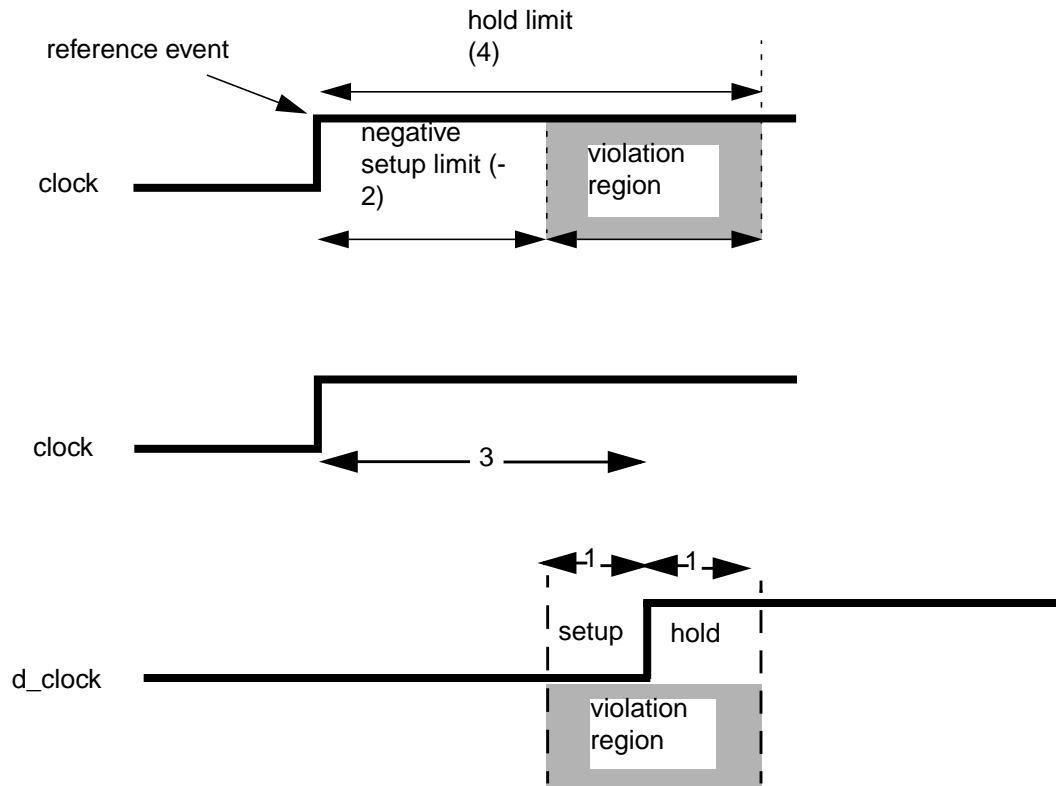
Because the violation region no longer extends from the reference signal, the constraints cannot be verified when the check events occur in the same way that they can be without negative limits. In the case of `$hold`, only the stamp events that occur after the value of the time offset (2 in the case of our example) should be considered. In the case of `$setup`, the check should be triggered before the actual check event is to occur, which cannot be predicted.

To solve this dilemma, the reference or data signals are delayed such that the violation region once again encloses the reference signal. The check can then be evaluated as if only positive limits were encountered. To accomplish this, signals must be delayed, and limits must be appropriately modified to verify the same constraint, as initially specified with negative limits.

For example, the following figure shows the violation region for a `$setuphold` timing check with a negative setup limit of -2 and a hold limit of 4. To verify the negative setup constraint

Verilog-XL Reference Timing Checks

shown in this example, the “equivalent constraint”, shown at the bottom of the figure, is verified.

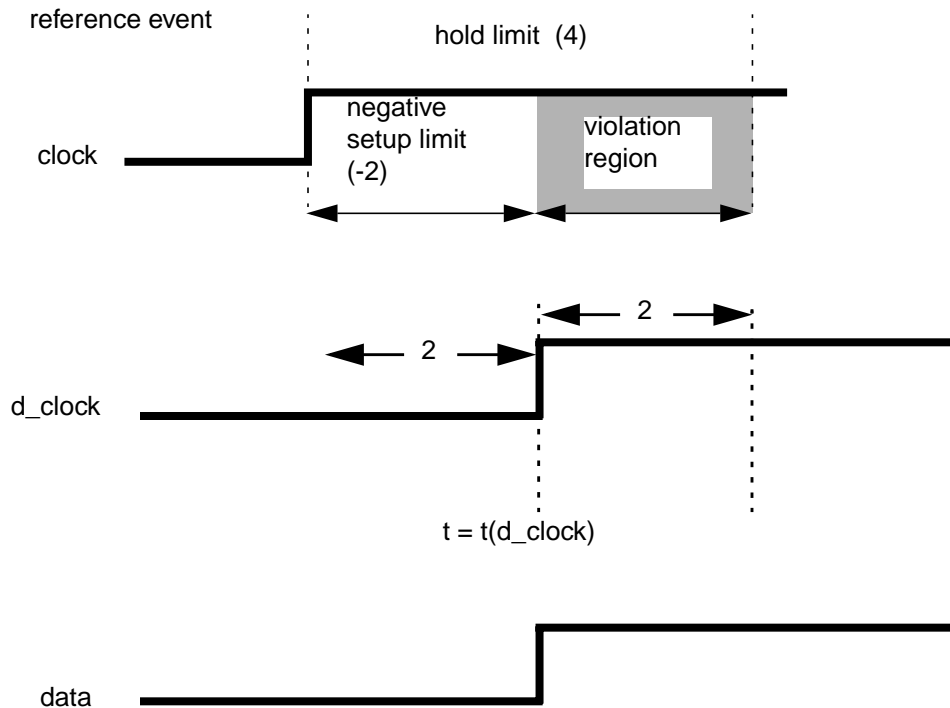


In this example, clock is delayed 3 time units, producing `d_clock`, which is used to verify the same constraint with a setup limit of 1 and a hold limit of 1. This constraint is equivalent to the original one.

Verilog-XL Reference

Timing Checks

Notice that `d_clock` was produced by delaying `clock` by 3 time units, and not by 2 time units, the value of the negative setup limit. The reason for this is best illustrated by the following diagram:



The above modified constraint implies that a data change at $t(d_clock)$ is not a violation. This implies that a change on the data signal at $t(d_clock)$ should be clocked in by a storage element in the model. However, if the data signal can change at the same time as `d_clock`, then it is not certain which value will be clocked in. Hence, `d_clock` has been delayed by an additional time unit. Verilog-XL uses the smallest simulation precision to determine this additional increment.

See [“Calculation of Delayed Signals and Limit Modification”](#) on page 316 for more details about how signals are delayed and limits adjusted.

Calculation of Delayed Signals and Limit Modification

This section contains more details about how signals are delayed and limits adjusted. This information is useful to the model developer and to someone designing a delay calculation algorithm that may compute negative timing check limits.

All timing checks are considered together. When a signal is delayed for a specific check, and that signal drives another timing check, the delayed version of the signal is used to trigger the other check, and the other check's limits are appropriately modified, even when the other

Verilog-XL Reference

Timing Checks

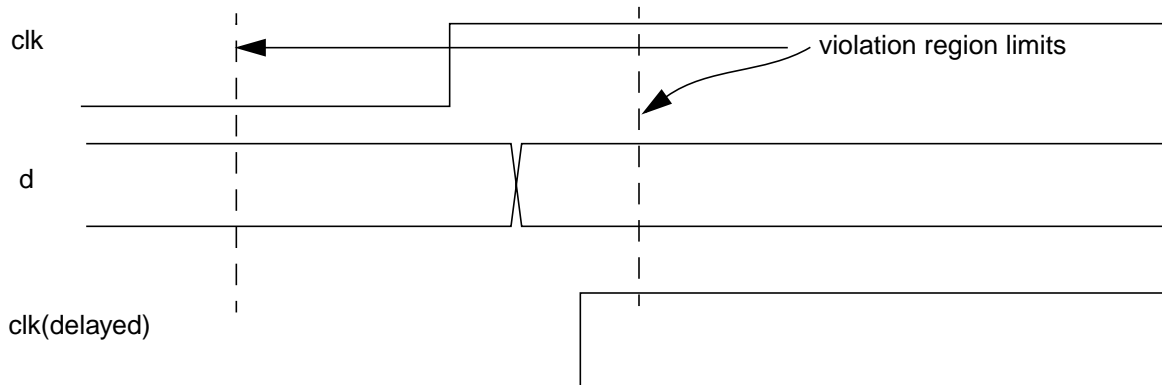
check does not have negative limits. For example, if multiple timing checks are driven with the signal `CLK`, then one delay is calculated for `CLK`, and each timing check is evaluated using this single delayed version of `CLK`.

The reason for not considering all checks independently is illustrated by the following example.

Consider the following set of timing checks:

```
$setuphold(posedge clk, d, 12.2, 4.3, ...);  
$recrem(posedge clr, posedge clk, 5.1, -2.1, ...);
```

As a result of the second timing check, the `clk` signal is delayed by 2.1 time units. If this delayed signal was not used to perform the first timing check, the following figure shows a set of inputs that could cause inaccurate results.



If the original, undelayed signal in the first timing check is used, the violation occurs at the edge on `d`. At the time of this violation, any notifier associated with the timing check will toggle, and the output of the device will be set to `x`. However, the device will not detect the edge on the delayed `clk` until after this has happened. This edge on the delayed `clk` will clock the device, and the output will incorrectly go to a known value, even though a violation has occurred.

If the delayed signal in the first timing check is used, Verilog-XL ensures that any violation and any functional evaluation of the device occur at the same time. Therefore, the functional evaluation cannot override the violation.

This requirement applies to any other timing check in the module, (for example `$setup` or `$width` checks). Therefore, when negative timing checks are being used, *any* timing check in the module being affected will use delayed signals. However, this implies that timing checks that have multiple signals need to have their limits adjusted accordingly. This adjustment is performed at the same time as the delays on the signals themselves are calculated.

Verilog-XL Reference

Timing Checks

The adjustment of limit values is performed such that a limit will never go to 0. The reason for this is to prevent a race condition when an explicit delayed signal drives the functional model. If, for example, a delayed clock signal were to change at the same time as a data signal, and the delayed clock feeds the functional model, the new value should be clocked. Hence, limits are adjusted by adding a delta value to ensure this situation never occurs.

The process of calculating the delay values and adjusting limits can be summarized with the following pseudo-code description:

```
count = 1;
while (any timing check limit is < 0) {
    if (count > number of checks)
        convergence error - delays cannot be calculated given current limit
values (see #1, below);
    count = count + 1;

    for each setuphold/2 limit recovery check:
    if (((setup limit is == 0) and has been modified) or (setup limit < 0))
        reference delay = 0 - setup limit;
        hold limit -= ((reference delay) + delta); (see #2, below)
        setup limit = delta;

        for every other timing check with the same reference signal:
            setup limit += (reference delay + delta);
            hold limit -= (reference delay + delta);

        for every other timing check with this reference signal as the data signal:
            hold limit += (reference delay + delta);
            setup limit -= (reference delay + delta);
            total delay for reference signal += (reference delay + delta);
    else if (((hold limit == 0) and has been modified) or (hold limit < 0))
        data delay = 0 - hold limit;
        setup limit -= ((data delay) + delta);
        hold limit = delta;

        for every other timing check with the same data signal:
            hold limit += (data delay + delta);
            setup limit -= (data delay + delta);

        for every other timing check with this data signal as the reference signal:
            setup limit += (data delay + delta);
            hold limit -= (data delay + delta);
            total delay for data signal += (data delay + delta);
```

#1: The whole process is repeated after the next setup limit of smallest magnitude is set to zero. If there are no negative setup limits left, then the next hold limit is set to zero.

#2: delta is the smallest simulation precision in the design.

Explicitly Defining Delayed Signals

The delayed versions of signals can be explicitly defined in the `$setuphold` and `$recrem` timing checks using the `delayed_reference` and `delayed_data` arguments, which are the delayed version of the reference and data signals, respectively. You may want to explicitly define the delayed signals in order to drive the functional model using the delayed version of these signals. The following example illustrates this. The negative timing check value causes

Verilog-XL Reference

Timing Checks

Verilog-XL to generate a delayed signal to use as input to the functional part of the UDP circuit. This ensures that the correct value for the data signal is present at the UDP input when the clock edge occurs.

```
module dff (q, d, clk);
    output q;
    input d, clk;
    dff_prim p1(q, dd, dclk, notfy);
    specify
        $setuphold(posedge clk, d, 12, -5, notfy, , , dclk, dd);
    endspecify
endmodule
```

If the delayed signals `dclk` and `dd`, were not explicitly defined, delayed versions of `clk` and/or `d` would still be used to evaluate the timing check. However, the functional model would utilize the undelayed signals.

The following is a slightly more complex example, which uses several delayed signals.

```
module device(q, d, clk, set, clr);
    output q;
    input d, clk, set, clr;
    prim p1(q, dd, dclk, dset, dclr);
    specify
        $setuphold(posedge clk, d, 12, -3, , , , dclk, dd);
        $recrem(posedge clr, posedge clk, 10, -7, , , , dclr, dclk);
        $recrem(posedge set, posedge clk, 13, -4, , , , dset, dclk);
    endspecify
endmodule
```

Verilog-XL iteratively analyzes the entire set of timing checks to generate a correct set of delay values for a device. The generated delay values for the model in the previous example are as follows:

```
clk  7
set  0
clr  0
d    10
```

Verilog-XL takes the limits from the entire set of timing checks into account when choosing a limit. You must list the delayed signal in *each* timing check that makes a contribution to the final delay generated for each signal.

Non-Convergence in Timing Checks

In some cases, when negative values are specified in `$setuphold` or `$recrem` timing checks (two limit), or in SDF `SETUPHOLD` or `RECREM` constructs, there is no overlap and negative values are set to zero with the `+neg_tchk` option, unexpected timing violations are reported.

Verilog-XL Reference

Timing Checks

This problem is resolved with the new relaxation algorithm. The two limit timing checks, with negative limits specified for `$setuphold` or `$recrem` timing checks, with the same data and reference limits (with or without different edge specifications), will converge without having to reset the negative time limits to zero.

Here is an example of a non-converging timing check pair:

```
(1) (SETUPHOLD (posedge td) (posedge clk) (145) (-5))
(2) (SETUPHOLD (negedge td) (posedge clk) (6) (-4))
```

Because these two timing checks rely on the same delayed signals, the algorithm must make the most negative timing value (-5) positive. It will make it positive by one base simulation time unit. The new setup and hold limits are:

```
(1) setup = 139
    hold = 1
    data_signal_delay = 5

(2) setup = 0      (6 - (data_signal_delay + delta))
    hold = 2      (-4 + (data_signal_delay + delta))
```

As the values cannot be negative or zero, the algorithm does not converge and non-convergence error messages such as the following are generated:

```
Warning! Non-convergence of NTC values in [Verilog-NTCNCN]
         <design_file>, <line_no>: module <module_name>;
```

The algorithm then forces convergence by setting one negative value in the timing checks to zero and then checking to see if the timing converged. The process is repeated until the timing converges or until all of the negative values are set to zero.

In this example, both negative hold values are set to zero, and after annotation the timing checks are as follows:

```
$setuphold(posedge clk, posedge td, 145, 0);
$setuphold(posedge clk, negedge td, 6, 0);
```

Note that the `posedge/negedge` have no bearing on convergence.

Another situation that results in non-convergence, and that is fairly common in deep submicron designs, is to have two different constraints for `posedge` and `negedge` of data with respect to the reference signal (as in the example above), and in which the constraints do not overlap. Because the violation regions created by the timing checks do not overlap each other, the negative timing check algorithm does not converge. This results in both of the negative limits being set to zero, thus underestimating the actual speed of the design.

Verilog-XL Reference

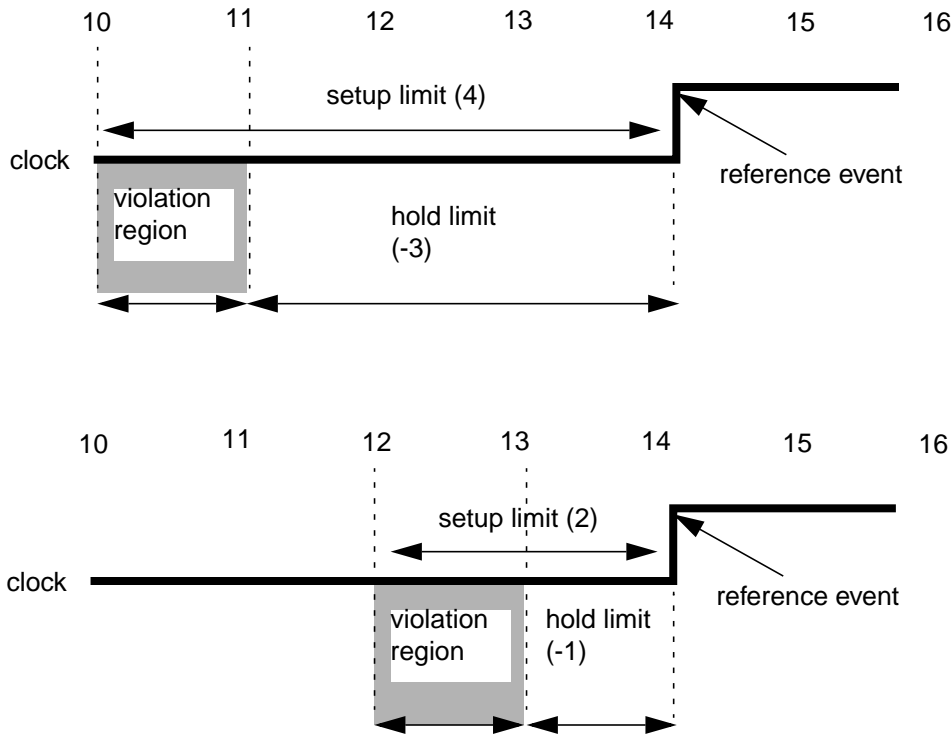
Timing Checks

For example, consider the timing checks in the following example of module `test` in the file `test.v`:

```
1  `timescale 1ns/1ps
2
3  module test;
4      reg data, clock;
5      wire q;
6
7  initial
8      begin
9      $monitor($realtime,"%b %b %b %b %b",clock,data,q,u1.d_del,u1.d_clk);
10     fork
11         #10 data = 0;
12         #10 clock = 0;
13         #13 data = 1'b1;
14         #14 clock = 1'b1;
15         #30 clock = 1'b0;
16         #38.5 data = 1'b0;
17         #40 clock = 1'b1;
18     join
19     #10 $finish;
20 end
21
22 dEdgeFF u1 ( q,clock, data);
23
24 endmodule
25
26 module dEdgeFF(q, clock, data);
27     output q;
28     reg q;
29     input clock, data;
30
31     initial
32         #10 q = 0;
33
34     always
35         @(negedge clock)#10 q=data;
36
37     specify
38         $setuphold(posedge clock, posedge data, 4, -3,,,,d_clk,d_del);
39         $setuphold(posedge clock, negedge data, 2, -1,,,,d_clk,d_del);
40     endspecify
```

Verilog-XL Reference Timing Checks

The first timing check establishes a violation region from 10 to 11 before the reference event (`posedge clock`). The second check establishes a violation region from 12 to 13 before the reference event. These violation regions do not overlap, as shown in the following figure:



This situation, where multiple timing checks use the same signals and where the timing violation regions do not overlap in time, prevents the negative timing check algorithm from converging. The delayed signals cannot be resolved because the value of the hold time for one check is greater than the setup for another. The violation regions for the `posedge` of data and the `negedge` of data must overlap to enable the tool to correctly place the delayed clock (that is, converge). Otherwise, it must set the negative value to 0.

When you use this model, non-convergence warning messages such as the following are generated:

```
Warning! Non-convergence of NTC values in [Verilog-NTCNNC]
         "test.v", 38: module test (q,data,clock);
```

Also, both negative hold values are set to zero. The two timing checks, in effect, are now as follows:

```
$setuphold(posedge clock, posedge data, 4, 0);
$setuphold(posedge clock, negedge data, 2, 0);
```

Verilog-XL Reference

Timing Checks

In the example, the `data` input is changed to 1 at time 13, one ns before the positive edge of `clock`, and then to 0 at time 38.5, 1.5 ns before the next positive edge of `clock`. Therefore, the following two timing check violations are reported:

```
"test.v", 38: Timing violation in test.ul
    $setuphold<setup>( posedge clock:14000, posedge data:13000, 4.000 : 4000,
0.000 : 0 );
```

```
"test.v", 39: Timing violation in test.ul
    $setuphold<setup>( posedge clock:40000, negedge data:38500, 2.000 : 2000,
0.000 : 0 );
```

You can avoid this non-convergence problem in two ways:

- By hand editing the values in the timing checks (or in the SDF file) so that the violation regions overlap by more than two delta simulation time units.

For example, you could hand edit the timing checks in the example to change the -2 hold time on the positive edge to -1.998. This makes the violation regions overlap by two delta simulation time units (the timescale is 1 ns / 1 ps).

You could also create some overlap in the violation regions by changing the setup time in the second timing check from 2 to 3.002.

- By using the `verilog +extend_tcheck_data_limit/<percentage_limit>` or `+extend_tcheck_reference_limit/<percentage_limit>` command-line options with the `+neg_tchk` option.

The `+extend_tcheck_data_limit/<percentage_limit>` or `+extend_tcheck_reference_limit/<percentage_limit>` command-line options automatically extend the violation regions by a specified percentage so that they overlap.

Syntax:

```
+extend_tcheck_data_limit/<percentage_limit>
+extend_tcheck_reference_limit/<percentage_limit>
```

The `+extend_tcheck_data_limit` option changes the hold or recovery limit in the timing checks and the `+extend_tcheck_reference_limit` option changes the setup or removal limit in the timing checks so that the violation regions overlap by at least two units of simulation precision.

The `percentage_limit` argument is the maximum percentage increase that is allowed in the timing violation window to achieve an overlap of at least two units of simulation precision.

However, the use of the `+extend_tcheck_data_limit` and the `+extend_tcheck_reference_limit` command-line options does not mean that there will be an overlap of at least two units of simulation precision in the violation region.

Verilog-XL Reference

Timing Checks

The overlap (whether or not it happens) and the extent of overlap depends on the following:

- ❑ The original timing check limits:

The values specified in the `SETUPHOLD` and `RECREM` constructs determine whether or not there will be an overlap in the violation region and the extent of overlap (at least two units of simulation precision) of the specified timing check pair. For example, the following timing check pair will give a non-convergence warning if you use the default timescale or a timescale of 1ns/1ns:

```
$setuphold( posedge clock, posedge data, 4, -3);  
$setuphold( posedge clock, negedge data, 2, -1);
```

- ❑ The timescale used:

The timescale specified before the module also determines whether or not there will be an overlap in the violation region and the extent of overlap (at least two units of simulation precision) of the specified timing check pair. If you use the default timescale, the timing check pair will give a non-convergence warning. For example, if you use a timescale of 1ns/1ns in [Example 1](#) on page 324, you will get a non-convergence warning.

- ❑ The limits specified by the user:

The limit specified by you in `<percentage_limit>` also determines whether or not there will be an overlap in the violation region and the extent of overlap (at least two units of simulation precision) of the specified timing check pair. For more information, see [Example 2](#) on page 325.

You cannot use the `+extend_tcheck_data_limit/<percentage_limit>` or `+extend_tcheck_reference_limit/<percentage_limit>` command-line options simultaneously. If you do, the Verilog-XL simulator ignores both the options, the relaxation algorithm is not implemented and a warning message such as the following is generated:

```
Warning! Cannot stretch both the limits,relaxation algorithm not used. Specify  
any one limit. [Verilog-CNSBL]
```

If a decimal value is specified as the limit in the `+extend_tcheck_data_limit/<percentage_limit>` or `+extend_tcheck_reference_limit/<percentage_limit>` command-line options, 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.

Example 1

In the example used above, the timescale used is 1ns/1ps and the timing checks are:

Verilog-XL Reference

Timing Checks

```
$setuphold( posedge clock, posedge data, 4, -3);  
$setuphold( posedge clock, negedge data, 2, -1);
```

You could extend the hold time of the first timing check up to 1 ns (that is, 100% of the width of the violation region) plus two units of precision with the following command:

```
% verilog +extend_tcheck_data_limit/100 +neg_tchk test.v
```

This command extends the violation region created by the first timing check by 1.002 to create some overlap. The new setup and hold limits are:

```
$setuphold( posedge clock, posedge data, 4, -1.998);  
$setuphold( posedge clock, negedge data, 2, -1);
```

Alternatively, you could extend the violation region created by the second timing check to create some overlap by extending the setup time of the second window up to 1.002 ns with the following command:

```
% verilog +extend_tcheck_reference_limit/100 +neg_tchk test.v
```

Then the new setup and hold limits are:

```
$setuphold( posedge clock, posedge data, 4, -3);  
$setuphold( posedge clock, negedge data, 3.002, -1);
```

Example 2

Suppose you have the following pair of timing checks, where there are two non-overlapping violation windows, each with a width of 2 ns and the timescale used is 1ns/1ps .

```
$setuphold( posedge clock, posedge data, 7, -5);  
$setuphold( posedge clock, negedge data, 3, -1);
```

The following command extends the setup time of the second window up to 2 ns (100% of the width of the violation region) plus two units of precision. In other words, the setup time is changed to 5.002.

```
% verilog +extend_tcheck_reference_limit/100 +neg_tchk test.v
```

The following command extends the hold time of the first window up to 1 ns (50% of the width of the violation region) plus two units of precision:

```
% verilog +extend_tcheck_data_limit/50 +neg_tchk test.v
```

Note that, because the region between the two violation regions is 2 ns, extending the hold time by 1.002 ns will not cause the timing violation regions to overlap, and you will get non-convergence warnings.

When you use the `+extend_tcheck_data_limit/<percentage_limit>` option, and if the specified relaxation percentage allows the timing checks to converge, the Verilog-XL simulator generates a warning message (Verilog-DLSS) that informs you that a pair of signals had non-overlapping two limit constraints for different edges, that this situation

Verilog-XL Reference

Timing Checks

caused non-convergence, and that the limits are being relaxed to make the constraints overlap.

When you use the `+extend_tcheck_reference_limit/<percentage_limit>` option, and if the specified relaxation percentage allows the timing checks to converge, the Verilog-XL simulator generates a warning message (`Verilog-RLSS`) that informs you that a pair of signals had non-overlapping two limit constraints for different edges, that this situation caused non-convergence, and that the limits are being relaxed to make the constraints overlap.

Explicitly Defining Delayed Signals

The delayed versions of signals can be explicitly defined in the `$setuphold` and `$recrem` timing checks using the `delayed_reference` and `delayed_data` arguments, which are the delayed version of the reference and data signals, respectively. You may want to explicitly define the delayed signals in order to drive the functional model using the delayed version of these signals. The following example illustrates this. The negative timing check value causes Verilog-XL to generate a delayed signal to use as input to the functional part of the UDP circuit. This ensures that the correct value for the data signal is present at the UDP input when the clock edge occurs.

```
module dff (q, d, clk);
    output q;
    input d, clk;
    dff_prim pl(q, dd, dclk, notify);
    specify
        $setuphold(posedge clk, d, 12, -5, notify, , , dclk, dd);
    endspecify
endmodule
```

If the delayed signals `dclk` and `dd`, were not explicitly defined, delayed versions of `clk` and `d` would still be used to evaluate the timing check. However, the functional model would utilize the undelayed signals.

The following is a slightly more complex example, which uses several delayed signals.

```
module device(q, d, clk, set, clr);
    output q;
    input d, clk, set, clr;
    prim pl(q, dd, dclk, dset, dclr);
    specify
        $setuphold(posedge clk, d, 12, -3, , , , dclk, dd);
        $recrem(posedge clr, posedge clk, 10, -7, , , , dclr, dclk);
        $recrem(posedge set, posedge clk, 13, -4, , , , dset, dclk);
    endspecify
endmodule
```

Verilog-XL Reference

Timing Checks

Verilog-XL iteratively analyzes the entire set of timing checks to generate a correct set of delay values for a device. The generated delay values for the model in the previous example are as follows:

```
clk  7
set  0
clr  0
d    10
```

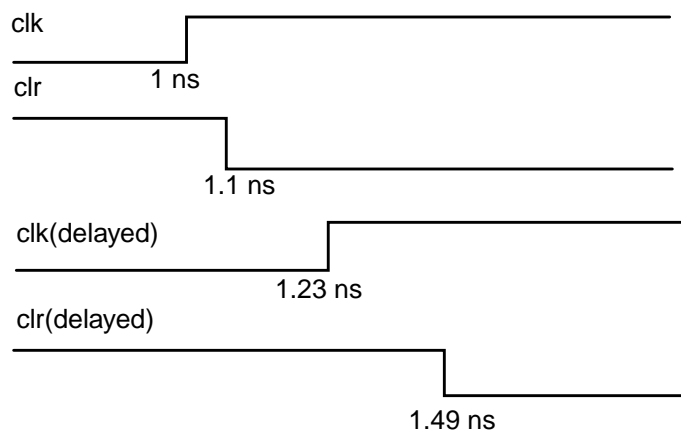
Verilog-XL takes the limits from the entire set of timing checks into account when choosing a limit. You must list the delayed signal in *each* timing check that makes a contribution to the final delay generated for each signal.

Effects of Delayed Signals on Path Delays

Delayed signals may affect path delays. When you specify a negative timing check, Verilog-XL chooses a path delay that may be different from the path delay that is chosen without negative timing checks.

To illustrate this, consider the following set of timing checks and path delays and the input waveforms (with delayed signals) in the following figure.

```
clk => q = 1.2;
clr => q = 0.33;
$setuphold(posedge clk, d, -0.23, 1.1, ...);
$recrem(posedge clr, posedge clk, -0.39, 0.77, ...);
```



If only undelayed signals are used, the output transition is scheduled at 2.2 ns because the functional part of the circuit immediately detects the `clk` transition at 1 ns and schedules the output at `q` 1.2 ns later.

Verilog-XL Reference

Timing Checks

If the delayed signals are used, the functional part of the circuit does not detect a transition (and a corresponding output change) until 1.23 ns. Because the path delay algorithm determines the path delay from the input with the most recent transition, it picks the path delay from `clr`, and the output transition is scheduled at time 1.43 ns (1.1 + 0.33).

To restore the original behavior, the delayed signals would need to be used as the input to the path delay algorithm in addition to the functional part of the circuit. However, the original behavior does not exactly represent the silicon Verilog-XL uses the undelayed signals as the inputs to the path delays because the output results depend on whether the delayed or undelayed signals are used, and the path delay may not be any less accurate than when using the delayed signals.

The delay calculated for a delayed signal should not be longer than a path delay with that signal as a source. After the delays for the delayed signals are calculated, all path delays in a module are scanned, and if any are longer than the delayed signal for their source, a warning is issued if the `+ntc_warn` option is provided to the elaborator. Furthermore, when this condition is detected (regardless of the presence of `+ntc_warn`), the entire process of calculating delays is started again, just as when a convergence error is detected, as described in [“Calculation of Delayed Signals and Limit Modification”](#) on page 316.

Restrictions

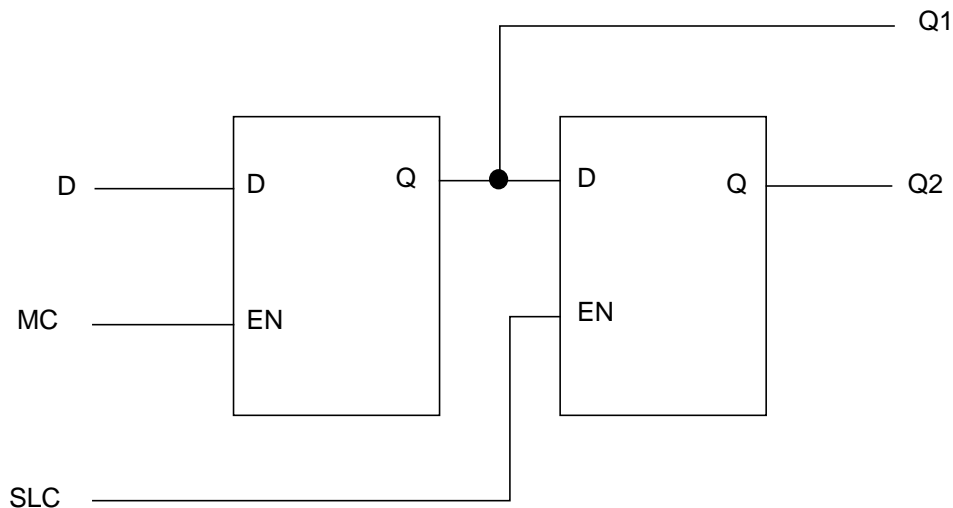
The delayed signal algorithm for negative timing check values cannot resolve the following situations for nonconverging timing check limits or signal delays that are bigger than the path delay for any signal:

- A signal that has a relationship to other signals where the other signals have no relationship to each other. For example, see the following figure where `MC` between `Q1` and `SLC`, where there is no relationship between `Q1` and `SLC`.

Verilog-XL Reference

Timing Checks

- Multiple timing checks that are based on relations that have no correlation with each other. For example, see the following figure where the relationship between MC and D has no correlation with the relationship between MC and SLC.



```
MC => Q1 = (4.2);
$setuphold(negedge SLC, posedge MC, 10.2, -9.4, ...);
```

You cannot use a single delayed signal to simultaneously model two different relationships. In the previous figure, the `$setuphold` timing check shows that the MC signal needs to be delayed by 9.4 to maintain the functional relationship between the SLC and MC signals. However, because you can create only one delayed version of MC, the effect of MC on Q1 is also delayed by 9.4., but this delay is longer than the needed path delay between MC and Q1.

A similar situation can arise between sets of timing checks. Consider the following set of timing checks, using the model in the previous figure.

```
$setuphold(negedge MC, D, 0.18, 0.03, ...);
$setuphold(negedge SLC, D, 1.28, -0.69, ...);
$setuphold(posedge MC, negedge SLC, -0.13, 0.21, ...);
```

Even though all timing check limits are valid by themselves, you cannot have a single delay value to satisfy all of the timing checks because the functional relationship between sets of signals (MC and D, and MC and SLC) are independent of each other. The timing check value between MC and SLC has no effect on the timing check value between MC and D, and vice versa. Therefore the two cannot be modeled simultaneously.

Exception Handling

When delayed signals cannot be resolved exactly, or when a signal delay is longer than a path delay, Verilog-XL approximates the set of delay values by setting the setup limit with the smallest magnitude to 0 then reapplying the algorithm. If the signals do not converge, the process is repeated on successive setup limits from smallest to largest. If delayed signals still do not converge, the process begins with hold limits. This method is guaranteed to eventually succeed because eventually all negative limits are set to 0.

You can display a warning message when Verilog-XL uses this approximation algorithm by specifying the `+ntc_warn` option on the command line. By default, a warning is not printed.

System Tasks and Functions

This chapter describes the following:

- [Filename Parameters](#) on page 332
- [Display and Write Tasks](#) on page 333
- [Strobed Monitoring](#) on page 340
- [Continuous Monitoring](#) on page 341
- [Monitoring Interconnect Delay Signal Values](#) on page 342
- [File Output](#) on page 343
- [Default Base](#) on page 345
- [Signed Expressions](#) on page 346
- [Simulation Time](#) on page 346
- [Stop and Finish](#) on page 347
- [Random Number Generation](#) on page 347
- [Tracing](#) on page 348
- [Saving and Restarting Simulations](#) on page 352
- [Command History](#) on page 355
- [Command Input Files](#) on page 356
- [Log File](#) on page 356
- [Key File](#) on page 357
- [Setting the Interactive Scope](#) on page 358
- [Showing the Hierarchy](#) on page 358
- [Showing Variable Status](#) on page 358

- [Showing Net Expansion Status](#) on page 359
- [Showing Module Port Status](#) on page 360
- [Showing Number of Drivers](#) on page 360
- [Displaying the Delay Mode](#) on page 362
- [Storing Interactive Commands](#) on page 362
- [Interactive Source Listing—Decompilation](#) on page 363
- [Disabling and Enabling Warnings](#) on page 365
- [Loading Memories from Text Files](#) on page 368
- [Setting a Net to a Logic Value](#) on page 369
- [Fast Processing of Stimulus Patterns](#) on page 370
- [Incremental Pattern File Tasks](#) on page 372
- [Functions and Tasks for Reals](#) on page 380
- [Functions and Tasks for Timescales](#) on page 380
- [Protecting Data in Memory](#) on page 381
- [Value Change Dump File Tasks](#) on page 382
- [Running the Behavior Profiler](#) on page 382
- [Resetting Verilog-XL—Starting Simulation Over Again](#) on page 384
- [SDF Annotation](#) on page 392
- [Using the \\$dlc System Task](#) on page 403
- [Using the \\$system System Task](#) on page 404

Filename Parameters

Many of the Verilog-XL system tasks and functions take filenames as parameters. Each filename must adhere to the conventions of the host platform. Any filename that is not acceptable to the host platform results in run-time errors. You may use string variables instead of parameters to specify filenames to system tasks and functions. For example:

```
//string variable to pass in filename
reg [ 50*8 : 1 ] mem_file;
integer memory_dump;
```

```
initial
begin
  case ($reset_count)
    0 : mem_file = "./test_1/stimuli/boot_file";
    1 : mem_file = "./test_2/stimuli/boot_file";
    2 : mem_file = "./test_3/stimuli/boot_file";
  endcase
end

initial
begin
  memory_dump = $fopen(mem_file); // open memory output file
end
```

This technique permits dynamic file selection under the control of the Verilog-XL language.

Display and Write Tasks

The `$display` and `$write` tasks are the main system task routines for displaying information. The two tasks are identical except that `$display` automatically adds a new line character to the end of its output, whereas `$write` does not. Thus, if you want to print several messages on a single line, use `$write`.

The `$display` and `$write` tasks display their parameters in the same order that they appear in the parameter list. Each parameter can be a quoted string, an expression that returns a value, or a null parameter. The syntax for these tasks is as follows:

```
$display(P1, P2, ... , Pn);
$write(P1, P2, ... , Pn);
```

The contents of string parameters are output literally, except when certain escape sequences are inserted to display special characters or to specify the display format for a subsequent expression.

Escape sequences are inserted into a string in two ways:

- The backslash character (\) indicates that the character to follow is a literal or non-printable character (see [“Escape Sequences for Special Characters”](#) on page 334).
- The percentage character (%) indicates that the next character is a format specification that establishes the display format for a subsequent expression parameter (see the table in [“Format Specifications”](#) on page 334). For each % character that appears in a string, a corresponding expression parameter must be supplied after the string.

Two percentage characters (%%) indicate the display of the percentage character % (see [“Escape Sequences for Special Characters”](#)).

A null parameter produces a single space character in the display. (A null parameter is characterized by two adjacent commas in the parameter list.)

Verilog-XL Reference

System Tasks and Functions

The `$display` task, when invoked without parameters, prints a new line character. A `$write` task invoked without parameters prints nothing.

Note: Because `$write` does not produce a new line character after outputting its text, most operating systems simply buffer the text rather than flush it directly to the output. For these operating systems, to 'see' the text in the output immediately, use the `$display` instead of `$write`, or include an explicit new line character (`\n`) in the `$write` task.

Escape Sequences for Special Characters

The following escape sequences, when included in a string parameter, cause special characters to be displayed:

```
\n  is the new line character
\t  is the tab character
\\  is the \ character
\"  is the " character
\o  is a character specified in 1-3 octal digits
%%  is the percent character
```

The following example shows these escape sequences in a string parameter:

```
module disp;
  initial
  begin
    $display("\\\t%%\n\"123");
  end
endmodule
```

The following are the results of the above example. Here `S` is the octal equivalent of `123`:

```
Highest level modules:
  disp
  \
  "S
```

Format Specifications

The following table shows the escape sequences used for format specifications.

<code>%h</code> or <code>%H</code>	Display in hexadecimal format
<code>%d</code> or <code>%D</code>	Display in decimal format
<code>%o</code> or <code>%O</code>	Display in octal format
<code>%b</code> or <code>%B</code>	Display in binary format

Verilog-XL Reference

System Tasks and Functions

The format specifications in the following table are used with real numbers and have the full formatting capabilities available in the C language. For example, the format specification `%10.3g` specifies a minimum field width of 10, with 3 fractional digits.

<code>%e</code> or <code>%E</code>	Display real number in an exponential format.
<code>%f</code> or <code>%F</code>	Display real number in a decimal format.
<code>%g</code> or <code>%G</code>	Display real number in exponential or decimal format, whichever format results in the shorter printed output.

Size of Displayed Data

For expression parameters, the values written to the output file (or terminal) are usually sized automatically. Verilog-XL reserves just enough characters to hold the largest possible value that can be returned by the expression, given the expression's bit length and specified display format.

For instance, the result of a 12-bit expression would be allocated three characters when displayed in hexadecimal format and four characters when displayed in decimal format since the largest possible value the expression is FFF in hexadecimal format and 4095 in decimal format.

When displaying decimal values, leading zeros are suppressed and replaced by spaces. In other radices, leading zeros are always displayed.

You can override the automatic sizing of displayed data by inserting a zero between the `%` character and the letter that indicates the radix, as shown below:

```
$display("d=%0h a=%0h", data, addr);
```

In response, Verilog-XL allocates the exact number of characters required to display the current expression result, instead of the number of characters in the expression's largest possible value. Consider the following Verilog-XL description and results:

```
module printval;
  reg [11:0] r1;
  initial
  begin
    r1 = 10;
    $display( "Printing with maximum size - :%d: :%h:",r1,r1 );
    $display( "Printing with minimum size - :%0d: :%0h:",r1,r1 );
  end
endmodule
```

The following are the results of the above example:

```
Highest level modules:
printval
```


Verilog-XL Reference

System Tasks and Functions

```
Printing with maximum size - : 10: :00a:
Printing with minimum size - :10: :a:
6 simulation events
```

In this example, the result of a 12-bit expression is displayed. The first call to `$display` uses the standard format specifier syntax and produces results requiring four and three columns for the decimal and hexadecimal radices, respectively. The second `$display` call uses the `%0` form of the format specifier syntax and produces results requiring two and one column, respectively.

Unknown and High-Impedance Values

When the result of an expression contains an unknown or high-impedance value, the following rules apply to displaying that value:

In decimal (`%d`) format:

- If all bits are at the unknown value, a single lowercase `x` character is displayed.
- If all bits are at the high-impedance value, a single lowercase `z` character is displayed.
- If some but not all bits are at the unknown value, the uppercase `X` character is displayed.
- If some but not all bits are at the high-impedance value, the uppercase `Z` character is displayed.
- Decimal numerals always appear right-justified in a fixed-width field. (The fixed-width format is used so that the output produced is consistent with the `$monitor` task output, which requires a fixed columnar format.)

In hexadecimal (`%h`) and octal (`%o`) formats:

- Each group of 4 bits is represented as a single hexadecimal digit; each group of 3 bits is represented as a single octal digit.
- If all bits in a group are at the unknown value, a lowercase `x` is displayed for that digit.
- If all bits in a group are at a high-impedance state, a lowercase `z` is printed for that digit.
- If some but not all bits in a group are unknown, an uppercase `X` is displayed for that digit.
- If some but not all bits in a group are at a high-impedance state, then an uppercase `Z` is displayed for that digit.

In binary (`%b`) format, each bit is printed separately using the characters `0`, `1`, `x`, and `z`.

Verilog-XL Reference

System Tasks and Functions

Some of these rules are illustrated in the following example:

STATEMENT	RESULT
<code>\$display("%d", 1'bx);</code>	x
<code>\$display("%h", 14'bx01010);</code>	xxXa
<code>\$display("%h %o", 12'b001xxx101x01, 12'b001xxx101x01);</code>	XXX 1x5X

Strength Format

The `%v` format specification is used to display the strength of scalar nets. For each `%v` specification that appears in a string, a corresponding scalar reference must follow the string in the parameter list. The parameter must be an explicit scalar reference; that is, it cannot be an expression or a bit-select.

The strength of a scalar net is reported in a three-character format. The first two characters indicate the strength. The third character indicates the scalar's current logic value and may be any one of the following:

0	for a logic 0 value
1	for a logic 1 value
X	for an unknown value
Z	for a high impedance value
L	for a logic 0 or high impedance value
H	for a logic 1 or high impedance value

The first two characters—the strength characters—are either a two-letter mnemonic or a pair of decimal digits. Usually, a mnemonic is used to indicate strength information; however, in less typical cases, a pair of decimal digits may be used to indicate a range of strength levels.

The following table shows the mnemonics used to represent the various strength levels:

Mnemonic	Strength Name	Strength Level
Su	Supply drive	7
St	Strong drive	6

Verilog-XL Reference

System Tasks and Functions

Mnemonic	Strength Name	Strength Level
Pu	Pull drive	5
La	Large capacitor	4
We	Weak drive	3
Me	Medium capacitor	2
Sm	Small capacitor	1
Hi	High impedance	0

Note that there are four driving strengths and three charge storage strengths. The driving strengths are associated with gate outputs and continuous assignment outputs. The charge storage strengths are associated with the `triereg` type net. “[Logic Strength Modeling](#)” on page 114 provides information on strength modeling.

For the logic values 0 and 1, a mnemonic is used when there is no range of strengths in the signal. Otherwise, the logic value is preceded by two decimal digits, which indicate the maximum and minimum strength levels.

For an unknown value, a mnemonic is used when both the 0 and 1 strength components are at the same strength level. Otherwise, the unknown value X is preceded by two decimal digits, which indicate the 0 and 1 strength levels respectively.

The high-impedance strength cannot have a known logic value; the only logic value allowed for this level is Z.

For the values L and H, a mnemonic is always used to indicate the strength level.

Consider the following call to `$monitor`:

```
$monitor($time,, "group=%b signals=%v %v %v",
        {sig1,sig2,sig3}, sig1, sig2, sig3);
```

The following example shows the output that might result from such a call.

```
0 group=111 signals=St1 Pu1 St1
15 group=011 signals=Pu0 Pu1 St1
30 group=0xz signals=520 PuH HiZ
31 group=0xx signals=Pu0 65X StX
45 group=000 signals=Me0 St0 St0
```

The table given below explains the various strength formats that appear in the output.

St1	Means a strong driving 1 value.
-----	---------------------------------

Verilog-XL Reference

System Tasks and Functions

Pu0	Means a pull driving 0 value.
HiZ	Means the high impedance state.
Me0	Means a 0 charge storage of medium capacitor strength.
StX	Means a strong driving unknown value.
PuH	Means a pull driving 1 or high impedance.
65X	Means an unknown value with a strong driving 0 component and a pull driving 1 component.
520	Means a 0 value with a range of possible strength from pull driving to medium capacitor.

Hierarchical Name Format

The `%m` format specifier does not accept a parameter. Instead, it causes Verilog-XL to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is very useful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier will pinpoint the module instance responsible for generating the timing check message.

String Format

You can use the `%s` format specifier to print ASCII codes as characters. For each `%s` specification that appears in a string, a corresponding parameter must follow the string in the parameter list. Verilog-XL interprets the associated parameter as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the parameter is a variable, you should right-justify its value so that the right-most bit of the value is the least-significant bit of the last character in the string. No termination character or value is required at the end of a string, and leading zeros are never printed.

Strobed Monitoring

When Verilog-XL encounters the `$strobe` system task, it displays the specified information at the end of the time unit. The parameters for this task are specified in exactly the same manner as for the `$display` system task—including the use of escape sequences for special characters and format specifications (see [“Display and Write Tasks”](#) on page 333). The syntax is as follows:

```
$strobe(P1, P2, ..., Pn);
```

Verilog-XL Reference

System Tasks and Functions

The following example shows how the `$strobe` system task is used:

```
forever @(negedge clock)
$strobe ("At time %d, data is %h", $time, data);
```

In this example, `$strobe` writes the time and data information to the standard output and to the log file at each negative edge of the clock. The action occurs just before simulation time is advanced, after all other events at that time have occurred, so that the data written is sure to be the correct data for that simulation time.

The strobe tasks produce output when they are executed, and there is no on/off control necessary.

Continuous Monitoring

The `$monitor` task provides the ability to monitor and display the values of any variables or expressions specified as parameters to the task. The parameters for this task are specified in exactly the same manner as for the `$display` system task—including the use of escape sequences for special characters and format specifications. See [“Display and Write Tasks”](#) on page 333. The syntax is as follows:

```
$monitor(P1, P2, ..., Pn);
$monitor;
$monitoron;
$monitroff;
```

When you invoke a `$monitor` task with one or more parameters, the simulator sets up a mechanism whereby each time a variable or an expression in the parameter list changes value, with the exception of the `$time`, `$stime`, or `$realtime` system functions, the entire parameter list is displayed at the end of the time step, as if reported by the `$display` task. If two or more parameters change value at the same time, however, only one display is produced that shows all of the new values.

Note that only one `$monitor` display list can be active at any one time; however, you can issue a new `$monitor` task with a new display list any number of times during simulation.

The `$monitoron` and `$monitroff` tasks control a monitor flag that enables and disables the monitoring, so that you can easily control when monitoring should occur. Use `$monitroff` to turn off the flag and disable the monitoring. Use `$monitoron` to turn on the flag so that monitoring is enabled and the most recent call to `$monitor` can resume its display.

A call to `$monitoron` always produces a display immediately after it is invoked, regardless of whether a value change has taken place; this is used to establish the initial values at the beginning of a monitoring session. By default, the monitor flag is turned on at the beginning of simulation.

If you use the `$monitor` command with no parameters, monitoring is turned off. Using `$monitor` with no parameters differs from using `$monitoroff`. If you turn off monitoring using `$monitoroff`, you can restart monitoring using `$monitoron`. If you turn off monitoring using `$monitor` with no parameters, you must execute a new `$monitor` task with a new display list to restart monitoring.

For `$monitor` tasks issued interactively, there is an alternative method for controlling when monitoring occurs. The method involves using the `disable` command to turn off a `$monitor` command and then re-executing the command to turn monitoring back on. The following example illustrates this technique in which monitoring is allowed to occur for the first 100 time units of the simulation before the `disable` command is issued at C5. The `disable` command is issued by identifying the command number of the interactive command you wish to disable and then typing a minus sign before it. Here, by typing `-3`, we disable command 3, which invokes the `$monitor` task.

```
C3> $monitor($time,,"rxd=%b txd=%b",rxd,txd);
C4> #100 $stop;
    0 rxd=1 txd=1
   20 rxd=0 txd=1
   60 rxd=0 txd=0
   80 rxd=0 txd=1
C4: $stop at simulation time 100
C5> -3
```

Later in the simulation, by typing a 3 at the interactive command prompt, we can re-execute command 3 to resume monitoring.

Monitoring Interconnect Delay Signal Values

The `$post_int_delay` system task provides monitoring for a specified signal value after an interconnect delay. For more information about interconnect delays, see [Chapter 16, "Interconnect Delays."](#)

The syntax for `$post_int_delay` is as follows:

```
$post_int_delay(<input_port_name>)
```

Note: You can only specify the `$post_int_delay` system task as an argument to the following system tasks:

- `$display` and derivatives (`$fdisplay`, for example)
- `$monitor` and derivatives (`$fmonitor`, for example)
- `$write` and derivatives (`$fwrite`, for example)
- `$strobe` and derivatives (`$fstrobe`, for example)

Also, `$post_int_delay` is activated only when you specify the `+transport_int_delays` plus option on the command line.

The following `$monitor` system task uses the `$post_int_delay` system task:

```
$monitor("%0t: net source = %b net destination = %b",  
        $realtime, U1.clk, $post_int_delay(U1.clk));
```

File Output

Each of the four formatted display tasks—`$display`, `$write`, `$monitor`, and `$strobe`—has a counterpart that writes to specific files as opposed to the log file and standard output. These counterpart tasks—`$fdisplay`, `$fwrite`, `$fmonitor`, and `$fstrobe`—accept the same type of parameters as the tasks they are based upon, with one exception: The first parameter must be a multichannel descriptor that indicates where to direct the file output. A multichannel descriptor is either a variable or the result of an expression that takes the form of a 32-bit unsigned integer value. This value determines the open files to which the task writes. The syntax is as follows:

```
$fdisplay(<multi_channel_descriptor>, P1, P2, ... , Pn);  
$fwrite(<multi_channel_descriptor>, P1, P2, ... , Pn);  
$fstrobe(<multi_channel_descriptor>, P1, P2, ... , Pn);  
$fmonitor(<multi_channel_descriptor>, P1, P2, ... , Pn);  
$fopen("<name_of_file>")  
$fclose(<multichannel_descriptor>);
```

The function `$fopen` opens the file specified as a parameter and returns a 32-bit unsigned multichannel descriptor that is uniquely associated with the file. It returns 0 if the file could not be opened for writing.

Think of the multichannel descriptor as a set of 32 flags, where each flag represents a single output channel. The least significant bit (bit 0) of a multichannel descriptor always refers to the standard output—that is, the log file and the screen (unless it has been redirected to a file).

The standard output is also called channel 0. The other bits refer to channels that have been opened by the `$fopen` system function.

The first call to `$fopen` opens channel 1 and returns a multichannel descriptor value of 2—that is, bit 1 of the descriptor is set. A second call to `$fopen` opens channel 2 and returns a value of 4—that is, only bit 2 of the descriptor is set. Subsequent calls to `$fopen` open channels 3, 4, 5, and so on and return values of 8, 16, 32, and so on, up to a maximum of 31 channels. Thus, a channel number corresponds to an individual bit in a multichannel descriptor.

The advantage of multichannel descriptors is that they allow a single system task to write the same information to multiple outputs simultaneously. This is accomplished by setting more than one bit in the multichannel descriptor, which is done by combining the values returned by `$fopen` in a bit-wise OR operation. Another advantage of multichannel descriptors is that

Verilog-XL Reference

System Tasks and Functions

it is easy to set up descriptions where the channels that receive diagnostic information can be dynamically altered during simulation, and even controlled with interactive commands.

Note: The number of simultaneous output channels that may be active at any one time is dependent on the operating system and is not determined by Verilog-XL.

Verilog-XL does not buffer writes via the `$fwrite` or `$fdisplay` system tasks. However, it is good programming practice to use `$fclose` to assure that all data has been written to the file. The `$fclose` system task closes the channels specified in the multichannel descriptor, and does not allow any further output to the closed channels. The `$fopen` task will reuse channels that have been closed.

The following example shows how to set up multichannel descriptors. In this example, three different channels are opened using the `$fopen` function. The three multichannel descriptors that are returned by the function are then combined in a bit-wise OR operation and assigned to the integer variable `messages`. The `messages` variable can then be used as the first parameter in a file output task to direct output to all three channels at once. To create a descriptor that directs output to the standard output as well, the `messages` variable is bit-wise ORed with the constant 1, which effectively enables channel 0.

```
integer
  messages,
  broadcast,
  cpu_chann,
  alu_chann,
  mem_chann;
initial
  begin
    cpu_chann = $fopen("cpu.dat"); if(cpu_chann == 0) $finish;
    alu_chann = $fopen("alu.dat"); if(alu_chann == 0) $finish;
    mem_chann = $fopen("mem.dat"); if(mem_chann == 0) $finish;
    messages = cpu_chann | alu_chann | mem_chann;
    broadcast = 1 | messages; // includes standard output
  end
```

The following file output tasks show how the channels opened in the previous example can be used:

```
$fdisplay( broadcast, "system reset at time %d", $time );

$fdisplay( messages, "Error occurred on address bus at time %d, address = %h",
$time, address );

forever @(posedge clock)
  $fdisplay( alu_chann, "acc= %h f=%h a=%h b=%h",acc, f, a, b );
```

The following interactive dialog is a further example of the use of multichannel descriptors:

```
C6 > $display( $fopen( "debug.lis" ) );
      16
C7 > forever @(negedge clock)
      > $fdisplay( 17, "At time %h, data is %h", $time, data );
```


Verilog-XL Reference

System Tasks and Functions

In this dialog, `$fopen` opens the file `debug.lis` and returns a value of 16. Then `$fdisplay` uses a multichannel descriptor of 17 to direct its output to both the standard output and the newly opened file.

Another way to control file output interactively is to dynamically alter multichannel descriptors. For example, the multichannel descriptor called `messages` in the previous example could be altered to include the standard output as follows:

```
C8 > messages[0]=1;
```

Later in the simulation, the `messages` descriptor could be restored to its original form with the following command:

```
C13 > messages[0]=0;
```

The `$fstrobe` and `$fmonitor` system tasks work just like their counterparts, `$strobe` and `$monitor`, except that they write to files using the multichannel descriptor for control. Unlike `$monitor`, you can set up any number of `$fmonitor` tasks to be simultaneously active. Thus, if you need to have more than one monitor task report to the standard output, then use a `$fmonitor` system task with a multichannel descriptor of 1.

Default Base

To avoid an excessive use of format specifiers, you can change the default format specification from decimal to either hexadecimal, binary, or octal. To do so, simply append one of the letters `h`, `b`, or `o` to the name of any of the formatted output system tasks. Consider the following example:

```
$displayh(var,,a,,b);
```

With the letter `h` appended to its name, this task displays each of its three variables in hexadecimal format.

The complete list of formatted output system tasks is as follows:

<code>\$display</code>	<code>\$fdisplay</code>	<code>\$write</code>	<code>\$fwrite</code>
<code>\$displayh</code>	<code>\$fdisplayh</code>	<code>\$writeh</code>	<code>\$fwriteh</code>
<code>\$displayb</code>	<code>\$fdisplayb</code>	<code>\$writeb</code>	<code>\$fwriteb</code>
<code>\$displayo</code>	<code>\$fdisplayo</code>	<code>\$writeo</code>	<code>\$fwriteo</code>
<code>\$strobe</code>	<code>\$fstrobe</code>	<code>\$monitor</code>	<code>\$fmonitor</code>
<code>\$strobeh</code>	<code>\$fstrobeh</code>	<code>\$monitorh</code>	<code>\$fmonitorh</code>
<code>\$strobeb</code>	<code>\$fstrobeb</code>	<code>\$monitorb</code>	<code>\$fmonitorb</code>
<code>\$strobeo</code>	<code>\$fstrobeo</code>	<code>\$monitro</code>	<code>\$fmonitro</code>

Signed Expressions

The `$signed()` and `$unsigned()` system tasks evaluate the input expression and return a value with the same size and value as the input expression with the type defined by the function.

Simulation Time

The `$time`, `$stime`, and `$realtime` system functions return the current simulation time. The function `$time` returns a 64-bit integer value, while `$stime` returns a 32-bit integer, and `$realtime` returns a real number. Each function returns a value that is scaled to the time unit of the module that invoked it. If the value of the simulation time is greater than the decimal 4294967295, then `$stime` returns a value expressed in modulus 2^{32} . The syntax is as follows:

```
$time
$stime
$realtime
```

These functions are useful when used as parameters to the formatted output tasks for establishing simulation time along with the display output. Consider the following example:

```
$monitor($time,, "areg=%h", areg);
```

The above call to `$monitor` produces a continuous display similar to the following:

```
0 abus=01
1000 abus=2f
1010 abus=1f
1013 abus=10
2000 abus=8a
```

Note that the `$time` parameter does not trigger monitoring whenever its value changes; time values are only displayed when one of the other parameters changes value. The same is true for `$stime` and `$realtime`.

Note also that `$time` and its relatives do not trigger event controls; to wait for a particular time, use a delay control, as in the following example. This delay control delays the assignment statement until simulation time reaches the desired time.

```
begin
    #(desired_time - $time) n=0;
end
```

Stop and Finish

The `$stop` system task puts the simulator into a halt mode, issues an interactive command prompt, and passes control to the user. This task takes an optional expression parameter that determines what type of diagnostic message is printed before the interactive command prompt is issued. If no parameter is supplied, then the task defaults to a parameter value of 1. The syntax is as follows:

```
$stop;  
$stop(n);  
$finish;  
$finish(n);
```

The following table shows the valid parameter values and the diagnostic information they produce:

Parameter Value	Diagnostic Message
0	Prints nothing.
1	Prints simulation time and location.
2	Prints simulation time, location, and statistics about the memory and CPU time used in simulation.

The `$finish` system task simply causes the simulator to exit and pass control back to the host operating system. The `$finish` task accepts the same optional parameter as `$stop` and produces the same type of diagnostic information.

Random Number Generation

The system function `$random` provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative. For further information on random number generators, see [Appendix D, Stochastic Analysis](#). The syntax is as follows:

```
$random;  
$random(<seed>);
```

The `<seed>` parameter controls the numbers that `$random` returns. The `<seed>` parameter must be either an integer or a register type and must be defined prior to calling `$random`.

Verilog-XL Reference

System Tasks and Functions

The random numbers generated are the same across all machines and operating systems, and are reproduced correctly when `$save` and `$restart` are used to save and restart the simulation data structure (see [“Saving and Restarting Simulations”](#) on page 352).

Where $b > 0$, the expression `($\$random \% b$)` gives a number in the range: $[(-b+1) : (b-1)]$. The following code fragment shows an example of 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;
```

Tracing

The `$settrace` system task enables the tracing of simulation activity. This trace consists of various information, including the current simulation time, the location in the source file description of the active statement, a full decompilation of the statement, and the result of the execution of the statement. The syntax is as follows:

```
$settrace;
$cleartrace;
```

You can turn off the trace using the `$cleartrace` system task and then turn it back on using `$settrace` any number of times during the simulation. For example, the following interactive command enables a trace for 20 time units after each change of the variable `s85.i85.acc`:

```
C8 > forever begin
> @s85.i85.acc $settrace;
> #20 $cleartrace;
> end
```

The following example is a source file that invokes `$settrace`:

```
module settrace;
reg [23:0] cond1, cond2;
  initial
  begin
    $settrace;
    cond1=0;
    cond2=0;
    #200 $finish;
  end
  always #5
  begin
    cond1=~cond1;
    #5;
    cond2=~cond2;
  end
end
```

Verilog-XL Reference

System Tasks and Functions

```
always @cond1
  if(cond1)
    $display("cond1-->true");
always @cond2
  if(cond2)
    $display("cond2-->true");
  else
    $display("cond2-->false");
endmodule
```

Note that the first conditional statement in this example has no `else`; the second conditional statement does have an `else`.

The following example shows some of the results of simulating the code in the example just shown. The strings such as L15 and L16 that begin the lines of the `$settrace` output indicate the lines of source code that Verilog-XL executes. The word `CONTINUE` indicates that simulation activity begins again after a timing control. The output for L14 and L16 show the values as both 32-bit hexadecimal values and decimal values.

```
SIMULATION TIME IS 180
L15 "settrace": #5 >>> CONTINUE
L15 "settrace": ;
L16 "settrace": cond2 = ~cond2; >>> cond2 = 24'h0, 0;
L17 "settrace": end
L12 "settrace": always
L12 "settrace": #5
L23 "settrace": @cond2 >>> CONTINUE
L24 "settrace": if(cond2) >>> FALSE
L27 "settrace": $display("cond2-->false");
cond2-->false
L23 "settrace": always
L23 "settrace": @cond2
SIMULATION TIME IS 185
L12 "settrace": #5 >>> CONTINUE
L13 "settrace": begin
L14 "settrace": cond1 = ~cond1; >>> cond1 = 24'hfffffff, 16777215;
L15 "settrace": #5
L19 "settrace": @cond1 >>> CONTINUE
L20 "settrace": if(cond1) >>> TRUE
L21 "settrace": $display("cond1-->true");
cond1-->true
L19 "settrace": always
L19 "settrace": @cond1
SIMULATION TIME IS 190
L15 "settrace": #5 >>> CONTINUE
L15 "settrace": ;
L16 "settrace": cond2 = ~cond2; >>> cond2 = 24'hfffffff, 16777215;
L17 "settrace": end
L12 "settrace": always
L12 "settrace": #5
L23 "settrace": @cond2 >>> CONTINUE
L24 "settrace": if(cond2) >>> TRUE
L25 "settrace": $display("cond2-->true");
cond2-->true
L23 "settrace": always
L23 "settrace": @cond2
SIMULATION TIME IS 195
L12 "settrace": #5 >>> CONTINUE
```

Verilog-XL Reference

System Tasks and Functions

```
L13 "settrace": begin
L14 "settrace": cond1 = ~cond1; >>> cond1 = 24'h0, 0;
L15 "settrace": #5
L19 "settrace": @cond1 >>> CONTINUE
L20 "settrace": if(cond1) >>> SKIPPING
L19 "settrace": always
L19 "settrace": @cond1
SIMULATION TIME IS 200
```

When the L21 or L24 conditional statement evaluates `cond1` or `cond2` as TRUE because it is nonzero, Verilog-XL executes the L21 or L25 `$display` statement. The `$settrace` data shows the standard output. When the L24 conditional statement evaluates `cond2` as FALSE, Verilog-XL executes the L27 `$display` statement. The case of a FALSE value for `cond1` has a different result: `$settrace` generates the message SKIPPING. This SKIPPING message is the result of a false conditional statement with no `else`.

The following example includes a task, a function, and named blocks. The task, `delay_task`, (line 18) and the function, `delay_func`, (line 28) control the value of `temp2` (line 13). The value of `temp2` increases in a nonlinear fashion to a maximum of 10 as the value of the `for` loop variable `temp` (line 10) increments to a maximum of 5. The delay implemented with the value of `temp2` (line 14) affects the iteration of the `for` loop (line 10) that makes an assignment to the register `value`. The wire `delayed_value` receives the value of register `value` immediately in a net declaration assignment (line 5).

```
1 module top;
2 reg [3:0] value;
3 reg [3:0] temp;
4 reg [3:0] temp2;
5 wire [3:0] delayed_value=temp;
6
7 initial
8 begin :assign_block
9 $settrace;
10 for (temp=0; temp<6; temp=temp+1)
11 begin
12 value=temp;
13 delay_task(temp,temp2);
14 #temp2;
15 end
16 end
17
18 task delay_task;
19 input [3:0] a;
20 output [3:0] b;
21 begin :delay_block
22 if ((5>a)&&(a>2)) b=5;
23 else if (a >= 5) b=10;
24 else b=delay_func(a);
25 end
26 endtask
27
28 function [3:0] delay_func;
29 input [3:0] data;
30 begin :func_block
31 if (data<=1)delay_func=1;
32 else delay_func=data;
```

Verilog-XL Reference

System Tasks and Functions

```
33 end
34 endfunction
35
36 endmodule
```

Execution begins with the `for` loop on line 10 and proceeds line by line until the `delay_task` invocation on line 13. Execution of the `delay_task` begins in the named block `delay_block` on line 21. The conditional statement on lines 22 and 23 proves `FALSE`, and line 24 invokes the `delay_func` function.

Execution of the `delay_func` function begins in the named block `func_block` on line 30. The `if` component of the conditional statement on line 30 proves `TRUE`, which finishes significant activities in `func_block`. The trace moves to the end of the `func_block` on line 33 and then to the beginning of the `func_block` on line 28 and issues the message `>>>RETURNING` to indicate that the function has executed.

Execution returns to line 24 which invoked the `delay_func` function, and the `delay_task` output `b` receives the value of 1, completing significant `delay_task` activity. The trace moves to the end of the `delay_block` on line 25 and then to the beginning of the `delay_task` on line 18 and issues the message `>>>RETURNING` to indicate that execution of the task is complete. The delay implemented by `temp2` on line 14 now has a known value and executes, causing simulation time to advance to 1.

Execution continues after the delay, reaching the end of the `for` loop on line 15 and reverting to the beginning of the `for` loop.

The following output shows the beginning of the `$settrace` from simulating the code in the previous example:

```
L10 "$settrace": for(temp = 0; temp < 6; ) >>> temp = 32'h0, 0
L10 "$settrace": for(temp = 0; temp < 6; ) >>> TRUE
L11 "$settrace": begin
L12 "$settrace": value = temp; >>> value = 4'h0, 0;
L13 "$settrace": delay_task(temp, temp2);
L21 "$settrace": begin :delay_block
L22 "$settrace": if((5 > a) && (a > 2)) >>> FALSE
L23 "$settrace": if(a >= 5) >>> FALSE
L24 "$settrace": delay_func(a)
L30 "$settrace": begin :func_block
L31 "$settrace": if(data <= 1) >>> TRUE
L31 "$settrace": delay_func = 1;
L33 "$settrace": end :func_block
L28 "$settrace": delay_func; >>> RETURNING
L24 "$settrace": b = delay_func(a); >>> b = 4'h1, 1;
L25 "$settrace": end :delay_block
L18 "$settrace": delay_task; >>> RETURNING
L14 "$settrace": #temp2 >>> #4'h1, 1
SIMULATION TIME IS 1
L14 "$settrace": #temp2 >>> CONTINUE
L14 "$settrace": ;
L15 "$settrace": end
L10 "$settrace": for(; temp < 6; temp = temp + 1) >>> temp = 32'h1, 1
```

Note: No message shows the initiation of activity in the named block `assign_block` as the simulation begins. Messages that `$settrace` generates at the end of the simulation show the execution of `assign_block` after exiting from `assign_block`.

Saving and Restarting Simulations

The `$save`, `$incsave`, and `$restart` system tasks let you save the complete simulation data structure into a file. This file can be then reloaded at a later time, so that you can continue the simulation where it left off at the time it was saved. The `$save`, `$incsave`, and `$restart` system tasks are often used during large simulation runs to save checkpoint versions of the data structure at regular intervals. You can also use them to perform quick “try and see” experiments without having to repeat the entire simulation each time. The syntax for these system tasks is as follows:

```
$save("<name_of_file>");  
$incsave("<incremental_filename>");  
$restart("<name_of_file>");
```

All three system tasks take a filename as a parameter. That filename must be supplied as a string enclosed in quotation marks.

The `$save` system task saves the complete data structure into the host operating system file specified as a parameter. Interactive commands are also saved by the `$save` system task; thus, when you use `$restart` to restore a simulation data structure, you also replace the current set of commands with the saved set of commands.

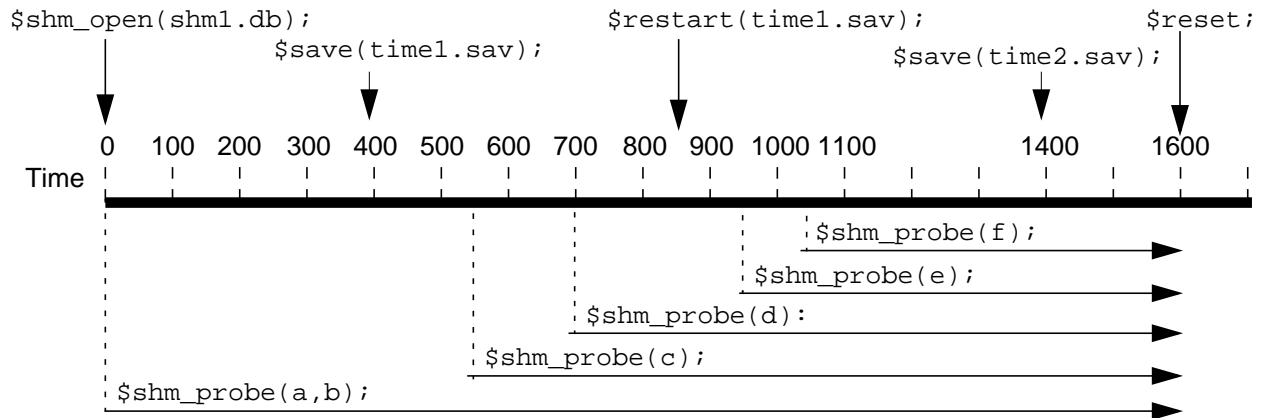
The `$incsave` system task marks the simulation time and saves only the information that has changed since the last invocation of `$save`. You supply a unique filename for each incremental save. An incremental save applies only to the file produced by the previous `$save`.

The `$restart` system task restores a previously saved data structure from a specified file. The data structure description to be restarted does not have to be related in any way to the description being replaced.

Verilog-XL Reference

System Tasks and Functions

The following figure and subsequent explanation show how `$save` and `$restart` work during a simulation:



1. At time 0, the `$shm_open` system task opens a waveform database called `shm1.db`.
2. At time 0, probes for signals `a` and `b` begin, recording them in `shm1.db`.
3. At time 400, the `$save` system task creates a Verilog save file called `time1.sav` where it stores a list of the current probes (`a` and `b`) and the simulation information.
4. At time 550, a probe for signal `c` begins, recording it in `shm1.db`.
5. At time 700, a probe for signal `d` begins, recording it in `shm1.db`.
6. At time 850, the `$restart` system task opens the `time1.sav` file to recall the state of the simulation at time 400.
 - ❑ Because signals `c` and `d` are being probed at the time the `$restart` system task is issued, they are added to the probe list.
 - ❑ The time in the waveform database (`shm1.db`) is set to time 400, and the waveform data that previously existed after time 400 is discarded.
 - ❑ When the simulation restarts at time 400, signals `a`, `b`, `c`, and `d` are probed.
7. At time 950, a probe for signal `e` begins, recording it in `shm1.db`.
8. At time 1050 a probe for signal `f` begins, recording it in `shm1.db`.
9. At time 1400, the `$save` system task creates a Verilog save file called `time2.sav` where it stores a list of the current probes (`a`, `b`, `c`, `d`, `e`, and `f`) and the simulation information.
10. At time 1600, `$reset` changes the clock to 0; simulation stops.

Verilog-XL Reference

System Tasks and Functions

To guard against host machine failures during simulation, use the following module in the main source description; it runs concurrently and independently of the rest of the description.

```
module checkpoint;
    always #100000 $save("run.dat");
endmodule
```

At any time after a failure, you can retrieve the simulation data structure and continue the simulation from the last checkpoint by issuing the following interactive commands:

```
C1> $restart("run.dat");
C2> .
```

Incremental Save and Restart

Restarting from an incremental save is similar to restarting from a full save, except that you specify the name of the incremental save file in the restart command. The full save file that the incremental save file is based upon must still be present, as it is required for a successful restart.

If the full save file has been changed in any way since the incremental save was performed, errors result. Many checks are performed during a restart to ensure that the data in the two files are consistent.

The incremental restart is useful for going back in simulation time. If a full save is performed near the beginning of simulation, and an incremental save is done at regular intervals, then going back in time is performed by restarting from the appropriate file.

The module shown in the following example saves the incremental state of the simulation every 10,000 time units. The files are recycled as time advances. Note that the first save is not performed at simulation time 0. This is because the incremental files are smaller if simulation is allowed to proceed before the first save is performed.

```
module checkpoint;
    initial
        #500 $save("save.dat");
    always
    begin
        #100000 $incsave("incl.dat");
        #100000 $incsave("inc2.dat");
        #100000 $incsave("inc3.dat");
        #100000 $incsave("inc4.dat");
    end
endmodule
```

Restarting from the `inc3.dat` files is performed as follows:

```
C23> $restart("inc3.dat");
```

Command-Line Restart

Another way to restart a simulation is to use the `-r` command-line option when you invoke Verilog-XL, as follows:

```
verilog -r run.dat
```

In this example, the simulation is restarted from the file `run.dat`, which contains a previously saved simulation data structure.

Note: When you restart a simulation with the `-r` (restart) option using a previously saved Verilog save file (instead of using the `$restart` system task), you must reprobe the signals you want to see.

Limitations for Saving and Restarting

The following are the limitations in saving and restarting a Verilog-XL simulation:

- You cannot save the state of simulation on a host of one type and then restart the simulation on a host of a different type.
- You cannot save the state of simulation using one version of Verilog-XL and then restart the simulation using another version.
- You cannot save the state of a simulation in batch mode and then restart the same simulation in the GUI mode or vice versa.

Command History

The `$history` system task prints out a list of all of the interactive commands that have been entered. The printout includes the number of each command so that you may use it in the re-execute and disable commands. The list also indicates which commands are active by flagging the command numbers with an asterisk (`*`) character. The syntax is as follows:

```
$history;
```

The following interactive dialog shows a sample command history printout:

```
C8> 1
    Command history:
    C1* $history;
    C2 $settrace;
    C3 $cleartrace;
    C4* forever
        @s85.i85.acc
        $stop;
    C5 force s85.ready = 0;
    C6 force s85.ready = 1;
```

```
C7 $save("85a.dat");
C8> -4
C8> 1
Command history:
C1* $history;
C2 $settrace;
C3 $cleartrace;
C4 forever
    @s85.i85.acc
    $stop;
C5 force s85.ready = 0;
C6 force s85.ready = 1;
C7 $save("85a.dat");
C8> $display("acc=%h", s85.i85.acc);
    acc=7f
C9> .
```

Command Input Files

The `$input` system task allows command input text to come from a named file instead of from the terminal. At the end of the command file or when an asynchronous interrupt is issued from the terminal, the input is automatically switched back to the terminal. The syntax is as follows:

```
$input("<filename>");
```

If an `$input` task is executed while a previous command file is being read, then the old file is closed and the new file is read in its place. At the end of the new file, input is switched back to the terminal, not to the old file.

Another way to specify a command input file is to use the command-line option `-i` when you first invoke Verilog-XL. The name of the input command file must follow the `-i` option. Again, at the end of this file, or when an asynchronous interrupt is issued from the terminal, the input is automatically switched back to the terminal.

Note: A previously generated key file containing asynchronous interrupt information cannot be read by the `$input` command because the interrupt information in the file will cause syntax errors. Such a file can only be read by using the `-i` command-line option at the beginning of a Verilog-XL run.

Log File

Each time Verilog-XL runs, a log file is created. 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. The syntax is as follows:

Verilog-XL Reference

System Tasks and Functions

```
$log("<filename>");  
$log;  
$nolog;
```

The `$nolog` and `$log` system tasks are used to disable and re-enable output to the log file. The `$nolog` task disables output to the log file, while the `$log` task re-enables the output. If you supply an optional filename to `$log`, then the old log file is closed, a new log file is created, and output is directed to the new log file.

The default log filename is `verilog.log`, which can be changed using the `-l` command option, as in the following example:

```
verilog src1.v -l src1.log
```

The preceding invocation of Verilog-XL changes the name of the log file to `src1.log`. Use this method to provide a unique name for each log file you intend to keep. Otherwise, with each new run, Verilog-XL overwrites any previously created log file that uses the default name `verilog.log`.

Key File

Verilog-XL creates a key file whenever it enters interactive mode for the first time. The key file contains all of the text that has been typed in from the standard input. The file also contains information about asynchronous interrupts so that, in conjunction with the `-i` input command file option, an exact simulation recovery is possible. See [Interactive Recovery](#) in the Verilog-XL User Guide for more details. The syntax is as follows:

```
$key("<filename>");  
$key;  
$nokey;
```

The `$nokey` and `$key` system tasks are used to disable and re-enable output to the key file. The `$nokey` task disables output to the key file, while the `$key` task re-enables the output. If you supply an optional filename to `$key`, then the old key file is closed, a new key file is created, and output is directed to the new file.

The default key filename is `verilog.key`, which can be changed by the `-k` command option, as in the following example:

```
verilog src1.v -k src1.key
```

The above invocation changes the name of the key file to `src1.key`. Use this method to provide a unique name for each key file you intend to keep. Otherwise, with each new run, Verilog-XL overwrites any previously created key file that uses the default name `verilog.key`.

Setting the Interactive Scope

The `$scope` system task lets you assign a particular level of hierarchy as the interactive scope for identifying objects. The `<name>` parameter must be the complete hierarchical name of a module, task, function, or named block. Once `$scope` is executed, you no longer need to reference the full hierarchical name of any object in or below the specified level of hierarchy. In subsequent interactive commands, these objects may be referenced in relation to the interactive scope established by `$scope`. The initial setting of the interactive scope is the first top-level module. The syntax is as follows:

```
$scope (<name>);
```

Showing the Hierarchy

The `$showscopes` system task produces a complete list of modules, tasks, functions, and named blocks that are defined *at the current scope level*. The parameter `n` is a switch. When it is nonzero, all of the modules, tasks, functions, and named blocks *in or below* the current hierarchical scope are listed. When `n` is zero, the operation is identical to `$showscopes` without a parameter—that is, only objects at the current scope level are listed. At the end of the list, the full hierarchical name of the current level is printed, along with a list of all the top level modules. This information is sufficient to allow traversal of the hierarchy either up or down by using the `$scope` system task.

The system task `$showallinstances` displays the number of instances of each module, gate, and primitive in the entire design hierarchy. Also, for modules and UDPs, this task identifies the name of the file that contains their definitions.

The syntax for the `$showscopes` and `$showallinstances` tasks is as follows:

```
$showscopes;  
$showscopes(n);  
$showallinstances;
```

Showing Variable Status

The `$showvars` system task produces status information for register and net variables, both scalar and vector. This information is very useful for examining the contribution of all the drivers on a net, especially when the net's value is `x`. When invoked without parameters, `$showvars` displays the status of all variables in the current scope. The syntax is as follows:

```
$showvars;  
$showvars(<list_of_variables>);  
$showvariables(control);  
$showvariables(control,<list_of_variables>);
```

Verilog-XL Reference

System Tasks and Functions

When invoked with a *<list_of_variables>*, `$showvars` shows only the status of the specified variables. If the *<list_of_variables>* includes a bit-select or part-select of a register or net, Verilog-XL displays status information for all the bits of that register or net. The information produced for each variable includes the following:

- name of variable
- scope of variable
- type of variable
- current value
- future value if scheduled
- whether the variable is forced
- decompilation of drivers, with output values
- future value of drivers, if scheduled

The system task `$showvariables` displays information similar to that of `$showvars`, but allows more control over the information displayed. The *control* parameter accepts an integer value from 0 to 7 and determines the amount of information displayed, as shown in the following table.

Control Value	Information Displayed
0	Display the same information as <code>\$showvars</code> (default).
1	Display all information for variables in or below the current scope.
2	Display all information except driver information for variables in, but not below, the current scope.
3	Display all information except driver information for variables in or below the current scope.
4	Display information for unknown variables in, but not below, the current scope.
5	Display information for unknown variables in or below the current scope.
6	Display all information except driver information for unknown variables in, but not below, the current scope.
7	Display all information except driver information for unknown variables in or below the current scope.

Showing Net Expansion Status

This system task lists all of the vector nets that have been expanded during compilation. The syntax is as follows:

```
$showexpandednets;
```

Showing Module Port Status

This system task lists all module ports that have not been collapsed during compilation. The syntax is as follows:

```
$showportsnotcollapsed;
```

Showing Number of Drivers

The `$countdrivers` system function is provided to count the number of drivers on a specified net so that bus contention can be identified. The syntax is as follows:

```
$countdrivers(net, net_is_forced, number_of_01x_drivers,  
             number_of_0_drivers, number_of_1_drivers, number_of_x_drivers);
```

This system function returns a 0 if there is no more than one driver on the net and returns a 1 otherwise (indicating contention). The net specified by *net* must be a scalar or a bit-select of an expanded vector net.

The number of parameters to the system function may vary according to how much information is desired. The *net* parameter is required; the rest are optional. Include a comma to hold the place of parameters you are not using. For example:

```
$countdrivers(net, , , number_of_0_drivers,  
             number_of_1_drivers, number_of_x_drivers);
```

The parameters *net_is_forced*, and *number_of_01x_drivers* are not used by this call to `$countdrivers`.

The optional parameters must be predefined so that `$countdrivers` can assign values to them. The parameter *net_is_forced*, if supplied, returns a scalar 1 or 0 and may be predefined as a `reg` variable. The parameters *number_of_01x_drivers*, *number_of_0_drivers*, *number_of_1_drivers*, and *number_of_x_drivers* should be predefined as `integer` variables.

If you supply additional parameters to the `$countdrivers` function, each parameter returns the information described in the following table.

Parameter	Return Value
<code>net_is_forced</code>	Returns a 1 if the net is forced and a 0 if the net is not forced.

Verilog-XL Reference

System Tasks and Functions

Parameter	Return Value
<code>number_of_01x_drivers</code>	Returns an integer representing the number of drivers that are in a 0, 1, or x state; this represents the total number of drivers on the net that are not forced.
<code>number_of_0_drivers</code>	Returns an integer representing the number of drivers on the net that are in the 0 state.
<code>number_of_1_drivers</code>	Returns an integer representing the number of drivers on the net that are in the 1 state.
<code>number_of_x_drivers</code>	Returns an integer representing the number of drivers on the net that are in the x state.

The following example shows how you can use `$countdrivers` to determine the drivers of a net:

```
module two_drivers (node, ina, inb);
output node;
input ina, inb;
buf a (node, ina);           // buffers a and b
buf b (node, inb);          // both drive output node\
endmodule

module drive_contention;
reg contention;
reg is_forced;
integer driver_total, d0, d1, dx;
reg ina, inb;

two_drivers c (node, ina, inb);

always @ (ina or inb)
  if (node === 1'bx)
    begin          // contention is set to 1 if there are
                  // two or more drivers, and 0 otherwise
      contention = $countdrivers (node, is_forced,
                                driver_total,      d0, d1, dx);
      if (contention)
        $display("Contention---\n",
                "is_forced: %b\n", is_forced,
                "driver_total: %0d\n", driver_total,
                "num_0_drivers: %0d\n", d0,
                "num_1_drivers: %0d\n", d1,
                "num_x_drivers: %0d\n", dx);
    end

initial $monitor("time: %0d", $time, "node:%b", node,
                "ina:%b", ina, "inb:%b", inb);

initial
  begin
    #10 ina = 1'b1;
    #10 inb = 1'b0;
    #10 ina = 1'b0; inb = 1'b1;
  end
endmodule
```

Verilog-XL Reference

System Tasks and Functions

```
end  
endmodule
```

The following output comes from running the code in the previous example:

```
Compiling source file "contention.v"  
Highest level modules:  
drive_contention
```

```
time: 0 node:x ina:x inb:x  
Contention---  
is_forced: 0  
driver_total: 2  
num_0_drivers: 0  
num_1_drivers: 0  
num_x_drivers: 2
```

```
time: 10 node:x ina:1 inb:x  
Contention---  
is_forced: 0  
driver_total: 2  
num_0_drivers: 0  
num_1_drivers: 1  
num_x_drivers: 1
```

```
time: 20 node:x ina:1 inb:0  
Contention---  
is_forced: 0  
driver_total: 2  
num_0_drivers: 1  
num_1_drivers: 1  
num_x_drivers: 0
```

```
time: 30 node:x ina:0 inb:1  
49 simulation events
```

Displaying the Delay Mode

The `$showmodes` system task displays the delay modes in effect for particular modules during simulation. When invoked with a non-zero constant argument, `$showmodes` displays the delay modes of the current scope, as well as the delay modes of all module instances beneath it in the hierarchy. If a zero argument or no argument is supplied to `$showmodes`, the system task displays only the delay mode of the current scope. The syntax is as follows:

```
$showmodes;  
$showmodes(<non_zero_constant>);
```

Storing Interactive Commands

The `$keepcommands` and `$nokeepcommands` system tasks control whether or not Verilog-XL saves interactive commands in its history stack. You can access this stack with the `$history` system task. The system task `$nokeepcommands` tells Verilog-XL to add no more interactive commands to its history stack. The system task `$keepcommands` tells

Verilog-XL to add all subsequent interactive commands to its history stack. The syntax is as follows:

```
$keepcommands;  
$nokeepcommands;
```

Interactive Source Listing—Decompilation

Two system tasks produce a line-numbered listing of your source description. Producing this listing of your source description is called decompilation. These system tasks are `$list` and `$listcounts`. A third system task, `$list_forces`, lists the currently active `force` statements.

\$list

When invoked without a parameter, `$list` produces a listing of the module, task, function, or named block that is defined as the current scope setting. When an optional parameter `i` is supplied, it must refer to a specific module, task, function or named block, in which case, the specified object is listed. The syntax is as follows:

```
$list;  
$list (<name>);
```

The listing provides, apart from the source text itself, the following information:

- References to the original source file line numbers
- An asterisk (*) appearing next to the line number of any source line that has a current simulation event associated with it
- The current value of declared data items listed as a comment after the declared name

\$listcounts

The `$listcounts` system task is an enhancement of `$list`. The `$listcounts` system task is disabled by default to accelerate simulation. You must include the `+listcounts` option on the command line to enable the task, unless the `+no_speedup` option is specified on the command line. The task produces a line-numbered source listing that includes an execution count, which is the number of times Verilog-XL has executed the statements in each line thus far in the simulation. The syntax is as follows:

```
$listcounts;  
$listcounts(<hierarchical_name>);
```

The `$listcounts` system task takes an optional hierarchical name argument. If you do not include an argument, `$listcounts` produces a listing of the source description at the scope level from which you called the task.

`$list_forces`

The `$list_forces` system task lists the currently active `force` statements. The list produced by the `$list_forces` system task shows `force` statements with the full hierarchical names of the nets and registers subject to the `force` statements. The syntax is as follows:

```
$list_forces;  
$list_forces("<file_name>");
```

The following example shows an interactive `force` statement and the `$list_forces` system task:

```
C1 > force d = 1;  
C2 > $list_forces;  
force top.d = 1;
```

The first interactive entry is a `force` statement that forces `d` to 1. The list produced by the `$list_forces` system task shows the full hierarchical name of `d`.

If you are making a design work correctly by adding `force` statements to it, the `$list_forces` and the `$reset` system tasks provide the following efficient method:

1. View all the currently active `force` statements with the `$list_forces` task to understand changes or additions that the design requires in the `force` statements.
2. Reset the simulation to time 0 with the `$reset` system task.
3. Enter the changes or additions to the `force` statements.
4. Simulate again.

The `$list_forces` system task does not show inactive `force` statements or the `release` statements that made them inactive unless a `force` statement has assigned a value to a concatenation of nets or registers and a subsequent `release` statement has inactivated the `force` statement on one or more of the nets or registers in the concatenation.

The following example shows an interactive `force` statement that forces a value onto a concatenation of registers, a `release` statement that releases the `force` statement on one of the registers, and the `$list_forces` system task:

```
C1 > force {a,b,c} = 3'b111;  
C2 > #100 $stop  
C3 > .  
C2: $stop at simulation time 210
```

Verilog-XL Reference

System Tasks and Functions

```
C3 > release b;
C4 > #100 $stop;
C5 > .
C4: $stop at simulation time 310
C5 > $list_forces;
force {top.a, top.b, top.c} = 3'b111;
release top.b;
```

You can specify a filename to which Verilog-XL writes the list of active `force` statements. Verilog-XL formats this file so that you can input its contents with the `$input` system task. You can, for example, input this file after you reset Verilog-XL to apply these `force` statements before the simulation begins again. The following example shows an interactive `force` statement, a `$list_forces` system task, a `$reset` system task, and the application of the `force` statement before the simulation begins again:

```
C1 > force d = 1;
C2 > $list_forces("forces.cmd");
C3 > $reset;
C3: $reset at simulation time 110
C4 > $input("forces.cmd");
C5 > force top.d = 1;
C6 >
```

The `$list_forces` system task takes an argument. This argument can be a character string that specifies a filename or a constant that is an ASCII value. You enclose a filename in quotation marks. In the above example, Verilog-XL writes the active `force` statement to `reg d` to a file named `forces.cmd`. The `$reset` system task resets the simulation time to 0, returns `d` to its initial value, and resumes the interactive mode. The `$input` system task tells Verilog-XL to execute the `force` statement in `forces.cmd`.

By design, the `$list_forces` task does not list `force` statements written in protected code. As a result, `$list_forces` does not always list all the `force` statements in effect at the time at which it is issued.

The `$list_forces` task does not list any `force` or `release` statements for a concatenation subject to a `force` statement if all members of the concatenation have been released. In such a case the `$list_forces` task without an argument lists the currently active `force` statements both in the log file and in the standard output.

Note: If you issue `force` statements interactively, but you have `$nokeepcommands` in effect, the behavior of `$list_forces` is unpredictable.

Disabling and Enabling Warnings

The `$disable_warnings` and `$enable_warnings` system tasks make it possible to suppress or generate warning messages. See [Appendix H, Verilog-XL Messages](#), of the *Verilog-XL User Guide* for more information about messages.

\$disable_warnings

The `$disable_warnings` system task tells Verilog-XL to stop displaying warnings about the following:

- Timing check violations, which Verilog-XL displays by default
- The triregs that acquire a value of `x` due to charge decay, which Verilog-XL displays by default

The syntax is as follows:

```
$disable_warnings;  
$disable_warnings (<keyword>"?<, <module_instance>>*?);
```

This system task's arguments are optional. The task can have only one `keyword` argument, but it can have any number of `module_instance` arguments.

In this example, Verilog-XL disables all timing check violation messages and all `trireg` charge decay notifications for the rest of the simulation, unless the `$enable_warnings` system task (see the following section) reenables them.

The keyword argument

Adding an argument to the `$disable_warnings` system task specifies the type of warning that Verilog-XL disables. The following table shows the valid arguments:

"decay"	Specifies that you want to disable charge decay warnings
"timing"	Specifies that you want to disable timing check violation warnings

You must enclose the keyword in quotation marks. The following example shows the use of the `decay` keyword argument:

```
$disable_warnings("decay");
```

In this example, Verilog-XL stops displaying notifications of `trireg` charge decay but continues to display warnings about timing check violations.

The module instance argument

An argument of one or more module instance names, or the top-level module name placed after the keywords in the previous table, specifies where in the source description Verilog-XL disables warnings. When you enter a `<module_instance>` argument Verilog-XL displays no warnings of the type designated by the `keyword` argument in the specified module

Verilog-XL Reference

System Tasks and Functions

instances in the module hierarchy. The following example shows a use of the module instance argument:

```
$disable_warnings ("decay",precharge,dram2);
```

In this example, Verilog-XL displays no charge decay warnings in the modules with the instance names `precharge` or `dram2`, or in the module instances hierarchically below `precharge` or `dram2`.

Note: Port collapsing can disable warnings of charge decay for a `triereg` that is not in the specified module. The disabling of charge decay warnings in a module and the collapsing of that module's ports can result in the disabling of charge decay warning messages about a `triereg` that is the external net connected by the collapsed port.

\$enable_warnings

The `$enable_warnings` system task without arguments enables the display of the warnings listed in the previous table.

The `$enable_warnings` system task with appended arguments enables the display of any warning that you disabled with the `$disable_warnings` system task. The `$enable_warnings` system task takes a module instance name or the top-level module name as arguments. The syntax is as follows:

```
$enable_warnings (<"keyword">?<,<module_instance>>*?);
```

For example, the following line enables notifications of `triereg` charge decay for the entire simulation:

```
$enable_warnings ("decay");
```

The following example shows uses of the `$enable_warnings` system task:

```
$disable_warnings ("decay");  
$enable_warnings ("decay",dram_bitcell);
```

The `$disable_warnings` system task tells Verilog-XL not to display charge decay warnings about `triereg`s in the source description; the `$enable_warnings` system task tells Verilog-XL to make an exception and to display charge decay warnings about all `triereg`s in the module instance named `dram_bitcell` and about all `triereg`s in the modules instantiated hierarchically below that module instance.

Loading Memories from Text Files

Two system tasks—`$readmemb` and `$readmemh`—read and load data from a specified text file into a specified memory. Either task may be executed at any time during a simulation. The syntax is as follows:

```
$readmemb("<filename>", <memname>);  
$readmemb("<filename>", <memname>, <start_addr>);  
$readmemb("<filename>", <memname>, <start_addr>, <finish_addr>);  
$readmemh("<filename>", <memname>);  
$readmemh("<filename>", <memname>, <start_addr>);  
$readmemh("<filename>", <memname>, <start_addr>, <finish_addr>);
```

The text file to be read must contain only the following:

- white space (spaces, new lines, tabs, and form-feeds)
- comments (both types of comments are allowed)
- binary or hexadecimal numbers

The numbers must have neither the length nor the base format specified. For `$readmemb`, each number must be binary. For `$readmemh`, the numbers must be hexadecimal. The unknown value (`x` or `X`), the high impedance value (`z` or `Z`), and the underscore (`_`) can be used in specifying a number as in a Verilog-XL source description (see [Chapter 2, “Lexical Conventions.”](#)). White space and/or comments must be used to separate the numbers.

In the following description, the term “address” refers to an index into the array that models the memory.

As the file is read, each number encountered is assigned to a successive word element of the memory. Addressing is controlled both by specifying start and/or finish addresses in the system task invocation, and by specifying addresses in the data file.

When addresses appear in the data file, the format is an “at” character (`@`) followed by a hexadecimal number as follows:

```
@hh...h
```

Both upper- and lower-case digits are allowed in the number. No white space is allowed between the `@` and the number. You may use as many address specifications as you need within the data file. When the system task encounters an address specification, it loads subsequent data starting at that memory address.

If no addressing information is specified within the system task, and no address specifications appear within the data file, then the default start address is the left-hand address given in the declaration of the memory, and consecutive words are loaded until either the memory is full or the data file is completely read. If the start address is specified in the task without the finish

Verilog-XL Reference

System Tasks and Functions

address, then loading starts at the specified start address and continues towards the right-hand address given in the declaration of the memory.

If both start and finish addresses are specified as parameters to the task, then loading begins at the start address and continues towards the finish address, regardless of how the addresses are specified in the memory declaration.

When addressing information is specified both in the system task and in the data file, the addresses in the data file must be within the address range specified by the system task parameters. Otherwise, an error message is issued, and the load operation is terminated.

A warning message is issued if the number of data words in the file differs from the number of words in the range implied by the start and finish addresses.

For example, consider the following declaration of memory `mem`:

```
reg[7:0] mem[1:256];
```

Given this declaration, each of the following statements will load data into `mem` in a different manner:

```
initial $readmemb("mem.data", mem);  
initial $readmemb("mem.data", mem, 16);  
initial $readmemb("mem.data", mem, 128, 1);
```

The first statement loads up the memory at simulation time 0 starting at the memory address 1. The second statement begins loading at address 16 and continues on towards address 256. For the third and final statement, loading begins at address 128 and continues down towards address 1.

In the third case, when loading is complete, Verilog-XL performs a final check to ensure that exactly 128 numbers are contained in the file. If the check fails, the simulator issues a warning message.

Note: Errors or warnings that result from the execution of the system tasks `$readmemb` and `$readmemb` do not terminate simulation.

Setting a Net to a Logic Value

The `$deposit` system task allows you to set a net to a particular value and then to simulate with the net set to that new value. The value change is propagated throughout the nets and registers being driven by the variable that has been set. The syntax is as follows:

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

Verilog-XL Reference

System Tasks and Functions

The `$deposit` task can be used within any Verilog-XL procedural block. You can define the time at which the net is to be given a new value using the standard procedural constructs. The task can also be used on the interactive command line.

Use this system task as a debugging or design initialization aid. You should not use it as a representation of actual circuitry.

Common uses for the `$deposit` system task include the following:

- To initialize large portions or all of a circuit either at the beginning of or during a simulation. You can select the nodes to be deposited to yourself, or use PLI code to extract the node names.
- To stop the simulator during a debugging session and to use the command on the interactive command line to set a new value.
- To reset a circuit to a known state after simulation in order to retry a different debug route.
- To set parts of a circuit to analyze intricate circuit details (common for switch level simulation).
- To break feedback loops to set them to a known state.

In the syntax, *variable* is the name of the net or register whose value is being changed. The variable can be a net or register type but not a parameter, and it can be a vector or scalar object that can be expanded or compacted.

The second parameter, *value*, is a numerical or logical value in standard Verilog-XL notation. Bit and part selects are not allowed.

If the width of the value is smaller than the range of the variable, an error message is generated. If the width of the value is larger than the range of the variable, the MSBs are truncated and a warning is issued.

X and Z states can also be deposited.

Here are some examples of using `$deposit`:

```
$deposit(sig, 1);  
$deposit(bus, 'hA2);  
$deposit(bus4, 'bZ01x);
```

Fast Processing of Stimulus Patterns

The system function `$getpattern` provides for the fast processing of stimulus patterns that must be propagated to a large number of scalar inputs. The function reads stimulus patterns

Verilog-XL Reference

System Tasks and Functions

that have been loaded into a memory using the `$readmemb` or `$readmemh` system tasks. The syntax is as follows:

```
$getpattern (<mem_element>);
```

The use of this function is limited; it may only be used in a continuous assignment statement in which the left-hand side is a concatenation of scalar nets, and the parameter to the system function is a memory element reference. There is no limit on the number of `$getpattern` functions in a simulation, but only one `$getpattern` function is permissible per memory instance.

The following example shows how stimuli stored in a file can be read into a Verilog-XL memory using `$readmemb` and applied to the circuit, one pattern at a time, using `$getpattern`:

```
module top;
  parameter in_width=10,
    patterns=200,
    step=20;
  reg [1:in_width] in_mem[1:patterns];
  integer index;
  // declare scalar inputs
  wire i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;
  // assign patterns to circuit scalar inputs (a new pattern
  // is applied to the circuit each time index changes value)
  assign {i1,i2,i3,i4,i5,i6,i7,i8,i9,i10}
    = $getpattern(in_mem[index]);
  initial
  begin
    // read stimulus patterns into memory
    $readmemb("patt.mem", in_mem);
    // step through patterns (note that each assignment
    // to index will drive a new pattern onto the circuit
    // inputs from the $getpattern system task specified
    // above
    for(index = 1; index <= patterns; index = index + 1)
      #step;
  end
  // instantiate the circuit module
  mod1 cct(o1,o2,o3,o4,o5,o6,o7,o8,o9,o10,
    i1,i2,i3,i4,i5,i6,i7,i8,i9,i10);
endmodule
```

In the above example, the memory `in_mem` is initialized with the stimulus patterns by the `$readmemb` task. The integer variable `index` selects which pattern is being applied to the circuit. The `for` loop increments the integer variable `index` periodically to sequence the patterns.

Incremental Pattern File Tasks

One of the system tasks that this section discusses stores selected simulation events in a special purpose file called an incremental pattern file. Other system tasks that this section discusses manipulate incremental pattern files for the following purposes:

- Stimulus for simulations
- Comparison of simulation results during a time window
- Comparison of simulation results at the end of a time unit

\$incpattern_write

The `$incpattern_write` system task writes an incremental pattern file that records the value changes of specified nets or registers during simulation. The file is called an incremental pattern file due to its particularly compact format, which stores only changes in the values of arguments at the end of each time unit and ignores redundant information. The size of the file is limited only by the available disk space.

The syntax is as follows:

```
$incpattern_write(<filename>,<list_of_variables>);
```

The `<filename>` is the name of the file that stores the value changes on items in the `<list_of_variables>`. The `<list_of_variables>` in the `$incpattern_write` syntax is a list of nets or registers separated by commas. The variables can be either scalars or vectors.

The system task writes value changes and the simulation time at the end of each time unit. The task records only the final values of variables.

Under some circumstances, a simulation is more efficient if it contains more than one `$incpattern_write` task. Therefore, if a task is writing more than 64 variables, writing two files results in faster simulation than writing one file. A single file holds more than 64 variables, but in such a case, the first 64 signals should be the most active signals. Writing separate files for stimulus and response data can simplify the use of incremental pattern file tasks other than `$incpattern_write`.

Multiple `$incpattern_write` tasks can write files simultaneously.

The following example illustrates a way to terminate the task with the `disable` command:

```
...
task writer;
begin
    $incpattern_write(.....);
```

```
end
endtask
...
initial
begin
    writer;
    #1000 disable writer;
end
```

Distinguishing between input and output signals on inouts while writing incremental pattern files is one of the topics discussed in [“Examples of Response Checking”](#) on page 378, which presents a general usage example for the `$incpattern_write`, `$incpattern_read`, and `$compare` system tasks documented in this chapter.

`$incpattern_read`

The `$incpattern_read` task reads an incremental pattern file and places values in that file on expanded or accelerated nets. The syntax is as follows:

```
$incpattern_read(<filename>, <eof>, <list_of _variables>);
```

- `<filename>` is the name of the incremental pattern file read by this system task.
- `<eof>` is a notifier register you declare, which toggles to signal the task's completion.
- The `<list of variables>` is a list of nets, scalar or vectored, separated by commas.

The `$incpattern_read` task reads the values of the variables in the `<list_of_variables>` for the `$incpattern_write` task that created an incremental pattern file. It applies those values to its own `<list_of_variables>`, matching the positions of variables in the two lists to determine which values to apply. The number of entries in the two lists must be the same, and entries holding matching positions in the two lists must have identical bit-widths.

The `$incpattern_read` task places values on nets named for the variables in its `<list_of_variables>`. You can think of these nets as constituting the fanout of the system task. These nets typically connect to a design or to the `$compare` or `$strobe_compare` system tasks. Any Verilog monitoring system task can monitor the nets in the task's fanout.

The value changes on the task's fanout nets occur at the same simulation times that they did in the simulation that provided data to `$incpattern_write`.

Verilog-XL Reference

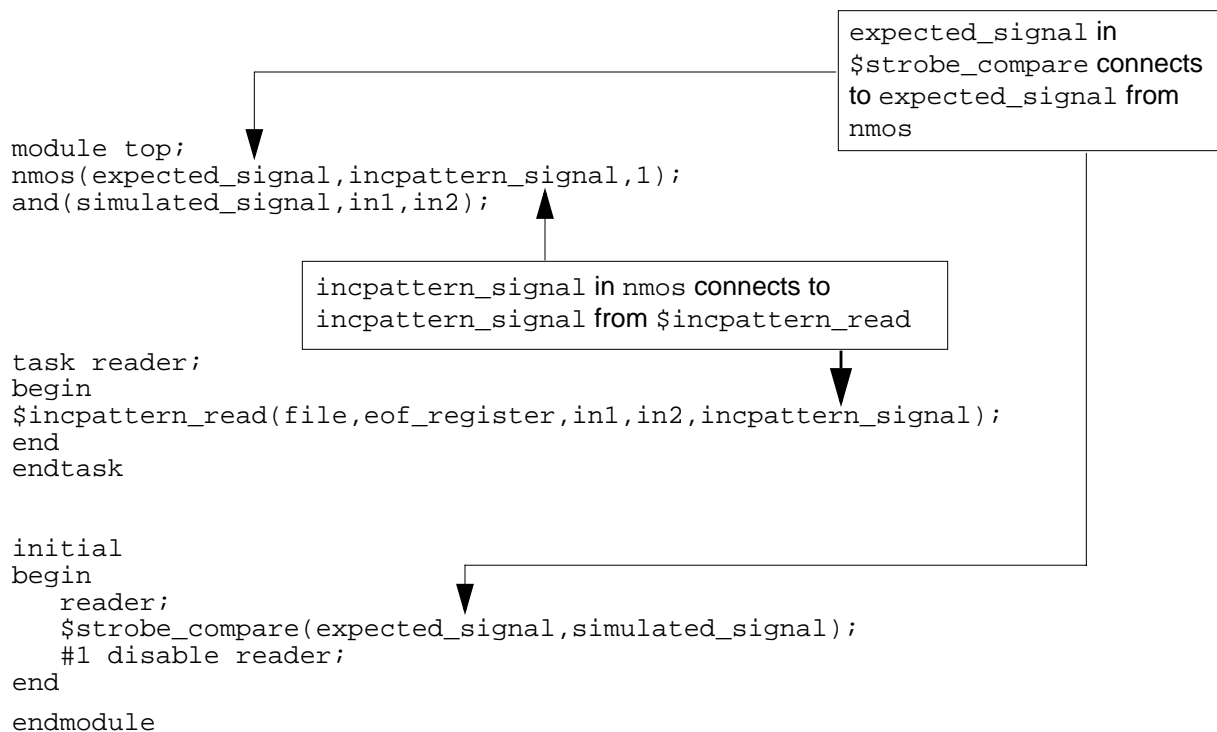
System Tasks and Functions

The following table shows the values of the `<eof>` notifier that toggles to indicate the task's completion.

Notifier Values	
BEFORE completion	AFTER completion
x	1
0	1
1	0
z	z

Fanout from the `$incpattern_read` task to the `$compare` or `$strobe_compare` system tasks must be scalared vector nets or scalar nets.

The following example illustrates the easiest way to treat the `$incpattern_read` fanout, which is buffering each output bit of the `$incpattern_read` task with an `nmos` gate:



Declaring the control input of the `nmos` as 1 makes the gate function as a buffer. The incremental pattern file, `file`, serves as a source of stimulus and as a source of values for comparisons that `$strobe_compare` makes.

This example shows how the `disable` command can terminate the task in the same way that it terminates the `$incpattern_write` task in the example in “[\\$incpattern_write](#)” on page 372.

See “[Examples of Response Checking](#)” on page 378 for an illustration of the use of the `$incpattern_read`, `$incpattern_write`, and `$compare` system tasks documented in this section.

\$compare

The `$compare` system task issues a discrepancy message if an expected value and a simulated value differ during the time window in which the task runs. The time window can be of any length. The syntax is as follows:

```
$compare(<enable>, <expected_value>, <simulated_value>  
        <,<expected_value>, <simulated_value>>*) ;
```

- `<enable>` is a Verilog expression that enables the `$compare` task when its value equals TRUE or 1, and disables the task when it has another value. Declaring it as a null enables the task throughout simulation; it can not be a constant.
- `<expected_value>` and `<simulated_value>` are the first and second members of a pair of equal bit-width scalar or vectored nets that includes the `<simulated_value>` as its second member.

The nets that carry expected values into the task can be the fanout of the `$incpattern_read` task, in which case, the nets must be expanded vector nets or scalar nets. The example in [\\$incpattern_read](#) shows how to connect the `$strobe_compare` task to the `$incpattern_read` task; the considerations and methods are identical for connecting the `$compare` task to the `$incpattern_read` task.

The nets which carry simulated values into the task connect to models. There is no limit on the number of pairs of expected and simulated values that the task compares.

The task perceives a discrepancy between an expected value and a simulated value in the circumstances in the following table:

Expected Value	Simulated Value
0	Non-0

Verilog-XL Reference

System Tasks and Functions

Expected Value	Simulated Value
1	Non-1
z	Non-Z

When the simulator finds a discrepancy between an expected value and a simulated value it displays a discrepancy message. The discrepancy message includes the following information:

- Name of the file that contains the `$compare` task invocation that generates the discrepancy message
- Line number that contains the `$compare` task invocation that generates the discrepancy message
- Instance name of the module that contains the `$compare` task invocation that generates the discrepancy message
- Time at which the discrepancy occurred
- Expected and simulated values

An invocation of the `$compare` task, and the discrepancy message that the task can produce while responding to that invocation appear in the following example:

```
initial
$compare(enable, bgrant_exp, bgrant_sim, back_exp, back_sim);
"src.v", 15: Simulated value does not match Expected value in top.block.lcpu
Expected bgrant_exp: 0, Simulated bgrant_sim: 1 at time 32450
```

When the task finds a discrepancy on a vector, it reports only one discrepancy message for the entire bus. The discrepancy message in such a case presents both the expected and the simulated values for the entire bus as hexadecimal numbers.

[“Examples of Response Checking”](#) on page 378 illustrates the `$compare`, `$incpattern_write`, and `$incpattern_read` system tasks documented in this section.

\$strobe_compare

The `$strobe_compare` system task issues a discrepancy message if a value produced by a simulation and a corresponding expected value differ at the end of the single time unit in which the task runs. The syntax is as follows:

```
$strobe_compare(<expected_value>, <simulated_value>
<, <expected_value>, <simulated_value>>*) ;
```


Verilog-XL Reference

System Tasks and Functions

- `<expected_value>` is the first member of a pair of equal bit-width scalar or vectored nets that includes the `<simulated_value>` as its second member.
- `<simulated_value>` is the second member of a pair of equal bit-width scalar or vectored nets that includes the `<expected_value>` as its first member.

The nets that carry expected values into the task can be the fanout of the `$incpattern_read` task, in which case the nets must be expanded vector nets or scalar nets. The example in “[\\$incpattern_read](#)” on page 373 shows how to connect the `$strobe_compare` task to the `$incpattern_read` task.

The nets that carry simulated values into the `$strobe_compare` task connect to models. There is no limit on the number of pairs of expected and simulated values.

The `$strobe_compare` task finds a discrepancy between an expected value and a simulated value in the circumstances in the following table:

Expected Value	Simulated Value
0	Non-0
1	Non-1
z	Non-Z

When the simulator finds a discrepancy between an expected value and a simulated value, it displays a discrepancy message. The discrepancy message includes the following information:

- Name of the file that contains the `$strobe_compare` task invocation that generates the message
- Line number that contains the `$strobe_compare` task invocation that generates the message
- Instance name of the module that contains the `$strobe_compare` task invocation that generates the message
- Time at which the discrepancy occurred
- Expected and simulated values

An invocation of the `$strobe_compare` task and a discrepancy message that the task can produce while responding to that invocation appear in the following example:

```
always @CLK #(PERIOD-1)
  $strobe_compare( bgrant_exp, bgrant_sim, back_exp, back_sim);
```

Verilog-XL Reference

System Tasks and Functions

```
"src.v", 15: Simulated value does not match Expected value in top.block.lcpu
Expected bgrant_exp: 0, Simulated bgrant_sim: 1 at time 32450
```

When the task finds a discrepancy on a vector, it reports only one discrepancy message for the entire bus. The discrepancy message in such a case presents both the expected and simulated values for the entire bus as hexadecimal numbers.

Examples of Response Checking

Ascertaining whether or not the responses of a gate-level implementation are identical to the responses of a behavioral model is an application that calls for the `$incpattern_write`, `$incpattern_read`, and `$compare` tasks. The following example shows a system-level simulation. The top-level module is named `system`. An ASIC described at the RTL level bearing the name `asic1_rtl_level a1` is instantiated in the top-level module `system`. The purpose of the simulation is to capture the stimulus and response of `asic1_rtl_level a1` in order to verify a gate-level implementation of `asic1_rtl_level a1` that appears as an instantiation named `asic_gate_level a1`.

Example 1

```
module system;
    asic1_rtl_level a1 (in1,out1,io);
    asic2_rtl_level a2 (in2,out2,io);
    initial $incpattern_write("ASIC1_PAT1",in1,out1,io,a1.io_control);
endmodule;
```

In addition to the signals on the ports of `asic1_rtl_level a1`, the simulation also captures the signal `a1.io_control`, which determines whether the inout `io` is serving as an input or an output of `asic1_rtl_level a1`. When the inout `io` is serving as an input, `a1.io_control` has a value of 1, which enables either a `bufif1` or its behavioral equivalent to place the value of a driver or a register on `io`. When `a1.io_control` has a value of 0, `io` serves as an output of `a1`.

The signal `a1.io_control` is read as `control` by the `$incpattern_read` task in [Example 2](#) because `control` is the last entry in that task's list, and because `a1.io_control` is the last entry in the `$incpattern_write` task's list in [Example 1](#).

Example 2

```
module system;
    reg EOF;
    wire in1,out1,io,control,io_bus,io_exp,io_sim;
    nmos (compare_out1_exp,out1_exp,1),(io,io_bus,control);
    pmos (io_exp,io_bus,control),(io_sim,io,control);
    asic_gate_level a1 (in1,out1,io);
    initial
    $incpattern_read("ASIC1_PAT1", EOF,in1,out1_exp,io_bus,control);
    initial @EOF $finish(2);
endmodule;
```

Verilog-XL Reference

System Tasks and Functions

```
initial
$compare(,compare_out1_exp,out1,io_exp,io_sim);
endmodule
```

This example is explained as follows:

- The register `EOF` declared is part of the mechanism that ends the simulation.
- The wires and gates declared connect the tasks to each other or connect the gate-level implementation to the tasks.
- The signal `in1` in the list of the `$incpattern_read` task applies stimulus from the incremental pattern file to the gate-level model.
- The signal `out1` in [Example 1](#) corresponds to `out1_exp` in the `$read` task's list in [Example 2](#). The signal `out1_exp` passes through the first of two buffering `nmos` gates and becomes `compare_out1_exp` in the `$compare` task's list. The `compare_out1_exp` variable forms a pair with the `out1` output of the gate-level model for comparison by the `$compare` task.
- The variable `control` in the `$read` task's list corresponds to the signal `a1.io_control` in [Example 1](#). The `control` variable controls the second of the two `nmos` gates and the two `pmos` gates. The `nmos` gate places the value stored as `io` in [Example 1](#) on the `io` inout of the gate-level model under test when `control` is 1.
- When `control` is 0, the two `pmos` gates propagate signals. The first `pmos` gate passes the value of `io` in [Example 1](#), which corresponds to the variable `io_bus` in the `read` task's list of variables. The value appears as `io_exp` in the variable list of the `$compare` task. Simultaneously, the second `pmos` gate passes the value on the `io` inout port of the gate-level model under test to the variable `io_sim` in the `$compare` task's variable list. The `$compare` task compares `io_exp` and `io_sim`.

Using gates to make the connections between incremental pattern tasks and models enables the tasks to function properly, and it makes simulations more efficient.

The following line begins the `$compare` task without delay because the first entry in the `$compare` task's list enables the task continuously when the entry is a null.

```
$compare(,out1_exp,out1,io_exp,io_sim);
```

The following line ends the simulation when the `$incpattern_read` task toggles the notifier.

```
initial @EOF $finish(2);
```

Functions and Tasks for Reals

The following functions handle real values:

<code>\$rtoi</code>	Converts real values to integers by truncating the real value. (For example, 123.45 becomes 123.)
<code>\$itor</code>	Converts integers to real values. (For example, 123 becomes 123.0.)
<code>\$realtobits</code>	Passes bit patterns across module ports; converts from a real number to the 64-bit representation (vector) of that real number.
<code>\$bitstoreal</code>	The reverse of <code>\$realtobits</code> ; converts from the bit pattern to a real number.

The following example shows how the `$realtobits` and `$bitstoreal` functions are used in port connections:

```
module driver (net_r);
    output net_r;
    real r;
    wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
    input net_r;
    wire [64:1] net_r;
    real r;
    initial assign r = $bitstoreal(net_r);
endmodule
```

Functions and Tasks for Timescales

The following are the system tasks and functions that support timescales. For more information about these constructs see [Chapter 17, “Timescales.”](#)

The following are timescale functions:

<code>\$time</code>	Returns an integer that is a 64-bit time, scaled to the time unit of the module that invoked it.
<code>\$realtime</code>	Returns a real number that is scaled to the time unit of the module that invoked it.
<code>\$scale</code>	Allows you to take a time value from a module with one time unit and use it in a module with a different time unit. The time value is converted from the time unit of one module to the time unit of the module that invokes <code>\$scale</code> .

Verilog-XL Reference

System Tasks and Functions

The following are timescale system tasks:

`$printtimescale` Displays the time unit and precision of a particular module. The syntax is as follows:

```
$printtimescale(<hierarchical_path_name>?);  
$timeformat(<units_number>, <precision_number>,  
            <suffix_string>, <minimum_field_width>);
```

`$timeformat` Specifies the following:

- the way that the `%t` format specification reports time information
- the time unit for delays entered interactively

Protecting Data in Memory

The system tasks `$sreadmemb` and `$sreadmemh` load data into memory from a Verilog-XL source character string, thus supporting the protection of that data.

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`. The syntax is as follows:

```
$sreadmemb(<mem_name>, <start_addr>, <finish_addr>,  
           <string1>, <string2>, ...);
```

```
$sreadmemh(<mem_name>, <start_addr>, <finish_addr>,  
           <string1>, <string2>, ...);
```

The following table defines the syntax variables:

<code><mem_name></code>	Name of the memory structure
<code><start_addr></code>	Memory start address
<code><finish_addr></code>	Memory end address
<code><stringN></code>	The string value containing the actual data to be placed into memory, beginning at <code><start_addr></code>

In the following example, the memory `memx` is loaded with 10 values, starting at address 0 and ending at address 9:

```
reg [3:0] memx [0:15] /* a memory made up of 16 four bit cells */  
$sreadmemh (memx, 0, 9, "6 7 8 9 A B C D E F");
```

Value Change Dump File Tasks

Seven system tasks are provided to create and format the value change dump file. For more information about these tasks see [Chapter 20, “The Value Change Dump File.”](#) The syntax is as follows:

```
$dumpall;  
$dumpfile(<filename>);  
$dumpflush;  
$dumplimit(<filesize>);  
$dumpoff;  
$dumpon;  
$dumpvars(<levels> <,<module|var>>* );
```

The `$dumpall` system task creates a checkpoint in the value change dump file that shows the current values of its variables.

The `$dumpfile` system task specifies the name of the value change dump file. If you do not specify a dump filename, Verilog-XL uses the default name `verilog.dump`.

The `$dumpflush` system task empties the dump file buffer and ensures that all the data in that buffer is stored in the value change dump file.

The `$dumplimit` system task sets the size of the value change dump file.

The `$dumpoff` system task stops Verilog-XL from recording value changes in the value change dump file.

The `$dumpon` system task allows Verilog-XL to resume recording value changes in the value change dump file.

The `$dumpvars` system task specifies the variables whose changing values Verilog-XL records in the value change dump file.

The `$dumpports` system task scans the (*arg1*) ports of a module instance and monitors the ports for both value and drive level.

Running the Behavior Profiler

The behavior profiler identifies the modules and statements in your source description that use the most CPU time during simulation. See [Chapter 19, “The Behavior Profiler.”](#) for details on the behavior profiler and for examples of profiler output.

There are four system tasks for the behavior profiler:

- [\\$startprofile](#) on page 383

- [\\$reportprofile](#) on page 383
- [\\$listcounts](#) on page 384
- [\\$stopprofile](#) on page 384

If you want to know the number of times each statement has executed at any point during simulation, use the `$listcounts` system task and the `+listcounts` command-line option or the `+no_speedup` command line option.

\$startprofile

The `$startprofile` system task invokes the behavior profiler. It tells the behavior profiler to begin or to continue to take samples of the simulation. The syntax is as follows:

```
$startprofile;  
$startprofile (<sampling_factor>);
```

This task takes an integer argument, called the sampling factor, that specifies the multiple of 100 microseconds of CPU time that is the interval between samples. By default, the behavior profiler takes a sample every 100 microseconds—a sampling factor of 1. This default sampling factor can increase the time it takes to simulate your source description. You can minimize this increase by specifying a sampling factor greater than 1.

\$reportprofile

The behavior profiler always displays its data at the end of simulation unless you use the `$reportprofile` system task to produce the following data reports *before* the end of a simulation.

- Profile ranking by statement
- Profile ranking by module instance
- Profile ranking by statement class
- Profile ranking by statement type

See “[Behavior Profiler Data Report](#)” on page 471 for information about the data reports.

The syntax is as follows:

```
$reportprofile (<max_lines>?);
```

This task takes an integer argument that specifies the maximum number of lines the behavior profiler prints in the tables. The argument is optional. The default maximum number of lines is 100.

\$listcounts

The `$listcounts` system task produces a source listing with both the line numbers and the execution count for each line. You can enter `$listcounts` before or after `$startprofile`. The `$listcounts` task is disabled unless you include the `+listcounts` option or the `+no_speedup` option on the command line. The syntax is as follows:

```
$listcounts (<hierarchical_name>?);
```

The `$listcounts` system task takes an optional hierarchical name argument. If you do not include an argument, `$listcounts` produces a listing of the source description at the scope level from which you called the task.

\$stopprofile

The `$stopprofile` system task tells the behavior profiler to stop taking samples before the end of the simulation. This system task takes no arguments as shown in the following syntax:

```
$stopprofile;
```

Resetting Verilog-XL—Starting Simulation Over Again

Verilog-XL includes the `$reset` system task to enable you to reset Verilog-XL to its "Time 0" state so that you can begin to simulate over again. The `$reset` system task works at least three times faster than any other means of restarting a simulation: compiling your source description again, or entering a `$restart` system task (if Verilog-XL executed a `$save` system task immediately after it compiled your source description).

Verilog-XL also includes the `$reset_count` system function to keep track of the number of times you reset Verilog-XL, and the `$reset_value` system function to allow you to pass an integer value that you can access after you reset the simulation time.

The following are some of the simulation methods that you can employ with the `$reset` system task and its related system functions:

- Determine the `force` statements your design needs to operate correctly, reset the simulation time to 0, enter these `force` statements, and start to simulate again.
- Reset the simulation time to 0 and apply new stimuli.
- Determine that debug system tasks, such as `$monitor` and `$strobe`, are keeping track of the correct nets or registers, reset the simulation time to 0, and begin the simulation again.

Verilog-XL Reference

System Tasks and Functions

The following section explains the syntax and an example of the use of the `$reset` system task and the `$reset_count` and `$reset_value` system functions.

\$reset

The `$reset` system task tells Verilog-XL to return the simulation of your design to its logical state at simulation time 0. When Verilog-XL executes the `$reset` system task, it takes the following actions to stop the simulation:

- Disables all concurrent activity, initiated in either `initial` and `always` procedural blocks in the source description, or through interactive mode (disables, for example, all `force` and `assign` statements, the current `$monitor` system task, and any other active task).
- Cancels all scheduled simulation events.
- Displays, and writes in the log file, a message about resetting Verilog-XL.
- Closes any active graphics windows.
- Clears all profiling information if you invoke the behavior profiler utility.
- Closes any open value dump file if you opened a value change dump file.

While Verilog-XL prepares to begin the simulation again, it takes the following actions:

- Reprocesses the options that you entered on the command line, with the exception of the following options:
 - `s` to enter interactive mode after compilation
 - `r` to restart a simulation
 - `l` to specify a log filename
 - `k` to specify a key filename

Note: Verilog-XL does not reprocess command-line options if it executes a `$reset` system task after a `$restart` system task. Verilog-XL does not save command-line option information when it executes a `$save` system task.

- Enters interactive mode if you specify that it enter interactive mode after executing the `$reset` system task.

After Verilog-XL executes the `$reset` system task, the simulation is in the following state:

- The simulation time is 0.

Verilog-XL Reference

System Tasks and Functions

- All registers and nets contain their initial values.
- Verilog-XL begins to execute the first procedural statements in all `initial` and `always` blocks.

The syntax is as follows:

```
$reset;  
$reset(<stop_value>);  
$reset(<stop_value>,<reset_value>);  
$reset(<stop_value>,<reset_value>,<diagnostics_value>);
```

The `$reset` system task takes these arguments:

`<stop_value>` Indicates whether you want to enter interactive mode after resetting Verilog-XL, or begin simulation immediately. The following table shows the mode specified by the `<stop_value>` argument:

Argument Value	Mode
A value of 0 or no argument	Enters interactive mode after resetting Verilog-XL.
A non-zero value	Does not enter interactive mode and begins simulation immediately after resetting Verilog-XL.

`<reset_value>` Is an integer that you specify and whose value is returned by the `$reset_value` system function after you reset Verilog-XL. You cannot declare an integer that keeps its value after a reset. All declared integers return to their initial values after reset. Entering an integer as the `<reset_value>` argument allows you to access the value of the integer as it was before the reset with the `$reset_value` system function. This argument provides you with means of presenting information from before the reset of Verilog-XL, to be used after the reset of Verilog-XL.

`<diagnostic_value>` Specifies the kind of diagnostic messages Verilog-XL displays before it resets the simulation time to 0. The following table describes the diagnostic messages that are specified by these integers:

Verilog-XL Reference

System Tasks and Functions

Argument Value	Diagnostic Messages
0	<ul style="list-style-type: none">■ No diagnostic messages
1	<ul style="list-style-type: none">■ The simulation time when Verilog-XL executes the <code>\$reset</code> system task■ The location in the source description file of the <code>\$reset</code> system task
2 or > 2	<ul style="list-style-type: none">■ The simulation time when Verilog-XL executes the <code>\$reset</code> system task■ The location in the source description file of the <code>\$reset</code> system task■ Statistics about memory used by the design■ Statistics about the CPU time since simulation began

The `<diagnostic_value>` argument specifies the same information as the integer argument to the `$finish` system task. The default `<diagnostic_value>` argument is 1.

Verilog-XL Reference

System Tasks and Functions

Examples

On the following pages, the two examples show the top-level module and the log file of the simulation of a half adder. There is a design mistake in the half adder. The log file shows how the design is debugged. The simulation is started again with the `$reset` system task.

```
module reset2;
reg in1,in2;
wire s,c;
initial
begin
    $monitor("%0d in1=%b,int2=%b,s=%b,c=%b",
            $time,in1,in2,s,c);
    #100 $stop;
end

initial
begin
    #5 forever
    begin
        if ({c,s} != in1 + in2)
        begin
            $display("ADDITION ERROR\n");
            $showvars(c,s);
            $stop;
        end
    end
    #10 ;
end

initial
begin
    {in1,in2} = 0;
    repeat (3)
    #10 {in1,in2} = {in1,in2} + 1;
end

halfadd hal (in1,in2,s,c);
endmodule
```

Monitor half adder inputs and outputs

The initial block checks the addition, reports mistakes and half adder output values, and stops the simulation

Half adder instance named hal

In the above example, the top-level module contains an instance of a module of a half adder. The instance name is `hal`. The top-level module monitors and checks the results of the half adder's addition. If Verilog-XL finds an addition error, it displays a warning, displays the input and output values, and stops the simulation.

The following example shows interactive entries during the simulation of the half adder:

```
0 in1=0,int2=0,s=1,c=0
ADDITION ERROR
c (reset2) wire = St0
  St0 <- (top.hal): and and1(c, in1, in2);
s (reset2) wire = St1
  St1 <- (top.hal): or or2(s, int1, int2);
L18 "reset2.v": $stop at simulation time 5
```

Verilog-XL Reference

System Tasks and Functions

```
Type ? for help
C1 > $list(hal);
// halfadder.v
34 module halfadd(in1, in2, s, c);
35     input
35         in1, // = St0
35         in2; // = St0
36     output
36         s, // = St1
36         c; // = St0
38     and
38         and1(c, in1, in2);
39     not
39         not1(int1, c);
40     or
40         or1(int2, in1, in2),
41         or2(s, int1, int2);
43 endmodule

C2 > force s = hal.int1 & hal.int2;

C3 > .
5 in1=0,int2=0,s=0,c=0
10 in1=0,int2=1,s=1,c=0
20 in1=1,int2=0,s=1,c=0
30 in1=1,int2=1,s=0,c=1
L8 "halfadder.v": $stop at simulation time 100

C3 > $reset;
C3: $reset at simulation time 100

C4 > force s = hal.int1 & hal.int2;

C4 > .
0 in1=0,int2=0,s=0,c=0
10 in1=0,int2=1,s=1,c=0
20 in1=1,int2=0,s=1,c=0
30 in1=1,int2=1,s=0,c=1
```

In the previous example, the following sequence of steps occurs:

1. At simulation time 5, Verilog-XL displays the warning of the addition error and the values of the half-adder outputs, stops the simulation, and enters the interactive mode.
2. The `$list` system task shows the contents of the half adder module. The design error is that the fanin of the `s` output should be an AND gate instead of an OR gate.
3. A `force` statement corrects the error, and simulation resumes.
4. Verilog-XL displays that the addition results are now correct.
5. An interactive entry of the `$reset` system task resets Verilog-XL to the time 0 state. This system task does not include a first argument of 1 or more than 1, so Verilog-XL stays in interactive mode after the reset.
6. With another entry of the `force` statement, Verilog-XL releases all forces when it resets the simulation time to 0.

7. Verilog-XL displays that the simulation results are once again correct.

\$reset_count

The `$reset_count` system function returns an integer that represents the number of times you called the `$reset` system task since you invoked Verilog-XL. The initial value of this integer is 0.

You can use this integer value, for example, to specify the stimuli that Verilog-XL applies to a design. The following example shows a top-level module that applies stimulus to a design. In this top-level module, the number of times that Verilog-XL executes the `$reset` system task determines the stimulus that Verilog-XL applies.

```
module reset;
reg in1,in2;
wire out;
design design1 (in1,in2,out);           // instance of a module
initial
begin
    $monitor("in1=%b,in2=%b,out=%b",in1,in2,out);
    case ($reset_count)                // integer returned by $reset_count
                                        // determines the stimulus
        0 : begin
            {in1,in2} = 2'd0;           // first stimulus
            #10 $reset(1);
        end
        1 : begin
            {in1,in2} = 2'd1;           // second stimulus
            #10 $reset(1);
        end
        2 : begin
            {in1,in2} = 2'd2;           // third stimulus
            #10 $reset(1);
        end
        3 : begin
            {in1,in2} = 2'd3;           // fourth stimulus
            #10 $reset(0);
        end
    endcase
end
endmodule
```

In the above example, Verilog-XL applies a stimulus and resets the simulation time to 0 four times. If Verilog-XL has not yet executed the `$reset` system task, it applies the first stimulus. After Verilog-XL executes the `$reset` system task once, it applies the second stimulus. It applies the third stimulus and the fourth stimulus after each subsequent execution of the `$reset` system task.

\$reset_value

The `$reset_value` system function extracts the reset value that is an argument to the `$reset` task. The `$reset_value` function returns a zero if there has not yet been a reset; otherwise, it extracts the value of the `<reset_value>` argument from the `$reset` call. You can use this system function to communicate information from one reset run to the next.

The following example shows a use of the `$reset_value` system function:

```
module reset_value;
reg [7:0] op1,op2;
reg ci;
wire [7:0] s;
wire co;
integer [31:0] reset_integer;
byte_adder byte_adder1(co,s,ci,op1,op2);
task what_next;
input carry_in,carry_out;
input [7:0] operand1,operand2,sum;
    // task checks the addition
if (operand1 + operand2 + carry_in != {carry_out,sum})
    $reset(0,reset_integer); // if addition not correct, reset and go to
interactive
    else
    begin
        reset_integer = reset_integer + 1;
        $reset (1,reset_integer); // if addition is correct,
                                // increment reset_integer and
                                // reset, without interactive
    end
endtask
initial
begin
    reset_integer = $reset_count;
    case ($reset_value)
        0 : begin
            {ci,op1,op2} = 17'd0;
            #10 what_next(ci,co,op1,op2,s);
            end
        1 : begin
            {ci,op1,op2} = 17'd625;
            #10 what_next(ci,co,op1,op2,s);
            end
        2 : begin
            {ci,op1,op2} = 17'd5025;
            #10 what_next(ci,co,op1,op2,s);
            end
        3 : begin
            {ci,op1,op2} = 17'd129225;
            #10 what_next(ci,co,op1,op2,s);
            end
    endcase
end
endmodule
```

Verilog-XL Reference

System Tasks and Functions

Verilog-XL applies a new stimulus only when the addition before Verilog-XL reset the simulation time was correct.

When the addition of the previous stimulus proves incorrect, Verilog-XL enters interactive mode and applies the previous stimulus again.

The previous example shows a top-level module that applies stimulus to an 8-bit full adder. The task `what_next` checks the addition of the adder. This check can then have one of the following results:

- If the addition is not correct, the task does not increment the value of the integer `<reset_integer>`. The task resets the simulation time to 0 and tells Verilog-XL to enter interactive mode.
- If the addition is correct, the task increments the value of the integer `<reset_integer>` and resets the simulation time to 0. The task does not tell Verilog-XL to enter interactive mode.

The results of the check of the addition described above determine the value of `<reset_integer>`. After the reset, the value of `<reset_integer>` returns to its initial value, but you can obtain its value from before the reset, using the `<reset_value>` argument to the `$reset` system task and through the `$reset_value` system function. The case statement in the `initial` block that applies the stimulus accesses the value of `<reset_integer>` before the reset using the `$reset_value` system function. If the task `what_next` has incremented the value of `reset_integer`, the case statement applies a new stimulus. If the task has not incremented that value, Verilog-XL enters the interactive mode and schedules the application of the stimulus that failed the addition check.

SDF Annotation

Tools used before or after Verilog-XL in the design process, such as pre-layout and post-layout tools, produce Standard Delay Format (SDF) files. These files can include timing information for the following:

- Delays for module iopaths, devices, ports, and interconnect delays
- Timing checks
- Timing constraints
- Scaling, environmental, technology, and user-defined parameters

SDF files are the input for the SDF annotator, which uses the PLI as an interface to backannotate timing information into Verilog HDL designs. A configuration file that you write controls how the backannotation occurs.

Verilog-XL Reference

System Tasks and Functions

This section covers only the system task aspect of SDF backannotation. Refer to the *SDF Annotator Guide* for complete information.

\$sdf_annotate

The `$sdf_annotate` system task invokes the SDF Annotator. You can run the SDF Annotator any number of times from a Verilog family tool. You can call this system task interactively or by typing it in the simulation stimuli file in an initial block. If you want to allow SDF backannotation at times other than time 0, use the `+annotate_any_time` plus option on the command line. The `$sdf_annotate` arguments are shown in the following syntax:

```
$sdf_annotate ( <"sdf_file">, <module_instance>?,  
               <"config_file">?, <"log_file">?, <"mtm_spec">?,  
               <"scale_factors">?, <"scale_type">? );
```

All arguments other than the initial `<"sdf_file">` are optional. All the arguments except `<module_instance>` must be in quotation marks. If you omit optional arguments, the commas that would have surrounded them must remain, unless the omitted arguments are consecutive and include the last argument, in which case, the closing parenthesis can follow the last argument present.

The `$sdf_annotate` arguments are as follows:

- | | |
|--------------------------------------|--|
| <code><"sdf_file"></code> | Literal string that identifies the name of the SDF file. The SDF Annotator reads this SDF file. This file does not appear on the Verilog command line. |
| <code><module_instance></code> | Name of the module instance. Where applicable, an instance can have an array index (for example, <code>x.y[3].p</code>). The SDF Annotator uses the hierarchy level of the module instance for running the annotation. If you do not specify <code><module_instance></code> , the SDF Annotator uses the module containing the call to the <code>\$sdf_annotate</code> system task as the <code><module_instance></code> for annotation. The names in the SDF file are relative paths to the <code><module_instance></code> or full paths with respect to the entire Verilog HDL description. |
| <code><"config_file"></code> | Literal string that identifies the name of the configuration file. The SDF Annotator reads this configuration file. See "Using the Configuration File" in the <i>SDF Annotator User Guide</i> for more information. If you do not specify this argument, the SDF Annotator uses the default settings. |

Verilog-XL Reference

System Tasks and Functions

`<"log_file">` Literal string that identifies the name of the annotation log file. The SDF Annotator generates this file during annotation. The default name for the log file is `sdf.log`. The SDF Annotator writes status information, warnings, and error messages to the log file during the annotation process. These messages also include the configuration of the annotator, assumptions made during annotation, and warnings or errors due to inconsistencies found during annotation. The SDF Annotator also prints warning and error messages to a standard output.

`<"mtm_spec">` Literal string that specifies the delay values that are annotated to the Verilog family tool as one of the following keywords:

MINIMUM	Annotates the minimum delay value.
TYPICAL	Annotates the typical delay value.
MAXIMUM	Annotates the maximum delay value.
TOOL_CONTROL	Delay value is determined by the command line options of the Verilog Family tool (+mindelays, +typdelays, or +maxdelays).

This argument overrides the `mtm` command in the configuration file. The default setting is `TOOL_CONTROL`, but if none of the `TOOL_CONTROL` command line options is specified, the default is `TYPICAL`.

Note: This argument applies only for backannotation to Verilog-XL and Verifault-XL. Minimum, typical, and maximum values are always annotated to Veritime.

`<"scale_factors">` Set of three real number multipliers in the form of `min_mult:typ_mult:max_mult` that the SDF Annotator uses to scale the minimum, typical, and maximum timing data from the SDF file before they are annotated to the Verilog Family tool. The `min_mult`, `typ_mult`, and `max_mult` variables each represent a positive real number. For example, `1.6:1.4:1.2`. This argument overrides the `scale` command in the configuration file. The default value is `1.0:1.0:1.0` for minimum, typical, and maximum values. See "Scaling the Timing Data" in the *SDF Annotator User*

Verilog-XL Reference

System Tasks and Functions

Guide for an example of scaling delay values using the `<"scale_factors">` and `<"scale_type">` parameters.

`<"scale_type">`

Literal string that specifies how the SDF Annotator scales the timing specifications in SDF that are annotated to the Verilog family tool as one of the following keywords:

FROM_MINIMUM	Scales from the minimum timing specification.
FROM_TYPICAL	Scales from the typical timing specification.
FROM_MAXIMUM	Scales from the maximum timing specification.
FROM_MTM (default)	Scales from the minimum, typical, and maximum timing specifications.

This argument overrides the scale command in the configuration file.

Controlling \$sdf_annotate Output

Three options enable you to control the output from the SDF Annotator:

- `+sdf_verbose`
Writes detailed information about the backannotation process to the annotation log file.
- `+sdf_error_info`
Displays PLI error messages.
- `+sdf_no_warnings`
Suppresses all warning messages from the SDF Annotator.

\$sdf_annotate Examples

The three examples in this section show the following aspects of running `$sdf_annotate`:

- Creating new delay triplets
- Invoking multiple system tasks and log files

■ Specifying interconnect delays

Creating new Delay Triplets

The following example shows how to create new minimum-typical-maximum delay triplets scaled to the Verilog design and how to annotate the minimum, the typical, or the maximum members of those triplets to the specified scope.

The argument `"my.sdf"` is the SDF file, and `m1` is the module instance that the `$sdf_annotate` task annotates. The following example omits the third argument, the configuration file that can determine how annotation occurs. The following example also omits the fourth argument, the log file, so the default `sdf.log` is used:

```
module top;
  ...
  circuit m1(i1,i2,i3,o1,o2,o3);
  initial
  $sdf_annotate("my.sdf",m1,, "MAXIMUM", "1.6:1.4:1.2", "FROM_MTM");
  //stimulus and response checking
  ...
endmodule
```

The `<"mtm_spec">` argument's value `"MAXIMUM"` specifies that the maximum delays in the new minimum-typical-maximum triplets are annotated to the Verilog design. The last two arguments are scale factors and scale types, which determine how SDF makes new minimum-typical-maximum delay triplets. The three scale factors are `1.6`, `1.4`, and `1.2`. The scale factors multiply members of delay triplets in the SDF file to create new triplets whose members can be annotated to the Verilog design. Like the other `<"scale_types">` value, `"FROM_MTM"` specifies the SDF triplet members to be multiplied by the scale factors. `"FROM_MTM"` specifies that scale factor `1.6` multiplies the minimum members in the SDF file delay triplets, `1.4` multiplies typical members, and `1.2` multiplies the maximum members. The three `<"scale_types">` values other than `"FROM_MTM"` work differently because they make all three of the `<"scale_factors">` values multiply only the minimum, or only the typical, or only the maximum members of delay triplets in the SDF file.

Invoking Multiple System Tasks

The following example shows separate annotations to distinct portions of a design hierarchy. There is no configuration file specification; therefore, the SDF Annotator uses the defaults. When performing multiple annotations, specify a different log file for each annotation for easier verification of the results.

```
module top;
  ...
  cpu m1(i1,i2,i3,o1,o2,o3);
  fpu m2(i4,o1,o3,i2,o4,o5,o6);
  dma m3(o1,o4,i5,i6,i2);
  ...
endmodule
```

Verilog-XL Reference

System Tasks and Functions

```
// perform annotation
initial
begin
    $sdf_annotate("cpu.sdf",m1,, "cpu.log");
    $sdf_annotate("fpu.sdf",m2,, "fpu.log");
    $sdf_annotate("dma.sdf",m3,, "dma.log");
end
// stimulus and response-checking
...
endmodule
```

Specifying Interconnect Delays

The following example shows an interconnect delay that occurs between instances `u1` and `u2` of `jbuf`:

- The [“Interconnect Delay Annotation Example”](#) on page 397 shows the code module.
- The [“Hierarchy of Interconnect Delay Annotation Example”](#) on page 398 shows the hierarchy of the code module.
- The [“SDF file for Interconnect Delay Annotation Example”](#) on page 399 shows the SDF file, `top.sdf`, that supplies delay information.
- The [“Effective Hierarchy Specifications for Annotation”](#) on page 399 shows the configurations of hierarchical information that can yield effective SDF annotation for the combination of the code module and the SDF file.

The delay between the instance ports is indicated in the SDF file by the descriptor `INTERCONNECT` followed by port descriptions. The first port is an output or inout, and the second port is an input or inout. You must describe each port hierarchically, but the descriptions in the SDF file are only one method of doing so. The lowest level in the description of each port in the SDF file is the name of the port in the module definition. The other information identifies the instances involved in the delay. You can divide the instance information between the `$sdf_annotate` task and the SDF file, with the information in the task constituting the beginning of the information. You can either include or omit the level containing the task.

Interconnect Delay Annotation Example

```
`timescale 1ns/1ns
module top ( );
reg t;
lower array1 (f,t);
initial
begin
    $sdf_annotate ("top.sdf");
end
initial
begin
    fork
```

Verilog-XL Reference

System Tasks and Functions

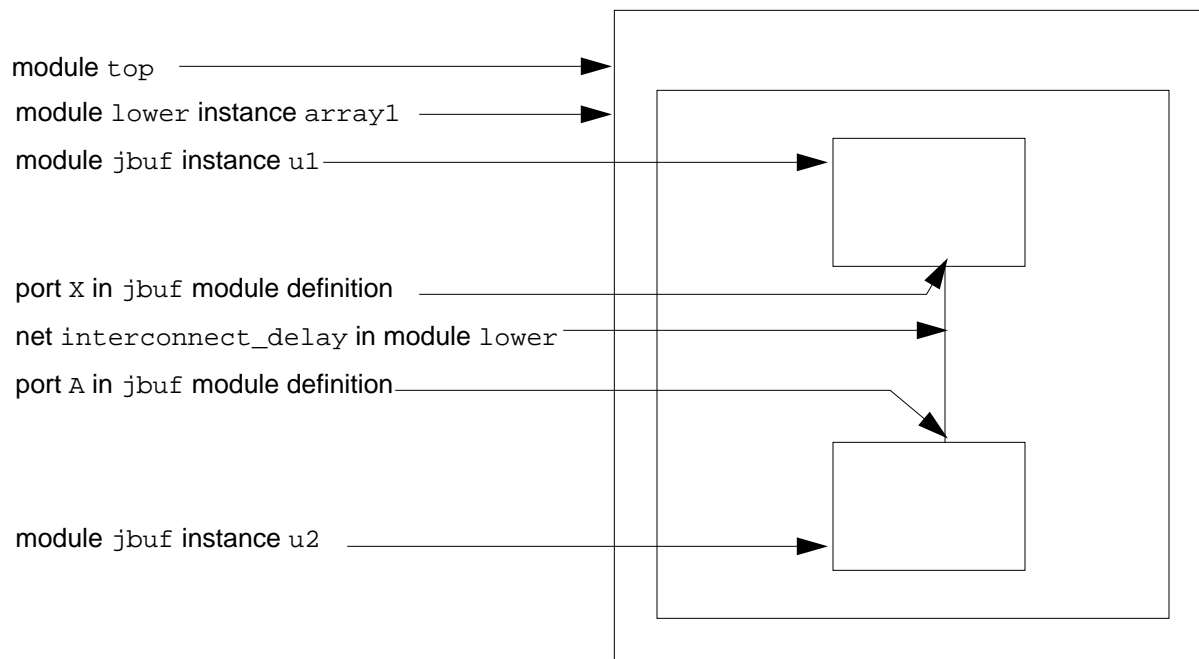
```
#00 t =0;
#20 t = 1;
#60 t = 0;
#100 $stop;
join
end
endmodule

`timescale 1ns/1ns
module lower (out1,in1);
input in1;
output out1;
wire delay_connection;
jbuf u1 (delay_connection,in1);
jbuf u2 (out1, delay_connection);
endmodule

`timescale 1ns/1ns
module jbuf (X,A);
input A;
output X;
buf u2(X,A);

specify
(A *> X) = (0:0:0);
endspecify
endmodule
```

Hierarchy of Interconnect Delay Annotation Example



You can consider interconnect delays as sitting on the MIPD of the target, which is the MIPD of u2 in this example.

SDF file for Interconnect Delay Annotation Example

```
(DELAYFILE
(DESIGN "top" )
  (DATE  " " )
  (VENDOR  " " )
  (PROGRAM  " " )
  (VERSION  " " )
  (DIVIDER  . )
  (VOLTAGE  )
  (PROCESS  " " )
  (TEMPERATURE  )
  (TIMESCALE  )
  (CELL
  (CELLTYPE "top" )
  (INSTANCE )
  (DELAY
  (ABSOLUTE
    (INTERCONNECT array1.u1.X array1.u2.A (6:10:16) (8:12:18) )
  ) // end delay
  ) // end absolute
  ) // end cell
  ) // end delayfile
```

Effective Hierarchy Specifications for Annotation

<code>\$sdf_annotate <module_instance></code> argument	Interconnect ports in SDF file
<code>top.array1</code>	<code>u1.X u2.A</code>
<code>top</code>	<code>array1.u1.X array1.u2.A</code>
<code>array1</code>	<code>u1.X u2.A</code>
<code>(no entry)</code>	<code>array1.u1.X array1.u2.A</code>

Annotating Path Delay or Timing Check Vector Bits in Specify Blocks

In specify blocks, you can annotate the path delays or timing check structures to individual bits of vectors on a module-by-module basis or for an entire design. Verilog-XL does this by expanding vectors to all possible statements before annotating. You can display expanded specify blocks using the `$list` system task, or specify the `-d` command line option.

Annotating Vector Bits on a Module-by-Module Basis

Use the ``expand_specify_vectors` and ``noexpand_specify_vectors` compiler directives to annotate vector bits on a module-by-module basis with SDF. These compiler directives must be specified outside of module descriptions.

Annotating Vector Bits for an Entire Design

Use the `+expand_specify_vectors` command line plus option to annotate vector bits for an entire design. If you use this plus option, the specify blocks in all modules in a design are expanded, regardless of the compiler directives you set on a module-by-module basis.

Expanding Path Delays

The following example shows a parallel connection path delay before and after it is expanded. Path Delays are described in [Chapter 12, “Using Specify Blocks and Path Delays.”](#)

```
/* Unexpanded Path Delay */
output [3:0] o;
input [3:0] i;
specify
  (i => o) = 0;
endspecify

/* Expanded Path Delay */

output [3:0] o;
input [3:0] i;
specify
  (i[0] => o[0]) = 0;
  (i[1] => o[1]) = 0;
  (i[2] => o[2]) = 0;
  (i[3] => o[3]) = 0;
endspecify
```

You can expand full connection path delays, but the number of path delays is exponentially expanded. For example, expanding a 32-bit to 32-bit full connection path delay results in 1024 paths.

With SDF, you must annotate each bit in an expanded specify block. For example, to annotate the specify block in the previous example, you would specify the following SDF syntax:

```
(IOPATH i[0] o[0] ...)
(IOPATH i[1] o[1] ...)
(IOPATH i[2] o[2] ...)
(IOPATH i[3] o[3] ...)
```

You cannot annotate with SDF using `(IOPATH i o ...)`.

The following example shows a full connection path delay before and after it is expanded.

```
/* Unexpanded Path Delay */
output [3:0] o;
input [3:0] i;
specify
  (i *> o) = 0;
endspecify
```


Verilog-XL Reference

System Tasks and Functions

```
/* Expanded Path Delay */

output [3:0] o;
input [3:0] i;
specify
  (i[0] => o[0]) = 0;
  (i[0] => o[1]) = 0;
  (i[0] => o[2]) = 0;
  (i[0] => o[3]) = 0;
  (i[1] => o[0]) = 0;
  (i[1] => o[1]) = 0;
  (i[1] => o[2]) = 0;
  (i[1] => o[3]) = 0;
  (i[2] => o[0]) = 0;
  (i[2] => o[1]) = 0;
  (i[2] => o[2]) = 0;
  (i[2] => o[3]) = 0;
  (i[3] => o[0]) = 0;
  (i[3] => o[1]) = 0;
  (i[3] => o[2]) = 0;
  (i[3] => o[3]) = 0;
endspecify
```

Expanding Timing Checks

You can expand timing checks by distributing the timing check across all possible combinations of the two signals. The following example shows a timing check before and after it is expanded. Timing Checks are described in [Chapter 13, “Timing Checks”](#).

```
/* Unexpanded Timing Check */
input [3:0] i;
input clk;
specify
  $setuphold(clk,i,0,0);
endspecify

/* Expanded Timing Check */
input [3:0] i;
input clk;
specify
  $setuphold(clk,i[0],0,0);
  $setuphold(clk,i[1],0,0);
  $setuphold(clk,i[2],0,0);
  $setuphold(clk,i[3],0,0);
endspecify
```

The following example shows a timing check with two vectored input signals before and after it is expanded.

```
/* Unexpanded Timing Check */
input [3:0] x;
input [3:0] y;
```

Verilog-XL Reference

System Tasks and Functions

```
specify
  $setuphold(x,y,0,0);
endspecify

/* Expanded Timing Check */
input [3:0] i;
input clk;
specify
  $setuphold(x[0],y[0],0,0);
  $setuphold(x[0],y[1],0,0);
  $setuphold(x[0],y[2],0,0);
  $setuphold(x[0],y[3],0,0);
  $setuphold(x[1],y[0],0,0);
  $setuphold(x[1],y[1],0,0);
  $setuphold(x[1],y[2],0,0);
  $setuphold(x[1],y[3],0,0);
  $setuphold(x[2],y[0],0,0);
  $setuphold(x[2],y[1],0,0);
  $setuphold(x[2],y[2],0,0);
  $setuphold(x[2],y[3],0,0);
  $setuphold(x[3],y[0],0,0);
  $setuphold(x[3],y[1],0,0);
  $setuphold(x[3],y[2],0,0);
  $setuphold(x[3],y[3],0,0);
endspecify
```

When delayed signals, which are used with negative timing checks, are vectors, Verilog-XL uses the numbered bit of the delayed signal to set the corresponding numbered bit of the timing check as shown in the following example. See [“Using Negative Timing Check Limits in \\$setuphold and \\$recrem”](#) on page 313 for information about using delayed signals.

```
/* Unexpanded Timing Check */
input [7:0] d;
wire [7:0] d_d;
specify
  $setuphold(posedge clk,d,-1,5,ntfy,,,clk_d,d_d);
endspecify

/* Expanded Timing Check */
input [3:0] i;
input clk;
specify
  $setuphold(posedge clk, d[7], -1, 5, ntfy,,,clk_d, d_d[7]);
  $setuphold(posedge clk, d[6], -1, 5, ntfy,,,clk_d, d_d[6]);
  $setuphold(posedge clk, d[5], -1, 5, ntfy,,,clk_d, d_d[5]);
  $setuphold(posedge clk, d[4], -1, 5, ntfy,,,clk_d, d_d[4]);
  $setuphold(posedge clk, d[3], -1, 5, ntfy,,,clk_d, d_d[3]);
  $setuphold(posedge clk, d[2], -1, 5, ntfy,,,clk_d, d_d[2]);
  $setuphold(posedge clk, d[1], -1, 5, ntfy,,,clk_d, d_d[1]);
  $setuphold(posedge clk, d[0], -1, 5, ntfy,,,clk_d, d_d[0]);
endspecify
```

Using the \$dlc System Task

The `$dlc` system task invokes the delay calculator. The syntax is as follows:

```
$dlc(<GCF_filename>,<Pearl_command_filename>{, <top_module_name>});
```

You must specify the general constraint format (GCF) filename and the Pearl filename arguments. If you omit the `<top_module_name>` argument, Verilog-XL uses the first top-level module found in the description.

You create the GCF file to specify the operating conditions and the timing library format (TLF) libraries. The following is an example GCF file:

```
(GCF
  (HEADER
    (VERSION "1.2")
    (DESIGN "small0")
    (TIME_SCALE 1.0E-9)
  )
  (GLOBALS
    (GLOBALS_SUBSET ENVIRONMENT
      (VOLTAGE_THRESHOLD 10.0 90.0)
      (OPERATING_CONDITIONS "" 1.00 3.13 100.00)
      (EXTENSION "CTLF_FILES" (timing.ctlf))
    )
  )
  (CELL ()
    (SUBSET TIMING
      (ENVIRONMENT
        (INPUT_SLEW 1.00 1.00)
      )
    )
  )
)
```

You create the Pearl command file to specify the commands and options that you want to use for delay calculation. If you have a CDC `dlcinit` file, you must first convert it with the `dlc2pearl.pl` script. For example:

```
dls2pearl.pl dlcinit pearl.cmd
```

This example generates a Pearl command file called `pearl.cmd` and a GCF file called `pearl.gcf` for the constraints. The following is an example Pearl command file:

```
ReadSpf small0.rspf
EstimateWireLoads -rc -topology best
TopLevelCell top
WriteSDFDelays -precision 6 -ns top.sdf
TopLevelCell adder
WriteSDFDelays -precision 6 -ns adder.sdf
```

Note: The Pearl command file can only contain commands that relate to delay calculation. Otherwise, Pearl generates error messages. For more details about Pearl-related information, see the *Pearl User Guide*.

Verilog-XL Reference

System Tasks and Functions

The following Pearl commands are always executed before those in the user command file:

```
ReadGCFTimingLibraries(GCF filename)
ReadVerilog(filename1)
ReadVerilog(filename2) ...
TopLevelCell(top module name)
ReadGCFConstraints(GCF filename)
```

Then, the user command file is included using:

```
Include <user cmd file>
```

The `$dlc` task generates an SDF file. You can specify the SDF filename in the Pearl command file. You can then backannotate the SDF file to the Verilog design using the `$sdf_annotate` system task.

Using the `$system` System Task

The `$system` system task is an in-built verilog system task that makes a call to the C function `system()`. The C function, `system()`, executes the argument passed to it as if the argument was executed from the terminal. Consider the example given below. It uses the `$system` task to rename a file.

```
module top;
    initial $system("mv design.v adder.v");
endmodule
```

Programmable Logic Arrays

This chapter describes the following:

- [Overview](#) on page 405
- [Syntax](#) on page 405
- [Array Types](#) on page 406
- [Array Logic Types](#) on page 406
- [Logic Array Personality Declaration and Loading](#) on page 407
- [Logic Array Personality Formats](#) on page 407
- [PLA Examples](#) on page 409

Overview

Programmable logic array (PLA) devices are modeled by a group of system tasks. This chapter describes the formats of the logic array personality file.

Syntax

The syntax for PLA system tasks is as follows:

```
$<array_type>$<logic>$<format>( <memname>, {A1, ..., An}, {B1, ..., Bn} );
<array_type>
  ::= sync
  || = async
<logic>
  ::= and
  || = or
  || = nand
  || = nor
<format>
  ::= array
  || = plane
```

Verilog-XL Reference

Programmable Logic Arrays

```
<memname>  
  ::=<IDENTIFIER>
```

You must put PLA input terms, output terms, and memory in ascending order, as demonstrated in this chapter since other orders are not supported consistently.

The PLA syntax allows for the following system tasks:

```
$sync$and$array(...)  
$sync$or$array(...);  
$sync$nand$array(...);  
$sync$nor$array(...);  
$async$and$array(...)  
$async$or$array(...);  
$async$nand$array(...);  
$async$nor$array(...);  
$sync$and$plane(...)  
$sync$or$plane(...);  
$sync$nand$plane(...);  
$sync$nor$plane(...);  
$async$and$plane(...)  
$async$or$plane(...);  
$async$nand$plane(...);  
$async$nor$plane(...);
```

Array Types

Verilog allows the modeling of both synchronous and asynchronous arrays. The synchronous forms allow you to control the time at which the logic array is evaluated and the outputs are updated. For the asynchronous forms, the evaluations are automatically performed whenever an input term changes value and when any word in the personality memory is changed.

For both the synchronous and asynchronous versions, the output terms (<Bi>'s) are updated with zero delays.

An example of an asynchronous system call is as follows:

```
$async$and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

An example of a synchronous system call is as follows:

```
$sync$or$plane(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

Note: The input terms (<Ai>'s) and the output terms (<Bi>'s) are always represented as concatenations.

Array Logic Types

Verilog allows the modeling of arrays with and, or, nand, and nor logic planes. This applies to all array types and formats.

An example of a nor plane system call is as follows:

```
$async$nor$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

An example of a nand plane system call is as follows:

```
$sync$nand$plane(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
```

Logic Array Personality Declaration and Loading

The logic array personality is declared as an array of registers that is as wide as the number of input terms and as deep as the number of output terms. The following example shows a logic array with *n* input terms and *m* output terms:

```
reg[1:n] mem[1:m];
```

The personality of the logic array is normally loaded into the memory from a text data file using the system tasks `$readmemb` or `$readmemh`. Alternatively, it is possible to write the personality data directly into the memory using the normal Verilog assignment statements. PLA personalities may be changed dynamically at any time during simulation, simply by changing the contents of the memory. The new personality will be reflected on the outputs of the logic array at the next evaluation.

Logic Array Personality Formats

Two separate personality formats are supported by Verilog and are differentiated by using either an array system call or a plane system call. The array system call allows for a 1 or a 0 in the memory that has been declared. A 1 means take the input value, and a 0 means do not take the input value.

The following example illustrates an array with logic equations:

```
b1 = a1 & a2  
b2 = a3 & a4 & a5  
b3 = a5 & a6 & a7
```

The PLA personality is as follows:

```
1100000 in mem[1]  
0011100 in mem[2]  
0000111 in mem[3]
```

The module for the PLA is as follows:

```
module async_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);  
input a1,a2,a3,a4,a5,a6,a7;  
output b1,b2,b3;  
    reg[1:7] mem[1:3];  
    // memory declaration for  
    // array personality
```

Verilog-XL Reference

Programmable Logic Arrays

```
reg b1, b2, b3;
initial
begin
  // setup the personality from the file array.dat
  $readmemb("array.dat", mem);
  // setup an asynchronous logic array with the input
  // and output terms expressed as concatenations
  $async$and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
end
endmodule
```

The file `array.dat` contains the binary data for the PLA personality:

```
1100000
0011100
0000111
```

The plane system call complies with the University of California at Berkeley format for espresso. Each bit of the data stored in the array has the following meaning:

- 0 take the complemented input value
- 1 take the true input value
- x take the "worst case" of the input value
- z don't-care; the input value is of no significance
- ? same as z

An example of the usage of the new tasks follows. The logical function of this PLA is shown first, followed by the PLA personality in the new format, the Verilog description using the `$async$and$plane` system task, and finally the results of the simulation.

The logical function of the PLA is as follows:

```
b[1] = a[1] & ~a[2];
b[2] = a[3];
b[3] = ~a[1] & ~a[3];
b[4] = 1;
```

The PLA personality is as follows:

```
3'b10?
3'b??1
3'b0?0
3'b???
```

The following example shows the Verilog description using the `$async$and$plane` system task:

```
module pla;
`define rows 4
`define cols 3
reg [1:`cols] a, mem[1:`rows];
```



```
reg [1:\rows] b;
initial
begin
  // PLA system call
  $async$and$plane(mem,
    {a[1],a[2],a[3]},
    {b[1],b[2],b[3],b[4]});
  mem[1] = 3'b10?;
  mem[2] = 3'b??1;
  mem[3] = 3'b0?0;
  mem[4] = 3'b???;
  // stimulus and display
  #10 a = 3'b111;
  #10 $displayb(a, " -> ", b);
  #10 a = 3'b000;
  #10 $displayb(a, " -> ", b);
  #10 a = 3'bxxx;
  #10 $displayb(a, " -> ", b);
  #10 a = 3'b101;
  #10 $displayb(a, " -> ", b);
end
endmodule
```

The output for the previous code is as follows:

```
111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101
```

PLA Examples

This section contains the following PLA examples:

- [“Synchronous Example”](#) on page 409
- [“And-Or Array Example”](#) on page 410
- [“PAL16R8 Example”](#) on page 411
- [“PAL16R4 Example”](#) on page 416

Synchronous Example

An example of synchronized version of PLA is as follows:

```
module sync_array(a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1,a2,a3,a4,a5,a6,a7;
output b1,b2,b3;
reg[1:7] mem[1:3]; // memory declaration
reg b1, b2, b3;
initial
begin
  // setup the personality
```

Verilog-XL Reference Programmable Logic Arrays

```
    $readmemb("array.dat", mem);
// setup a synchronous logic array to be evaluated
// when a positive edge on the clock occurs
    forever @(posedge clk)
        $sync$and$array(mem, {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
    end
endmodule
```

And-Or Array Example

To model a double PLA device, two logic array system tasks with two separate memories are used. For example, if the previous logic equation defines the and-plane, the logic equations for the or-plane are as follows:

```
c1 = b1 | b2;
c2 = b1 | b3;
c3 = b1 | b2 | b3;
c4 = 0;
c5 = b1;
c6 = b2;
c7 = b3;
```

The asynchronous version of this PLA has the following form:

```
module and_or_async(a1,a2,a3,a4,a5,a6,a7,c1,c2,c3,c4,c5,c6,c7);
input a1,a2,a3,a4,a5,a6,a7; // input terms
output c1,c2,c3,c4,c5,c6,c7; // output terms
reg[1:7] and_plane[1:3]; // and-plane memory
reg[1:3] or_plane[1:7]; // or-plane memory
reg b1, b2, b3;
reg c1, c2, c3, c4, c5, c6, c7;
    initial
    begin
        $readmemb("and_plane.dat", and_plane);
        $readmemb("or_plane.dat", or_plane);
        $async$and$array(and_plane,
            {a1,a2,a3,a4,a5,a6,a7}, {b1,b2,b3});
        $async$or$array(or_plane,
            {b1,b2,b3}, {c1,c2,c3,c4,c5,c6,c7});
    end
endmodule
```

The file `and_plane.dat` contains the following binary data:

```
1100000
0011100
0000111
```

and the file `or_plane.dat` contains the binary data:

```
110
101
111
000
100
010
001
```

Verilog-XL Reference

Programmable Logic Arrays

PAL16R8 Example

```
//Test program taken from MMI's "PAL (Programmable Array Logic)
//Handbook", Third Edition, shaft encoder no. 2, page 6-262.
module test_PAL16R8;
  reg
    clk, oc_, set_, phi0,phi90, x4;
  wire
    ud, s4, s3, s2, s1, count;
  //instantiation of the PAL device
  PAL16R8
    cct (clk, phi0,phi90,x4,,,,,set_,,oc_,
        ud,,s4,s3,s2,s1,,count);
  parameter
    p = 100,
    L = 1'b0, H = 1'b1, X = 1'bx;
  always
  begin
    #(p/2) clk = 1;
    #(p/2) clk = 0;
  end
  initial
  begin
    $display("--CONTROLS-- --INPUTS-- x ssss -OUTPUTS-");
    $display("clk oc_ set_ phi0 phi90 4 1234 count ud");
    $display("-|---|---|----|-----|---|---|||---|----|-");
    $monitor(,clk,,,,oc_,,,,set_,,,,,phi0,,,,,phi90,,,,x4,
        ,,s1,s2,s3,s4,,,,count,,,,,ud);
    $monitoroff;
    #(p-1) forever
    begin
      $monitor;
      #p;
    end
  end
  initial
  begin
    $display("clear registers");
    clk=0; oc_=L; set_=L;
    #p $display("count up x4=L");
    phi0=L; phi90=L; x4=L; set_=H;
    #p phi0=H; phi90=L; #p;
    #p phi0=H; phi90=H; #p;
    #p phi0=L; phi90=H; #p;
    #p phi0=L; phi90=L; #p;
    #p phi0=H; phi90=L; #p;
    #p phi0=H; phi90=H; #p;
    #p phi0=L; phi90=H; #p;
    #p $display("count up x4=H");
    phi0=L; phi90=L; x4=H; #p;
    #p phi0=H; phi90=L; #p;
    #p phi0=H; phi90=H; #p;
    #p phi0=L; phi90=H; #p;
    #p phi0=L; phi90=L; #p;
    #p phi0=H; phi90=L; #p;
    #p phi0=H; phi90=H; #p;
    #p $display("clear registers");
    phi0=X; phi90=X; x4=X; set_=L;

    #p $display("count down x4=L");
```

Verilog-XL Reference

Programmable Logic Arrays

```
        phi0=L; phi90=L; x4=L; set_=H;
#p phi0=L; phi90=H; #p;
#p phi0=H; phi90=H; #p;
#p phi0=H; phi90=L; #p;
#p phi0=L; phi90=L; #p;
#p phi0=L; phi90=H; #p;
#p phi0=H; phi90=H; #p;
#p phi0=H; phi90=L; #p;
#p $display("count down x4=H");
    phi0=L; phi90=L; x4=H; #p;
#p phi0=L; phi90=H; #p;
#p phi0=H; phi90=H; #p;
#p phi0=H; phi90=L; #p;
#p phi0=L; phi90=L; #p;
#p phi0=L; phi90=H; #p;
#p phi0=H; phi90=H; #p;
#p $display("test Hi-Z");
    phi0=X; phi90=X; x4=X; set_=X; oc_=H;
#p $finish;
end
endmodule

module PAL16R8(clock, i0,i1,i2,i3,i4,i5,i6,i7,,enable,
              o7,o6,o5,o4,o3,o2,o1,o0);
    input
        clock, i0,i1,i2,i3,i4,i5,i6,i7, enable;
    output
        o7,o6,o5,o4,o3,o2,o1,o0;
    reg //AND array outputs
        b0,b1,b2,b3,b4,b5,b6,b7,
        b8,b9,b10,b11,b12,b13,b14,b15,
        b16,b17,b18,b19,b20,b21,b22,b23,
        b24,b25,b26,b27,b28,b29,b30,b31,
        b32,b33,b34,b35,b36,b37,b38,b39,

        b40,b41,b42,b43,b44,b45,b46,b47,
        b48,b49,b50,b51,b52,b53,b54,b55,
        b56,b57,b58,b59,b60,b61,b62,b63;
    reg[0:31] //memory for array personality
        person[0:63];
    notif0 #20 //output enable drivers
        (o0, q0, enable), (o1, q1, enable),
        (o2, q2, enable), (o3, q3, enable),
        (o4, q4, enable), (o5, q5, enable),
        (o6, q6, enable), (o7, q7, enable);
    or //fixed OR array
        (q0, b0,b1,b2,b3,b4,b5,b6,b7),
        (q1, b8,b9,b10,b11,b12,b13,b14,b15),
        (q2, b16,b17,b18,b19,b20,b21,b22,b23),
        (q3, b24,b25,b26,b27,b28,b29,b30,b31),
        (q4, b32,b33,b34,b35,b36,b37,b38,b39),
        (q5, b40,b41,b42,b43,b44,b45,b46,b47),
        (q6, b48,b49,b50,b51,b52,b53,b54,b55),
        (q7, b56,b57,b58,b59,b60,b61,b62,b63);
    always @(posedge clock)
        //programmable AND array evaluated every positive clock edge
        $sync$and$array(person,
            {i0,!i0, !q0,q0, i1,!i1, !q1,q1,
             i2,!i2, !q2,q2, i3,!i3, !q3,q3,
             i4,!i4, !q4,q4, i5,!i5, !q5,q5,
             i6,!i6, !q6,q6, i7,!i7, !q7,q7},
            {b0,b1,b2,b3,b4,b5,b6,b7,
```

Verilog-XL Reference Programmable Logic Arrays

```
    b8,b9,b10,b11,b12,b13,b14,b15,  
    b16,b17,b18,b19,b20,b21,b22,b23,  
    b24,b25,b26,b27,b28,b29,b30,b31,  
    b32,b33,b34,b35,b36,b37,b38,b39,  
    b40,b41,b42,b43,b44,b45,b46,b47,  
    b48,b49,b50,b51,b52,b53,b54,b55,  
    b56,b57,b58,b59,b60,b61,b62,b63});  
    // read the PAL personality at the beginning of simulation  
    initial $readmemb("person16R8.dat", person, 0, 63);  
endmodule
```

The contents of file person16R8.dat are as follows:

```
// Addresses 0-7  
0000_0000_0010_0010_0001_0010_0000_0000  
0000_0000_0001_0001_0010_0001_0000_0000  
0000_0000_1001_0010_0001_0001_0000_0000  
0000_0000_1010_0001_0010_0010_0000_0000  
0000_0000_0010_0010_0010_0001_0000_0000  
0000_0000_0001_0001_0001_0010_0000_0000  
0000_0000_1001_0010_0010_0010_0000_0000  
0000_0000_1010_0001_0001_0001_0000_0000
```

```
// Addresses 8-15  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 16-23  
0100_0000_0000_0000_0000_0000_0000_0000  
0000_0000_0000_0000_0000_0000_0000_0100  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 24-31  
0000_0000_0010_0000_0000_0000_0000_0000  
0000_0000_0000_0000_0000_0000_0000_0100  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 32-39  
0000_0100_0000_0000_0000_0000_0000_0000  
0000_0000_0000_0000_0000_0000_0000_0100  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111  
1111_1111_1111_1111_1111_1111_1111_1111
```

Verilog-XL Reference Programmable Logic Arrays

```

1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111

// Addresses 40-47
0000_0000_0000_0000_0010_0000_0000_0000
0000_0000_0000_0000_0000_0000_0000_0100
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111

// Addresses 48-55
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111

// Addresses 56-63
0000_0000_0001_0010_0001_0010_0000_0000
0000_0000_0001_0010_0010_0010_0000_0000
0000_0000_0001_0010_0010_0001_0000_0000
0000_0000_0010_0010_0010_0001_0000_0000
0000_0000_0010_0001_0010_0001_0000_0000
0000_0000_0010_0001_0001_0001_0000_0000
0000_0000_0010_0001_0001_0010_0000_0000
0000_0000_0001_0001_0001_0010_0000_0000

```

The output from running this example is as follows:

```

Compiling source file "test_PAL16R8"
Highest level modules:
test_PAL16R8

--CONTROLS--  --INPUTS--  x  ssss  -OUTPUTS-
clk oc  set_  phi0 phi90 4  1234 count ud
-|---|---|-----|-----|---|---|---|---|---|---|
clear registers
 1  0  0    x    x  x  0000  x  x
count up x4=L
 1  0  1    0    0  0  0101  1  1
 1  0  1    1    0  0  1101  1  0
 1  0  1    1    0  0  1001  0  1
 1  0  1    1    1  0  1011  1  0
 1  0  1    1    1  0  1010  1  1
 1  0  1    0    1  0  0010  1  0
 1  0  1    0    1  0  0110  0  1
 1  0  1    0    0  0  0100  1  0
 1  0  1    0    0  0  0101  1  1
 1  0  1    1    0  0  1101  1  0
 1  0  1    1    0  0  1001  0  1
 1  0  1    1    1  0  1011  1  0
 1  0  1    1    1  0  1010  1  1

```

Verilog-XL Reference Programmable Logic Arrays

```

1 0 1 0 1 0 0010 1 0
1 0 1 0 1 0 0110 0 1
count up x4=H
1 0 1 0 0 1 0100 1 0
1 0 1 0 0 1 0101 0 1
1 0 1 1 0 1 1101 1 0
1 0 1 1 0 1 1001 0 1
1 0 1 1 1 1 1011 1 0
1 0 1 1 1 1 1010 0 1
1 0 1 0 1 1 0010 1 0
1 0 1 0 1 1 0110 0 1
1 0 1 0 0 1 0100 1 0
1 0 1 0 0 1 0101 0 1
1 0 1 1 0 1 1101 1 0
1 0 1 1 0 1 1001 0 1
1 0 1 1 1 1 1011 1 0
1 0 1 1 1 1 1010 0 1
clear registers
1 0 0 x x x 0000 1 0
count down x4=L
1 0 1 0 0 0 0101 1 1
1 0 1 0 1 0 0111 1 0
1 0 1 0 1 0 0110 1 0
1 0 1 1 1 0 1110 1 0
1 0 1 1 1 0 1010 0 0
1 0 1 1 0 0 1000 1 0
1 0 1 1 0 0 1001 1 0
1 0 1 0 0 0 0001 1 0
1 0 1 0 0 0 0101 0 0
1 0 1 0 1 0 0111 1 0
1 0 1 0 1 0 0110 1 0
1 0 1 1 1 0 1110 1 0
1 0 1 1 1 0 1010 0 0
1 0 1 1 0 0 1000 1 0
1 0 1 1 0 0 1001 1 0
count down x4=H
1 0 1 0 0 1 0001 1 0
1 0 1 0 1 0 0101 0 0
1 0 1 1 1 0 0111 1 0
1 0 1 1 1 0 0110 0 0
1 0 1 1 1 1 1110 1 0
1 0 1 1 1 1 1010 0 0
1 0 1 0 1 1 1000 1 0
1 0 1 0 1 1 1001 0 0
1 0 1 0 1 0 0001 1 0
1 0 1 0 1 0 0101 0 0
1 0 1 1 1 0 0111 1 0
1 0 1 1 1 0 0110 0 0
1 0 1 1 1 1 1110 1 0
1 0 1 1 1 1 1010 0 0
test Hi-Z
1 1 x x x x zzzz z z
L75 "test_PAL16R8": $finish at simulation time 6100
1663 simulation events
CPU time: 1 secs to compile and load + 2 secs in simulation

```

Verilog-XL Reference

Programmable Logic Arrays

PAL16R4 Example

```
// Test program taken from MMI's "PAL (Programmable Array Logic)
// Handbook", Third Edition, shaft encoder no. 1, page 6-259.
module test_PAL16R4;
  reg
    clk, oc_, set_, phi0,phi90;
  wire
    down, s4, s3, s2, s1, up;
  // instantiate the PAL device
  PAL16R4
    cct (clk, phi0,phi90,,,,,set_,,oc_,
        down,,s4,s3,s2,s1,,up);
  parameter
    p = 100,
    L = 1'b0, H = 1'b1, X = 1'bx;
  always
    begin
      #(p/2) clk = 1;
      #(p/2) clk = 0;
    end
  initial
    begin
      $display("--CONTROLS--  --INPUTS--      ssss -OUTPUTS-");
      $display("clk oc_ set_  phi0 phi90    4321  up  down");
      $display("-|---|---|----|-----|-----|||---|----|-");
      $monitor(,clk,,,,oc_,,,,set_,,,,,phi0,,,,,phi90,,,,
        ,,s4,s3,s2,s1,,up,,,,,down);
      $monitoroff;
      #(p-1) forever
        begin
          $monitor;
          #p;
        end
    end
  initial
    begin
      $display("clear registers");
      clk=0; oc_=L; set_=L;
      #p $display("count up");
      phi0=L; phi90=L; set_=H; #p;
      #p phi0=L; phi90=H; #p;
      #p phi0=H; phi90=H; #p;
      #p phi0=H; phi90=L; #p;
      #p phi0=L; phi90=L; #p;
      #p phi0=L; phi90=H; #p;
      #p phi0=H; phi90=H; #p;
      #p phi0=H; phi90=L; #p;
      #p phi0=L; phi90=L; #p;
      #p phi0=L; phi90=H; #p;
      #p phi0=H; phi90=H; #p;
      #p phi0=H; phi90=L; #p;
      #p phi0=L; phi90=L; #p;
      #p $display("clear registers");
      phi0=X; phi90=X; set_=L;
      #p $display("count down");
      phi0=L; phi90=L; set_=H; #p;
      #p phi0=H; phi90=L; #p;
      #p phi0=H; phi90=H; #p;
      #p phi0=L; phi90=H; #p;
      #p phi0=L; phi90=L; #p;
    end
end
```


Verilog-XL Reference

Programmable Logic Arrays

```
#p phi0=H; phi90=L; #p;
#p phi0=H; phi90=H; #p;
#p phi0=L; phi90=H; #p;
#p phi0=L; phi90=L; #p;
#p phi0=H; phi90=L; #p;
#p phi0=H; phi90=H; #p;
#p phi0=L; phi90=H; #p;
#p phi0=L; phi90=L; #p;
#p $display("test Hi-Z");
    phi0=X; phi90=X; set_=X; oc_=H;
#p $finish;
end
endmodule

module PAL16R4(clock, i0,i1,i2,i3,i4,i5,i6,i7,,enable,
    o7,o6,o5,o4,o3,o2,o1,o0);
    input
        clock, i0,i1,i2,i3,i4,i5,i6,i7, enable;
    output
        o7,o6,o5,o4,o3,o2,o1,o0;
    reg //AND array outputs
        b0,b1,b2,b3,b4,b5,b6,b7,
        b8,b9,b10,b11,b12,b13,b14,b15,
        b16,b17,b18,b19,b20,b21,b22,b23,
        b24,b25,b26,b27,b28,b29,b30,b31,
        b32,b33,b34,b35,b36,b37,b38,b39,
        b40,b41,b42,b43,b44,b45,b46,b47,
        b48,b49,b50,b51,b52,b53,b54,b55,
        b56,b57,b58,b59,b60,b61,b62,b63,
        //output latches
        q2,q3,q4,q5;
    reg[0:31] //memory for array personality
        person[0:63];
    notif1 #20 //output enable drivers
        (o0, f0, b0), (o1, f1, b8),
        (o2, q2, !enable), (o3, q3, !enable),
        (o4, q4, !enable), (o5, q5, !enable),
        (o6, f6, b48), (o7, f7, b56);
    or //fixed OR array
        (f0, b1,b2,b3,b4,b5,b6,b7),
        (f1, b9,b10,b11,b12,b13,b14,b15),
        (f2, b16,b17,b18,b19,b20,b21,b22,b23),
        (f3, b24,b25,b26,b27,b28,b29,b30,b31),
        (f4, b32,b33,b34,b35,b36,b37,b38,b39),
        (f5, b40,b41,b42,b43,b44,b45,b46,b47),
        (f6, b49,b50,b51,b52,b53,b54,b55),
        (f7, b57,b58,b59,b60,b61,b62,b63);
    always @(posedge clock)
        begin
            q2 = f2;
            q3 = f3;
            q4 = f4;
            q5 = f5;
        end
    initial
        begin
            // load personality into logic array memory at the
            // beginning of simulation
            $readmemb("person16R4.dat", person, 0, 63);
            // asynchronous programmable AND array
            // evaluated automatically when input terms change
```

Verilog-XL Reference Programmable Logic Arrays

```
$async$and$array(person,
  {i0,!i0, o0,!o0, i1,!i1, o1,!o1,
   i2,!i2, !q2,q2, i3,!i3, !q3,q3,
   i4,!i4, !q4,q4, i5,!i5, !q5,q5,
   i6,!i6, o6,!o6, i7,!i7, o7,!o7},
  {b0,b1,b2,b3,b4,b5,b6,b7,
   b8,b9,b10,b11,b12,b13,b14,b15,
   b16,b17,b18,b19,b20,b21,b22,b23,
   b24,b25,b26,b27,b28,b29,b30,b31,
   b32,b33,b34,b35,b36,b37,b38,b39,
   b40,b41,b42,b43,b44,b45,b46,b47,
   b48,b49,b50,b51,b52,b53,b54,b55,
   b56,b57,b58,b59,b60,b61,b62,b63});
end
endmodule
```

The contents of file person16R4.dat are as follows:

```
// Test program personality for the shaft encoder no. 1
/* Columns
   4      8      1      1      2      2      2
   2      2      6      0      4      8  */
// Addresses 0-7
0000_0000_0000_0000_0000_0000_0000_0000
0100_1000_0001_0001_0010_0001_0000_0000
1000_0100_0010_0010_0001_0010_0000_0000
1000_1000_0010_0001_0010_0010_0000_0000
0100_0100_0001_0010_0001_0001_0000_0000
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111

// Addresses 8-15
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111

// Addresses 16-23
0100_0000_0000_0000_0000_0000_0000_0000
0000_0000_0000_0000_0000_0000_0000_0100
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111

// Addresses 24-31
0000_0000_0001_0000_0000_0000_0000_0000
0000_0000_0000_0000_0000_0000_0000_0100
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
```

Verilog-XL Reference Programmable Logic Arrays

```
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 32-39
0000_0100_0000_0000_0000_0000_0000_0000
0000_0000_0000_0000_0000_0000_0000_0100
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 40-47
0000_0000_0000_0000_0001_0000_0000_0000
0000_0000_0000_0000_0000_0000_0000_0100
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 48-55
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
```

```
// Addresses 56-63
0000_0000_0000_0000_0000_0000_0000_0000
1000_1000_0010_0010_0010_0001_0000_0000
0100_0100_0001_0001_0001_0010_0000_0000
1000_0100_0010_0001_0001_0001_0000_0000
0100_1000_0001_0010_0010_0010_0000_0000
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
1111_1111_1111_1111_1111_1111_1111_1111
```

The output from running this example through Verilog-XL is as follows:

```
Compiling source file "test_PAL16R4"
Highest level modules:
test_PAL16R4

--CONTROLS--  --INPUTS--      ssss -OUTPUTS--
clk oc_ set_  phi0 phi90      4321 up  down
-|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
clear registers
 1  0  0      x    x      0000  1   1
count up
 1  0  1      0    0      0000  1   1
 1  0  1      0    0      0000  1   1
 1  0  1      0    1      0100  0   1
```

Verilog-XL Reference Programmable Logic Arrays

```

1 0 1 0 1 1100 1 1
1 0 1 1 1 1101 0 1
1 0 1 1 1 1111 1 1
1 0 1 1 0 1011 0 1
1 0 1 1 0 0011 1 1
1 0 1 0 0 0010 0 1
1 0 1 0 0 0000 1 1
1 0 1 0 1 0100 0 1
1 0 1 0 1 1100 1 1
1 0 1 1 1 1101 0 1
1 0 1 1 1 1111 1 1
1 0 1 1 0 1011 0 1
1 0 1 1 0 0011 1 1
1 0 1 0 0 0010 0 1
1 0 1 0 0 0000 1 1
1 0 1 0 1 0100 0 1
1 0 1 0 1 1100 1 1
1 0 1 1 1 1101 0 1
1 0 1 1 1 1111 1 1
1 0 1 1 0 1011 0 1
1 0 1 1 0 0011 1 1
1 0 1 0 0 0010 0 1
1 0 1 0 0 0000 1 1
clear registers
1 0 0 x x 0000 1 1
count down
1 0 1 0 0 0000 1 1
1 0 1 0 0 0000 1 1
1 0 1 1 0 0001 1 0
1 0 1 1 0 0011 1 1
1 0 1 1 1 0111 1 0
1 0 1 0 1 1111 1 1
1 0 1 0 1 1110 1 0
1 0 1 0 1 1100 1 1
1 0 1 0 0 1000 1 0
1 0 1 0 0 0000 1 1
1 0 1 1 0 0001 1 0
1 0 1 1 0 0011 1 1
1 0 1 1 1 0111 1 0
1 0 1 1 1 1111 1 1
1 0 1 0 1 1110 1 0
1 0 1 0 1 1100 1 1
1 0 1 0 0 1000 1 0
1 0 1 0 0 0000 1 1
1 0 1 1 0 0001 1 0
1 0 1 1 0 0011 1 1
1 0 1 1 1 0111 1 0
1 0 1 1 1 1111 1 1
1 0 1 0 1 1110 1 0
1 0 1 0 1 1100 1 1
1 0 1 0 0 1000 1 0
1 0 1 0 0 0000 1 1
test Hi-Z
1 1 x x x zzzz x x
L69 "test_PAL16R4": $finish at simulation time 5500
1643 simulation events
CPU time: 1 secs to compile and load + 4 secs in simulation

```

Interconnect Delays

This chapter describes the following:

- [Overview](#) on page 421
- [Module Input Port Delays \(MIPDs\)](#) on page 423
- [Single-Source/MultiSource Interconnect Transport Delays \(S/MITDs\)](#) on page 432

Overview

Interconnect delays are delays that affect signals as they pass through a module input or inout. For information about displaying and monitoring interconnect delay signal values, see [“Monitoring Interconnect Delay Signal Values”](#) on page 342. Verilog-XL offers three types of interconnect delays:

- **Module Input Port Delays (MIPDs)**—A MIPD describes the delay to a module input port. Delays are inertial and affect three transitions: to 1, to 0, and to Z.
- **Single-source Interconnect Transport Delays (SITDs)**—A SITD is like a MIPD, but with transport delays and with global and local pulse control. SITDs affect six transitions: 0 to 1, 1 to 0, 0 to Z, Z to 0, 1 to Z, and Z to 1.
- **Multi-source Interconnect Transport Delays (MITDs)**—A MITD is like a SITD, in that they are transport delays, are subject to global and local pulse control, and affect six transitions. However, MITDs allow you to specify unique delays for each source-load path.

Note: For brevity, SITDs and MITDs are sometimes grouped together in this chapter and are referred to as S/MITDs.

Verilog-XL Reference

Interconnect Delays

Important

S/MITDs are not part of the Verilog Hardware Description Language; PLI backannotation creates MIPDs as the default form of interconnect delay and it creates S/MITDs in response to command line options. For more information about the PLI access routines for MIPDs and S/MITDs, see the *PLI 1.0 User Guide and Reference* and the *VPI User guide and Reference*.

The following table summarizes MIPDs, S/MITDs, and their differences.

Aspect	MIPDs	S/MITDs
Default status	Default type of interconnect delay	Command line option required
Delayed path characteristics	Maximum of one MIPD for each input or inout. MIPDs affect all signals passing to acceleratable gates.	Affect signal paths beginning in any source, driven by acceleratable gates, and passing through inputs or inouts to acceleratable gates
Signals subject to delay	Same delay affects signals from all sources connected to a primitive that is connected to a port with a MIPD.	Source-to-load paths can have unique delays
Transitions affected	Affect three transitions: to 1, to 0, to Z	Affect six transitions: 0->1, 1->0, 0->Z, Z->0, 1->Z, Z->1
min:typ:max triplets	Available for each of the three delays	Available for each of the six delays
System tasks' ability to observe delays	No post-interconnect delay-monitoring. Delay in signal is visible only after it passes through a primitive.	Post-interconnect delay-monitoring available. Delay in signal is visible after it passes through the input or inout port.
Delay handling	Inertial delay	Inertial or transport delay
Pulse control	Not subject to pulse control	Global and local pulse control apply.
Memory	Small memory requirement	Slightly larger memory requirement
Performance	Small performance penalty	Slightly larger performance penalty

Aspect	MIPDs	S/MITDs
Specification	MIPDs and S/MITDs are specified by PLI routines, with SDF annotation possible	

Module Import Port Delays (MIPDs)

A MIPD defines a delay to a module input or inout port. If a port comprises more than one bit, you can assign a different MIPD for each bit. If you specify a MIPD on an inout port, the delay only works on propagations into a module.

A MIPD elapses before the following:

- A module path delay begins.
- Verilog-XL looks for timing check violations

In a MIPD, you can specify rise, fall, and high-impedance propagation delays.

MIPDs are inertial delays just like the delays on primitives. An inertial delay filters out pulse widths shorter than the specified delay, and schedules at a later simulation time pulse widths that are longer than the delay. If a pulse width is just as long as a delay, you cannot predict whether Verilog-XL schedules or filters out the pulse width.

Note: MIPDs affect timing checks. If a timing check's data or reference event propagates through a port with a MIPD, the event is delayed by the MIPD. If this delay means the event is no longer within the time limit of the timing check, the timing check will not report a timing violation.

Strength changes are propagated through MIPDs, but are not affected by them.

How MIPDs Work

The purpose of MIPDs is to distribute the same delay to each of the accelerated primitives in a port's fanout. MIPDs do *not* work like buffers on ports. This section discusses the following characteristics and limitations of MIPDs:

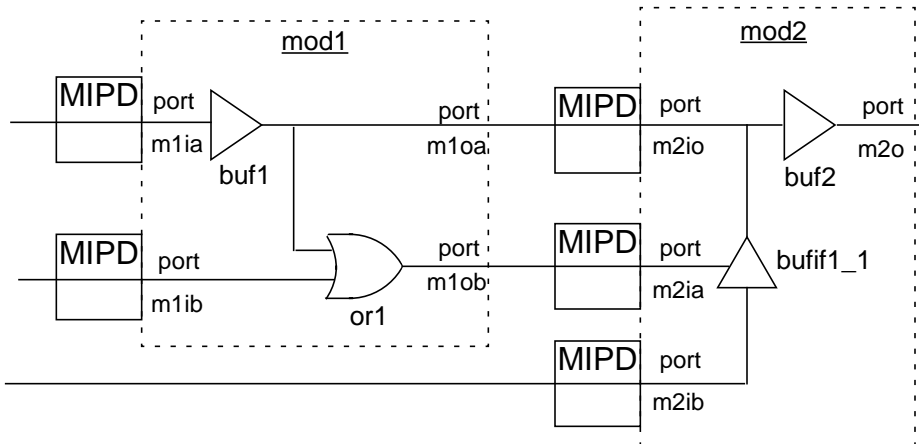
- MIPDs are specified only on input and inout ports.
- MIPDs affect all levels of the hierarchy.
- MIPDs are unidirectional.
- MIPDs are not wire path delays.

Verilog-XL Reference Interconnect Delays

- Each bit in a port can have only one MIPD.

MIPDs Are Specified Only on Input and Inout Ports

You can specify MIPDs only on input and inout ports. In the following figure, the valid ports for MIPDs are `m1ia`, `m1ib`, `m2io`, `m2ia`, and `m2ib`.



The ports are declared as follows:

Input Ports	Output Ports	Inout Ports
<code>m1ia</code>	<code>m1oa</code>	<code>m2io</code>
<code>m1ib</code>	<code>m1ob</code>	
<code>m2ia</code>	<code>m2o</code>	
<code>m2ib</code>		

As you can see in the previous figure, no MIPDs were placed on the output ports, because this is illegal.

MIPDs Affect All Levels of the Hierarchy

MIPDs are implemented so as to distribute delays to each load in a port's fanout; they require no specific hierarchical relationship between drivers and loads or between ports and loads. MIPDs do the following:

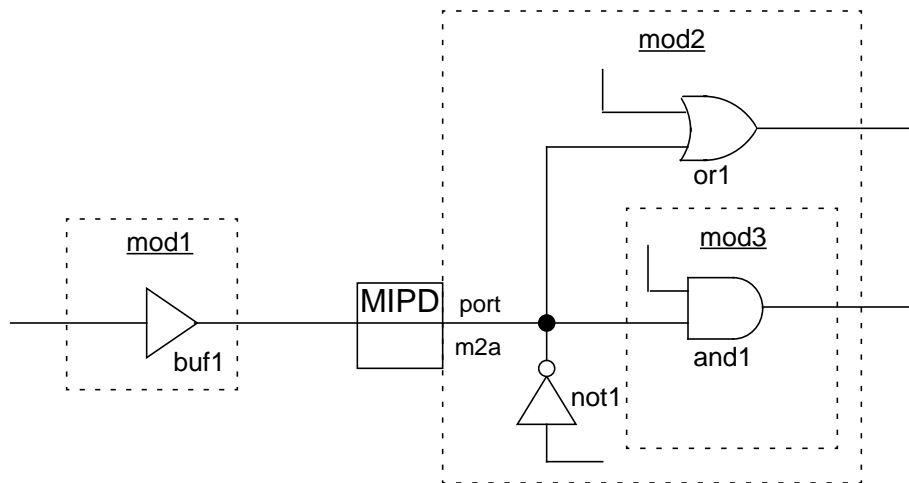
- They distribute delays to loads on the same hierarchical level as the port and to loads on a lower hierarchical level

Verilog-XL Reference

Interconnect Delays

- They delay the propagation of transitions between drivers and loads on the same or different hierarchical levels

The following figure shows the loads and drivers on different levels of a hierarchy that are affected by a MIPD:



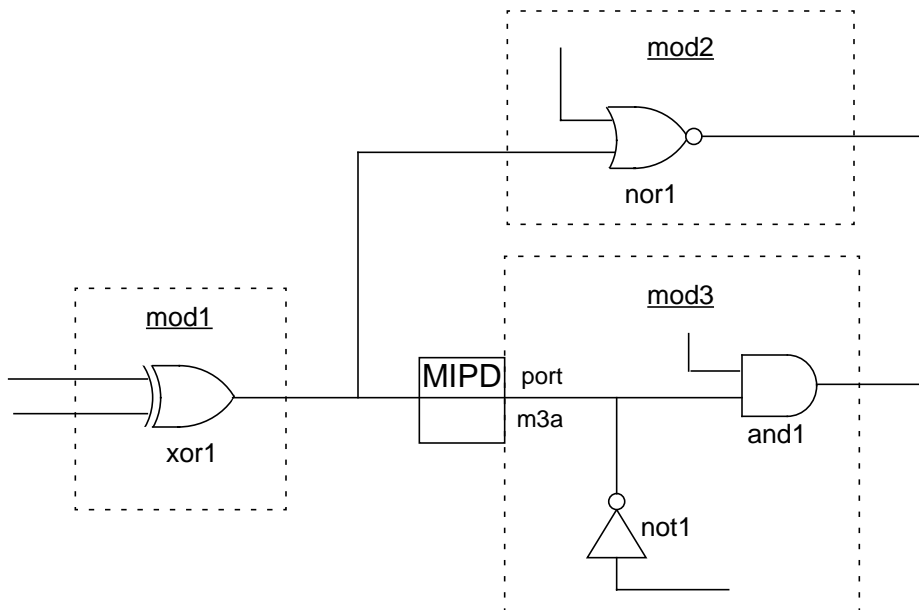
In this figure, a MIPD placed on input port `m2a` of module `mod2` specifies a delay between all the loads in port `m2a`'s fanout and each of the drivers connected to those loads. Here, the drivers are `buf1` in `mod1` and `not1` in `mod2`. The loads are `or1` in `mod2` and `and1` in `mod3`.

Notice that the MIPD specifies a delay between `not1` and both `or1` and `and1`, even though `not1` is inside of port `m2a`. The MIPD's delay value is distributed to `or1` and `and1` to delay the propagation of transitions from all their drivers, including `not1`.

Note: A MIPD can specify a delay between a driver and a load within the same module, as long as the load is in the fanout of the port on which you specify the MIPD.

MIPDs Are Unidirectional

MIPDs are unidirectional—that is, a MIPD distributes its delay only to loads *inside* the module that contains the MIPD. The following figure shows the loads that are and are not affected by a MIPD because of its unidirectional property:



In the previous figure, the MIPD on port `m3a` of `mod3` specifies a delay from drivers `xor1` and `not1` to the load `and1`. The MIPD does *not* specify a delay between these two drivers and `nor1` in module `mod2`. Even if port `m3a` were an inout port, the MIPD would not specify a delay between `not1` and `nor1`, because MIPDs are unidirectional.

MIPDs Are Not Wire Path Delays

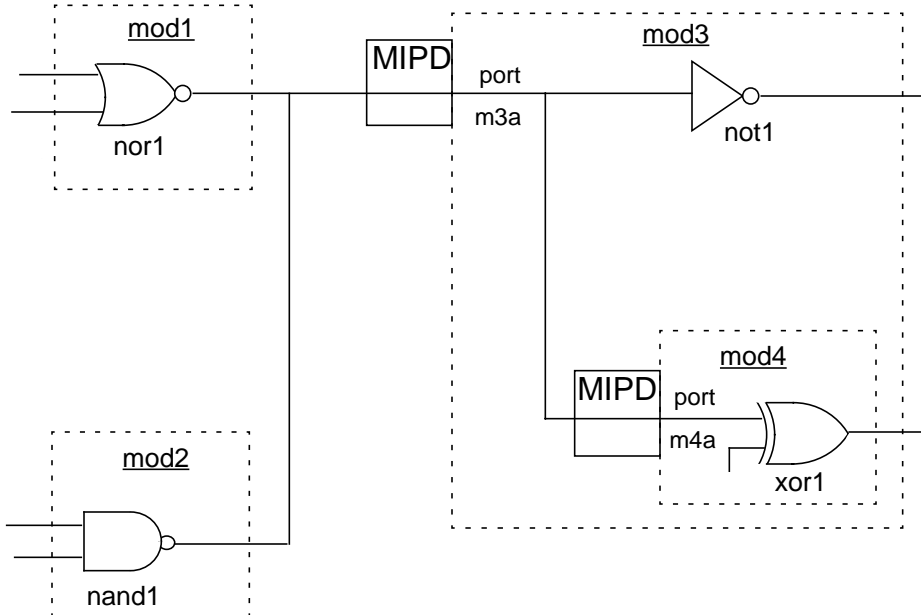
MIPDs are not specified for a wire path. Instead, they apply to all state transitions that propagate through a bit in a port. If different wire paths enter through a bit in a port, a MIPD on that bit applies equally to each wire path.

The following figure shows modules with single-bit ports and their valid MIPDs. In this figure, `mod3`'s port `m3a` and `mod4`'s port `m4a` are single-bit ports. You can insert only one MIPD on

Verilog-XL Reference

Interconnect Delays

port `m3a` that specifies a delay from both drivers `nor1` and `nand1`, to both loads `not1` and `xor1`.



You cannot assign a MIPD to port `m3a` to specify a delay between `nor1` and its loads, and then insert a second MIPD on the same port to specify a different delay between `nand1` and its loads. To do this, you must use multi-source interconnect transport delays (MITDs) (see [“An Application of MIPDs”](#) on page 431). Similarly, you cannot insert one MIPD on port `m3a` to specify a delay between the drivers and `not1`, and then insert another MIPD on port `m3a` to specify a delay between the drivers and `xor1`. A MIPD at port `m3a` affects both `not1` and `xor1`. To specify a delay between the drivers and only `xor1`, insert a MIPD at port `m4a`.

Note: If both MIPDs are specified, the propagation delay between the drivers and `xor1` is the sum of the MIPDs.

Each Bit in a Port Can Have Only One MIPD

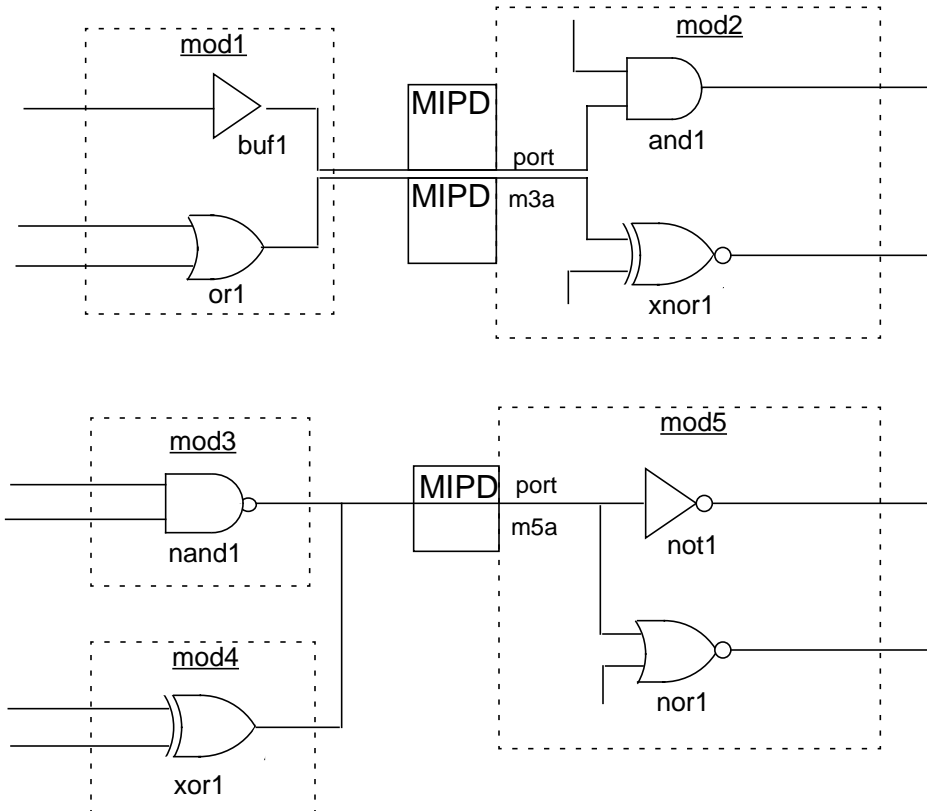
The number of MIPDs on a port cannot exceed its number of bits. Single-bit ports can only have one MIPD.

The following figure shows the valid MIPDs for multiple-bit and single-bit ports. In this figure, port `m3a` was declared to be 2 bits wide. One bit connects `buf1` to `and1` and the other bit connects `or1` to `xnor1`. You can specify one MIPD on port `m3a` that applies to both bits, or two MIPDs on port `m3a`, one for each bit. Port `m5a` is a single-bit port that can take only one

Verilog-XL Reference

Interconnect Delays

MIPD. Port `m5a`'s MIPD specifies the delay between the two drivers `nand1` and `xor1`, and the two loads `not1` and `nor1`.



Specifying MIPDs

You must specify MIPDs using PLI access routines because you do not specify MIPDs in your source description. The following list shows the access routines you need to specify MIPDs.

- `acc_append_delays` inserts delay values on an object if they do not exist, or adds delays to existing delay values on an object. Use it after you find the handle for the port.
- `acc_replace_delays` replaces existing delay values on an object with new values. It inserts new delay values if they do not exist. Use it when you need to change the delay values in a MIPD.
- `acc_fetch_delays` returns the delay values on an object. If the object has more than one bit, this routine returns the delay value for the most significant bit. To find the delay values for each bit, first get each bit's handle with `acc_next_bit`, then call `acc_fetch_delays`. Use it when you need to know the delay values of a MIPD.

Verilog-XL Reference

Interconnect Delays

- `acc_handle_object` returns a handle for a named object. Use it when you need to get the handle for a module instance.
- `acc_handle_port` returns a handle for a module's port. Use it after you get the instance handle.
- `acc_next_bit` returns a handle for the next bit in a vector port. Use it when you need to get handles for all the bits in a port.

For examples and additional information about the PLI access routines, see the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference*.

Restrictions on Ports for MIPDs

You can only insert a MIPD on an input or inout port that meets the following conditions:

- All loads in its fanout can be accelerated.
- Nets from the module input port to the loads inside the module are scalar nets or expanded vector nets.

A load can be accelerated if it is a user-defined primitive (UDP) or one of the following primitive types:

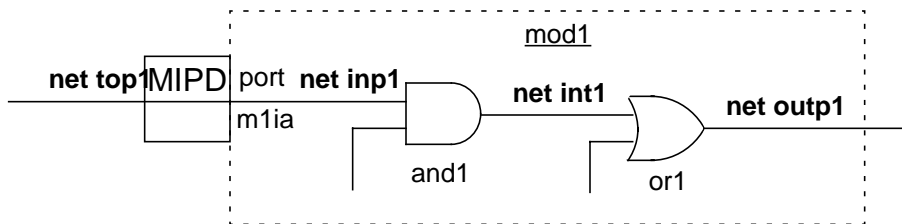
<code>buf</code>	<code>and</code>	<code>bufif0</code>	<code>nmos</code>
<code>not</code>	<code>nand</code>	<code>bufif1</code>	<code>pmos</code>
	<code>or</code>		<code>cmos</code>
<code>pullup</code>	<code>nor</code>	<code>notif0</code>	<code>rnmos</code>
<code>pulldown</code>	<code>xor</code>	<code>notif1</code>	<code>rpmos</code>
	<code>xnor</code>		<code>rcmos</code>

Behavioral constructs and bidirectional primitives are loads that cannot be accelerated. If you try to insert a MIPD on a port whose fanout includes a load that cannot be accelerated, the PLI generates a warning message and does not insert the MIPD.

You can insert a MIPD on a port of a module containing a load that is not accelerated if that load is not in the port's fanout. Therefore, to avoid using an illegal MIPD, you can insert buffers to separate a load that cannot be accelerated from an input port fanout.

Monitoring Nets Internal to MIPDs

MIPDs affect the way the `$monitor` system task displays the values and transition times of some of the nets in a module. The following figure shows how MIPDs affect the way these nets are monitored in a module:



MIPDs have no effect on the values and transition times displayed by the `$monitor` system task for a net that connects a port to a load in the port's fanout. This is because port collapsing applies the values and transition times of the net outside a port to the net inside a port.

MIPDs do not change port collapsing results. In the previous figure, nets `top1` and `inp1` are connected by port `m1ia`. Port collapsing requires the `$monitor` system task to display the same values and transition times for `top1` and `inp1`. However, MIPDs change the values and transition times displayed by the `$monitor` system task of all nets not affected by port collapsing that are in the path from the input port to the output. In the previous figure, the `$monitor` system task displays the values of `int1` and `outp1` changing at later simulation times because port `m1ia` has a MIPD.

Displaying Status Information for Nets Internal to MIPDs

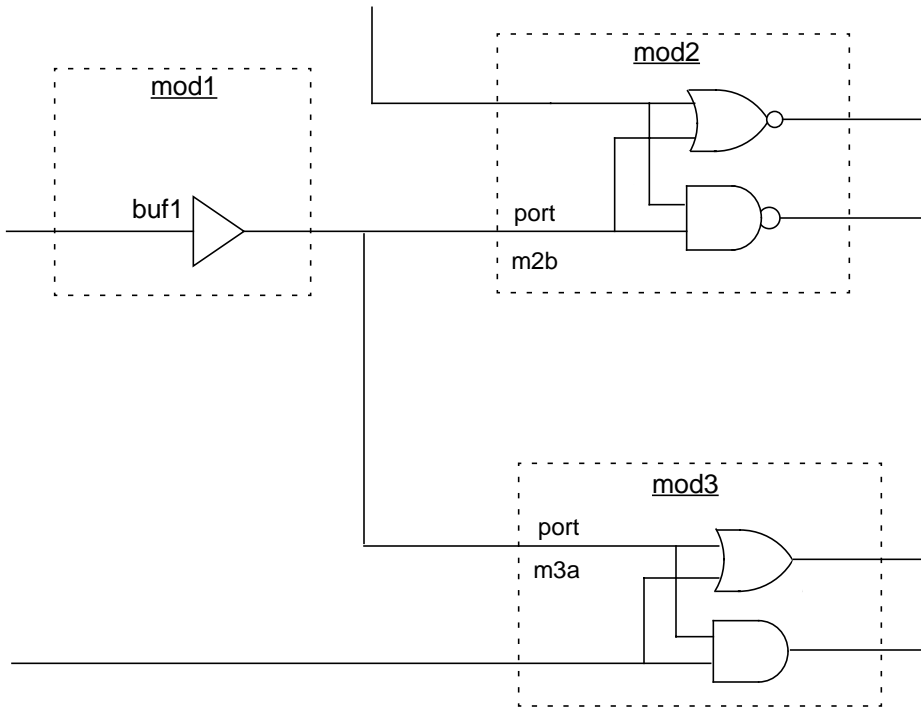
If you enter the `$showvars` system task for a net that connects a port with a MIPD to a load in the port's fanout, Verilog-XL displays status information with the following warning message:

```
Warning: input port delay exists between a net and its driver
```

Verilog-XL generates this message because the status information comes from the net on the other side of the port, but port collapsing applies it to the net inside the port. If you enter this system task for other nets in the path from the input port to the output, Verilog-XL issues no warning message.

An Application of MIPDs

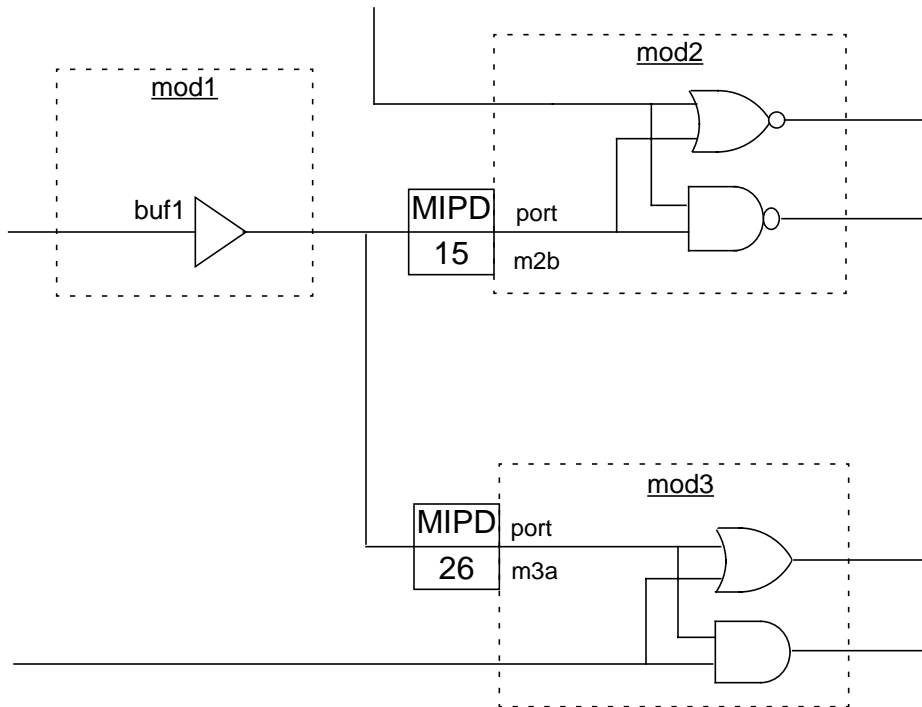
The following figure illustrates one use of MIPDs:



In this figure, `buf1` in module `mod1` drives loads in modules `mod2` and `mod3`. Suppose you want to model a propagation delay of 15 time units between `buf1` and `mod2`, and a propagation delay of 26 time units between `buf1` and `mod3`. A lumped delay of 26 on `buf1` could accurately model the propagation delay to `mod3`, but could not model the propagation delay to `mod2` as desired.

Verilog-XL Reference Interconnect Delays

The solution to this problem is shown in the next figure:



The MIPDs in this figure accurately model the propagation delays from `buf1` to both `mod2` and `mod3`. Here, the propagation delay from `buf1` to each of its loads in `mod2` is 15 time units, and the propagation delay from `buf1` to each of its loads in `mod3` is 26 time units.

Single-Source/MultiSource Interconnect Transport Delays (S/MITDs)

S/MITDs, like MIPDs, implement delay information for paths connecting sources with loads through module input or inout ports. For single source nets, a SITD is much like a MIPD in that it is a delay associated with a module input port, but a SITD supports transport delays and pulse limits. For multi-source nets, a MITD is used because it can handle unique delays and pulse limits between all source/load pairs.

A S/MITD can impose a delay on multiple paths from the same source to different loads, or it can implement different delays on multiple paths from different sources to different loads. There is no limit on the number of paths affected by S/MITDs that can pass through a module port.

The delay information to which a S/MITD refers is like the delay information for a module path delay: It can consist of one, two, three, or six types of transitions, and each transition can have

Verilog-XL Reference

Interconnect Delays

minimum, typical, and maximum variations. Unlike module path delays, S/MITDs cannot be conditional, but the ability to give 18 delay specifications (6 delays plus pulse reject and pulse error values for each delay) for every source-to-load path can still model hardware delays accurately.

S/MITDs are unidirectional. They affect signals passing into a module through the module's inout port, but then do not affect signals travelling in the opposite direction. Paths affected by S/MITDs preserve the strength of signals passing through them.

Note: S/MITDs are not part of the Verilog Hardware Description Language; PLI backannotation creates S/MITDs.

SITDs and MITDs have different characteristics designed to model hardware accurately while conserving memory and maintaining performance. The following table shows the similarities and differences between SITDs and MITDs.

Characteristics	SITDs	MITDs
Unique delays for each source-to-load path	Always available, because there is only one source	Available, but controlled by command line options
Observability by system tasks after the module port	Both the pre- and post- delay signal values are observable	Same as SITDs
Observability at primary output	Observable	Observable
6 delays plus 2 pulse limits per delay	Always available, but allocated only when needed	Always available
Transport delay	Always available, but allocated only when needed	Always available
Pulse control	Always available, but allocated only when needed	Always available

Controlling MIPD and S/MITD Creation

Two options control the creation of MIPDs and S/MITDs:

```
+transport_int_delays  
+multisource_int_delays
```

Verilog-XL Reference

Interconnect Delays

This section discusses the effects of using each option alone, of using them together, and of omitting both options.

Using `+transport_int_delays` Alone

MIPDs are the default type of interconnect delay. If you use the `+transport_int_delays` plus option without using the `+multisource_int_delays` option, Verilog-XL inserts SITDs instead of MIPDs everywhere. The following table summarizes the effects of using the `+transport_int_delays` option without the `+multisource_int_delays` option.

Characteristics	Single-source Nets	Multi-source Nets
Unique delays for each source-to-load path	Always available, because there is only one source	Unavailable. As with MIPDs, all signals propagating to primitives connected to the port have the same delay.
6 delays plus 2 pulse limits per delay	Available, but allocated only when needed	Available
Transport delay	Available	Available
Pulse control	Available	Available

Using `+multisource_int_delays` Alone

If you use the `+multisource_int_delays` plus option without the `+transport_int_delays` option, Verilog-XL inserts MITDs on all multi-source nets and MIPDs on all single-source nets. The following table summarizes the effects of using the `+multisource_int_delays` plus option without using the `+transport_int_delays` option:

Characteristics	Single-source Nets	Multi-source Nets
Unique delays for each source-to-load path	Unavailable; simulated as MIPD	Available
6 delays plus 2 pulse limits per delay	Unavailable; simulated as MIPD	Available
Transport delay	Unavailable; simulated as MIPD	Available

Verilog-XL Reference Interconnect Delays

Characteristics	Single-source Nets	Multi-source Nets
Pulse control	Unavailable; simulated as MIPD	Available

The following table shows how MITD delay specifications supply information for the delays in MIPDs:

S/MITD delay specification mappings

MITD Delay Specifications	MIPD to 1	MIPD to 0	MIPD to Z
Two (Rise and Fall)	Rise	Fall	max(Rise, Fall)
Three (Rise, Fall, Z)	Rise	Fall	Z
Six (0->1, 1->0, 0->Z, Z->0, 1->Z, Z->1)	0 -> 1	1 -> 0	0 -> Z

Using Both `+transport_int_delays` and `+multisource_int_delays`

Using the `+transport_int_delays` and the `+multisource_int_delays` options in combination enables maximum interconnect delay functionality, as shown in the following table.

Characteristics	Single-source Nets	Multi-source Nets
Unique delays for each source-to-load path	Always available, because there is only one source.	Available
6 delays plus 2 pulse limits per delay	Available	Available
Transport delay	Available	Available
Pulse control	Available	Available

Using Neither Option

In the absence of both the `+transport_int_delays` option and the `+multisource_int_delays` option, backannotation creates only MIPDs even if specifications sufficient to create S/MITDs exist. "" on page 435 shows how S/MITD delay specifications map to MIPD specifications in the absence of both the `+transport_int_delays` and `+multisource_int_delays` plus options.

S/MITDs and Pulse Handling

You can use global pulse control to make S/MITDs recognize pulses shorter than a specified percentage of a delay, and either reject such pulses or treat them as having the `e` value. Global pulse control can also make S/MITDs recognize pulses for rejection using the SLDI delays as criteria, or it can prevent S/MITDs from rejecting pulses.

“[Specifying Global Pulse Control on Module Paths](#)” on page 257 explains global pulse control for module paths. Global pulse control for S/MITDs is analogous to global pulse control for module paths. You can specify global path pulse limits for S/MITDs as follows:

- Use the `+pulse_r/m` and `+pulse_e/n` options to specify global path pulse control. The limits you specify apply to both S/MITDs and module paths in the same way.
- Specify reject and error limits for S/MITDs and for module paths separately in the same simulation. To do this, enter two pairs of options on the simulation command line:
 - The `+pulse_r/m` and `+pulse_e/n` options
 - The `+pulse_int_r/m` and `+pulse_int_e/n` options. The reject and error limits specified in these plus options affect S/MITDs only.

The `+transport_path_delays` option, which makes module paths show transport delay behavior has no effect on interconnect delays. Specific pulse control of the type that `PATHPULSE$` implements for module paths is not available for S/MITDs, but SDF annotation can uniquely specify pulse limits for all transition types.

Preventing Pulse Limit Backannotation

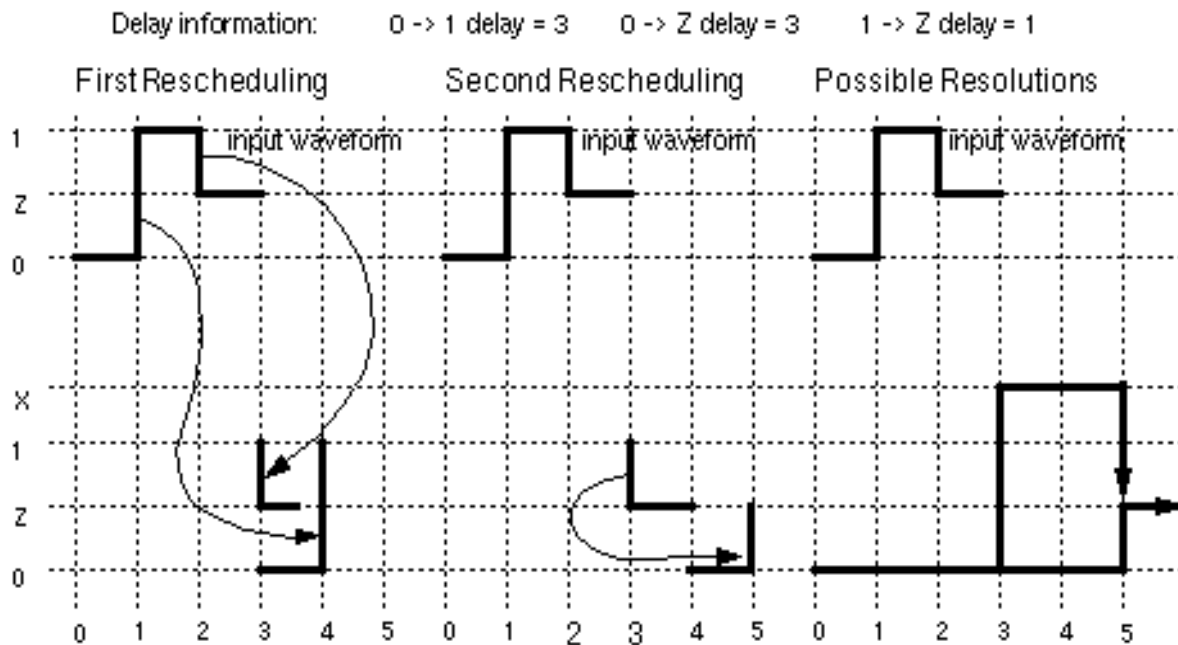
The `+no_pulse_int_backanno` option prevents PLI backannotation of pulse limits. Only one warning message is issued on the first attempt.

Resolving Ambiguous S/MITD Events

Verilog-XL maintains a table of currently scheduled events and their expected simulation times for each S/MITD output. A new event entering the table that has an expected simulation time earlier than the expected simulation times of other events on the table cancels the later events. This is similar to what happens with module path delays (see “[Understanding Path](#)”

Verilog-XL Reference Interconnect Delays

Delays" on page 239). This policy can have the paradoxical results shown in the following figure.



In this figure, the waveform at the S/MITD source is shown three times. In the first case, the output transition to Z cancels the later transition to 1. The transition to Z must then be rescheduled as a transition from 0, rather than from 1. Reinstating the 1 -> Z transition would make it appear with the wrong delay.

Using the delay for the 1 -> Z transition after cancelling the 1 -> Z transition would also be incorrect. Verilog-XL offers two resolutions:

- A transition to Z using the 0->Z delay time
- A transition to X using the 1 -> Z delay, followed by a transition to Z using the 0->Z delay time

If you want the X value, use the `+x_transport_pessimism` command line option. This option reflects the ambiguous delay situation by propagation of an X value during the ambiguous period. Without this option, the S/MITD propagates the value that would have followed the ambiguous period using the proper delay.

Note: The previous figure deals only with value changes. Handling for strength changes is different.

PLI Tasks for S/MITDs

The PLI routines for S/MITDs are the same as those for MIPDs, but S/MITDs have different and/or additional arguments. See the table in [“Specifying MIPDs”](#) on page 428 for a list of the routines.

A new `acc_configure` option called `accMultiSrcInt` must be set in order for S/MITDs to be used instead of MIPDs. The SDF annotator already sets this option.

For examples and additional information about the routines and their arguments, see the “Known Problems and Solutions” (KP & S) document that accompanies the release of this version.

Timescales

This chapter describes the following:

- [Overview](#) on page 439
- [The `'timescale` Compiler Directive](#) on page 440
- [Timescale System Functions](#) on page 442
- [The Timescale System Tasks](#) on page 445

Overview

Timescales let you use modules that were developed with different time units together in the same simulation. Verilog-XL can, for example, simulate a design that contains both a module whose delays are specified in nanoseconds and a module whose delays are specified in picoseconds.

To use modules with different time units in the same simulation, you need the following timescale constructs:

- The `'timescale` compiler directive specifies the unit of measurement for time and the degree of precision of the time in the modules in your design.
- The `$printtimescale` system task displays the time unit and precision specified for a given module.
- The `$time` and `$realtime` system functions, the `$timeformat` system task, and the `%t` format specification specify how Verilog-XL reports time information. The system function `$timeformat` also specifies the time unit you use in the delays entered interactively.
- The `$scale` system function allows the use of time values from one module with time values from another module that has a different time unit.

The `'timescale` Compiler Directive

The `'timescale` compiler directive specifies the unit of measurement for time and delay values, as well as the degree of accuracy, for the delays in all subsequent modules until Verilog-XL reads another `'timescale` compiler directive. This rule applies to modules in subsequent files also.

Usage Rules

A `'timescale` compiler directive applies to all module definitions that follow it in the source description until the compiler reads another `'timescale` compiler directive. It must be specified before and outside a module definition. The directive must not appear between the `module` and `endmodule` keywords nor should it be specified for an instantiation of a module.

The `'timescale` compiler directive is optional. However, if your design includes a `'timescale` compiler directive for any module definition, you must include it before and outside the first module definition.

Syntax

The syntax is as follows:

```
'timescale <time_unit> / <time_precision>
```

The `<time_unit>` argument specifies the unit of measurement for times and delays.

The `<time_precision>` argument specifies the degree of precision to which Verilog-XL rounds delay values before using them in simulation. The values Verilog-XL uses will be accurate to within the unit of time that you specify. The smallest `<time_precision>` argument of all the `'timescale` compiler directives in the design determines the time unit for the simulation.

The `<time_precision>` argument must be at least as precise as the `<time_unit>` argument; it cannot specify a longer unit of time than `<time_unit>`.

The integers in the above arguments specify an order of magnitude for the size of the value; the valid integers are 1, 10, and 100. The character strings represent units of measurement; the valid character strings are `s`, `ms`, `us`, `ns`, `ps`, and `fs`.

The units of measurement specified by these character strings are shown in the following table:

<code>s</code>	seconds
----------------	---------

Verilog-XL Reference Timescales

ms milliseconds
us microseconds
ns nanoseconds
ps picoseconds
fs femtoseconds

The following table shows how and when the ``timescale` compiler directive scales values:

Value of:	Scaled at compile time	Scaled at run time	Rounded by time precision	Not rounded by time precision
Accelerated gate delays	X		X	
Unaccelerated gate delays		X	X	
Time limits on \$setup and \$hold	X		X	
Simulation time \$time and \$realtime		X		X

The following example shows how this directive is used:

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

Here, all time values in the modules that follow are multiples of 1 nanosecond because the `<time_unit>` argument is `1 ns`. Delays are rounded to real numbers with three decimal places—that is, the values are precise to within one thousandth of a nanosecond—because the `<time_precision>` argument is `1 ps`, or one thousandth of a nanosecond.

Consider this example:

```
`timescale 10 us / 100 ns
```

The time values in the modules that follow this directive are multiples of 10 microseconds because the `<time_unit>` argument is `10 us`. Delays are rounded to within one tenth of a microsecond because the `<time_precision>` argument is `100 ns`, or one tenth of a microsecond.

The following example shows a ``timescale` directive in the context of an actual source description:

Verilog-XL Reference

Timescales

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter d = 1.55;
  initial
  begin
    #d set = 0;
    #d set = 1;
  end
endmodule
```

In the previous example, the ``timescale 10 ns / 1 ns` compiler directive specifies that the time unit for module `test` is 10 nanoseconds. As a result, the time values in the module are multiples of 10 nanoseconds rounded to the nearest nanosecond, and therefore, the value stored in parameter `d` is scaled to a delay of 16 nanoseconds. This means that Verilog-XL assigns the value 0 to `reg set` at simulation time 16 nanoseconds (1.6×10 ns), and then assigns it the value 1 at simulation time 32 nanoseconds.

Note: parameter `d` retains its value no matter which timescale is in effect.

The simulation times in the previous example are determined as follows:

1. The value of parameter `d` is rounded from 1.55 to 1.6 according to the time precision.
2. The time unit of the module is 10 nanoseconds, and the precision is 1 nanosecond, so Verilog-XL scales the delay of parameter `d` from 1.6 to 16.
3. Verilog-XL schedules the assignment of 0 to `reg set` at simulation time 16 nanoseconds (Verilog-XL adds 16 nanoseconds to the current simulation time of 0) and the assignment of 1 at simulation time 32 nanoseconds (Verilog-XL adds 16 nanoseconds to the current simulation time of 16 nanoseconds). Verilog-XL does not round time values when it schedules these assignments.

Effects of Timescales on Simulation Performance

Verilog-XL simulates with one time unit for the entire design. The simulation time unit is the smallest `<time_precision>` argument specified in all the ``timescale` compiler directives in the design. During compilation, Verilog-XL converts the delays on accelerated gates to the simulation time unit. This means that the larger the difference between the smallest `time_precision` argument and the typical delay on accelerated gates the more memory your design requires and the slower the simulation speed. Therefore, you should make your `<time_precision>` arguments no smaller than is necessary.

Timescale System Functions

Verilog-XL has the following timescale system functions:

- “`$time`” on page 443
- “`$realtime`” on page 444
- “`$scale`” on page 444

The `$time` and `$realtime` system functions allow you to access the current simulation time.

The `$scale` system function converts time values in a module that uses one time unit, so that these time values can be used in another module that uses a different time unit.

`$time`

The `$time` system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it. The following example shows a use of the `$time` system function:

```
`timescale 10 ns / 1 ns
module test;
  reg set;
  parameter p = 1.55;
  initial
  begin
    $monitor($time, , "set=%b", set);
    #p set = 0;
    #p set = 1;
  end
endmodule
```

The results from the previous example are as follows:

```
0 set=x
2 set=0
3 set=1
```

In this example, Verilog-XL assigns to `reg set` the value 0 at simulation time 16 nanoseconds, and the value 1 at simulation time 32 nanoseconds. Note that these times do not match the times reported by `$time`. The time values returned by the `$time` system function are determined as follows:

1. Verilog-XL scales the simulation times 16 and 32 nanoseconds to 1.6 and 3.2, because the time unit for the module is 10 nanoseconds, so time values reported by this module are in multiples of 10 nanoseconds.
2. Verilog-XL rounds 1.6 to 2 and 3.2 to 3, because the `$time` system function returns an integer. The `<time_precision>` argument does not cause Verilog-XL to round these values.

\$realtime

The `$realtime` system function returns a real number time that, like `$time`, is scaled to the time unit of the module that invoked it.

The following example shows a use of the `$realtime` system function:

```
`timescale 10 ns / 1 ns
module test;
  reg set;
  parameter p = 1.55;
  initial
  begin
    $monitor($realtime,,"set=%b",set);
    #p set = 0;
    #p set = 1;
  end
endmodule
```

The results from the previous example are as follows:

```
0 set=x
1.6 set=0
3.2 set=1
```

In this example, the event times in `reg set` are multiples of 10 nanoseconds, because 10 nanoseconds is the time unit for the module. They are real numbers because `$realtime` returns a real number.

\$scale

The `$scale` system function allows you to take a time value from a module that uses one time unit and use it in a module that uses a different time unit. This system function takes a hierarchical-name reference argument (such as a delay parameter) and converts its value to the time unit of the module that invokes it. This system function returns a real number. The syntax is as follows:

```
$scale <hierarchical_name>;
```

The following example shows the use of `$scale`:

```
`timescale 1 s / 1 ms
module seconds;
  initial
    $display("p=%10.5f\n", $scale(milli.p));
endmodule

`timescale 1 ms / 1 ms
module milli;
  parameter p = 1;
endmodule
```

Verilog-XL Reference

Timescales

In module `milli`, parameter `p` has the value of 1 millisecond because it is assigned the value 1, and the preceding ``timescale` directive specifies that all time values in the module be multiples of 1 millisecond.

In module `seconds`, the time unit is 1 second, so all time values in the module are in multiples of 1 second; the precision is to 1 millisecond, or one thousandth of a second, as specified by ``timescale 1 s / 1 ms`. The `$scale` system function converts the value of `p` from 1 millisecond in module `milli` to 0.001 second in module `seconds`.

The following example shows another use of the `$scale` system function:

```
`timescale 100 s / 1 ms
module first;
  initial
    $display("p=%10.5f\n", $scale(next.p));
endmodule

`timescale 10 ms / 10 ms
module next;
  parameter p = 1;
  ...
endmodule
```

In this, the output from the `$display` system task is as follows:

```
p= 0.00010
```

In module `next`, parameter `p` has a value of 10 milliseconds because it is assigned the value 1 and the preceding ``timescale` compiler directive specifies that all time values in the module be multiples of 10 milliseconds.

In module `first`, all time values in the module are multiples of 100 seconds, rounded to a precision of 1 millisecond, as specified by ``timescale 100 s / 1 ms`. The `$scale` system function takes the value stored in parameter `p` and converts it from a value of 1 in module `next`, to a value of 0.0001 in module `first`. This is because Verilog-XL assumes the argument passed to `$scale` is a delay parameter and converts it to the time unit of the module that invokes `$scale`.

The Timescale System Tasks

The following system tasks display and set timescale information:

- ["\\$sprinttimescale"](#) on page 446
- ["\\$timeformat"](#) on page 446

\$printtimescale

The `$printtimescale` system task displays the time unit and precision for a particular module. The syntax is as follows:

```
$printtimescale <hierarchical_name>;
```

This system task can be specified with or without an argument, as follows:

- When no argument is specified, `$printtimescale` displays the time unit and precision of the module that is the current scope (as set by `$scope`).
- When an argument is specified, `$printtimescale` displays the time unit and precision of the module passed to it.

The timescale information appears in the following format:

```
Time scale of (module_name) is unit / precision
```

The following example shows the use of the `$printtimescale` system task.

```
`timescale 1 ms / 1 us
module a_dat;
  initial
    $printtimescale(b_dat.c1);
endmodule

`timescale 10 fs / 1 fs
module b_dat;
  c_dat c1 ();
endmodule

`timescale 1 ns / 1 ns
module c_dat;
  ...
endmodule
```

In the previous example, module `a_dat` invokes the `$printtimescale` system task to display timescale information about another module `c_dat`, which is instantiated in module `b_dat`.

The information about `c_dat` is displayed in the following format:

```
Time scale of (b_dat.c1) is 1ns / 1ns
```

\$timeformat

The `$timeformat` system task performs the following two functions:

- Sets the time unit for all subsequent delays entered interactively

Verilog-XL Reference

Timescales

- Sets the time unit, precision number, suffix string, and minimum field width for all `%t` formats specified in all modules that follow it in the source description until another `$timeformat` system task is invoked. The `%t` format specification works with the `$timeformat` system task to specify the uniform time unit, the time precision, and the format that Verilog-XL uses to report timing information from various modules that have different time units and precisions. You can use `%t` with the `$display`, `$monitor`, `$write`, `$strobe`, `$fdisplay`, `$fmonitor`, `$fwrite`, and `$fstrobe` system tasks.

The syntax is as follows:

```
$timeformat (<units_number>,  
            <precision_number>,  
            <suffix_string>,  
            <minimum_field_width>);
```

The `$timeformat` system task arguments are described as follows:

Argument	Description
<code><units_number></code>	An integer in the range from 0 to -15 representing a time unit. The default is the smallest <code><time_precision></code> argument of all the <code>`timescale</code> compiler directives in the source description.
<code><precision_number></code>	An integer defining the precision of the <code><units_number></code> . For example, a value of 5 with a -9 <code><units_number></code> indicates a precision of 5 nanoseconds. The default value is 0.
<code><suffix_string></code>	A quoted string that you can use to clarify the output of the <code>\$timeformat</code> system task. For example, " ns" can be printed with the output to indicate nanoseconds. The default is a null character string.
<code><minimum_field_width></code>	An integer specifying the minimum field width to report the time of the event. The default is 20.

The `<units_number>` argument must be an integer in the range from 0 to -15. This argument represents the time unit as follows:

Unit Number	Time Unit	Unit Number	Time Unit
0	1 s	-8	10 ns
-1	100 ms	-9	1 ns

Verilog-XL Reference Timescales

Unit Number	Time Unit	Unit Number	Time Unit
-2	10 ms	-10	100 ps
-3	1 ms	-11	10 ps
-4	100 us	-12	1 ps
-5	10 us	-13	100 fs
-6	1 us	-14	10 fs
-7	100 ns	-15	1 fs

The following example shows the use of %t with the \$timeformat system task to specify a uniform time unit, time precision, and format for timing information.

Note: The following example requires a large amount of memory because of the large difference between the smallest time precision (1ps) and the largest delay (10ns). See [“Effects of Timescales on Simulation Performance”](#) on page 442 for more information.

```
`timescale 1 ms / 1 ns
module cntrl;
  initial
    $timeformat(-9, 5, " ns", 10);
endmodule

`timescale 1 fs / 1 fs
module al_dat;
  reg in1;
  integer file;
  buf #10000000 (o1,in1);
  initial
  begin
    file = $fopen("al.dat");
    #00000000 $fmonitor(file,"%m: %t in1=%d
    o1=%h", $realtime,in1,o1);
    #10000000 in1 = 0;
    #10000000 in1 = 1;
  end
endmodule

`timescale 1 ps / 1 ps
module a2_dat;
  reg in2;
  integer file2;
  buf #10000 (o2,in2);
  initial
  begin
    file2=$fopen("a2.dat");
    #00000 $fmonitor(file2,"%m: %t in2=%d
    o2=%h",
    $realtime,in2,o2);
    #10000 in2 = 0;
    #10000 in2 = 1;
  end
endmodule
```


Verilog-XL Reference

Timescales

The contents of file `a1.dat` are as follows:

```
a1_dat: 0.00000 ns in1= x o1=x
a1_dat: 10.00000 ns in1= 0 o1=x
a1_dat: 20.00000 ns in1= 1 o1=0
a1_dat: 30.00000 ns in1= 1 o1=1
```

The contents of file `a2.dat` are as follows:

```
a2_dat: 0.00000 ns in2=x o2=x
a2_dat: 10.00000 ns in2=0 o2=x
a2_dat: 20.00000 ns in2=1 o2=0
a2_dat: 30.00000 ns in2=1 o2=1
```

In this example, the times of events written to the files by the `$fmonitor` system task in modules `a1_dat` and `a2_dat` are reported as multiples of 1 nanosecond even though the time units for these modules are 1 femtosecond and 1 picosecond respectively. This is because the first argument of the `$timeformat` system task is `-9` and the `%t` format specification is included in the arguments to `$fmonitor`. This time information is reported after the module names with five fractional digits, followed by an `ns` character string in a space wide enough for 10 ASCII characters.

Timescales Examples

The following examples include three modules called `ts_1ms_1ns`, `ts_1us_1ns`, and `ts_1ns_1ns`. They each use the following timescale constructs:

- ``timescale` compiler directive
- `$time`, `$realtime`, and `$scale` system functions
- `$prinntimescale` and `$timeformat` system tasks
- `%t` format specification.

The comments in the code indicate the numbers to which you should refer in the explanation after each module.

Timescale Example `ts_1ms_1ns`

```
`timescale 1 ms / 1 ns // See 1
module ts_1ms_1ns;
  initial
  begin
    $timeformat (-9 ,3 ," ns" ,20 ); // See 2, 3, 4, 5
    $prinntimescale; // See 6
    $display;
    $prinntimescale(ts_1us_1ns); // See 7
    $display("p = %15.9f\n",
    $scale(ts_1us_1ns.p)); // See 8
```

Verilog-XL Reference

Timescales

```
    $prnttimescale(ts_1ns_1ns);           // See 9
    $display("p = %15.9f\n",
    $scale(ts_1ns_1ns.p));               // See 10
end
endmodule
```

1. This specifies that all module definitions that follow this directive have a time unit of 1 millisecond and a time precision of 1 nanosecond.
2. The first argument specifies that data displayed in the %t format specification appears in the 1 nanosecond time unit.
3. The second argument specifies that the event time is reported with three fractional digits.
4. The third argument specifies that the event time is followed by the ns character string to indicate the time unit is nanoseconds.
5. The fourth argument specifies at least a 20-character field width to report the time of the event. If the width exceeds 20, Verilog-XL adds additional character widths.
6. This \$prnttimescale system task displays the time unit and the time precision of the module that defines the current scope.
7. This \$prnttimescale system task displays the time unit and the time precision of the module ts_1us_1ns (defined below).
8. This \$scale system task reports the value of p in module ts_1us_1ns as a floating point number with up to a 15-character width and up to 9 fractional digits; since module ts_1us_1ns has a time unit of 1 us, Verilog-XL divides the value of p in ts_1us_1ns by 1000.
9. This \$prnttimescale system task displays the time unit and the time precision of the module ts_1ns_1ns.
10. This \$scale system task reports the value of p in module ts_1ns_1ns as a floating point number with up to a 15-character width and up to 9 fractional digits. Since module ts_1ns_1ns has a time unit of 1 nanosecond, it divides the value of p by 1,000,000.

Timescale Example ts_1us_1ns

```
`timescale 1us / 1ns                               // See 1
module ts_1us_1ns;
  reg in;
  parameter p = 123.456789;
  buf #(10.415111) (o1, in);                       // See 2
  initial
  begin
    #100 $display;
    #000 $fmonitor(1, "%m:                          // See 3
    %t %d %9.3f" ,                                  // See 4, 5, 6
    $realtime,                                       // See 7
  end
```

Verilog-XL Reference

Timescales

```
        $time,                // See 8
        $realtime            // See 9
        ,,, in,, ol);
    #100 in = 0;
    #100 in = 1;
end
endmodule
```

1. A new ``timescale` compiler directive overrides the previous one, and this new directive now applies to all subsequent module definitions.
2. This delay is in multiples of 1 microsecond because the 1 us time unit was specified in the preceding ``timescale` compiler directive.
3. This `$fmonitor` system task displays the name of the module in which the event occurs.
4. This `%t` displays timing information as specified in the last `$timeformat` system task in module `ts_1us_1ns`.
5. This `%d` displays a decimal value.
6. This `%9.3f` displays a value in a 9-character width with 3 fractional digits.
7. This `$realtime` system task returns a real value for time, corresponding to the `%t` format specification.
8. This `$time` system task returns an integer value for time, corresponding to the `%d` format specification.
9. This `$realtime` system task returns a real value for time, corresponding to the `%9.3f` format specification.

Timescale Example `ts_1ns_1ns`

```
`timescale 1ns / 1ns                // See 1
module ts_1ns_1ns;
    reg in;
    parameter p = 123.456789;
    buf #(10.415111) (ol, in);        // See 2
    initial
    begin
        #100 $display;
        #000 $fmonitor(1, "%m: %t %d %9.3f", $realtime, $time,
            $realtime,,,in,,ol);
        #100 in = 0;
        #100 in = 1;
    end
endmodule
```

1. A new ``timescale` compiler directive overrides the previous one. It sets the time unit for subsequent module definitions to 1 ns and the time precision to 1 nanosecond.

Verilog-XL Reference Timescales

2. This delay value is a multiple of 1 nanosecond; Verilog-XL rounds this delay to 10 because the time precision argument is the same as the time unit argument.

Timescales Examples Output

The output of the `ts_1ms_1ns`, `ts_1us_1ns`, and `ts_1ns_1ns` source modules is as follows:

```
Time scale of (ts_1ms_1ns) is 1ms / 1ns

Time scale of (ts_1us_1ns) is 1us / 1ns
p = 0.123456789
Time scale of (ts_1ns_1ns) is 1ns / 1ns
p = 0.000123457
/* See 1          See 2          See 3 */
ts_1ns_1ns:      100 ns          100          100.000          x x
ts_1ns_1ns:      200 ns          200           200.000          0 x
ts_1ns_1ns:      210 ns          210           210.000          0 0
ts_1ns_1ns:      300 ns          300           300.000          1 0
ts_1ns_1ns:      310 ns          310           310.000          1 1

/*          See 4          See 5 */
ts_1us_1ns:      100000 ns       100           100.000          x x
ts_1us_1ns:      200000 ns       200           200.000          0 x
ts_1us_1ns:      210415 ns       210           210.415          0 0
ts_1us_1ns:      300000 ns       300           300.000          1 0
ts_1us_1ns:      310415 ns       310           310.415          1 1
```

1. Events from module `ts_1ns_1ns` occur first because it has the smaller time unit and the same delay numbers as `ts_1us_1ns`.
2. These events are times in nanoseconds as specified in `$timeformat`. Fractions of nanoseconds in delay times are not included because the time precision is a single nanosecond.
3. These events are in nanoseconds, as specified by the ``timescale` compiler directive that precedes module `ts_1ns_1ns`.
4. These events are times in nanoseconds, as specified in `$timeformat`.
5. These events are in microseconds, as specified by the ``timescale` compiler directive that precedes module `ts_1us_1ns`. The `%t` format specification is not used here; the `%d` and `%9.3f` format specifications are used to format these values.

Delay Mode Selection

This chapter describes the following:

- [Overview](#) on page 453
- [Delay Modes](#) on page 454
- [Reasons to Select a Delay Mode](#) on page 456
- [Setting a Delay Mode](#) on page 456
- [Precedence in Selection](#) on page 457
- [Timescales and Simulation Time Units](#) on page 458
- [Overriding Delay Values](#) on page 459
- [Delay Mode Example](#) on page 461
- [Decompiling with Delay Modes](#) on page 462
- [\\$showmodes](#) on page 462
- [acc_fetch_delay_mode Access Routine](#) on page 462
- [Macro Module Expansion and Delay Modes](#) on page 462
- [Summary of Delay Mode Rules](#) on page 463

Overview

Delay modes provide command-line options and compiler directives that allow you to alter the delay values specified in Verilog-XL models. You can ignore all delays specified in your model or replace all delays with a value of one simulation time unit. You can also replace delay values in selected portions of the model.

You can specify delay modes on a global basis or a module basis. If you assign a specific delay mode to a module, then all instances of that module simulate in that mode. Moreover,

the delay mode of each module is determined at compile time and cannot be altered dynamically.

You can use selectable delay modes to speed up simulation during debugging. This chapter explains how to do this as well as how delay selection facilitates using library modules that contain delay specifications for both Verifault-XL and Veritime. [“Delay Modes”](#) on page 454 describes the four selectable delay modes. [“Reasons to Select a Delay Mode”](#) on page 456 discusses practical applications for selectable delay modes. The remaining sections explain how to use selectable delay modes.

Important

The selected delay mode controls only structural delays (structural delays include delays assigned to gate and switch primitives, UDPs, and nets), path delays, timing checks, and delays on continuous assignments. Other delays simulate as specified regardless of delay mode.

See [“Summary of Delay Mode Rules”](#) on page 463 for a summary of the rules governing delay modes.

Delay Modes

The following sections describe the four delay modes that can be explicitly selected in Verilog-XL, and the default mode in effect if no delay mode is selected.

Unit Delay Mode

In unit delay mode, Verilog-XL ignores all module path delay information and timing checks and converts all non-zero structural and continuous assignment delay expressions to a unit delay of one simulation time unit (see [“Timescales and Simulation Time Units”](#) on page 458). There are two ways to circumvent the effect of the unit delay mode for specific delays:

- Using PLI access routines (See the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference* for more information on PLI access routines.)
- Using the parameter attribute mechanism (See [“Parameter Attribute Mechanism”](#) on page 460.)

Zero Delay Mode

Zero delay mode is similar to unit delay mode in that all module path delay information, timing checks, and structural and continuous assignment delays are ignored.

Verilog-XL Reference

Delay Mode Selection

There are two ways to override the zero delay mode for specific delays:

- Using PLI access routines (See the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference* for more information.)
- Using the parameter attribute mechanism (See [“Parameter Attribute Mechanism”](#) on page 460.)

Distributed Delay Mode

Distributed delays are delays on nets, primitives, or continuous assignments—in other words, delays other than those specified in procedural assignments and specify blocks. In distributed delay mode, Verilog-XL ignores all module path delay information and uses all distributed delays and timing checks. You can override specified delay values with PLI access routines (see the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference*). Verilog-XL ignores the parameter attribute mechanism in the distributed delay mode. See [“Parameter Attribute Mechanism”](#) on page 460 for information about the parameter attribute mechanism.

The distributed delay mode in Verilog-XL produces results equivalent to “good machine” results in Verifault-XL. See the *Verifault-XL User Guide* and the *Verifault-XL Reference* for more information about Verifault-XL.

Path Delay Mode

In this mode, Verilog-XL derives its timing information from specify blocks. If a module contains a specify block with one or more module path delays, all structural and continuous assignment delays within that module except `triereg` charge decay times are set to zero. In path delay mode, `triereg` charge decay remains active. The module simulates with “black box” timing—that is, with module path delays only.

You can specify distributed delays that cannot be overridden by the path delay mode by using the parameter attribute mechanism (see [“Parameter Attribute Mechanism”](#) on page 460) or with PLI and VPI access routines (see the *PLI 1.0 User Guide Reference* and the *VPI User Guide and Reference*). When a path delay mode simulation encounters a distributed delay that is locked in by either mechanism, module path delays and the distributed delay simulate concurrently.

When path delay mode is selected, modules that contain *no* module path delays simulate in distributed delay mode.

Default Delay Mode

If no delay mode is explicitly selected, the model simulates in the default mode—that is, delays simulate as specified in the model's source description files. Note that you can specify path delays and distributed delays in the same module and they will simulate together *only* when simulation is in the default delay mode.

Reasons to Select a Delay Mode

Replacing integer path or distributed delays with global zero or unit delays can reduce simulation time by an appreciable amount. You can use delay modes during design debugging phases when checking circuit logic is more important than timing correctness. You can also speed up simulation during debugging by selectively disabling delays in sections of the model where timing is not currently a concern. If these are major portions of a large design, the time saved may be significant.

The *distributed* and *path* delay modes allow you to develop or use modules that define both path and distributed delays and then to choose either the path delay or the distributed delays at compile time. This feature allows you to use the same source description with all the Veritools and then to select the appropriate delay mode when using the sources with Verilog-XL. You can set the delay mode for Verilog-XL by placing a compiler directive for the distributed or path mode in the module source description file, or by specifying a global delay mode at run time.

Setting a Delay Mode

The following are the two ways to set a delay mode:

- Use compiler directives in the source file to set delay modes specific to particular modules.
- Use command-line plus options at compile time to set a global delay mode for the simulation run.

Compiler Directives

Use compiler directives to select a delay mode for all instances of the same module. The compiler directive must precede the module definition. The compiler directives are as follows:

- ``delay_mode_path`
- ``delay_mode_distributed`

Verilog-XL Reference

Delay Mode Selection

- ``delay_mode_unit`
- ``delay_mode_zero`

When the compiler encounters a delay mode directive in a source file, it applies that delay mode to all modules defined from that point on, until it encounters a directive specifying a different delay mode or the end of compilation. Note that you can use the ``resetall` compiler directive at any point to return the source to the default delay mode (no mode selected). The recommended usage is to place ``resetall` at the beginning of each source text file, followed immediately by the directives desired in the file. You may choose to manually override all compiler directives by using the plus options described in [“Command-Line Plus Options”](#) on page 457.

Delay modes specified with a compiler directive remain active across file boundaries. Therefore, if you want to ensure that the modules in a particular file operate with the correct delay mode, place a compiler directive for the correct mode at the top of a source file. You can use command-line plus options to override these compiler directives.

Command-Line Plus Options

Four command-line plus options enable you to set a global delay mode. If you use more than one plus option, the compiler issues a warning and selects the mode with the highest precedence. The plus options are listed in the following table from highest to lowest precedence:

<code>+delay_mode_path</code>	The design simulates in path delay mode—except for modules with no module path delays.
<code>+delay_mode_distributed</code>	The design simulates in distributed delay mode.
<code>+delay_mode_unit</code>	The design simulates in unit delay mode.
<code>+delay_mode_zero</code>	Modules simulate in zero delay mode.

Precedence in Selection

The order of precedence in delay mode selection from highest to lowest is as follows:

1. plus option selection
2. compiler directives
3. default — no delay mode

You can override the delay values established by any of these cases. See [“Overriding Delay Values”](#) on page 459.

Timescales and Simulation Time Units

When working with delay modes, consider the way delay modes use timescales and simulation time units. When you select the unit delay mode at runtime, each explicit delay gets converted to a value of one, *measured in simulation time units*—that is, the value of the smallest `time_precision` argument specified by a ``timescale` compiler directive in any of your model’s description files. For example, you can specify an explicit delay for a gate as follows:

```
nand #5 g1 (qbar, q, clear);
```

The nand gate (shown above) can be controlled by the following timescale directive:

```
`timescale 1 us/1 ns
```

The directive causes the simulation delay value for the nand gate `g1` to be five microseconds. When a model uses timescales, that delay of five units is measured in timescale units—that is, its simulation value is five times the unit of time specified in a controlling timescale directive. (In the absence of any timescale directives the delay is a relative value. It is used to schedule events in the correct relative order.)

The following timescale directive gives the smallest precision argument specified for the model:

```
`timescale 10 ns/1 ps
```

The previous code sets the simulation time unit as one picosecond, so the five microsecond delay on nand gate `g1` is measured as 5,000,000 picoseconds. When you select the unit delay mode, your five microsecond delay on `g1` gets converted to one picosecond.

The following example shows how delay times change from the default mode when the unit delay mode is selected.

```
`timescale 1 ns/1 ps
module alpha (a, b, c) ;
  input b, c ;
  output a ;
  and #2 (a, b, c) ;
endmodule

`timescale 100 ns/1 ns
module beta (q, a, d, e) ;
  input a, d, e ;
  output q ;
  wire f ;
  xor #2 (f, d, e) ;
  alpha g1 (q, f, a) ;
```

Verilog-XL Reference

Delay Mode Selection

```
endmodule

`timescale 10 ps/1 fs
module gamma (x, y, z) ;
    input y,z;
    output x;
    reg w ;
    initial
        #200 w = 3 ;
    ...
endmodule
```

Delay mode selection controls the delays in the previous example with the following results:

- zero delay mode — no delays on gates; delay on assignment to register `w` is 2 ns, as specified, because delay modes do not affect behavioral delays
- unit delay mode — delays of one femtosecond on gates; delay on assignment to register `w` is 2 ns, as specified, because delay modes do not affect behavioral delays
- path delay mode — distributed delays used because no module paths are defined
- distributed delay mode — distributed delays used
- default delay mode — distributed delays used

Overriding Delay Values

You can use one of the following two methods to override the effect of a delay mode selection:

- PLI access routines
- parameter attribute mechanism

The following two sections discuss these methods.

PLI 1.0 or VPI Access Routines and Delays

You can use a PLI 1.0 or VPI access routine to override a structural delay set by a delay mode. This method can provide structural delay values in a module regardless of the method used to define the module's delay mode. The access routines that set delay values are the following:

- `acc_append_delays`
- `acc_replace_delays`

Verilog-XL Reference

Delay Mode Selection

Refer to the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference* for more information.

Note: In a PLI access routine, the delay value is measured in the timescale units of the module containing the gate.

Parameter Attribute Mechanism

Modules frequently need distributed delays on sequential elements to prevent race conditions. Sometimes such a module also needs path delays. Use parameter attributes to ensure that these essential delays are not overridden in path, unit, or zero delay modes.

The `DelayOverride$` specparam implements the parameter attribute mechanism. This specparam allows you to specify a delay—on a particular instance of a primitive or UDP—that takes effect during the zero, unit, or path delay modes. The delay provided by this mechanism replaces the distributed delay that the zero, unit, or path delay mode overrides. You must also provide a distributed delay to take effect during the distributed and default delay modes.

To use `DelayOverride$`, include it in the `specify` block section of the module that contains the instance to be controlled. The specparam uses the `DelayOverride$` prefix followed by the primitive or UDP instance name, with no space between. The syntax is as follows:

```
specparam DelayOverride$object_name = literal_constant_value;
```

The *object_name* is a primitive or UDP instance name. If you specify no object name, then Verilog-XL overrides all delays on gate primitives and UDPs in that module. The *literal_constant_value* is the number that provides the value for the delay. The number can be any one of the following:

- a decimal integer
- a based number (for example, 2'b10)
- a real number
- a *min:typ:max* expression composed of any one of the above three number formats

The following is a syntax example for a `DelayOverride$` specparam:

```
module
  ...
  nand #5 g1 (q, qbar, preset) ;
  ...
  specify
    ...
    specparam DelayOverride$g1= 5;
    ...
  endspecify
```

Verilog-XL Reference

Delay Mode Selection

```
...
endmodule
```

Note: For the parameter attribute mechanism the delay override value is measured in simulation time units—that is, the module’s timescale is ignored.

Delay Mode Example

The following example illustrates the behavior of some delay mode features. The module simulates using the distributed delays on the gates unless you set a global delay mode by specifying a command-line plus option.

```
`delay_mode_distributed          // compiler directive controls
                                // all instances of ffnand

module ffnand (q, qbar, preset, clear);
    output q, qbar;
    input preset, clear;

    nand #1 g1 (q, qbar, preset) ;           // set to 5 in unit, zero,
                                           // and path delay modes

    nand #0 g2 (qbar, q, clear) ;           // zero in all modules

    specify
        (preset => q) = 10;                 // path delay from preset to q --
                                           // used only in path delay mode
        specparam DelayOverride$g1= 5;     // delay for g1 --
                                           // used only in unit, path,
                                           // and zero delay modes
    endspecify
endmodule
`resetall                                // returns delay mode to default delay mode
```

The following table shows the simulation delays executed on the example when you select one of the global delay modes:

unit delay	Gate g1 is assigned a delay value of five simulation time units because the specparam <code>DelayOverride\$g1</code> overrides the unit delay mode; gate g2 keeps its zero delay because unit delay mode affects only non-zero delays.
zero delay	Gate g1 gets a delay of five simulation time units, as specified by the specparam <code>DelayOverride\$g1</code> .
distributed delay	A global distributed delay mode has the same effect on <i>this</i> module as no global delay mode because the compiler directive selects distributed mode. In either case, g1 has a delay of one timescale unit because the distributed delay is used (the specparam and module path specification are both ignored).

Verilog-XL Reference

Delay Mode Selection

path delay The simulation uses the module path delay information and ignores distributed delays. The `g1` delay is five simulation time units, as specified by the `DelayOverride$g1` specparam.

You cannot simulate this module in the default mode because a delay mode compiler directive precedes it.

Decompiling with Delay Modes

When decompiling a Verilog-XL source using `$list` or the `-d` compile time option, the delay values displayed are the ones being simulated—not the ones in the original description. If delays have been added using PLI access routines, these are not displayed in the decompilation.

`$showmodes`

Use the `$showmodes` system task to display delay modes in effect for particular modules during simulation. When invoked with a non-zero constant argument, it displays the delay modes of the current scope as well as delay modes of all module instances beneath it in the hierarchy. If a zero argument or no argument is supplied to `$showmodes`, this system task displays only the delay mode of the current scope.

```
$showmodes ;
```

```
$showmodes (<non_zero_constant>);
```

`acc_fetch_delay_mode` Access Routine

An application can use the access routine `acc_fetch_delay_mode` to retrieve delay mode information from Verilog-XL.

Macro Module Expansion and Delay Modes

When a delay mode is in effect, all macro module instances within the scope of that delay mode are expanded before the delay mode information is processed. This rule means that a macro module instance inherits the delay mode of the module in which it is expanded.

Summary of Delay Mode Rules

The following table summarizes the rules governing the behavior of a module for which a particular delay mode is in effect.

	Unit	Zero	Distributed	Path**	Default
module path delays	ignored	ignored	ignored	used	used
timing checks	ignored	ignored	used	used	used
delays specified by access routine	PLI access routines work in all delay modes				
override by DelayOverride\$	used	used	ignored	used	ignored
treatment of distributed delays	set to 1*	set to 0	used as defined	ignored	used as defined

* non-zero values are set to one simulation time unit

** path mode is ignored in modules containing no path information

Verilog-XL Reference

Delay Mode Selection

The Behavior Profiler

This chapter describes the following:

- [How the Behavior Profiler Works](#) on page 465
- [Behavior Profiler System Tasks](#) on page 467
- [Behavior Profiler Data Report](#) on page 471
- [Recommended Modeling Practices](#) on page 483
- [How Verilog-XL Affects Profiler Results](#) on page 483
- [Behavior Profiler Example](#) on page 484

How the Behavior Profiler Works

The behavior profiler identifies the modules and statements in your Verilog HDL that use the most CPU time during simulation. When Verilog-XL executes the `$startprofile` system task, the behavior profiler begins to take *samples* of your source description. A *sample* is a “snapshot” of your design.

By default, the behavior profiler takes a sample every 100 microseconds of CPU time, recording the number of samples it takes of each line. When the behavior profiler takes a sample, it performs the following functions:

1. It interrupts the current process.
2. It scans the Verilog-XL data structure to determine which source file line the CPU is executing.
3. It continues the current process.

There is a relationship between the number of samples the behavior profiler records for a line and the amount of CPU time used by that line. The more samples of a line in your source description, the more CPU time is used by that line. The line that has the most samples thus uses more CPU time than any other line.

Verilog-XL Reference

The Behavior Profiler

If the behavior profiler takes more samples of one line in your source description than it finds of any other line, then that one line uses more CPU time than any other line.

The following example shows an elementary use of the behavior profiler. In this example, two registers are declared in different modules. The registers are toggled at different simulation time intervals. The output of the following example has four different formats shown in the section [“Behavior Profiler Data Report”](#) on page 471.

Behavior Profiler Sample Code

```
2  module test2;
3      rega
3      inst1();
4      regb
4      inst2();
5      initial
6          begin
7*         $list;
8             $list(inst1);
9             $list(inst2);
10            #10
10            $startprofile; // starts the behavior profiler
11            #10000
11            $finish;
12        end
13    endmodule
15    module rega;
16        reg
16        a; // = 1'h1, 1
17        initial
18        begin
19            a = 1;
20            forever
21            begin
22*           #1 // specifies that reg a toggles
23                a = ~a; // after every time unit
24            end
25        end
26    endmodule
28    module regb;
29        reg
29        b; // = 1'h1, 1
30        initial
31        begin
32            b = 1;
33            forever
34            begin
35*           #10 // specifies that reg b toggles
36                b = ~b; // after every 10 time units
37            end
38        end
39    endmodule
```

In the previous example, `reg a` changes value every time unit, and `reg b` changes value every 10 time units. The `$startprofile` system task invokes the behavior profiler after Verilog-XL has simulated for 10 time units.

The behavior profiler results show that lines 22 and 23 have almost 10 times more samples than lines 35 and 36. The lines that toggle `reg a` are executed most frequently and use most of the CPU time during simulation.

Behavior Profiler System Tasks

There are four system tasks for the behavior profiler. These system tasks are as follows:

- `$startprofile` on page 467
- `$reportprofile` on page 468
- `$stopprofile` on page 469
- `$listcounts` on page 469

The `$startprofile` system task tells Verilog-XL to begin or to resume collecting behavior profiler data, `$reportprofile` tells Verilog-XL to report this data before the end of the simulation, and `$stopprofile` tells Verilog-XL to stop collecting this data. Enter `$reportprofile` or `$stopprofile` after `$startprofile`.

The `$listcounts` system task produces a source listing with both the line numbers and the execution count for each line. You can enter `$listcounts` before or after `$startprofile`. The `$listcounts` task is disabled unless you include the `+listcounts` option on the command line.

This section describes the behavior profiler system tasks in detail and tells you when to use them.

`$startprofile`

Use the `$startprofile` system task to invoke the behavior profiler. It tells the behavior profiler to begin, or to continue to take samples of the simulation. The syntax is as follows:

```
$startprofile;
```

Note: When Verilog-XL is invoked in Turbo mode, the profiler is disabled as default. You must specify the `+profile` command-line option if you want to start the profiler.

By default, the behavior profiler takes a sample every 100 microseconds.

The following example shows a source description that includes the `$startprofile` system task. Verilog-XL loads three memories before it invokes the behavior profiler.

```
...  
initial
```

Verilog-XL Reference

The Behavior Profiler

```
begin
    $readmemb("mem1.dat", mem1);
    $readmemb("mem2.dat", mem2);
    $readmemb("mem3.dat", mem3);
    $startprofile;
end
...
```

Note: Instead of placing `$startprofile` in your design, you can place it in a separate file with the following module definition and include the filename in your Verilog-XL command line.

```
module profile;
initial
    $startprofile;
endmodule
```

\$reportprofile

The behavior profiler always displays its data at the end of simulation unless you use the `$reportprofile` system task to produce the following data reports *before* the end of a simulation.

- Profile ranking by statement
- Profile ranking by module instance
- Profile ranking by statement class
- Profile ranking by statement type

The syntax is as follows:

```
$reportprofile (<max_lines>?);
```

The `<max_lines>` variable is an optional integer argument that specifies the maximum number of lines of data that the behavior profiler reports. The default maximum number of lines is 100.

The following example shows a source description that includes the `$reportprofile` system task:

```
...
$startprofile;
...
    #1000000
    $reportprofile(200);
    $finish;
end
endmodule
```

Verilog-XL Reference

The Behavior Profiler

Default behavior profiler data reports never have more than 100 lines but you can produce reports with more than 100 lines with the `$reportprofile` system task. In the previous example, the specified maximum number of lines is 200.

\$stopprofile

Use the `$stopprofile` system task to stop taking samples before the end of the simulation. This system task takes no arguments. The syntax is as follows:

```
$stopprofile;
```

The following example shows a source description that includes the `$stopprofile` system task.

```
...
initial
  #10000
  begin
    $stopprofile;
    $readmemb("mem.dat",mem);
    $startprofile;
  end
initial
  ...
```

In this example, at simulation time 10000, Verilog-XL performs the following functions:

1. Stops the behavior profiler from taking samples.
2. Loads the memory.
3. Enables the behavior profiler to resume taking samples.

Verilog-XL interrupts sampling in this example, so the behavior profiler takes no sample of the `$readmemb` system task.

\$listcounts

The `$listcounts` system task is an enhancement of `$list`; it produces a line-numbered source listing that includes an execution count—that is, the number of times Verilog-XL executes the statements in the line. The syntax is as follows:

```
$listcounts (<hierarchical_name>?);
```

The `$listcounts` system task takes an optional hierarchical name argument. If you do not include an argument, `$listcounts` produces a listing of the source description at the scope level from which you called the task.

Verilog-XL Reference

The Behavior Profiler

The following example shows a source listing produced by the `$listcounts` system task for the “[Behavior Profiler Sample Code](#)” on page 466.

```
0: 1  module test2;
0: 2      rega
0: 3          inst1();
0: 4      regb
0: 5          inst2();
1: 6      initial
1: 7          begin
1: 8          #10
1: 9              $startprofile;
1: 10             #10000
1: 11*             $listcounts;
0: 12             $listcounts(test2.inst1);
0: 13             $listcounts(test2.inst2);
0: 14             $finish;
1: 15          end
0: 16  endmodule
// tst121
0: 17  module rega;
0: 18      reg
0: 19          a; // = 1'h0, 0
1: 20      initial
1: 21          begin
1: 22              a = 1;
1: 23              forever
10010: 24                  begin
10010: 25*                      #1
10009: 26                          a = ~a;
10010: 27                      end
1: 28                  end
0: 29  endmodule
// tst121
0: 30  module regb;
0: 31      reg
0: 32          b; // = 1'h1, 1
1: 33      initial
1: 34          begin
1: 35              b = 1;
1: 36              forever
1001: 37                  begin
1001: 38*                      #10
1000: 39                          b = ~b;
1001: 40                  end
1: 41              end
0: 42  endmodule
```

In this example, a `$listcounts` system task on line 11 for the top-level module and for each instance of its submodules produces a listing of the entire source description. You interpret the listings as follows:

- The first column lists the number of times that Verilog-XL has executed each statement.
- The second column lists the line numbers of each statement in the source description. As with `$list`, an asterisk (*) in this column indicates that the line contains a simulation event that Verilog-XL schedules for the current simulation time.

- The third column lists the source description.

The `$listcounts` system task provides information that helps you to analyze the behavior profiler data in the profile ranking by statement report. The output from the `$listcounts` system task shows you the line number and execution count of the statements in your source description. The behavior profiler uses these line numbers to produce the profile ranking by statement report. The execution count thus helps you to determine the cause of a high percentage and number of samples for a line in the profile ranking by statement report.

There are two causes for a high percentage and number of samples for a line in your source description. The execution count indicates which cause explains the high percentage and number, as shown in the following table:

Execution Count	Cause of a High Percentage and Number of Samples
low	Verilog-XL uses more CPU time to execute the statement on that line than it uses to execute other lines.
high	Verilog-XL executes the statement on that line more often than it executes other lines.

If a line has a high percentage and number of samples, but a low execution count, you should consider rewriting that line with a different construct.

Behavior Profiler Data Report

Verilog-XL creates the following behavior profiler data reports:

- [“Profile Ranking by Statement”](#) on page 471
- [“Profile Ranking by Module Instance”](#) on page 474
- [“Profile Ranking by Statement Class”](#) on page 475
- [“Profile Ranking by Statement Type”](#) on page 476

Profile Ranking by Statement

The profile ranking by statement report ranks the lines in your source description according to how many samples they represent.

Verilog-XL Reference

The Behavior Profiler

The following report shows the contents of the profile ranking by statement report produced by the behavior profiler during a simulation of the source listing in “[Behavior Profiler Sample Code](#)” on page 466:

Profile ranking by statement:

Self%	Cum. %	Samples	Statement
-----	-----	-----	-----
51.7%	51.7%	179	test2.v, L23, test2.inst1
26.6%	78.3%	92	test2.v, L22, test2.inst1
7.2%	85.5%	25	test2.v, L21, test2.inst1
4.6%	90.2%	16	test2.v, L36, test2.inst2
4.3%	94.5%	15	test2.v, L22, test2.inst1
3.8%	98.3%	13	test2.v, L35, test2.inst2
1.2%	99.4%	4	test2.v, L11, test2
0.3%	99.7%	1	XL
0.3%	100.0%	1	test2.v, L34, test2.inst2

To read this report, you must first look in the `Statement` column to find the desired line number, and then look in the other columns for the behavior profiler data about that line.

Note: The “XL” in the report stands for all the events in the design accelerated by the XL algorithm. Verilog-XL reports a single percentage and number of samples for all accelerated events, no matter how many lines are used to specify them in the source description.

The following table describes the information displayed in each column:

Column	Description
Statement	Lists from left-to-right the following information about a line in the source description: <ul style="list-style-type: none">■ the name of the file that contains the line■ the line number■ the hierarchical name of the module instance that contains the line
Self %	Indicates the percent of the total number of samples that represent the line listed in the <code>Statement</code> column.
Cum. %	Indicates the percent of samples that represent the line listed in the <code>Statement</code> column plus the preceding lines. Look in this column to see if a small number of source description lines use most of the CPU time. In the previous profile ranking by statement report , 98.3% of the samples represent lines 23, 22, 21, 36 and 35.
Samples	Indicates how many samples of the line were found.

Verilog-XL Reference

The Behavior Profiler

Note: The profile ranking by statement report can contain more than one entry for the same line in the source description. For example, in the report in [“Profile Ranking by Statement”](#) on page 471, a procedural delay (`test2.v, L22, test2.inst1`) has two entries, because procedural delays have two locations in the Verilog-XL data structure. One entry occurs when Verilog-XL schedules an event; the other entry occurs at the event's execution.

Interactive statements

The profile ranking by statement report can include samples of interactive statements. The behavior profiler lists samples of interactive statements by inserting a hyphen (-) between the uppercase letter L and the line number in the statement column. Verilog-XL counts interactive statements, and the behavior profiler uses this count to assign line numbers to the statements in the profile ranking by statement report. The statement column entry for interactive statements does not include a hierarchical name for a module instance. The following example shows the log file of a simulation that invokes the behavior profiler and includes interactive statements:

```
Compiling source file "beh_int.v"
Highest level modules:
beh_int

 1  module beh_int;
 2      reg
 2      a; // = 1'hx, x
 3      initial
 4          begin
 5*         $list;
 6             $startprofile;
 7             $stop;
 8         end
 9  endmodule

L7 "beh_int.v": $stop at simulation time 0
Type ? for help

C1 > $showscopes; // Line number L1 in the statement column
Directory of scopes at current scope level:
Current scope is (beh_int)
Highest level modules:
beh_int

C2 > $showvars; // Line number L2 in the statement column
Variables in the current scope:
a (beh_int) reg = 1'hx, x

C3 > $finish; // Line number L3 in the statement column
C3: $finish at simulation time 0
8 simulation events

CPU time: 0 secs to compile + 0 secs to link + 0 secs in simulation
Report limit: 100

Profile ranking by statement:

Self%  Cum.%  Samples  Statement
-----  -----  -
25.0%  25.0%    1        beh_int.v, L7, beh_int
```

Verilog-XL Reference

The Behavior Profiler

```
25.0%  50.0%      1      beh_int.v, L-1,
25.0%  75.0%      1      beh_int.v, L-2,
25.0% 100.0%      1      beh_int.v, L-3,
```

Profile Ranking by Module Instance

The profile ranking by module instance report ranks the module instances in your source description according to the number of samples of their contents. The following example shows the contents of the profile ranking by module instance report produced by the behavior profiler during a simulation of the source listing in [“Behavior Profiler Sample Code”](#) on page 466:

Profile ranking by module instance:

```
Self%   Cum.%   Samples   (Self + submodules)   Instance
-----  -----  -
89.9%   89.9%   311      ( 89.9%  311)          test2.inst1
 8.7%   98.6%   30       (  8.7%   30)          test2.inst2
 1.2%   99.7%   4        ( 99.7%  345)          test2
```

To read this report, look in the `Instance` column to find the desired module instance name, and then look in the other columns for the behavior profiler data about that module instance.

The following table describes the information displayed in each column:

Column	Description
Instance	Lists the hierarchical name of the module instance
Self %	Indicates the percentage of the total number of samples that contain the module instance
Cum. %	Indicates the percentage of the total number of samples that contain the module instance and the preceding module instances. Look in this column to see if a small number of module instances use most of the CPU time.
Samples	Indicates how many samples contains the module instance
(Self + submodules)	Indicates the percentage of the total number of samples, as well as the number of samples that contain the module instance and all module instances below it in the hierarchy

Profile Ranking by Statement Class

The profile ranking by statement class report organizes into six categories and reports the amount of CPU time spent in each category. The following example shows the contents of the profile ranking by statement class produced by the behavior profiler during a simulation of the source listing in [“Behavior Profiler Sample Code”](#) on page 466.

Profile by statement class:

Self%	Cum. %	Samples	Statement class
41.9%	41.9%	1757	Procedural RTL
28.6%	70.5%	1197	Continuous Assignments
18.7%	89.1%	782	Other
10.9%	100.0%	455	Gates

To read this report, first look in the `Statement class` column to find the desired statement class type, and then look in the other columns for the behavior profiler data about that type.

The following table describes the information displayed in each column:

Column	Description
Statement class	Lists the type of statement using the CPU.
Self %	Indicates the percent of the total number of samples that contain the corresponding statement class.
Cum. %	Indicates the percentage of the total number of samples that contain the corresponding statement class and the preceding lines. Look in this column to see which statement classes use most of the CPU time.
Samples	Indicates how many samples of the statement class were found.

The statement classes are defined as follows:

Class	Description
Continuous Assignments	Drive values onto nets, both vector and scalar. Assignments occur whenever the simulation causes the value of the right-hand side to change.

Verilog-XL Reference

The Behavior Profiler

Class	Description
Non-Blocking Assignments	Allow you to make several register assignments within the same time step without regard to order or dependence upon each other. The simulator evaluates the right-hand side immediately, but schedules the assignment of the new value to take place at a time specified by the Timing Control. In contrast, Blocking Procedural Assignments wait for the time specified by the Timing Control, then evaluate the right-hand side, and make the assignment.
PLI	The Programming Language Interface (PLI) allows you to interact with the simulation environment. You can pass information to and from the internal data structures of Verilog-XL. The PLI mechanism works with utility routines to interact dynamically with the Verilog simulation process and data structures. You write user-supplied routines in the C high-level programming language.
Procedural RTL	Occur within a structured procedural block and are specified within one of the following statements: <code>initial</code> , <code>always</code> , <code>task</code> , or <code>function</code> .
Others	Any statement type that is not grouped into one of the other statement classes. (Typically, there is very little time reported under this class.)
Gates	One of the following: <code>and nand or nor xor xnor</code> <code>buf bufif0 bufif1</code> <code>not notif0 notif1</code> <code>pulldown pullup</code> <code>nmos rnmos pmos rpms cmos rcmos</code> <code>tran rtran tranif0 rtranif0 tranif1 rtranif1</code>

Profile Ranking by Statement Type

The profile ranking by statement type reports the amount of CPU time spent in each type of Verilog HDL construct. The following example shows the contents of the profile ranking by statement type produced by the behavior profiler during a simulation of the source listing in “[Behavior Profiler Sample Code](#)” on page 466.

Profile by statement type:

Self%	Cum.%	Samples	Statement type
41.5%	41.5%	1877	<code>assign_delay_stat</code>
26.9%	68.4%	1217	<code>cont_assign</code>
18.6%	87.0%	844	<code>event_stat</code>
11.8%	98.8%	534	<code>if_else_stat</code>
1.1%	99.9%	51	OTHERS (XL)
0.0%	100.0%	1	<code>modport</code>

Verilog-XL Reference

The Behavior Profiler

```

0.0% 100.0%      1      delay_stat
0.0% 100.0%      1      assign_stat

```

To read this report, first look in the `Statement type` column to find the desired statement type, and then look in the other columns for the behavior profiler data about that type. The following table describes the information displayed in each column:

Column	Description
Statement type	Lists the specific type of statement using the CPU.
Self %	Indicates the percentage of the total number of samples that contain the corresponding statement type.
Cum. %	Indicates the percentage of the total number of samples that contain the corresponding statement type and the preceding lines. Look in this column to see which statement types use most of the CPU time.
Samples	Indicates how many samples of the statement type were found.

The following list shows the alphabetically ordered types of statements that the behavior profiler examines and the statement class to which each type belongs. The third column takes you to more information about the statement type.

Statement Type	Statement Class	For More Information, see...
<code>alone_stat</code>	Procedural RTL	“Structured Procedures” on page 164
<code>and_gate</code>	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
<code>assign_delay_stat</code>	Procedural RTL	“Intra-Assignment Timing Controls” on page 186
<code>assign_event_stat</code>	Procedural RTL	“Intra-Assignment Timing Controls” on page 186
<code>assign_multi_stat</code>	Procedural RTL	“Intra-Assignment Timing Controls” on page 186
<code>assign_stat</code>	Procedural RTL	“Blocking Procedural Assignments” on page 167
<code>bit_select</code>	Procedural RTL	“Net and Register Bit Addressing” on page 63

Verilog-XL Reference

The Behavior Profiler

Statement Type	Statement Class	For More Information, see...
block_par	Procedural RTL	“Parallel Blocks” on page 191
buf_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
bufif0_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
bufif1_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
case_stat	Procedural RTL	“case Statements” on page 176
casex_stat	Procedural RTL	“Using case Statements with Inconsequential Conditions” on page 178
casez_stat	Procedural RTL	“Using case Statements with Inconsequential Conditions” on page 178
cmos_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
comb_prim	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
concatenate	Procedural RTL	“Concatenations” on page 62
cont_assign	Continuous Assignments	“The Continuous Assignment Statement” on page 73
cont_assign_decl	Continuous Assignments	“The Net Declaration Assignment” on page 73
contassign_stat	Procedural RTL	“The assign and deassign Procedural Statements” on page 94
deassign_stat	Procedural RTL	“The assign and deassign Procedural Statements” on page 94

Verilog-XL Reference

The Behavior Profiler

Statement Type	Statement Class	For More Information, see...
delay_stat	Procedural RTL	“Delay Control” on page 183
disable_stat	Procedural RTL	Chapter 10, “Disabling of Named Blocks and Tasks.”
enable_task_stat	Procedural RTL, PLI	Chapter 9, “Tasks and Functions.” Also see the <i>PLI 1.0 User Guide and Reference</i>
enable_taskfunc	Procedural RTL, PLI	Chapter 9, “Tasks and Functions.” Also see the <i>PLI 1.0 User Guide and Reference</i>
event_stat	Procedural RTL	“Event Control” on page 184
for_stat	Procedural RTL	“for Loop” on page 181
force_stat	Procedural RTL	“The force and release Procedural Statements” on page 95
forever_stat	Procedural RTL	“forever Loop” on page 180
full_bit_select	Procedural RTL	“Net and Register Bit Addressing” on page 63
full_part_select	Procedural RTL	“Net and Register Bit Addressing” on page 63
full_ref	Procedural RTL	“Registers” on page 32
gen_event_stat	Procedural RTL	“Named Events” on page 184
gt_innode	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
gt_inout	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
gt_inoutnode	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
gt_input	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>

Verilog-XL Reference

The Behavior Profiler

Statement Type	Statement Class	For More Information, see...
gt_outnode	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
gt_output	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
if_else_stat	Procedural RTL	“if-else-if Statements” on page 175
if_stat	Procedural RTL	“Conditional Statements” on page 174
modport	Continuous Assignments	“Port Connection Rules” on page 224
nand_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
nmos_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
nor_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
not_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
notif0_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
notif1_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
null_stat	Procedural RTL	“Delay Control” on page 183
or_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>

Verilog-XL Reference

The Behavior Profiler

Statement Type	Statement Class	For More Information, see...
OTHERS (XL)	Gates	Items Supported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
par_block	Procedural RTL	“Parallel Blocks” on page 191
part_select	Procedural RTL	“Net and Register Bit Addressing” on page 63
pmos_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
pulldown_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
pullup_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
rcmos_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
refnode	Procedural RTL	“Registers” on page 32
release_stat	Procedural RTL	“The force and release Procedural Statements” on page 95
repeat_stat	Procedural RTL	“repeat Loop” on page 180
rnmos_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
rpmos_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
rtl_delay_stat	Non-Blocking Assignments	“Non-Blocking Procedural Assignments” on page 167
rtl_event_stat	Non-Blocking Assignments	“Non-Blocking Procedural Assignments” on page 167
rtl_multi_stat	Non-Blocking Assignments	“Non-Blocking Procedural Assignments” on page 167

Verilog-XL Reference

The Behavior Profiler

Statement Type	Statement Class	For More Information, see...
rtran_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
rtranif0_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
rtranif1_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
seq_block	Procedural RTL	“Sequential Blocks” on page 189
seq_prim	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
tran_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
tranif0_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
tranif1_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
vcl_trigger	PLI	See the <i>PLI 1.0 User Guide and Reference</i>
wait_stat	Procedural RTL	“Level-Sensitive Event Control” on page 186
while_stat	Procedural RTL	“while Loop” on page 181
xnor_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>
xor_gate	Gates	Items Unsupported by the Default XL Algorithm in the <i>Verilog-XL User Guide</i>

Recommended Modeling Practices

This section describes the modeling practices that you should use when you plan to invoke the behavior profiler to get the most useful information from it.

Invoke the Behavior Profiler After You Initialize Your Design

Enter `$startprofile` after those statements that Verilog-XL executes only when the simulation begins, so that the behavior profiler samples only those statements that Verilog-XL executes throughout the simulation.

Put Statements on Separate Lines

Enter your behavior statements on separate lines, so that you can see the percentage and the number of samples for each statement individually. If a line contains more than one statement, there is no way to determine which statement is represented by the sample.

How Verilog-XL Affects Profiler Results

Verilog-XL can alter behavior profiler data in ways described in this section.

Using a Variable to Drive Multiple Module Instances

If you use the behavior profiler to compare the performance of two module instances, be sure that different nets or registers drive them. When a net or register drives more than one module instance, the behavior profiler can attribute to one module instance the CPU time Verilog-XL uses to pass a value to the other module instance. This problem results in too high a number and percentage of samples for one module instance and that module's header, as well as too low a number and percentage of samples of the other module instance and its header. This anomaly is caused by port collapsing. There are solutions to this problem:

- Insert buffers between the drivers and the module instances.
- Declare different drivers for each module instance.

Expanded Vector Nets

A statement that contains an expanded vector net has multiple entries in the profile ranking by statement report. The more bits in the net, the more entries in the report. These multiple

Verilog-XL Reference

The Behavior Profiler

entries can collectively make up a large percentage of the samples, although none of these entries appear near the top of the report. If your source description contains an expanded vector net, check for multiple entries for the line that contains this net.

Accelerated Events

Accelerated events are events that require Verilog-XL to evaluate an accelerate primitive. The behavior profiler makes one entry in the profile ranking by statement report for all lines that contain accelerated events. That entry is labeled `XL` in the `Statement` column.

Behavior Profiler Example

The following example uses the behavior profiler to see how much CPU time Verilog-XL uses to execute two behavioral modules for a D flip-flop with a clear line. This example proves that one module is more efficient than the other. The modules are appropriately named `inefficient` and `efficient` and appear in the following examples:

```
module inefficient(clk1,d1,clr1,q1,qb1);
input  clk1,d1,clr1;
output q1,qb1;
reg q1,qb1;

always
  @(posedge clk1)          // A rising edge of clk1 triggers
  begin                    // the following proc. assignments:
    q1=#5 d1;              // 1. value of d1 to q1
    qb1=#1 ~q1;            // 2. unary negation of value of
  end                       // q1 to qb1

always
  wait (clr1 === 1'b0)     // if clr1 goes low (active), the
                          // procedural continuous assignments
                          // of 0 to q1 and 1 to qb1 override
                          // the procedural assignments

  begin
    #5 assign q1 = 0;
    #1 assign qb1 = 1;
    wait (clr1 === 1'b1)   // When clr1 goes high after it
                          // goes low, Verilog-XL deassigns the
                          // procedural continuous assignments
                          // so that d1 can once again drive q1,
                          // and q1 can once again drive qb1.

    begin
      deassign q1;
      deassign qb1;
    end
  end
endmodule

module efficient(clk2,d2,clr2,q2,qb2);
input  clk2,d2,clr2;
output q2,qb2;
reg q2,qb2;
```

Verilog-XL Reference

The Behavior Profiler

```
always
  wait (clr2 === 1'b1)           // when clr2 goes high (inactive),
                                // Verilog-XL enables a begin-end block
                                // named clock_trigger.

  begin:clock_trigger           // In clock_trigger,
    forever @(posedge clk2)     // a rising edge of clk2 triggers
      begin                     // a procedural assignment of
        q2=#5 d2;              // the value of d2 to q2 and
        qb2=#1 ~q2;           // the unary negation of q2 to qb2
      end
  end

always
  wait (clr2 ===1'b0)           // When clr2 goes low, Verilog-XL
                                // executes another begin-end block.
                                // In this begin-end block, Verilog-XL
                                // does the following:

  begin
    disable clock_trigger;     // disables the block clock_trigger
    q2=#5 0;                   // procedurally assigns 0 to q2
    qb2=#1 1;                  // procedurally assigns 1 to qb2
    wait (clr2===1'b1);        // waits until clr2 goes high before
                                // exiting the begin-end block

  end
endmodule
```

Unlike in module `inefficient`, procedural assignments are never evaluated in module `efficient` and then overridden by procedural continuous assignments.

The following example shows the contents of the top-level module that drives a concurrent simulation of the previous modules:

```
module behavior (q1,qb1,q2,qb2);
output q1,qb1,q2,qb2;
reg clk1, clk2, d1, d2, clr1, clr2;           // a separate data, clock, and
                                              // clear line for each module

inefficient ineff (clk1,d1,clr1,q1,qb1);
efficient eff (clk2,d2,clr2,q2,qb2);

initial
  fork
    begin
      clk1=0;
      clk2=0;
      forever
        begin
          #30
          clk1=~clk1;
          clk2=~clk2;
        end
    end
  begin
    d1=0;
    d2=0;
    #15
    forever
      begin
        #120
```

Verilog-XL Reference

The Behavior Profiler

```
        d1=~d1;
        d2=~d2;
    end
end
begin
    clr1=0;
    clr2=0;
    #15
    forever
    begin
        #240
        clr1=~clr1;
        clr2=~clr2;
    end
end
begin
#100 $startprofile;          // Behavior profiler starts after 100 time units
    repeat(10000)
    @(posedge clk1);
    $listcounts;             // The $listcounts system tasks produce
                            // a line-numbered source listing with
                            // an execution count for the stimulus
                            // module and for the two D flip-flop
                            // behavioral modules.
    $listcounts(behavior.ineff);
    $listcounts(behavior.eff);
    $finish;
end
join
endmodule
```

The following example shows the `$listcounts` source listing for the stimulus module. Note the large execution counts for some lines in this module.

```
0:    1  module behavior(q1, qb1, q2, qb2);
0:    2      output
0:    2      q1, // = St0
0:    2      qb1, // = St1
0:    2      q2, // = St0
0:    2      qb2; // = St1
0:    3      reg
0:    3      clk1, // = 1'h1, 1
0:    3      clk2, // = 1'h1, 1
0:    3      d1, // = 1'h0, 0
0:    3      d2, // = 1'h0, 0
0:    3      clr1, // = 1'h0, 0
0:    3      clr2; // = 1'h0, 0
0:    5      inefficient
0:    5      ineff(clk1, d1, clr1, q1, qb1);
0:    6      efficient
0:    6      eff(clk2, d2, clr2, q2, qb2);
1:    8      initial
1:    9      fork
1:   10          begin
1:   11              clk1 = 0;
1:   11              clk2 = 0;
1:   12              forever
20004: 12                  begin
20004: 13*                      #30
20003: 14                          clk1 = ~clk1;
```

Verilog-XL Reference The Behavior Profiler

```

20003: 15             clk2 = ~clk2;
20004: 16             end
1: 17             end
1: 19             begin
1: 20             d1 = 0;
1: 20             d2 = 0;
1: 21             #15
1: 22             forever
5001: 22             begin
5001: 23*            #120
5000: 24             d1 = ~d1;
5000: 25             d2 = ~d2;
5001: 26             end
1: 27             end
1: 29             begin
1: 30             clr1 = 0;
1: 30             clr2 = 0;
1: 31             #15
1: 32             forever
2501: 32             begin
2501: 33*            #240
2500: 34             clr1 = ~clr1;
2500: 35             clr2 = ~clr2;
2501: 36             end
1: 37             end
1: 39             begin
1: 40             #100
1: 40             $startprofile;
1: 41             repeat(10000)
10000: 42             @(posedge clk1)
10000: 42 ;
1: 43*            $listcounts;
0: 44             $listcounts(ineff);
0: 45             $listcounts(behavior.eff);
0: 46             $finish;
1: 47             end
1: 48             join
0: 50             endmodule

```

The following example shows the `$listcounts` source listing for module `inefficient`. Note that the procedural assignments in lines 60 and 61 are executed 10,001 times. Half of these assignments are overridden by the procedural continuous assignments in lines 67 and 69.

```

0: 52     module inefficient(clk1, d1, clr1, q1, qbl);
0: 53         input
0: 53             clk1, // = St0
0: 53             d1, // = St0
0: 53             clr1; // = St0
0: 54         output
0: 54             q1, // = 1'h0, 0
0: 54             qbl; // = 1'h1, 1
0: 55         reg
0: 55             q1, // = 1'h0, 0
0: 55             qbl; // = 1'h1, 1
1: 57         always
10002: 58*            @(posedge clk1)
10001: 59             begin
10001: 60                 q1 = #(5) d1;

```

Verilog-XL Reference The Behavior Profiler

```

10001: 61             qb1 = #(1) ~q1;
10001: 62             end
1:      64         always
1251:  65             wait(clr1 === 1'b0)
1251:  66                 begin
1251:  67                     #5
1252:  67                         assign q1 = 0;
1251:  68                         #1
1252:  68                             assign qb1 = 1;
1251:  69*                             wait(clr1 === 1'b1)
1250:  70                                 begin
1250:  71                                     deassign q1;
1250:  72                                     deassign qb1;
1250:  73                                 end
1251:  74                 end
0:      75     endmodule

```

The following example shows the \$listcounts source listing for module efficient. In this example, the procedural assignments in line 88 and 89 are executed only 5,000 times.

```

0:      77     module efficient(clk2, d2, clr2, q2, qb2);
0:      78         input
0:      78             clk2, // = St0
0:      78             d2, // = St0
0:      78             clr2; // = St0
0:      79         output
0:      79             q2, // = 1'h0, 0
0:      79             qb2; // = 1'h1, 1
0:      80         reg
0:      80             q2, // = 1'h0, 0
0:      80             qb2; // = 1'h1, 1
1:      82         always
1251:  83*             wait(clr2 === 1'b1)
1250:  84                 begin :clock_trigger
1250:  85                     forever
6250:  86                         @(posedge clk2)
5000:  87                             begin
5000:  88                                 q2 = #(5) d2;
5000:  89                                 qb2 = #(1) ~q2;
5000:  90                             end
1250:  91                 end
1:      93         always
1251:  94             wait(clr2 === 1'b0)
1251:  95                 begin
1251:  96                     disable clock_trigger;
1251:  97                     q2 = #(5) 0;
1251:  98                     qb2 = #(1) 1;
1251:  99*                     wait(clr2 === 1'b1)
1250:  99 ;
1251: 100                 end
0:      101     endmodule

```

The following example shows the profile ranking by statement report for modules inefficient and efficient:

Profile ranking by statement:

Self%	Cum.%	Samples	Statement
11.2%	11.2%	625	behavior6.v, L14, behavior

Verilog-XL Reference The Behavior Profiler

7.4%	18.5%	411	behavior6.v, L15, behavior
7.3%	25.9%	410	behavior6.v, L53, behavior.ineff
6.8%	32.6%	378	behavior6.v, L52, behavior.ineff
6.5%	39.2%	366	behavior6.v, L77, behavior.eff
6.2%	45.3%	344	behavior6.v, L78, behavior.eff
3.7%	49.0%	205	behavior6.v, L13, behavior
3.6%	52.6%	202	behavior6.v, L60, behavior.ineff
3.3%	55.9%	184	behavior6.v, L58, behavior.ineff
3.2%	59.1%	179	behavior6.v, L42, behavior
3.2%	62.3%	179	behavior6.v, L61, behavior.ineff
2.4%	64.7%	135	behavior6.v, L89, behavior.eff.clock_trigger
2.1%	66.9%	120	behavior6.v, L86, behavior.eff.clock_trigger
2.0%	68.9%	113	behavior6.v, L88, behavior.eff.clock_trigger
2.0%	70.9%	109	behavior6.v, L25, behavior
1.8%	72.6%	98	behavior6.v, L24, behavior
1.5%	74.1%	85	behavior6.v, L52, behavior.ineff
1.4%	75.6%	81	behavior6.v, L77, behavior.eff
1.0%	76.6%	56	behavior6.v, L53, behavior.ineff
0.9%	77.5%	51	behavior6.v, L77, behavior.eff
0.9%	78.4%	51	behavior6.v, L23, behavior
0.9%	79.3%	50	behavior6.v, L77, behavior.eff
0.9%	80.2%	48	behavior6.v, L78, behavior.eff
0.9%	81.0%	48	behavior6.v, L69, behavior.ineff
0.8%	81.8%	45	behavior6.v, L52, behavior.ineff
0.8%	82.6%	42	behavior6.v, L35, behavior
0.7%	83.3%	41	behavior6.v, L52, behavior.ineff
0.7%	84.0%	40	behavior6.v, L34, behavior
0.7%	84.7%	38	behavior6.v, L52, behavior.ineff
0.7%	85.4%	38	behavior6.v, L77, behavior.eff
0.6%	86.0%	36	behavior6.v, L45, behavior
0.6%	86.6%	32	behavior6.v, L12, behavior
0.6%	87.2%	31	behavior6.v, L96, behavior.eff
0.5%	87.7%	30	behavior6.v, L2, behavior
0.5%	88.2%	30	behavior6.v, L43, behavior
0.5%	88.8%	29	behavior6.v, L94, behavior.eff
0.5%	89.3%	28	behavior6.v, L83, behavior.eff
0.5%	89.8%	28	behavior6.v, L60, behavior.ineff
0.4%	90.2%	25	behavior6.v, L13, behavior
0.4%	90.7%	25	behavior6.v, L99, behavior.eff
0.4%	91.1%	25	behavior6.v, L78, behavior.eff
0.4%	91.5%	24	behavior6.v, L98, behavior.eff
0.4%	92.0%	24	behavior6.v, L33, behavior
0.4%	92.3%	21	behavior6.v, L61, behavior.ineff
0.4%	92.7%	21	behavior6.v, L53, behavior.ineff
0.4%	93.1%	21	behavior6.v, L67, behavior.ineff
0.3%	93.4%	19	behavior6.v, L68, behavior.ineff
0.3%	93.8%	19	behavior6.v, L94, behavior.eff
0.3%	94.1%	19	behavior6.v, L67, behavior.ineff
0.3%	94.4%	18	behavior6.v, L2, behavior
0.3%	94.7%	17	behavior6.v, L69, behavior.ineff
0.3%	95.0%	17	behavior6.v, L65, behavior.ineff
0.3%	95.3%	17	behavior6.v, L97, behavior.eff
0.3%	95.6%	16	behavior6.v, L2, behavior
0.3%	95.9%	16	behavior6.v, L99, behavior.eff
0.3%	96.2%	14	behavior6.v, L58, behavior.ineff
0.3%	96.4%	14	behavior6.v, L42, behavior
0.3%	96.7%	14	behavior6.v, L83, behavior.eff
0.3%	96.9%	14	behavior6.v, L2, behavior
0.3%	97.2%	14	behavior6.v, L68, behavior.ineff
0.2%	97.4%	13	behavior6.v, L22, behavior
0.2%	97.6%	13	behavior6.v, L23, behavior

Verilog-XL Reference The Behavior Profiler

0.2%	97.9%	13	behavior6.v, L65, behavior.ineff
0.2%	98.1%	13	behavior6.v, L59, behavior.ineff
0.2%	98.3%	13	behavior6.v, L44, behavior
0.2%	98.5%	11	behavior6.v, L87, behavior.eff.clock_trigger
0.2%	98.7%	10	behavior6.v, L86, behavior.eff.clock_trigger
0.2%	98.9%	9	behavior6.v, L88, behavior.eff.clock_trigger
0.2%	99.0%	9	behavior6.v, L89, behavior.eff.clock_trigger
0.1%	99.1%	6	behavior6.v, L67, behavior.ineff
0.1%	99.2%	6	behavior6.v, L32, behavior
0.1%	99.3%	5	behavior6.v, L97, behavior.eff
0.1%	99.4%	5	behavior6.v, L84, behavior.eff
0.1%	99.5%	5	behavior6.v, L72, behavior.ineff
0.1%	99.6%	4	behavior6.v, L71, behavior.ineff
0.1%	99.7%	4	behavior6.v, L85, behavior.eff.clock_trigger
0.1%	99.7%	3	behavior6.v, L42, behavior
0.1%	99.8%	3	behavior6.v, L33, behavior
0.0%	99.8%	2	behavior6.v, L95, behavior.eff
0.0%	99.8%	2	behavior6.v, L68, behavior.ineff
0.0%	99.9%	2	behavior6.v, L70, behavior.ineff
0.0%	99.9%	2	XL
0.0%	99.9%	2	behavior6.v, L46, behavior
0.0%	100.0%	1	behavior6.v, L84, behavior.eff.clock_trigger
0.0%	100.0%	1	behavior6.v, L66, behavior.ineff
0.0%	100.0%	1	behavior6.v, L98, behavior.eff

In the previous report, there are two listings for each of the lines numbered 60, 61, 88 and 89.

■ Line 60 has 230 samples. It has the following listings:

3.6%	52.6%	202	behavior6.v, L60, behavior.ineff
0.5%	89.8%	28	behavior6.v, L60, behavior.ineff

■ Line 61 has 200 samples. It has the following listings:

3.2%	62.3%	179	behavior6.v, L61, behavior.ineff
0.4%	92.3%	21	behavior6.v, L61, behavior.ineff

■ Line 88 has 122 samples. It has the following listings:

2.0%	68.9%	113	behavior6.v, L88, behavior.eff.clock_trigger
0.2%	98.9%	9	behavior6.v, L88, behavior.eff.clock_trigger

■ Line 89 has 144 samples. It has the following listings:

2.4%	64.7%	135	behavior6.v, L89, behavior.eff.clock_trigger
0.2%	99.0%	9	behavior6.v, L89, behavior.eff.clock_trigger

There are far more samples of the lines in module `inefficient` (lines 60 and 61) than there are of the lines in module `efficient` (lines 88 and 89).

The previous profile ranking by statement report also contains several listings of the module header lines numbered 52 and 77. The behavior profiler shows samples of the module headers even though `$listcounts` shows no execution of these headers. These header samples represent CPU activity used to pass values between modules that cannot be attributed to any statement inside the modules.

Verilog-XL Reference

The Behavior Profiler

The profile ranking by module instance report for this example is as follows:

Profile ranking by module instance:

Self%	Cum.%	Samples	(Self + submodules)	Instance
36.7%	36.7%	2052	(100.0% 5586)	behavior
34.1%	70.8%	1903	(34.1% 1903)	behavior.ineff
29.2%	100.0%	1631	(29.2% 1631)	behavior.eff

This report shows that module `inefficient` uses almost 5% more of the total CPU time than module `efficient`. This discrepancy can be attributed to the fact that module `inefficient` makes procedural assignments to `q` and `qb` twice as often as module `efficient`.

Verilog-XL Reference

The Behavior Profiler

The Value Change Dump File

This chapter describes the following:

- [Overview](#) on page 493
- [Creating the Value Change Dump File](#) on page 493
- [Format of the Value Change Dump File](#) on page 498
- [Using the \\$dumpports System Task](#) on page 510

Overview

Verilog-XL can produce a file called a *value change dump (VCD) file* that contains information about value changes on selected variables in a design. You can use this VCD file for developing various applications programs and postprocessing tools. Here are some examples:

- An applications program can graphically display the results of an overnight simulation.
- A postprocessor can process the simulation results and forward them to a device tester or to a board tester.

The value change dumper is more efficient than the `$monitor` task, both in performance and in storage space. With the VCD feature, you can save the value changes of variables in any portion of the design hierarchy during any specified time interval. You can also save these results globally, without having to explicitly name all signals involved.

Creating the Value Change Dump File

The steps involved in creating the VCD file are listed below and illustrated in the following figure:

Verilog-XL Reference

The Value Change Dump File

1. Insert the VCD system tasks in the Verilog source file to define the dump filename and to specify the variables to be dumped. You may also choose to invoke these tasks interactively during the simulation instead of adding them to your source file.
2. Run the simulation.

Verilog Source File

```
initial
$dumpfile( "dump1" );
.
.
.
$dumpvars( ... )
.
.
.
```

Simulation

VCD File dump1.dump

```
(Header
Information)
(Node
Information)
(Value
Changes)
```

User
Postprocessing

Verilog-XL produces an ASCII dump file that contains header information, variable definitions, and the value changes for all variables specified in the task calls.

Several system tasks can be inserted in the source description or invoked interactively to create the VCD file. The sections that follow describe the tasks listed in the following example:

```
$dumpfile(<filename> );
$dumpvars;
$dumpvars(<levels> <, <module_or_variable>>* );
$dumppoff;
$dumpon;
$dumppall;
$dumplimit( <filesize> );
$dumpflush;
```

Specifying the Dump File Name (\$dumpfile)

Use the `$dumpfile` task to specify the name of the VCD file as follows:

```
initial
    $dumpfile ( "module1.dump" )
```

The filename is optional. If you do not specify a dump filename, the program uses the default name "verilog.dump".

The `$dumpfile` task can be invoked only once during the simulation, and it must precede all other dump file tasks, as described in the sections that follow.

Specifying Variables for Dumping (\$dumpvars)

Use the `$dumpvars` task to determine which variables Verilog-XL dumps into the file specified by `$dumpfile`. You can invoke the `$dumpvars` task throughout the design as often as necessary but these `$dumpvars` tasks must all execute at the same simulation time. You cannot add variables to be dumped once dumping begins.

You can use the `$dumpvars` task with or without arguments. The syntax for calling the task without arguments is as follows:

```
$dumpvars;
```

When invoked with no arguments, `$dumpvars` dumps all variables in the design, except those in source-protected regions, to the VCD file. If you want to include a variable from a source-protected region in the VCD file, you must include the variable in the `$dumpvars` argument list.

The syntax for calling the `$dumpvars` system task with arguments is as follows:

```
$dumpvars(<levels> <, <module_or_variable>>* );
```

When you specify `$dumpvars` with arguments, the `<levels>` variable indicates the number of hierarchical levels below each specified module instance that `$dumpvars` affects. Subsequent `<module_or_variable>` variable arguments specify which scopes of the design to dump. These subsequent arguments can specify entire modules or individual variables within a module. Here is an example:

```
$dumpvars (1, top);
```

Because the first argument in the previous example is a 1, this invocation dumps all variables within the module `top`; it does not dump variables in any of the modules instantiated by module `top`.

Setting the first argument to 0 causes a dump of all variables in the specified module and in all module instances below the specified module. Note that the argument 0 applies only to subsequent arguments that specify module instances, and not to individual variables.

In the following example, the `$dumpvars` task dumps all variables in the module `top` and in all module instances below module `top` in the design hierarchy:

```
$dumpvars (0, top);
```

The next example shows how the `$dumpvars` task can specify both modules and individual variables:

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

Verilog-XL Reference

The Value Change Dump File

This call dumps all variables in module `mod1` and in all module instances below `mod1`, along with variable `net1` in module `mod2`. Note that the argument `0` applies only to the module instance `top.mod1`, and not to the individual variable `top.mod2.net1`.

If you wish to dump individual bits of a vector net, first make sure that the net is expanded. Declaring a vector net with the keyword `scalared` guarantees that it is expanded. Using the `-x` command-line option expands all nets, but this procedure is not recommended due to its negative impact on memory usage and performance.

If the `$dumpvars` task is invoked with no arguments, all variables in the design except those in a source-protected region are dumped. However, if you include the name of a variable in a source-protected region in the `$dumpvars` argument list, then that variable is dumped. For example, if the `$dumpvars` argument list contains the variable name `'top.mod2.net1'`, then that variable is dumped even though module `'top.mod2'` may be source protected.

Stopping and Resuming the Dump (\$dumpoff/\$dumpon)

Executing the `$dumpvars` task causes value change dumping to start at the end of the current simulation time unit. To suspend the dump, invoke the `$dumpoff` task. To resume the dump, invoke `$dumpon`.

When `$dumpoff` is executed, a checkpoint is made in which every variable is dumped as an `x` value. When `$dumpon` is later executed, each variable is dumped with its value at that time. In the interval between `$dumpoff` and `$dumpon`, no value changes are dumped.

The `$dumpoff` and `$dumpon` tasks allow you to specify the simulation period during which the dump takes place. Here is an example:

```
initial
begin
    #10      $dumpvars(0, top.mod1, top.mod2.net1);
    #200    $dumpoff;
    #800    $dumpon;
    #900    $dumpoff;
end
```

This example starts the value change dumper after 10 time units, stops it 200 time units later (at time 210), restarts it again 800 time units later (at time 1010) and stops it again 900 time units later (at time 1910).

Generating a Checkpoint (\$dumpall)

The `$dumpall` task creates a checkpoint in the dump file that shows the current value of all VCD variables. It has no arguments. An example is shown in [“Sample Source Description Containing VCD Tasks”](#) on page 498.

Verilog-XL Reference

The Value Change Dump File

When dumping is enabled, the value change dumper records the values of the variables that change during each time increment. Values of variables that do not change during a time increment are not dumped.

Periodically during a dump, an applications program might find it useful to check the values of all specified variables, as a convenient checkpoint. For example, a program can save considerable time obtaining a variable value by quickly backtracking to the most recent checkpoint, rather than returning to the last time the variable changed value.

Limiting the Size of the Dump File (`$dumplimit`)

Use `$dumplimit` to set the size of the VCD file as follows:

```
$dumplimit(<filesize>;
```

This task takes a single argument that specifies the maximum size of the dump file in bytes. When the dump file reaches this maximum size, the dumping stops and the system inserts a comment in the dump file indicating that the dump limit was reached. The simulation continues uninterrupted.

Reading the Dump File During Simulation (`$dumpflush`)

The `$dumpflush` task empties the operating system's dump file buffer and ensures that all the data in that buffer is stored in the dump file. After executing a `$dumpflush` task, the system resumes dumping as before, so that no value changes are lost.

A common application is to call `$dumpflush` to update the dump file so that an applications program can read the file during a simulation.

Here is an example of using the `$dumpflush` task in a Verilog source file:

```
initial
  begin
    $dumpvars
    ...
    $dumpflush
    $(applications program)
  end
```

There are two ways of flushing the dump file buffer:

- Insert the `$dumpflush` task in the Verilog source description, as described above.
- Include a call to the PLI C-function `tf_dumpflush()` in your applications program C code.

The `$dumpflush` task and the PLI `tf_dumpflush()` functions are equivalent.

Sample Source Description Containing VCD Tasks

This section contains a simple source description example that produces a value change dump file. In this example, the name of the dump file is "verilog.dump". Verilog-XL dumps value changes for variables in the circuit. Dumping begins when event `do_dump` occurs. The dumping continues for 500 clock cycles, then stops and waits for event `do_dump` to be triggered again. At every 10000 time steps, the current values of all VCD variables are dumped.

```
module dump;
    event do_dump;
    initial $dumpfile("verilog.dump");           //Same as default file
    initial @do_dump
    $dumpvars;                                   //Dump variables in the design
    always @do_dump                              //To begin the dump at event do_dump
    begin
        $dumpon;                                 //No effect the first time through
        repeat (500) @(posedge clock);          //Dump for 500 cycles
        $dumpoff;                                //Stop the dump
    end
    initial    @(do_dump)
               forever #10000 $dumpall;         //Dump all variables for checkpoint
endmodule
```

Format of the Value Change Dump File

The information in this section pertains to users who write their own application programs to postprocess the VCD file.

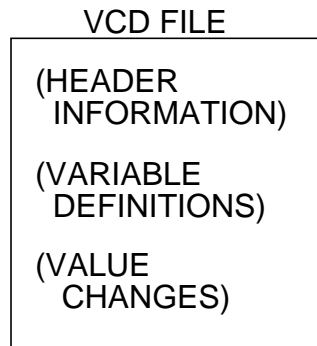
Contents of the Dump File

As shown in the following figure, the VCD file starts with the header information (the date, the version number of Verilog-XL used for the simulation, and the timescale used). Next, the file

Verilog-XL Reference

The Value Change Dump File

lists the definitions of the scope and the type of variables being dumped, followed by the actual value changes at each time increment.



Only the variables that change value during a time increment are listed. Value changes for non-real variables are specified by 0, 1, X, or Z values. Value changes for real variables are specified by real numbers. Strength information and memories are not dumped.

Note: You cannot dump *part* of a vector. For example, you cannot dump only bits 8 through 15 (8:15) of a 16-bit vector. You must dump the entire vector (0:15). In addition, you cannot dump expressions, such as $a + b$.

Structure of the Dump File

The dump file is structured in a free format. White space is used to separate commands and to make the file easily readable by a text editor. Output data in the VCD file is case sensitive.

To simplify postprocessing of the VCD file, the value change dumper automatically generates 1- to 4-character identifier codes (taken from the visible ASCII characters) to represent variables. The examples in “[Description of Keyword Commands](#)” on page 501 show these identifier codes, such as `*@` and `(k`.

Note that the value change dump file contains limited structural information, including information about the design hierarchy, but has no interconnection information. As a result, the VCD file by itself does not allow a postprocessor to display the drivers and loads on a net.

Formats of Dumped Variable Values

Variables may be either scalars or vectors. Each type has its own format in the VCD file. Dumps of value changes to scalar variables contain no white space between the value and the identifier code, as in this example:

```
1*@          // No space between the value 1 and the identifier code *@
```

Verilog-XL Reference

The Value Change Dump File

Dumps of value changes to vectors contain no white space between the base letter and the value digits, but they do contain white space between the value digits and the identifier code, as in this example:

```
b1100x01z (k //No space between the b and 1100x01z, but space between b1100x01z
and (k
```

The output format for each value is right-justified. Vector values appear in the shortest form possible: the VCD eliminates redundant bit values that result from left-extending values to fill a particular vector size.

The rules for left-extending vector values are as follows:

When the value is:	VCD left-extends with:
1	0
0	0
Z	Z
X	X

The following table shows how the VCD shortens values:

The binary value:	Extends to fill a 4-bit register as:	Appears in the VCD file as:
10	0010	b10
X10	XX10	bX10
ZX0	ZZX0	bZX0
0X10	0X10	b0X10

Events are dumped in the same format as scalars (for example, 1*%). For events, however, the value (1 in this example) is irrelevant. Only the identifier code (*% in this example) is significant. It appears in the VCD file as a marker to indicate that the event was triggered during the time step.

Using Keyword Commands

Much of the general information in the value change dump file is presented as a series of keyword commands that a postprocessor can parse. Refer to [“Syntax of the VCD File”](#) on page 506 for information about keyword commands use.

Verilog-XL Reference

The Value Change Dump File

If a postprocessor reads a keyword command that it does not recognize, it can perform error processing such as displaying a warning message and ignoring the text that appears between the keyword command and the `$end`.

Description of Keyword Commands

Keyword commands provide a means of inserting information into the VCD file. Keyword commands can be inserted either by the dumper or by you, as shown in the example in [“Value Change Dump File Format Example”](#) on page 507. This section deals with the following keyword commands:

<code>\$comment</code>	<code>\$dumpoff</code>	<code>\$enddefinitions</code>	<code>\$upscope</code>
<code>\$date</code>	<code>\$dumpon</code>	<code>\$scope</code>	<code>\$var</code>
<code>\$dumpall</code>	<code>\$dumpvars</code>	<code>\$timescale</code>	<code>\$version</code>

Applications programs that read the value change dump file must be able to recognize and process the standard keyword commands defined in the following pages. You also can define additional keyword commands for each application.

\$comment

The `$comment` keyword provides a means of inserting a comment in the VCD file.

Syntax

```
$comment
    <comment_text>
$end
```

Examples

```
$comment    This is a single-line comment    $end
$comment    This is a
multiple-line comment
$end
```

\$date

The date stamp allows the system to indicate the date on which the VCD file was generated.

Verilog-XL Reference

The Value Change Dump File

Syntax

```
$date  
    <date_text>  
$end
```

Example

```
$date  
    June 25, 1989 09:24:35  
$end
```

\$dumpall

`$dumpall` lists the current values of all the variables dumped.

Syntax

```
$dumpall    $end
```

Example

```
$dumpall    1*@    x*#    0*$    bx    (k    $end
```

\$dumpoff

`$dumpoff` lists all the variables dumped with X values and then stops dumping.

Syntax

```
$dumpoff    $end
```

Example

```
$dumpoff    x*@    x*#    x*$    bx    (k    $end
```

\$dumpon

`$dumpon` resumes dumping and list current values of all variables dumped.

Syntax

```
$dumpon    $end
```

Verilog-XL Reference

The Value Change Dump File

Example

```
$dumpon    x*@    0*#    x*$    b1    (k    $end
```

\$dumpvars

`$dumpvars` lists the initial values of all the variables dumped.

Syntax

```
$dumpvars <value_changes>* $end
```

Example

```
$dumpvars    x*@    z*$    b0    (k    $end
```

\$enddefinitions

`$enddefinitions` marks the end of the header information and definitions.

Syntax

```
$enddefinitions    $end
```

\$scope

Scope definition defines the scope of the dump.

Syntax

```
$scope  
    <scope_type> <identifier>  
$end
```

Definitions

`<scope_type>` is one of the following keywords:

<code>module</code>	for top-level module and module instances
<code>task</code>	for a task
<code>function</code>	for a function
<code>begin</code>	for named sequential blocks

Verilog-XL Reference

The Value Change Dump File

<code>fork</code>	for named parallel blocks
-------------------	---------------------------

Example

```
$scope
  module top
$end
```

\$timescale

`$timescale` specifies the timescale that Verilog-XL used for the simulation.

Syntax

```
$timescale <number> <time_dimension> $end
```

Definitions

`<number>` is one of the following:

```
1  10  100
```

`<time_dimension>` is one of the following:

```
s  ms  us  ns  ps  fs
```

For more information see [Chapter 17, “Timescales.”](#)

Example

```
$timescale 10 ns $end
```

\$upscope

`$upscope` changes the scope to the next higher level in the design hierarchy.

Syntax

```
$upscope $end
```

\$var

`$var` prints the names and identifier codes of the variables being dumped.

Verilog-XL Reference

The Value Change Dump File

Syntax

```
$var
    <var_type> <size> <identifier_code>
    <reference>
$end
```

Definitions

<var_type> specifies the variable type and can be one of the following keywords:

event	integer	parameter	real	reg
supply0	supply1	time	tri	triand
trior	trireg	tri0	tril	wand
wire	wor			

<size> is a decimal number that specifies how many bits are in the variable.

<identifier_code> specifies the name of the variable using printable ASCII characters, as described in [“Structure of the Dump File”](#) on page 499.

<reference> is a reference name you specify in the source file. More than one *<reference>* name may be mapped to the same *<identifier_code>*. For example, net10 and net15 may be interconnected in the circuit, and therefore will have the same *<identifier_code>*. A *<reference>* can have the following components:

```
::= <identifier>
| | = <identifier> [ <bit_select_index> ]
| | = <identifier> [ <MSI> : <LSI> ]
```

- *<identifier>* is the Verilog name of the saved variable, including any leading backslash (\) characters for escape identifiers.
- Verilog-XL dumps each bit of an expanded vector net individually. That is, each bit has its own *<identifier_code>* and is dumped only when it changes, not when other bits in the vector change.
- *<MSI>* indicates the most significant index; *<LSI>* indicates the least significant index.
- *<bit_select_index>*, *<MSI>*, and *<LSI>* are all decimal numbers.

Example

```
$var
    integer 32 (2 index
$end
```

\$version

\$version indicates the version of the simulator that was used to produce the VCD file.

Verilog-XL Reference

The Value Change Dump File

Syntax

```
$version
  <version_text>
$end
```

Example

```
$version
  VERILOG-XL 1.5a
$end
```

Syntax of the VCD File

The following example shows the syntax of the output VCD file:

```
<value_change_dump_definitions>
  := <declaration_command>*<simulation_command>*
<declaration_command>
  ::= <keyword_command>
  (NOTE: Keyword_commands are described in the next section.)
<simulation_command>
  ::= <keyword_command>
  || = <simulation_time>
  || = <value_change>
<keyword_command>
  ::= $<keyword> <command_text> $end
<simulation_time>
  ::= #<decimal_number>
<value_change>
  ::= <scalar_value_change>
  || = <vector_value_change>
<scalar_value_change>
  ::= <value><identifier_code>
  <value> is one of the following: 0 1 x X z Z
<vector_value_change>
  ::= b<binary_number> <identifier_code>
  ::= B<binary_number> <identifier_code>
  ::= r<real_number> <identifier_code>
  ::= R<real_number> <identifier_code>
```

- *<binary_number>* is a number composed of the following characters: 0 1 x X z Z
- *<real_number>* is a real number will be dumped using a `%.16g printf()` format. This format preserves the precision of the number by outputting all 53 bits in the mantissa of a 64-bit C-code 'double'. Applications programs can read a real number using a `%g` format to `scanf()`.
- *<identifier_code>* is a code from 1 to 4 characters long composed of the printable characters that are in the ASCII character set from ! to ~ (decimal 33 to 126).

Verilog-XL Reference

The Value Change Dump File

Value Change Dump File Format Example

The following example illustrates the format of the value change dump file. A description of the file format follows the example. With the exception of the `$comment` command, all other keyword commands in this example are generated by the value change dumper.

```
$date
  June 26, 1998 10:05:41
$end
$version
  VERILOG-XL 2.7
$end
$timescale
  1 ns
$end
$scope      module      top      $end
$scope      module      m1      $end
$var        trireg 1     *@ net1  $end
$var        trireg 1     *# net2  $end
$var        trireg 1     *$ net3  $end
$upscope    $end
$scope      task        t1      $end
$var        reg 32 (k accumulator[31:0] $end
$var        integer 32 {2 index $end
$upscope    $end
$upscope    $end
$enddefinitions $end
$comment
  Note:          $dumpvars was executed at time '#500'.
                All initial values are dumped at this time.
$end
#500
$dumpvars x*@ x*# x*$ bx (k bx {2 $end
#505
0*@
1*#
1*$
b10zx1110x11100 (k b1111000101z01x {2
#510
0*$
#520
1*$
#530
0*$
bz (k
#535
$dumpall 0*@ 1*# 0*$
bz (k b1111000101z01x {2 $end
#540
1*$
#1000
$dumppoff x*@ x*# x*$ bx (k bx {2 $end
#2000
$dumpon z*@ 1*# 0*$ b0 (k bx {2 $end
#2010
1*$
```

Verilog-XL Reference

The Value Change Dump File

Note: In general, the VCD does not automatically include comments in the dump file. An exception is when the dump file reaches the limit set by the `$dumplimit` task. Then, the VCD includes a comment to that effect.

The following sections describe the keyword commands, variables, and values that appear in the previous example.

\$date...\$end \$version...\$end \$timescale...\$end

These keyword commands are generated by the VCD to provide information about the dump file and the simulation.

\$scope...\$end

This keyword command indicates the scope of the defined signals that follow. In this example, the scope includes two modules (`top` and `m1`) and one task (`t1`).

\$var...\$end

This keyword command defines each variable that is dumped. It specifies the variable's type (`trireg`, `reg`, `integer`), size (1, 32, 32), identifier code (`*@`, `(k`, `{2}`), and reference name as specified by the user in the source file (`net1`, `accumulator`, `index`). To make the dump file machine-independent, compact, and capable of being edited, the VCD assigns each variable in the circuit a 1- to 4-character code called an identifier code. These characters are taken from the visible ASCII characters '!' to '~' (decimal 33 to 126).

\$upscope...\$end

For each `$scope` there is a matching `$upscope` to signify the end of that scope.

\$enddefinitions...\$end

This keyword command indicates the end of the header information and definitions, and marks the start of the value change data.

Verilog-XL Reference

The Value Change Dump File

#500

```
$dumpvars x*@ x*# x*$ bx (k bx {2 $end
```

At time 500, `$dumpvars` is executed to show the initial values of all the variables dumped. Identification codes (such as `*@`, `*#`) are used for conciseness and are associated with user reference names in the `$var ... $end` sections of the VCD file. In this example, all initial values are X (unknown).

#505

```
0*@
```

```
1*#
```

```
1*$
```

```
b10zx1110x11100 (k          b1111000101z01x {2
```

This display shows the new values of all the variables that changed at time 505: `net1` (which has an identifier code of `*@`) changed to 0, `net2` (identifier code `*#`) and `net3` (identifier code `*$`) changed to 1, the vector `accumulator[31:0]` and the integer `index` changed to the binary values shown.

#510

```
0*$
```

#520

```
1*$
```

#530

```
0*$
```

```
bz (k
```

At time 510, only `net3` changed to a 0. All other variables remained unchanged. At time 520, `net3` changed to a 1, and at time 530 it changed back to a 0. Also at time 530, all bits of the vector `accumulator` changed to the high-impedance (Z) state.

#535

```
$dumpall 0*@ 1*# 0*$
```

```
bz (k b1111000101z01x {2 $end
```

The source file calls a `$dumpall` task at time 535 to dump the latest values of all the specified variables as a checkpoint.

#540

1*\$

At time 540, `net3` changed to a 1.

#1000

\$dumpoff x*@ x*# x*\$ bx (k bx {2 \$end

At time 1000, a `$dumpoff` is executed to dump all the variables as X values and to suspend dumping until the next `$dumpon`.

#2000

\$dumpon z*@ 1*# 0*\$ b0 (k bx {2 \$end

Dumping resumes at time 2000; `$dumpon` dumps all the variables with their values at that time.

#2010

1*\$

At time 2010, the value of `net3` changes to 1.

Using the \$dumpports System Task

The procedures described in this section are deliberately broad and generic. The requirements for your specific design may dictate procedures slightly different from those described here.

\$dumpports Syntax

The `$dumpports` system task scans the (`arg1`) ports of a module instance and monitors the ports for both value and drive level. The `$dumpports` system task also generates an output file that contains the value, direction, and strength of all the ports of a device. The output file generated by `$dumpports` is similar to the output file generated by the value change dump (VCD). For information about the VCD file, see [“Overview”](#) on page 493. The syntax for `$dumpports` is as follows:

```
$dumpports(<DUT> <,"filename"> <,ID>);
```

Verilog-XL Reference

The Value Change Dump File

The table given below describes the arguments of `$dumpports`.

Arguments	Description
<i>DUT</i>	Device under test (DUT); the name of the module instance to be monitored.
<i>filename</i>	String containing the name of the output file. The filename argument is optional. If you do not specify a filename, then a <code>verilog.evcd</code> (the default filename) file will be generated. Consider the following examples: Example 1: <pre> \$dumpports(dut, , id);</pre> In this example, the filename argument is not specified. No error will be reported and a <code>verilog.evcd</code> file will be generated. Example 2: <pre> \$dumpports(dut, "testVec.file", id);</pre> In this example, as the filename argument is specified, a <code>testVec.file</code> will be generated.
<i>ID</i>	An integer data type that identifies a running <code>\$dumpports</code> task with the <code>\$dumpports_close</code> system task. For more information, see “\$dumpports_close” on page 515.

Consider the example given below.

```
....
....
module top;
reg A;
integer id;
.....
.....
initial
    begin
        $dumpports(dut, "testVec.file", id);
        #5 $dumpports_close(id);
    end
endmodule
....
....
```

\$dumpports Output

The following example shows an output from using `$dumpports`.

```
#100
pDDBF 6566 0066 <1
```

Verilog-XL Reference

The Value Change Dump File

The table given below describes each component of the output.

Component	Description
#100	Simulation time
p	Output type of the driver. p indicates the value, strength, and collision detection for the ports.
DDBF	Value of the port. See “Port Value Character Identifiers” on page 513 for more information.
6566	0's strength component of the value. See “Strength Mapping” on page 514 for more information.
0066	1's strength component of the value. See “Strength Mapping” on page 514 for more information.
<1	Signal identifier

Port Names

Port names are recorded in the output file as follows:

- A port that is explicitly named is recorded in the output file.
- If a port is not explicitly named, the name of the object used in the port definition is recorded.
- If the name of a port cannot be determined from the object used in the port definition, the port index number is used (the first port being 0).

Drivers

A driver is anything that can drive a value onto a net including the following:

- primitives
- continuous assigns
- forces
- ports with objects of type other than net, such as the following:

```
module foo(out, ...)
  output out;
  reg out;
```


Verilog-XL Reference

The Value Change Dump File

If a net is forced, a comment is placed into the output file stating that the net connected to the port is being forced, and the scope of the force is given. Forces are treated differently because the existence of a force is not permanent, even though a force is a driver.

While a force is active, driver collisions are ignored and the level part of the output is determined by the scope of the force definition. When the force is released, a note is again placed into the log file.

Port Value Character Identifiers

The following table shows the characters (middle column) that identify the value of a port in the `$dumpports` output:

Level	Char.	Value
DUT	L	(0) low
DUT	l	(0) low with more than 2 active drivers
DUT	H	(1) high
DUT	h	(1) high with more than 2 active drivers
DUT	T	(Z) tri-state
DUT	X	(X) unknown
DUT	x	(X) unknown because of a 1-0 collision
unknown	?	(X) unknown state
unknown	0	(0) unknown direction; both the input and the output are active with the same value.
unknown	1	(1) unknown direction; both the input and the output are active with the same value.
unknown	A	(0-1) Input is a 0 and output is a 1.
unknown	a	(0-X) Input is a 0 and output is an X
unknown	B	(1-0) Input is a 1 and output is a 0.
unknown	b	(1-X) Input is a 1 and output is an X.
unknown	C	(X-0) Input is a X and output is a 0.
unknown	c	(X-1) Input is a X and output is a 1.
unknown	F	(Z) tri stated, nothing is driving the net.

Verilog-XL Reference

The Value Change Dump File

Level	Char.	Value
unknown	f	(Z) tri stated, both internal and external.
test fixture	D	(0) low
test fixture	d	(0) low with more than 2 active drivers
test fixture	U	(1) high
test fixture	u	(1) high with more than 2 active drivers
test fixture	N	(X) unknown
test fixture	n	(X) unknown because of a 1-0 collision
test fixture	Z	(Z) tri-state

The level of a driver is determined by the scope of the driver's placement on a net. Port type has no influence on the level of a signal. Any driver whose definition is outside of the scope of the DUT is at the test fixture level.

In the following example, because the driver is outside the scope of a DUT, the continuous assignment is at the test fixture level:

```
module top;
  reg regA;
  assign dut1.out = regA;
  dut dut1(out, ....);
  initial
    $dumpports(dut1, "testVec.file");
  ...
endmodule
module dut(out, ...
  output out;
  wire out;
  ...
endmodule
```

Strength Mapping

Strength values in the `$dumpports` output are as follows:

```
0 HiZ
1 Sm Cap
2 Md Cap
3 Weak
4 Lg Cap
5 Pull
6 Strong
7 Supply
```

\$dumpports Restrictions

The following restrictions apply to the `$dumpports` system task:

- The `$dumpports` system task does not work with the `$save` and `$restart` system tasks.
- Continuous assignments cannot have delays.
- The following wire types are the only ones permitted to be connected to the ports:
`wire, tri, tri0, tri1, reg, trireg`
- Directional information can be lost when a port is driven by one or more drivers of the different `tran` elements (`tran, rtran, rtranif0, ...`).

In the following example, the direction of the port `out` cannot be determined because the value that the `tran` gate is transporting from its other terminal is unknown.

```
bufif1 ul(out, 1'b1, 1'b1);
DUT    udut(out);
...
module DUT(out)
    tran t1(out, int);
```

\$dumpports_close

The `$dumpports_close` system task stops a running `$dumpports` system task. The syntax is as follows:

```
$dumpports_close(<ID>);
```

The optional `<ID>` argument is an integer that identifies a particular `$dumpports` system task. If only one `$dumpports` system task is running, the `<ID>` can be left blank. Valid `<ID>` values are specified in the syntax of the `$dumpports` system task.

Consider the example given below.

```
....
....
module top;
reg A;
integer id;
.....
.....
initial
begin
    $dumpports(dut, "testVec.file", id);
    #5 $dumpports_close(id);
end
endmodule
```

Verilog-XL Reference
The Value Change Dump File

....
....

Formal Syntax Definition

This appendix describes the following:

- [Summary of Syntax Descriptions](#) on page 517
- [Source Text](#) on page 518
- [Declarations](#) on page 521
- [Primitive Instances](#) on page 523
- [Module Instantiations](#) on page 523
- [Behavioral Statements](#) on page 524
- [Specify Section](#) on page 526
- [Expressions](#) on page 529
- [General Syntax Definition](#) on page 530
- [Switch-Level Modeling](#) on page 531

Summary of Syntax Descriptions

The following items summarize the format of the formal syntax descriptions in this appendix:

1. Spaces may be used to separate lexical tokens.
2. Angle brackets around each description item are added for clarity and are *not* literal symbols—that is, they do not appear in a source example of a syntax item.
3. <ITEM> in upper case is a lexical token item. Its definition is a terminal node in the description hierarchy—that is, its definition does not contain any syntax construct items.
4. <item> in lower case is a syntax construct item defined by other syntax construct items (<item>) or by lexical token items (<ITEM>).
5. <item>? indicates an optional item.

Verilog-XL Reference

Formal Syntax Definition

6. `<item>*` indicates that an item occurs zero or more times in the syntax construct.
7. `<item>+` indicates that an item occurs one or more times in the syntax construct.
8. `<item> <,<item>>*` is a comma-separated list of items with at least one item in the list.
9. `iff [condition]` is a condition placed on one of several definitions
10. `::=` the right side of this syntax construct defines the `<item>` on the left side.
11. `||=` like `::=`, but this syntax construct indicates an alternative syntax definition.
12. `item` is a literal (a keyword). For example, the definition `<event_declaration> ::= event <name_of_event>` indicates that the keyword “event” precedes the name of an event in an event declaration.
13. `(...)` parenthesis symbols in a definition are required literals by the syntax being defined. Other literal symbols can also appear in a definition (for example, the period `(.)` and the colon `(:)`).

Note: In Verilog syntax, a period `(.)` may not be preceded or followed by a space.

Source Text

```
<source_text>
  ::= <description>*

<description>
  ::= <module>
  ||= <primitive>

<module>
  ::= module <name_of_module> <list_of_ports>? ;
     <module_item>*
     endmodule
  ||= macromodule <name_of_module> <list_of_ports>? ;
     <module_item>*
     endmodule

<name_of_module>
  ::= <IDENTIFIER>
     See "Identifiers, Keywords, and System Names" on page 28

<list_of_ports>
  ::= ( <port> <,<port>>* )

<port>
  ::= <port_expression>?
  ||= . <name_of_port> ( <port_expression>? )

<port_expression>
  ::= <port_reference>
  ||= { <port_reference> <,<port_reference>>* }

<port_reference>
  ::= <name_of_variable>
```

Verilog-XL Reference

Formal Syntax Definition

```
|| = <name_of_variable> [ <constant_expression> ]
|| = <name_of_variable>
  [ <constant_expression> : <constant_expression> ]
<name_of_port>
  ::= <IDENTIFIER>
<name_of_variable>
  ::= <IDENTIFIER>
<module_item>
  ::= <parameter_declaration>
    See "Wired Nets" on page 41
  || = <input_declaration>
    See "Port Declarations" on page 220
  || = <output_declaration>
    See "Port Declarations" on page 220
  || = <inout_declaration>
    See "Port Declarations" on page 220
  || = <net_declaration>
    See "Signed Objects" on page 33
  || = <reg_declaration>
    See "Signed Objects" on page 33
  || = <time_declaration>
    See "Integers and Times" on page 47
  || = <integer_declaration>
    See "Integers and Times" on page 47
  || = <real_declaration>
    See "Real Number Declaration Syntax" on page 48
  || = <event_declaration>
    See "Event Control" on page 184
  || = <gate_declaration>
    See "Gate and Switch Declaration Syntax" on page 98
  || = <UDP_instantiation>
    See "UDP Syntax" on page 144
  || = <module_instantiation>
    See "Module Instantiation" on page 211
  || = <parameter_override>
    See "Overriding Module Parameter Values" on page 213
  || = <continuous_assign>
    See "Continuous Assignments" on page 72
  || = <specify_block>
    See "Using Specify Blocks and Path Delays" on page 237
  || = <initial_statement>
    See "initial Statement" on page 165
  || = <always_statement>
    See "always Statement" on page 165
  || = <task>
  || = <function>
<UDP>
  ::= primitive <name_of_UDP> ( <name_of_variable>
    <, <name_of_variable>>* ) ;
    <UDP_declaration>+
    <UDP_initial_statement>?
    <table_definition>
  endprimitive
<name_of_UDP>
  ::= <IDENTIFIER>
<UDP_declaration>
  ::= <output_declaration>
    See "UDP Syntax" on page 144
```

Verilog-XL Reference

Formal Syntax Definition

```
||= <reg_declaration>
    See "UDP Syntax" on page 144
||= <input_declaration>
    See "UDP Syntax" on page 144
<UDP_initial_statement>
    ::= initial <output_terminal_name> = <init_val> ;
<init_val>
    ::= 1'b0
       || = 1'b1
       || = 1'bx
       || = 1
       || = 0
<table_definition>
    ::= table <table_entries> endtable
<table_entries>
    ::= <combinational_entry>+
       || = <sequential_entry>+
<combinational_entry>
    ::= <level_input_list> : <OUTPUT_SYMBOL> ;
<sequential_entry>
    ::= <input_list> : <state> : <next_state> ;
<input_list>
    ::= <level_input_list>
       || = <edge_input_list>
<level_input_list>
    ::= <LEVEL_SYMBOL>+
<edge_input_list>
    ::= <LEVEL_SYMBOL>* <edge> <LEVEL_SYMBOL>*
<edge>
    ::= ( <LEVEL_SYMBOL> <LEVEL_SYMBOL> )
       || = <EDGE_SYMBOL>
<state>
    ::= <LEVEL_SYMBOL>
<next_state>
    ::= <OUTPUT_SYMBOL>
       || = - (This is a literal hyphen)
           See "User-Defined Primitives \(UDPs\)" on page 143.
<OUTPUT_SYMBOL> is one of the following characters:
    0  1  x  X
<LEVEL_SYMBOL> is one of the following characters:
    0  1  x  X  ?  b  B
<EDGE_SYMBOL> is one of the following characters:
    r  R  f  F  p  P  n  N  *
<task>
    ::= task <name_of_task> ;
       <tf_declaration>*<statement_or_null>
       endtask
<name_of_task>
    ::= <IDENTIFIER>
<function>
    ::= function <range_or_type>? <name_of_function> ;
       <tf_declaration>+
```


Verilog-XL Reference

Formal Syntax Definition

```
    <statement_or_null>
endfunction

<range_or_type>
 ::= <range>                               See "Defining a Function" on page 201
  || = integer
  || = real

<name_of_function>
 ::= <IDENTIFIER>

<tf_declaration>
 ::= <parameter_declaration>                See "Parameters" on page 50
  || <input_declaration>                   See "Port Declarations" on page 220
  || <output_declaration>                  See "Port Declarations" on page 220
  || <inout_declaration>                   See "Port Declarations" on page 220
  || <reg_declaration>
      See "Net and Register Declaration Syntax" on page 35
  || <time_declaration>                    See "Integers and Times" on page 47
  || <integer_declaration>                 See "Integers and Times" on page 47
  || <real_declaration>
      See "Real Number Declaration Syntax" on page 48
  || <event_declaration>                   See "Event Control" on page 184
```

Declarations

```
<parameter_declaration>
 ::= parameter <list_of_param_assignments> ;

<list_of_param_assignments>
 ::= <param_assignment> <, <param_assignment> *

<param_assignment>
 ::= <<identifier> = <constant_expression>>

<input_declaration>
 ::= input <range>? <list_of_variables> ;

<output_declaration>
 ::= output <range>? <list_of_variables> ;

<inout_declaration>
 ::= inout <range>? <list_of_variables> ;

<net_declaration>
 ::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
  || = trireg <charge_strength>? <expandrange>? <delay>?
      <list_of_variables> ;

<NETTYPE> is one of the following keywords:
  wire tri tril supply0 wand triand tri0
  supply1 wor trior trireg

<expandrange>
 ::= <range>
  || = scalared <range>
      iff [the data type is not a trireg]
          the following syntax is available:
  || = vectored <range>

<delay>
 ::=
      See "Gate and Switch Declaration Syntax" on page 98
```

Verilog-XL Reference

Formal Syntax Definition

```
<reg_declaration>
  ::= reg <range>? <list_of_register_variables> ;

<time_declaration>
  ::= time <list_of_register_variables> ;

<integer_declaration>
  ::= integer <list_of_register_variables> ;

<real_declaration>
  ::= real <list_of_variables> ;

<event_declaration>
  ::= event <name_of_event> <,<name_of_event>>* ;

<continuous_assign>
  ::= assign <drive_strength>? <delay>? <list_of_assignments> ;
  ||= <NETTYPE> <drive_strength>? <expandrange>? <delay>?
      <list_of_assignments> ;
      <parameter_override>
  ::= defparam <list_of_param_assignments> ;

<list_of_variables>
  ::= <name_of_variable> <,<name_of_variable>>*

<name_of_variable>
  ::= <IDENTIFIER>

<list_of_register_variables>
  ::= <register_variable> <,<register_variable>>*

<register_variable>
  ::= <name_of_register>
  ||= <name_of_memory>
      [ <constant_expression> : <constant_expression> ]

<constant_expression>
  ::=
    See "Expressions" on page 51

<name_of_register>
  ::= <IDENTIFIER>
    See "Identifiers, Keywords, and System Names" on page 28

<name_of_memory>
  ::= <IDENTIFIER>
    See "Identifiers, Keywords, and System Names" on page 28

<name_of_event>
  ::= <IDENTIFIER>
    See "Identifiers, Keywords, and System Names" on page 28

<charge_strength>
  ::= ( small )
  ||= ( medium )
  ||= ( large )

<drive_strength>
  ::= ( <STRENGTH0> , <STRENGTH1> )
  ||= ( <STRENGTH1> , <STRENGTH0> )

<STRENGTH0> is one of the following keywords:
  supply0 strong0 pull0 weak0 highz0

<STRENGTH1> is one of the following keywords:
  supply1 strong1 pull1 weak1 highz1

<range>
  ::= [ <constant_expression> : <constant_expression> ]
```

Verilog-XL Reference

Formal Syntax Definition

```
<list_of_assignments>
 ::= <assignment> <,<assignment>>*</pre><pre>
<expression>
 ::=
   See "Expressions" on page 51
<assignment>
 ::=
   See "Assignments" on page 71
```

Primitive Instances

```
<gate_declaration>
 ::= <GATETYPE> <drive_strength>? <delay>? <gate_instance>
   <,<gate_instance>>*</pre><pre>
<GATETYPE> is one of the following keywords:
  and buf bufif0 bufif1 cmos nand nmos nor not notif0 notif1
  or pmos pulldown pullup rcmos rmos rpmos rtran rtranif0 rtranif1
  tran tranif0 tranif1 xnor xor
<drive_strength>
 ::= ( <STRENGTH0>,<STRENGTH1> )
   || = ( <STRENGTH1>,<STRENGTH0> )
<delay>
 ::= # <number>
   || = # <identifier>
   || = # ( <mintypmax_expression> <,<mintypmax_expression>>? )
   <,<mintypmax_expression>>? )
<gate_instance>
 ::= <name_of_gate_instance>? ( <terminal> <,<terminal>>*</pre><pre>
<name_of_gate_instance>
 ::= <IDENTIFIER>
   See "Identifiers, Keywords, and System Names" on page 28
<UDP_instantiation>
 ::= <name_of_UDP> <drive_strength>? <delay>? <UDP_instance>
   <,<UDP_instance>>*</pre><pre>
<name_of_UDP>
 ::= <IDENTIFIER>
   See "Identifiers, Keywords, and System Names" on page 28
<UDP_instance>
 ::= <name_of_UDP_instance>? ( <terminal> <,<terminal>>*</pre><pre>
<name_of_UDP_instance>
 ::= <IDENTIFIER>
   See "Identifiers, Keywords, and System Names" on page 28
<terminal>
 ::= <expression>
   || = <IDENTIFIER>
```

Module Instantiations

```
<module_instantiation>
 ::= <name_of_module> <parameter_value_assignment>?
   <module_instance> <,<module_instance>>*</pre><pre>
<name_of_module>
 ::= <IDENTIFIER>
   See "Identifiers, Keywords, and System Names" on page 28
```

Verilog-XL Reference

Formal Syntax Definition

```
<parameter_value_assignment>
 ::= # ( <expression> <, <expression>>* )

<module_instance>
 ::= <name_of_instance> ( <list_of_module_connections>? )

<name_of_instance>
 ::= <IDENTIFIER>
    See "Identifiers, Keywords, and System Names" on page 28

<list_of_module_connections>
 ::= <module_port_connection> <,<module_port_connection>>*
    || = <named_port_connection> <,<named_port_connection>>*

<module_port_connection>
 ::= <expression>
    See "Expressions" on page 51
    || = <NULL>

<NULL>
 ::= nothing - this form covers the case of an empty item
    in a list -for example:
    (a, b, , d)

<named_port_connection>
 ::= .< IDENTIFIER> ( <expression?> )

<expression>
 ::=
    See "Expressions" on page 51
```

Behavioral Statements

```
<initial_statement>
 ::= initial <statement>

<always_statement>
 ::= always <statement>

<statement_or_null>
 ::= <statement>
    || = ;

<statement>
 ::= <assignment> ;
    || = if ( <expression> ) <statement_or_null>
    || = if ( <expression> ) <statement_or_null>
        else <statement_or_null>
    || = case ( <expression> ) <case_item>+ endcase
    || = casez ( <expression> ) <case_item>+ endcase
    || = casex ( <expression> ) <case_item>+ endcase
    || = forever <statement>
    || = repeat ( <expression> ) <statement>
    || = while ( <expression> ) <statement>
    || = for ( <assignment> ; <expression> ; <assignment> )
        <statement>
    || = <delay_control> <statement_or_null>
        See "Delay Control" on page 183
    || = <event_control> <statement_or_null>
        See "Event Control" on page 184
    || = <lvalue> = <delay_control> <expression> ;
    || = <lvalue> = <event_control> <expression> ;
    || = wait ( <expression> ) <statement_or_null>
    || = -> <name_of_event> ;
```

Verilog-XL Reference

Formal Syntax Definition

```

| = <seq_block>
| = <par_block>
| = <task_enable>
| = <system_task_enable>
| = disable <name_of_task> ;
| = disable <name_of_block> ;
| = assign <assignment> ;
| = deassign <lvalue> ;

<assignment>
 ::= <lvalue> = <expression>

<lvalue>
 ::=
   See "Procedural Assignments" on page 166

<expression>
 ::=
   See "Expressions" on page 51

<case_item>
 ::= <expression> <,<expression>>* : <statement_or_null>
    || = default : <statement_or_null>
    || = default <statement_or_null>

<seq_block>
 ::= begin <statement>* end
    || = begin : <name_of_block> <block_declaration>*
        <statement>* end

<par_block>
 ::= fork <statement>* join
    || = fork : <name_of_block> <block_declaration>*
        <statement>* join

<name_of_block>
 ::= <IDENTIFIER>

<block_declaration>
 ::= <parameter_declaration> See "Parameters" on page 50
    || = <reg_declaration>
        See "Net and Register Declaration Syntax" on page 35
    || = <integer_declaration> See "Integers and Times" on page 47
    || = <real_declaration> See "Real Numbers" on page 48
    || = <time_declaration> See "Integers and Times" on page 47
    || = <event_declaration> See "Event Control" on page 184

<task_enable>
 ::= <name_of_task> ; See "Defining a Task" on page 198
    || = <name_of_task> ( <expression> <,<expression>>* ) ;

<system_task_enable>
 ::= <name_of_system_task> ;
    || = <name_of_system_task> ( <expression> <,<expression>>* ) ;

<name_of_system_task>
 ::= $<SYSTEM_IDENTIFIER>
    The $ cannot be followed by a space.

<SYSTEM_IDENTIFIER>
 ::= An <IDENTIFIER> assigned to an existing system task or function.
```

Specify Section

```

<specify_block>
  ::= specify <specify_item>* endspecify

<specify_item>
  ::= <specparam_declaration>
     | <path_declaration>
     | <level_sensitive_path_declaration>
     | <edge_sensitive_path_declaration>
     | <system_timing_check>
     | <sdpd>

<specparam_declaration>
  ::= specparam <list_of_param_assignments> ;

<list_of_param_assignments>
  ::= <param_assignment><,<param_assignment>>*

<param_assignment>
  ::= <<identifier>=<constant_expression>>

<path_declaration>
  ::= <path_description> = <path_delay_value> ;

<path_description>
  ::= ( <specify_input_terminal_descriptor> =>
        <specify_output_terminal_descriptor> )
     || ( <list_of_path_inputs> * > <list_of_path_outputs> )
     || <list_of_path_inputs>
  ::= <specify_input_terminal_descriptor>
     <,<specify_input_terminal_descriptor>>*

<list_of_path_outputs>
  ::= <specify_output_terminal_descriptor>
     <,<specify_output_terminal_descriptor>>*

<specify_input_terminal_descriptor>
  ::= <input_identifier>
     || <input_identifier> [ <constant_expression> ]
     || <input_identifier> [ <constant_expression> :
         <constant_expression> ]

<specify_output_terminal_descriptor>
  ::= <output_identifier>
     || <output_identifier> [ <constant_expression> ]
     || <output_identifier> [ <constant_expression> :
         <constant_expression> ]

<input_identifier>
  ::= the <IDENTIFIER> of a module input or inout terminal

<output_identifier>
  ::= the <IDENTIFIER> of a module output or inout terminal.
     See "Describing Module Paths" on page 247.

<path_delay_value>
  ::= <path_delay_expression>
     || ( <path_delay_expression> )
     || <path_delay_expression>, <path_delay_expression>
     || ( <path_delay_expression>, <path_delay_expression> )
     || <path_delay_expression>, <path_delay_expression>,
         <path_delay_expression>

<path_delay_expression>
  || <path_delay_expression>, <path_delay_expression>,

```

Verilog-XL Reference Formal Syntax Definition

```

<path_delay_expression> )
  || = <path_delay_expression>, <path_delay_expression>,
      <path_delay_expression>, <path_delay_expression>,
      <path_delay_expression>, <path_delay_expression>
  || = ( <path_delay_expression>, <path_delay_expression>,
      <path_delay_expression>, <path_delay_expression>,
      <path_delay_expression>, <path_delay_expression> )
      <path_delay_expression>
  ::= <expression>

<system_timing_check>
  ::= $setup( <timing_check_event>, <timing_check_event>,
      <timing_check_limit> <,<notify_register>>? ) ;
  || = $hold( <timing_check_event>, <timing_check_event>,
      <timing_check_limit> <,<notify_register>>? ) ;
  || = $period( <controlled_timing_check_event>,
      <timing_check_limit> <,<notify_register>>? ) ;
  || = $width( <controlled_timing_check_event>,
      <timing_check_limit>
      <,<constant_expression>,<notify_register>>? ) ;
  || = $skew( <timing_check_event>, <timing_check_event>,
      <timing_check_limit> <,<notify_register>>? ) ;
  || = $recovery( <controlled_timing_check_event>,
      <timing_check_event>,
      <timing_check_limit> <,<notify_register>>? ) ;
  || = $setuphold( <timing_check_event>, <timing_check_event>,
      <timing_check_limit>, <timing_check_limit>
      <,<notify_register>>? ) ;

<timing_check_event>
  ::= <timing_check_event_control>? <specify_terminal_descriptor>
      &&& <timing_check_condition>?

<specify_terminal_descriptor>
  ::= <specify_input_terminal_descriptor>
  || = <specify_output_terminal_descriptor>

<controlled_timing_check_event>
  ::= <timing_check_event_control> <specify_terminal_descriptor>
      &&& <timing_check_condition>?

<timing_check_event_control>
  ::= posedge
  || = negedge

<timing_check_condition>
  ::= <SCALAR_EXPRESSION>
  || = ~<SCALAR_EXPRESSION>
  || = <SCALAR_EXPRESSION> == <scalar_constant>
  || = <SCALAR_EXPRESSION> === <scalar_constant>
  || = <SCALAR_EXPRESSION> != <scalar_constant>
  || = <SCALAR_EXPRESSION> !== <scalar_constant>

<SCALAR_EXPRESSION> is a one bit net or a bit select
of an expanded vector net.
  ::= <timing_check_limit>
  ::= <expression>

<scalar_constant>
  ::= 1'b0
  || = 1'b1
  || = 1'B0
  || = 1'B1

```

Verilog-XL Reference

Formal Syntax Definition

```
<notify_register>
 ::= <identifier>

<level_sensitive_path_declaration>
 ::= if (<conditional_port_expression>)
      (<specify_terminal_descriptor> <polarity_operator>?=>
       <specify_terminal_descriptor>) = <path_delay_value>
 || = if (<conditional_port_expression>)
      (<list_of_path_inputs> <polarity_operator>? *>
       <list_of_path_outputs>) = <path_delay_value>
```

Note: The following two symbols are literal symbols, not syntax description conventions:

```
*>    =>

<conditional_port_expression>
 ::= <port_reference>
 || = <UNARY_OPERATOR><port_reference>
 || = <port_reference><BINARY_OPERATOR><port_reference>

<polarity_operator>
 ::= +
 || = -

<edge_sensitive_path_declaration>
 ::=if (<expression>)? (<edge_identifier>?
   <specify_terminal_descriptor>=>
   (<specify_terminal_descriptor> <polarity_operator> ? :
    <data_source_expression>)) = <path_delay_value>
 ||=<if (<expression>)? (<edge_identifier>?
   <specify_terminal_descriptor> *>
   (<list_of_path_outputs> <polarity_operator> ? :
    <data_source_expression>)) =<path_delay_value>

<data_source_expression>
 Any expression, including constants and lists. Its width must
 be one bit or equal to the destination's width. If the
 destination is a list, the data source must be as wide as
 the sum of the bits of the members.

<edge_identifier>
 ::= posedge
 || = negedge

<edge_control_specifier>
 ::= edge [ <edge_descriptor><, <edge_descriptor>>* ]

<edge_descriptor>
 ::= 01
 || 10
 || 0x
 || x1
 || 1x
 || x0

<pulse_control_specparam>
 ::=PATHPULSE$( <r_value>, <e_value> );
 || =PATHPULSE$<module_path_source>$
   <module_path_destination>=( <r_value>, <e_value> );

<sdpd>
 ::=if(<sdpd_conditional_expression>)(<path_description>)=
   (<path_delay_value>)
 || = if(<sdpd_conditional_expression>)
   (<edge_sensitive_path_declaration>)
 || = <ifnone_path>
```


Verilog-XL Reference Formal Syntax Definition

```
<sdpd_conditional_expresssion>
  ::= <expression>

<ifnone_path>
  ::= ifnone(<path_description>)=(<path_delay_value>)
```

Expressions

```
<lvalue>
  ::= <identifier>
     See "Identifiers, Keywords, and System Names" on page 28
  || = <identifier> [ <expression> ]
  || = <identifier> [ <constant_expression>:<constant_expression> ]
  || = <concatenation>

<constant_expression>
  ::= <expression>

<mintypmax_expression>
  ::= <expression>
  || = <expression> : <expression> : <expression>

<expression>
  ::= <primary>
  || = <UNARY_OPERATOR> <primary>
  || = <expression> <BINARY_OPERATOR> <expression>
  || = <expression> <QUESTION_MARK> <expression> : <expression>
  || = <STRING>

<UNARY_OPERATOR> is one of the following tokens:
  + - ! ~ & ~& | ^ | ^ ~^

<BINARY_OPERATOR> is one of the following tokens:
  + - * / % == != === !== && || < <=
  > >= & | ^ ^~ >> <<

<QUESTION_MARK> is ? (a literal question mark).

<STRING> is text enclosed in " " and contained on one line.

<primary>
  ::= <number>
  || = <identifier>
     See "Identifiers, Keywords, and System Names" on page 28
  || = <identifier> [ <expression> ]
  || = <identifier> [ <constant_expression>:<constant_expression> ]
  || = <concatenation>
  || = <multiple_concatenation>
  || = <function_call>
  || = ( <mintypmax_expression> )

<number>
  ::= <DECIMAL_NUMBER>
  || = <UNSIGNED_NUMBER>? <BASE> <UNSIGNED_NUMBER>
  || = <DECIMAL_NUMBER>.<UNSIGNED_NUMBER>
  || = <DECIMAL_NUMBER><.<UNSIGNED_NUMBER>>? E<DECIMAL_NUMBER>
  || = <DECIMAL_NUMBER><.<UNSIGNED_NUMBER>>? e<DECIMAL_NUMBER>
```

Note: Embedded spaces are illegal in Verilog numbers, but embedded underscore characters can be used for spacing in any type of number.

Verilog-XL Reference

Formal Syntax Definition

<DECIMAL_NUMBER>
 ::= A number containing a set of any of the following characters, optionally preceded by
 + - 0 1 2 3 4 5 6 7 8 9 _

<UNSIGNED_NUMBER>
 ::= A number containing a set of any of the following characters:
 0 1 2 3 4 5 6 7 8 9 _

<NUMBER>
 Numbers can be specified in decimal, hexadecimal, octal or binary, and may optionally start with a + or -. The <BASE> token controls what number digits are legal. <BASE> must be one of **d**, **h**, **o**, or **b**, for the bases **d**ecimal, **h**exadecimal, **o**ctal, and **b**inary respectively. A number can contain any set of the following characters that is consistent with <BASE>:
 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F x X z Z ?

<BASE> is one of the following tokens:
 'b 'B 'o 'O 'd 'D 'h 'H

<concatenation>
 ::= { <expression> <,<expression>>* }

<multiple_concatenation>
 ::= { <expression> { <expression> <,<expression>>* } }

<function_call>
 ::= <name_of_function> (<expression> <,<expression>>*)
 || = <name_of_system_function> (<expression> <,<expression>>*)
 || = <name_of_system_function>

<name_of_function>
 ::= <identifier>

<name_of_system_function>
 ::= \$<SYSTEM_IDENTIFIER>
 The \$ cannot be followed by a space.

<SYSTEM_IDENTIFIER>
 ::= An <IDENTIFIER> assigned to an existing system task or function

General Syntax Definition

<identifier>
 ::= <IDENTIFIER><.<IDENTIFIER>>*
 The period cannot be preceded or followed by a space.

<IDENTIFIER>
 An identifier is any sequence of letters, digits, dollar signs (\$), and underscore (_) symbol, except that the first must be a letter or the underscore; the first character may not be a digit or \$. Upper and lower case letters are considered to be different. Identifiers may be up to 1024 characters long. Verilog-XL, Veritime and Verifault-XL do not recognize identifier characters beyond the 1024th as a significant part of the identifier. Escaped identifiers start with the backslash character (\) and may include any printable ASCII character. An escaped identifier ends with white space. The leading backslash character is not considered to be part of the identifier.

<delay>
 ::= # <number>

Verilog-XL Reference

Formal Syntax Definition

See "[Numbers](#)" on page 24

||= # <identifier>

||= # (<mintypmax_expression> <,<mintypmax_expression>>? <,<mintypmax_expression>>?)

<mintypmax_expression>

::=

See "[min/typ/max Delays](#)" on page 136

<delay_control>

::= # <number>

See "[Numbers](#)" on page 24

||= # <identifier>

||= # (<mintypmax_expression>)

See "[min/typ/max Delays](#)" on page 136

<event_control>

::= @ <identifier>

||= @ (<event_expression>)

<event_expression>

::= <expression>

See "[Expressions](#)" on page 51

||= posedge <SCALAR_EVENT_EXPRESSION>

||= negedge <SCALAR_EVENT_EXPRESSION>

||= <event_expression> or <event_expression>*

<SCALAR_EVENT_EXPRESSION> is an expression that resolves to a one bit value.

Switch-Level Modeling

rs_technology definition

<rs_technology_directive>+

rs_technology_directive

::= 'rs_technology <technology_statement>

technology statement

::= name <technology_name>

||= default

||= lowthresh <default_low_threshold>

||= highthresh <default_high_threshold>

||= resistance< type> <context> <width> <length> <switch_resistance>

||= mapres <driving_strength> <map_resistance>

||= mapcap <charging_strength> <map_capacitance>

||= cox <oxide_capacitance>

||= delta1 <length_change>

||= deltaw <width_change>

||= ldiff <lateral_diffusion>

||= cgo <gate_overlap_capacitance>

||= xa <default_diffusion_length>

||= cdiff <diffusion_capacitance>

<technology_name>

::= <IDENTIFIER>

<type> is one of the following keywords:

n_channel rn_channel p_channel rp_channel depletion rdepletion pullup pulldown

<context> is one of the following keywords:

dynamic_high dynamic_low slope

Verilog-XL Reference

Formal Syntax Definition

```
<switch_declaration>
 ::= <SWITCHTYPE><drive_strength>?<delay>?
    <instance_attributes>?<switch_instance>
    <,<switch_instance>>*</pre>

<SWITCHTYPE> is one of the following keywords  
tranif0 tranif1 rtranif0 rtranif1 pmos nmos cmos  
rpmos rnmos rcmos tran rtran pullup pulldown



```
<drive_strength>
 ::= (<constant_expression>)

<instance_attributes>
 ::= (*const real <attribute_spec_list>;*)

<attribute_spec_list>
 ::= <length_spec>? , <width_spec>?

<length_spec>
 ::= length = <constant_expression>

<width_spec>
 ::= width = <constant_expression>

<net_declaration>
 ::= trireg <charge_strength>?<expandrange>?<delay>?
 <instance_attributes>?<triereg_instance>
 <,<triereg_instance>>*</pre>

||= <NETTYPE><drive_strength>?<expand_range>?<delay>?
<instance_attributes>?<net_instance>
<,<net_instance>>*


```
<charge_strength>
 ::= (small)
    || = (medium)
    || = (large)
    || = strength (<constant_expression>)

<instance_attributes>
 ::= (*const real <attribute_spec_list>;*)

<attribute_spec_list>
 ::= <lowthresh_spec>? , <highthresh_spec>? , <capacitance_spec>?

<lowthresh_spec>
    lowthresh = <constant_expression>

<highthresh_spec>
    highthresh = <constant_expression>

<capacitance_spec>
    capacitance = <constant_expression>
```


```


```

Verilog-XL Keywords

This appendix describes words that have special meaning to Verilog-XL. Do not use them unless you intend these special meanings.

- [Keywords from Compiler Directives](#) on page 533
- [Keywords from Specify Blocks](#) on page 535
- [Keywords from Neither Compiler Directives nor Specify Blocks](#) on page 535

Keywords from Compiler Directives

<code>define</code>	<code>endmacro</code>	<code>macro</code>
<code>optimize_data</code>	<code>nooptimize_data</code>	
<code>remove_gatenames</code>	<code>noremove_gatenames</code>	
<code>remove_netnames</code>	<code>noremove_netnames</code>	
<code>accelerate</code>	<code>noaccelerate</code>	
<code>default_nettyp</code>	<code>e</code>	
<code>unconnected_drive</code>	<code>nounconnected_drive</code>	
<code>pull0</code>	<code>pull1</code>	
<code>autoexpand_vectornets</code>	<code>expand_vectornets</code>	<code>noexpand_vectornets</code>
<code>automacmods</code>	<code>noautomacmods</code>	
<code>timescale</code>	<code>s</code>	<code>ms</code>
<code>us</code>	<code>ns</code>	<code>ps</code>
<code>fs</code>		
<code>typdelays</code>	<code>mindelays</code>	<code>maxdelays</code>
<code>celldefine</code>	<code>endcelldefine</code>	

Verilog-XL Reference

Verilog-XL Keywords

nocheck_behavior	check_behavior	
suppress_faults	nosuppress_faults	
delay_mode_path	delay_mode_distribute	
delay_mode_unit	delay_mode_zero	
enable_portfaults	disable_portfaults	
default_rswitch_strength	default_switch_strength	default_trireg_strength
ifdef	else	endif
include	default_decay_time	infinite
undef		
pre_16a_paths	end_pre_16a_paths	
switch	default	
XL	resistive	
rs_technology	name	
lowthresh	highthresh	
resistance	mapres	
mapcap	slope	
dynamic_low	dynamic_high	
n_channel	rn_channel	
p_channel	rp_channel	
depletion	rdepletion	
pullup	pulldown	
strong1	strong0	
weak1	weak0	
large1	large0	
medium1	medium0	
small1	small0	
cox		

Verilog-XL Reference

Verilog-XL Keywords

deltal	deltaw
ldiff	
cgo	xa
cdiff	
uselib	
bpi	nobpi

Keywords from Specify Blocks

specify	endspecify
when	always
follow	invert
unknown	posedge
negedge	latchhigh
latchlow	pulselow
pulsehigh	ifnone
femtosecs	picosecs
nanosecs	microsecs
millisecs	seconds
picofarads	volts
megahertz	centigrade

Keywords from Neither Compiler Directives nor Specify Blocks

package	endpackage
module	macromodule
endmodule	primitive
endprimitive	parameter

Verilog-XL Reference

Verilog-XL Keywords

input	output
inout	reg
integer	time
real	event
task	endtask
function	endfunction
table	endtable
defparam	when
wire	wand
wor	tri
triand	trior
trireg	tri0
tril	supply0
supply1	
buf	not
and	nand
or	nor
xor	xnor
bufif0	bufif1
notif0	notif1
nmos	rnmos
pmos	rpmos
cmos	rcmos
pulldown	pullup
rtran	rtranif0
rtranif1	tran
tranif0	tranif1
strong0	strong1
pull0	pull1

Verilog-XL Reference

Verilog-XL Keywords

weak0	weak1
highz0	highz1
small	medium
large	
scalared	vectored
signed	
assign	deassign
force	release
initial	always
begin	end
fork	join
if	else
case	casez
casex	endcase
default	forever
repeat	while
for	wait
posedge	negedge
edge	disable
specparam	realtime
strength	
attribute	endattribute
const	use

Verilog-XL Reference

Verilog-XL Keywords

Verilog-XL and Standards' Compliance

This appendix describes the following:

- [Supported Standards](#) on page 539
- [Known Exceptions](#) on page 539

Supported Standards

Verilog-XL supports the following Standards:

- *IEEE Standard 1364-1995 - IEEE Standard Description Language Based on the Verilog™ Hardware Description Language*
- *IEEE Standard 1499-1998 - IEEE Standard Interface for Hardware Description Models of Electronic Components (Open Models Interface)*
- *OVI Standard Delay Format (SDF) Version 3.0*

Known Exceptions

VPI Routines

You cannot use the following methods for the `vpi_iterate()` and `vpi_scan()` routines:

- `vpiUse`
- `vpiAttribute` (This method is defined in the `vpi_user.h` include file, which Cadence provides in addition to *IEEE 1364* functionality.)

You cannot use the following reasons for the `vpi_register_cb()` routine:

- `cbError`
- `cbTchkViolation`

■ `cbUnresolvedSysf`

Wire with same name as a Port

If you declare a wire with same name as an existing port and the wire size declaration is different from the port size declaration, then the wire size declaration supersedes the port size declaration.

For example, consider the following code fragment:

```
module (a,...)
  input [4:6] a; //bus width is 3
  ...
  wire [0:8] a; //bus width is 9
endmodule
```

The second declaration of `a` supersedes the first declaration.

This behavior is different from the *IEEE 1364* interpretation.

Index

- - in state table [151](#)
- Symbols**
- !
 - compared to '==0' [57](#)
 - evaluation of [57](#)
 - logical negation operator [52](#)
- !=
 - logical inequality operator [52](#), [56](#)
- !==
 - case inequality operator [52](#)
 - evaluation of [56](#)
- <
 - evaluation of [56](#)
- <<
 - evaluation of [60](#)
 - left shift operator [53](#)
- <=
 - evaluation of [56](#)
 - relational less-than-or-equal operator [52](#)
- ""
 - null string [66](#)
- \$async\$and\$array [406](#)
- \$async\$and\$plane [406](#)
- \$async\$nand\$array [406](#)
- \$async\$nand\$plane [406](#)
- \$async\$nor\$array [406](#)
- \$async\$nor\$plane [406](#)
- \$async\$or\$array [406](#)
- \$async\$or\$plane [406](#)
- \$bitstoreal
 - definition of [380](#)
 - use in port connections [223](#)
- \$cleartrace [348](#)
- \$compare [375](#)
 - syntax [375](#)
- \$countdrivers [360](#)
 - syntax [360](#)
- \$deposit [369](#)
- \$disable_warnings [366](#)
 - syntax [366](#)
- \$display [333](#) to [340](#)
 - and mnemonic strength format [130](#)
 - and simulation time [346](#)
 - compared to \$monitor [341](#)
 - compared to \$write [333](#)
 - escape sequences [334](#)
 - format specifications [334](#) to [335](#)
 - size of displayed data [336](#) to [337](#)
 - syntax [333](#)
- \$dlc [403](#), [404](#)
- \$dumpall
 - definition and syntax [382](#)
 - example [502](#)
 - how to use [496](#)
- \$dumpfile [494](#)
 - definition and syntax [382](#)
- \$dumpflush [497](#)
 - definition and syntax [382](#)
- \$dumplimit [497](#)
 - definition and syntax [382](#)
- \$dumpoff
 - definition and syntax [382](#)
 - example [502](#)
 - how to use [496](#)
- \$dumpon
 - definition and syntax [382](#)
 - example [502](#)
 - how to use [496](#)
- \$dumpports [510](#)
 - character identifiers [513](#)
 - drivers [512](#)
 - output [511](#)
 - restrictions [515](#)
 - strength mapping [514](#)
- \$dumpports_close [515](#)
- \$dumpvars
 - definition and syntax [382](#)
 - example [503](#)
 - how to use [495](#) to [496](#)
- \$enable_warnings [367](#)
 - syntax [367](#)
- \$eventcond [279](#)
- \$fclose [343](#) to [345](#)
 - syntax [343](#)
- \$fdisplay [343](#) to [345](#)
 - syntax [343](#)
- \$finish [347](#)

Verilog-XL Reference

- syntax [347](#)
- \$fmonitor [343 to 345](#)
 - syntax [343](#)
- \$fopen [343 to 345](#)
 - syntax [343](#)
- \$fstrobe [343 to 345](#)
 - syntax [343](#)
- \$fwrite [343 to 345](#)
 - syntax [343](#)
- \$getpattern [370, 371](#)
- \$history [355](#)
 - syntax [355](#)
- \$hold [295](#)
- \$incpattern_read [373](#)
 - syntax [373](#)
- \$incpattern_write [372](#)
 - syntax [372](#)
- \$incsave [352](#)
- \$input [356](#)
 - and asynchronous interrupts [356](#)
 - and reading key files [356](#)
 - and reading previous command file [356](#)
 - syntax [356](#)
- \$itor [380](#)
- \$keepcommands [363](#)
- \$key [357](#)
 - syntax [357](#)
- \$list
 - and decompiling macro modules [218](#)
 - for debugging [175](#)
 - syntax [363](#)
- \$list_forces [364](#)
- \$listcounts [363, 384](#)
 - syntax and example [469](#)
- \$log
 - syntax [357](#)
- \$monitor
 - and fixed width format [337](#)
 - and simulation time [346](#)
 - compared to \$display [341](#)
 - syntax [341](#)
 - turn off [342](#)
- \$monitoroff [341](#)
 - syntax [341](#)
- \$monitoron [341](#)
 - syntax [341](#)
- \$no_show_cancelled_e [265](#)
- \$nochange [297](#)
- \$noeventcond [279](#)
- \$nokeepcommands [363](#)
- \$nokey [357](#)
 - syntax [357](#)
- \$nolog
 - syntax [357](#)
- \$period [298](#)
- \$post_int_delay [342](#)
- \$prnttimescale [446](#)
 - definition and syntax [381](#)
- \$pulsestyle_ondetect [260](#)
- \$pulsestyle_onevent [260](#)
- \$readmemb [368 to 369](#)
 - and \$getpattern [371](#)
 - and \$sreadmemb [381](#)
 - and loading logic array personality [407](#)
 - syntax [368](#)
- \$readmemh [368 to 369](#)
 - and \$getpattern [371](#)
 - and \$sreadmemh [381](#)
 - and loading logic array personality [407](#)
 - syntax [368](#)
- \$realtime
 - and triggering monitoring [346](#)
 - definition and example [444](#)
 - definition of [380](#)
 - how to use [346](#)
- \$realtobits
 - definition of [380](#)
 - use in port connections [223](#)
- \$recovery [299](#)
- \$recrem [301](#)
- \$removal [304](#)
- \$reportfile [383, 468](#)
- \$reportprofile
 - definition and syntax [383, 384](#)
 - syntax and example [468](#)
- \$reset
 - description, syntax, and examples [385 to 390](#)
- \$reset_count [390](#)
- \$reset_value [391 to 392](#)
- \$restart
 - description, syntax, and examples [352](#)
- \$rtoi [380](#)
- \$save
 - description, syntax, and examples [352](#)
 - syntax [352](#)
- \$scale
 - definition of [380](#)
 - description, syntax, and examples [444 to 445](#)
- \$scope
 - syntax [358](#)

Verilog-XL Reference

- [\\$sdf_annotate](#) [393](#)
- [\\$settrace](#)
 - and tracing statements inside macro modules [218](#)
 - syntax [348](#)
- [\\$setup](#) [305](#)
- [\\$setuphold](#) [307](#)
- [\\$show_cancelled_e](#) [265](#)
- [\\$showallinstances](#)
 - syntax [358](#)
- [\\$showexpandednets](#)
 - syntax [359](#)
- [\\$showmodes](#) [462](#)
- [\\$showportsnotcollapsed](#)
 - syntax [360](#)
- [\\$showscopes](#)
 - syntax [358](#)
- [\\$showvariables](#)
 - syntax [358](#)
- [\\$showvars](#)
 - syntax [358](#)
- [\\$skew](#) [310](#)
- [\\$sreadmemb](#)
 - definition and syntax [381](#)
- [\\$sreadmemh](#)
 - definition, syntax, and example [381](#)
- [\\$startprofile](#) [383](#)
 - how it works [465](#)
 - syntax and example [467](#)
- [\\$stime](#) [346](#)
 - and triggering monitoring [346](#)
 - as parameters to [\\$display](#) and [\\$monitor](#) [346](#)
 - syntax [346](#)
- [\\$stop](#) [347](#)
- [\\$stopprofile](#) [384](#)
 - definition and syntax [384](#)
 - syntax and example [469](#)
- [\\$strobe](#) [341](#)
 - syntax [340](#)
- [\\$strobe_compare](#) [376](#)
 - syntax [376](#)
- [\\$sync\\$and\\$array](#) [406](#)
- [\\$sync\\$and\\$plane](#) [406](#)
- [\\$sync\\$and\\$array](#) [406](#)
- [\\$sync\\$and\\$plane](#) [406](#)
- [\\$sync\\$nor\\$array](#) [406](#)
- [\\$sync\\$nor\\$plane](#) [406](#)
- [\\$sync\\$or\\$array](#) [406](#)
- [\\$sync\\$or\\$plane](#) [406](#)
- [\\$time](#)
 - and triggering event controls [346](#)
 - and triggering monitoring [346](#)
 - as parameters to [\\$display](#) and [\\$monitor](#) [346](#)
 - data type used with [47](#)
 - definition of [380](#)
 - description and examples [443](#)
- [\\$timeformat](#) [446 to 449](#)
 - definition and syntax [381](#)
- [\\$width](#) [311](#)
 - use of threshold argument [312](#)
- [\\$write](#) [333 to 340](#)
 - compared to [\\$display](#) [333](#)
 - escape sequences [334](#)
 - format specifications [334 to 335](#)
 - size of displayed data [336 to 337](#)
 - syntax [333](#)
- [%](#)
 - in format specifications [333](#)
 - in format specifications for real numbers [336](#)
 - modulus operator [52](#)
- [&](#)
 - bit-wise AND operator [52](#)
 - reduction AND operator [52](#)
- [&&](#)
 - evaluation of [57](#)
 - logical AND operator [52](#)
- [*](#)
 - arithmetic multiplication operator [52](#)
- [-d](#)
 - effect on decompilation and tracing [218](#)
- [+accu_path_delay](#) [279](#)
- [+annotate_any_time](#) [393](#)
- [+autonaming](#) [233](#)
- [+delay_mode_distributed](#) [457](#)
- [+delay_mode_path](#) [457](#)
- [+delay_mode_unit](#) [457](#)
- [+delay_mode_zero](#) [457](#)
- [-i](#)
 - and reading key files [356](#)
 - for simulation recovery [357](#)
 - to read command input file [356](#)
- [-k](#)
 - for changing key file name [357](#)
- [-l](#)
 - to change log file name [357](#)
- [+listcounts](#)
 - how to use [363](#)
- [+maxdelays](#) [67](#)
- [+mindelays](#) [67](#)

Verilog-XL Reference

- +multisource_int_delays [433 to 435](#)
- +neg_tchk [304, 309](#)
- +no_cancelled_e_msg [262](#)
- +no_pulse_int_backanno [436](#)
- +no_show_cancelled_e [261, 262, 265](#)
- +nosdfwarn [395](#)
- +notimingchecks [289](#)
- +pathpulse
 - and overriding global pulse control [259](#)
- +pre_16a_paths [274](#)
- +pulse_e/n [436](#)
- +pulse_e_style_ondetect [261](#)
- +pulse_e_style_onevent [261](#)
- +pulse_int_e/n [436](#)
- +pulse_int_r/m [436](#)
- +pulse_r/m [436](#)
- r
 - for command-line restart [355](#)
- +sdf_error_info [395](#)
- +sdf_verbose [395](#)
- +show_cancelled_e [261, 265](#)
- +transport_int_delays [267, 433 to 435](#)
 - monitoring interconnect delay signal values [342](#)
- +transport_path_delays [244, 267](#)
- +typdelays [67](#)
- w
 - use with non-downward path names [236](#)
- +x_transport_pessimism [287, 437](#)
- ”
 - commas in null expressions [333](#)
- /
 - arithmetic division operator [52](#)
- <
 - relational less-than operator [52](#)
- =
 - in assignment statement [71](#)
- ==
 - logical equality operator [52](#)
- ===
 - case equality operator [52](#)
- >
 - evaluation of [56](#)
 - relational greater-than operator [52](#)
- >=
 - evaluation of [56](#)
 - relational greater-than-or-equal operator [52](#)
- >>
 - evaluation of [60](#)
- ?
 - equivalent to z in literal number values [25, 179](#)
 - in state table [149, 151](#)
- ?:
 - conditional operator [53](#)
- @
 - for addressing memory [368](#)
 - for event control [184](#)
- \
 - for escape sequences in strings [333](#)
- ^
 - bit-wise exclusive OR operator [52](#)
 - for hierarchical names in macro module instances [231](#)
 - reduction XOR operator [53](#)
- ^~
 - bit-wise equivalence operator [52](#)
 - reduction XNOR operator [53](#)
- |
 - bit-wise inclusive OR operator [52](#)
 - reduction OR operator [53](#)
- ||
 - logical OR operator [52](#)
 - evaluation of [57](#)
- ~
 - bit-wise negation operator [52](#)
- ~&
 - reduction NAND operator [52](#)
- ~^
 - bit-wise equivalence operator [52](#)
 - reduction XNOR operator [53](#)
- ~|
 - reduction NOR operator [53](#)
- ‘default_nettype
 - syntax [114](#)
- ‘delay_mode_distributed
 - how to use [456](#)
- ‘delay_mode_path
 - how to use [456](#)
- ‘delay_mode_unit
 - how to use [457](#)
- ‘delay_mode_zero
 - how to use [457](#)
- ‘end_pre_16a_paths [274](#)
- ‘noremove_gatenames
 - how to use [140](#)
- ‘noremove_netnames
 - how to use [140](#)
- ‘pre_16a_paths [274](#)
- ‘remove_gatenames

- how to use [140](#)
- 'remove_netnames
 - how to use [140](#)
- 'timescale [440](#)
 - effect on performance [442](#)
 - usage rules [440](#), [442](#)

Numerics

- 0
 - for minimizing bit lengths of expressions [336](#)
 - logic 0 [31](#), [338](#)
- 01 transition [151](#)
- 1
 - logic 1 [32](#), [338](#)
- 12436
 - Section
 - 4.12 [127](#)

A

- accelerated continuous assignments [80](#)
 - compilation speed [91](#)
 - different results [92](#) to [93](#)
 - effects [87](#) to ??
 - memory usage [91](#)
 - restrictions [80](#)
 - delay expressions [86](#)
 - left-hand side [81](#) to [82](#)
 - right-hand side [83](#) to [86](#)
 - simulation speed [88](#) to [91](#)
- acceleration
 - and module path destinations [248](#)
 - and specify path declarations [239](#)
- accu_path algorithm [279](#)
 - applying [280](#)
 - invoking [279](#)
 - limitations of [285](#)
- addressing memory [368](#) to [369](#)
- always
 - and activity flow [163](#), [164](#)
 - as structured procedure [164](#)
 - syntax [165](#)
- ambiguous strength [117](#)
- and gate [105](#) to ??
- annotating
 - vector bits [399](#)
- are [239](#)

- arithmetic operators
 - [54](#)
 - % [54](#)
 - * [54](#)
 - + [54](#)
 - / [54](#)
 - and unknown logic values [54](#)
- arithmetic shift left operators
 - <<< [60](#)
- arithmetic shift right operators
 - >>> [60](#)
- array of instances [99](#), [101](#)
 - rules [102](#)
 - syntax [99](#)
 - UDPs [156](#)
- arrays
 - element [46](#)
 - format [407](#)
 - index [46](#)
 - no multiple dimension [46](#)
 - of integers [47](#)
 - of time variables [47](#)
 - word [46](#)
- assign keyword [94](#) to [95](#)
- assignment [71](#) to [96](#)
 - continuous [72](#) to [79](#)
 - using functions in [78](#)
 - versus procedural [166](#)
 - left hand side [71](#)
 - of delays to module paths [247](#) to [253](#)
 - and driving wired logic outputs [242](#)
 - syntax [247](#)
 - procedural [166](#)
 - versus continuous [166](#)
 - right hand side [71](#)
- asynchronous arrays [406](#)
- asynchronous control signal
 - detecting changes in [299](#), [304](#)
- automatic naming [233](#) to [234](#)
 - +autonaming option [233](#)
 - for gates [101](#)
 - for user-defined primitives [156](#)

B

- b
 - binary number format [24](#)
- base format
 - binary [24](#)
 - decimal [24](#)

Verilog-XL Reference

- hexadecimal [24](#)
- octal [24](#)
- begin-end block statement
 - description, syntax, and examples [189 to 190](#)
 - used with conditional statement [174](#)
- Behavior Profiler [465 to 491](#)
 - data table [471 to 477](#)
 - by module instance [474, 475, 476](#)
 - by statement [471 to 474](#)
 - example [484 to 491](#)
 - system tasks [467 to 471](#)
 - \$listcounts [469](#)
 - \$reportprofile [468](#)
 - \$startprofile [467](#)
 - \$stopprofile [469](#)
- behavioral modeling [163 to 194](#)
 - block statements [189 to 193](#)
 - case statements [176](#)
 - conditional statements [174](#)
 - looping statements [179 to 182](#)
 - multi-way decision statements [175](#)
 - overview of [163](#)
 - procedural assignments [166 to 173](#)
 - procedural timing controls [182 to 189](#)
 - structured procedures [164 to 194](#)
- bidirectional pass gate [110](#)
- binary display format [24](#)
 - and high impedance state [337](#)
 - and unknown logic value [337](#)
- binary operators [53](#)
 - &&
 - evaluation of [57](#)
 - { } [62](#)
 - ||
 - evaluation of [57](#)
 - logic table for [58](#)
- bit annotation [399](#)
- bit-select
 - and vector ports [220](#)
 - of vector net or register [63](#)
 - out of bounds [63, 64](#)
 - references of real numbers [49](#)
- bit-wise operators
 - AND [52](#)
 - compared to logical operators [58](#)
 - equivalence [52](#)
 - exclusive OR [52](#)
 - inclusive OR [52](#)
 - negation [52](#)
- blank module terminal [212](#)

- block statement [189 to 193](#)
 - definition [189](#)
 - fork-join [189](#)
 - naming of [192](#)
 - parallel [189](#)
 - sequential [189 to 190](#)
 - start and finish times [192 to 193](#)
 - timing for embedded blocks [192](#)
- blocking procedural assignment [167](#)
 - processing assignments [173](#)
 - syntax [167](#)
- buf gate [106 to ??](#)
- bufif gate [107](#)

C

- capacitive networks [42 to 45](#)
- case equality operator [52](#)
- case inequality operator [52](#)
- case statement [176](#)
 - compared to if-else-if statement [177](#)
 - with don't-care [178](#)
- casex [178](#)
- casez [178](#)
- cells [209](#)
- changing default base in formatted output
 - system tasks [345](#)
- charge decay
 - description and examples [137 to 140](#)
- charge storage strength
 - strength levels for [116](#)
- checkpoints [352](#)
- cmos [112](#)
- cmos gate [112](#)
- collapsing nets [225](#)
- collapsing ports [223 to 227](#)
 - chart of resulting net types [226](#)
 - rules [224 to 226, 235](#)
 - that connect nets of different types [225](#)
- combinational UDPs
 - compared to level-sensitive sequential [150](#)
 - description and examples [148 to 150](#)
 - input and output fields in state table [147](#)
- combined signal strengths [116](#)
- combined signal values [116](#)
- command
 - history [355](#)
 - input files [356](#)
- command line

Verilog-XL Reference

- plus options
 - +annotate_any_time [393](#)
 - +listcounts [363](#)
 - +neg_tcheck [304](#)
 - +neg_tchk [309](#)
 - +nosdfwarn [395](#)
 - +pathpulse
 - and overriding global pulse control [259](#)
 - +pre_16a_paths [274](#)
 - +pulse_e/n [436](#)
 - +pulse_int_e/n [436](#)
 - +pulse_int_r/m [436](#)
 - +pulse_r/m [436](#)
 - +sdf_error_info [395](#)
 - +sdf_verbose [395](#)
 - +x_transport_pessimism [287](#)
 - command-line restart [355](#)
 - comments [24](#)
 - compare
 - string operation [65](#)
 - compilation
 - user-defined primitives [157](#)
 - concatenation
 - and macro module instances [217](#)
 - and repetition multiplier [62](#)
 - and unsized numbers [62](#)
 - of names [228](#)
 - of operands [63](#)
 - of terms in synchronous and asynchronous system calls [406](#)
 - string operation [65](#)
 - concurrency
 - of activity flow [163](#)
 - of procedures [201](#)
 - condition
 - deterministic [294](#)
 - non-deterministic [294](#)
 - conditional operator [61](#)
 - and ambiguous results [61](#)
 - modeling tri-state output busses [62](#)
 - syntax [61](#)
 - conditional statement [174](#)
 - syntax [174](#)
 - conditioned event
 - constraints [294](#)
 - conditioning signals
 - multiple [294](#)
 - conflicts [40, 41](#)
 - connecting ports
 - between modules [225](#)
 - by name [221 to 223](#)
 - by position with ordered list [220](#)
 - in macro modules [227](#)
 - rules [224 to 226](#)
 - connection
 - difference between full and parallel [250](#)
 - full [250](#)
 - parallel [249](#)
 - consistency
 - in user-defined primitive state tables [157](#)
 - constant expression [51](#)
 - continuous assignment [72 to 79](#)
 - accelerated [80 to 93](#)
 - and \$getpattern [371](#)
 - and connecting ports [224](#)
 - and driving strength [115, 339](#)
 - and net variables [166](#)
 - and supply nets [45](#)
 - and wire nets [40](#)
 - examples [73 to 74](#)
 - explicit declaration [73](#)
 - implicit declaration [73](#)
 - syntax [72](#)
 - using functions in [78](#)
 - versus procedural assignment [79](#)
 - continuous monitoring [341](#)
 - copy
 - string operation [65](#)
 - counting number of drivers [360](#)
- ## D
- d
 - decimal number format [24](#)
 - data structures [231](#)
 - data types [31 to 50](#)
 - deassign keyword [94 to 95](#)
 - decay
 - charge [137](#)
 - decimal display format [24](#)
 - and high impedance state [337](#)
 - and unknown logic value [337](#)
 - compatibility with \$monitor [337](#)
 - decimal notation [48](#)
 - declaring
 - events [185](#)
 - multiple module paths in a single statement [250](#)
 - decompilation [363 to 365](#)

Verilog-XL Reference

- default
 - base in formatted output [345](#)
 - in case statement [177](#)
 - in if-else-if statements [176](#)
 - word size [25](#)
- default delay mode [456](#)
- defparam [214](#)
 - compared to module parameter assignment [215](#)
- delay
 - calculating for high impedance (z) transitions [132](#)
 - calculating for unknown logic value (x) transitions [132](#), [133](#)
 - control
 - and intra-assignment delay [186](#)
 - definition of [182](#)
 - syntax and examples [183](#)
 - zero-delay [183](#)
 - distributed [240](#) to [244](#)
 - expanding vector bits for full connection path [400](#)
 - expanding vector bits for parallel connection path [400](#)
 - expanding vector bits for path [400](#)
 - fall
 - definition of [132](#)
 - falling [136](#)
 - for continuous assignment [74](#)
 - gate [132](#) to [137](#)
 - inertial [77](#)
 - interconnect [421](#)
 - minimum:typical:maximum values [136](#) to [137](#)
 - mixing distributed and module path [241](#)
 - module path [239](#)
 - net [132](#) to [135](#)
 - propagation [101](#), [132](#)
 - rise
 - assigning values for [136](#)
 - definition of [132](#)
 - specification [101](#)
 - specify one value [132](#)
 - specify three values [133](#)
 - specify two values [132](#)
 - syntax for delay control [183](#)
 - triereg charge decay
 - description and examples [138](#) to [140](#)
 - turn-off [136](#)
- delay calculator [403](#)
- delay mode selection [453](#)
 - and macro module expansion [462](#)
 - and timescales [458](#) to [459](#)
 - command line plus options [457](#)
 - compiler directives [456](#)
 - decompiling with delay modes [462](#)
 - default delay mode [456](#)
 - distributed delay mode [455](#)
 - overriding delay values [459](#)
 - path delay mode [455](#)
 - precedence [457](#)
 - reasons to select a delay mode [456](#)
 - the \$showmodes system task [462](#)
 - the acc_fetch_delay_mode access routine [462](#)
 - the parameter attribute mechanism [460](#) to [461](#)
 - unit delay mode [454](#)
 - zero delay mode [454](#)
- delays
 - path [239](#)
- diagnostic messages
 - from \$stop and \$finish [347](#)
- disable
 - and turning off monitoring tasks [342](#)
 - named blocks [205](#) to [208](#)
 - syntax [205](#)
 - tasks [205](#) to [208](#)
 - use of [205](#)
- disable timing check [289](#)
- disabling
 - warnings [366](#) to [367](#)
- displaying information [333](#) to [340](#)
- displaying the delay mode [362](#)
- distributed delay mode [455](#)
- dominating net [225](#)
- don't-care bits
 - in case statements [179](#)
- don't-care condition
 - in state table [149](#)
- drive strength specification [100](#)
- drivers
 - for \$dumpports [512](#)
- driving strength [115](#)
 - compared to charge storage strength [339](#)
 - keywords [77](#)
- dynamic file selection [333](#)

Verilog-XL Reference

E

- edge control specifiers [291](#)
- edge descriptors [49](#)
- edge-sensitive UDPs [150](#) to [151](#)
 - compared to level-sensitive UDPs [150](#)
- element
 - of array [46](#)
- embedding modules
 - and hierarchy [209](#)
 - by instantiating [211](#)
- enable [186](#)
- enabling
 - tasks [198](#) to [199](#)
 - when already active [201](#)
 - warnings [367](#)
- endmodule keyword [210](#)
- endprimitive keyword [145](#)
- endtable keyword [146](#)
- equality operators [56](#)
 - != [56](#)
 - !== [56](#)
 - == [56](#)
 - === [56](#)
 - and ambiguous results [57](#)
 - and operands of different sizes [56](#)
 - precedence [56](#)
- escape sequences
 - for displaying special characters [334](#)
 - inserting in string [333](#)
- espresso format [408](#)
- event
 - control
 - definition of [183](#)
 - description, syntax, and examples [184](#)
 - declaration syntax [185](#)
 - explicit [183](#)
 - expression [182](#)
 - implicit [183](#)
 - in timing checks [295](#)
 - level sensitive control [186](#)
 - named [184](#) to [185](#)
 - OR construct [185](#)
 - syntax for event control [184](#)
 - syntax for triggering statement [185](#)
- event control
 - repeat [188](#) to [189](#)
- event-driven simulation [160](#)
- examples
 - "joining" events [193](#)
 - \$hold timing check [296](#)
 - \$monitor [339](#)
 - \$printtimescale system task [446](#)
 - \$realtime system function [444](#)
 - \$recovery timing check [300](#)
 - \$recrem [303](#)
 - \$removal timing check [305](#)
 - \$scale system function [445](#)
 - \$setup timing check [306](#)
 - \$setuphold [309](#)
 - \$skew timing check [311](#)
 - \$strobe [341](#)
 - \$time system function [443](#)
 - \$timeformat system task [448](#)
 - \$width calls, legal and illegal [312](#)
 - \$width timing check [312](#)
 - %t format specification [448](#)
 - 'timescale compiler directive [441](#)
 - adder
 - using zero-delay buf gates [255](#)
 - AND-OR gate as user-defined primitive [161](#)
 - AND-OR PLA [410](#)
 - array with logic equations [407](#)
 - asynchronous system call [406](#)
 - begin-end block [190](#)
 - Behavior Profiler [484](#) to [491](#)
 - behavioral model [164](#)
 - bit-select [63](#)
 - case statement [178](#)
 - casex [179](#)
 - casez in instruction decoder [179](#)
 - changing default base in formatted output [345](#)
 - combinational primitive [149](#)
 - combinational UDP [149](#)
 - command history [355](#)
 - command line
 - restart [355](#)
 - conditioned events [294](#)
 - connecting ports by name [222](#)
 - declaring memory and registers in one statement [46](#)
 - decompiling a macro module with \$list [218](#) to ??
 - defparam [214](#)
 - delay control [183](#)
 - delay mode selection [461](#) to [462](#)
 - disable statement [206](#) to [208](#)
 - disabling all timing checks [289](#)

Verilog-XL Reference

- disabling the \$incpattern_write task [372](#)
- displaying unknown logic value in different radix formats [338](#)
- edge control specifiers [290](#), [291](#)
- edge-sensitive UDP [151](#)
- escaped identifiers [28](#)
- establishing simulation time with display output [346](#)
- event OR construct [185](#)
- factorial function [203](#)
- for loop [181](#)
- for loop in multiplier [182](#)
- fork-join block [191](#)
- function definition [202](#)
- hierarchical name
 - in macro module instance [232](#)
 - in module instance [232](#)
 - referencing [230](#)
- hierarchical path names [229](#)
- identifiers [28](#)
- incremental save and restart [354](#)
- infinite zero-delay loop [165](#)
- intra-assignment timing controls [187](#)
- invoking \$compare [376](#)
- J-K flip-flop [154](#)
- latch [150](#)
- latch module with tri-state outputs [135](#) to [136](#)
- level-sensitive sequential primitive [150](#)
- loading memories from text files [369](#)
- logic array personality declaration [407](#)
- macro module specification [217](#)
- memory addressing [64](#)
- memory declaration [46](#)
- minimum:typical:maximum values [67](#), [136](#)
- mixing level- and edge-sensitive user-defined primitives [154](#)
- module instance [211](#), [213](#)
- module instance parameter value assignment [215](#)
- module parameter declaration [50](#)
- module path declarations with polarity [254](#)
- multiplexer [149](#)
- NAND plane system [407](#)
- NOR plane system [407](#)
- notifiers [292](#)
- notifiers in edge sensitive UDP [292](#) to [293](#)
- overriding module parameter values [214](#)
- PAL16R4 [416](#) to [420](#)
- PAL16R8 [411](#) to [415](#)
- part-select [64](#)
- passing module parameters to tasks [200](#)
- PATHPULSES\$ [260](#)
- PLA module [407](#)
- PLA system tasks [408](#) to [409](#)
- port declarations [220](#)
- problem in string value padding [66](#)
- processing stimulus patterns with \$getpattern [371](#)
- race condition [187](#)
- real numbers [49](#)
- real numbers in port connections [223](#)
- reducing pessimism in a user-defined J-K flip-flop primitive [158](#)
- reducing pessimism in a user-defined latch UDP [158](#)
- register and net declarations [37](#), [38](#)
- repeat loop in multiplier [180](#)
- SDPDs [272](#)
- sized constant numbers [26](#)
- source description containing VCD tasks [498](#)
- specify block [238](#)
- specify parameters [238](#)
- specparams [238](#)
- strength outputs [339](#)
- strings [26](#)
- synchronous PLA [409](#)
- synchronous system call [406](#)
- template of a data structure [231](#)
- timescales [449](#) to [452](#)
- time-sequenced waveform [190](#), [191](#)
- traffic light sequencer [193](#)
- traffic light sequencer using tasks [200](#)
- tri-state output bus [62](#)
- turn off monitoring [342](#)
- two sequential events working in parallel [193](#)
- two-channel multiplexer as user-defined primitive [161](#)
- use of multi-channel descriptors [344](#)
- user-defined primitive instance [156](#)
- using \$realtobits and \$bitstoreal in port connections [380](#)
- value change dump file format [507](#)
- variable delays for synchronizing

Verilog-XL Reference

- clock [194](#)
- vector XOR [77](#)
- waveform [165](#)
- while loop in counter [181](#)
- writing formatted output to files [344](#)
- zero-delay control [183](#)
- exit simulator [347](#)
- expansion
 - of macro modules
 - definition of [216](#)
 - when expanded [228](#)
 - of vector nets
 - specified in port definition [220](#)
- explicit event [183](#)
- expressions
 - bit lengths [67](#)
 - constant [51](#)
 - self-determined [68](#)

F

- fall delay
 - assigning values for [136](#)
 - definition of [132](#)
- files
 - output to [343](#) to [345](#)
- filtering pulses
 - and cancelled schedules [262](#)
- filtering pulses on module path delays [263](#)
- finish time
 - in parallel block statements [192](#)
 - in sequential block statements [192](#)
- for loop
 - syntax [180](#)
- force keyword [93](#) to [96](#)
 - precedence over assign [95](#)
- forever loop
 - syntax [180](#)
- fork-join block statement [189](#)
- format specifications [334](#) to [335](#)
 - ASCII character [335](#)
 - b or B [334](#)
 - binary [334](#)
 - c or C [335](#)
 - d or D [334](#)
 - decimal [334](#)
 - h or H [334](#)
 - hexadecimal [334](#)
 - hierarchical name [335](#)
 - m or M [335](#)

- net signal strength [338](#) to [340](#)
 - escape sequences for [335](#)
- o or O [334](#)
- octal [334](#)
- s or S [335](#)
- string [335](#), [340](#)
- t or T [335](#)
- time format [335](#)
- timescales [335](#)
- v or V [335](#)
- formats
 - array [407](#)
 - of logic array personality [407](#) to [409](#)
 - plane [408](#)
- formatted output system tasks [345](#)
- full connection [250](#)
 - expanding vector bits [400](#)
- function
 - syntax [201](#)
- functions [201](#) to [204](#)
 - and scope [235](#)
 - as structured procedures [164](#)
 - definition [164](#)
 - in continuous assignments [78](#)
 - purpose [197](#)
 - returning a value [202](#)
 - rules [203](#)
 - syntax [198](#)
 - syntax for function call [202](#)

G

- g [201](#)
- gate level modeling [97](#) to [141](#)
 - logic gate syntax [98](#) to [102](#)
- gate type specification [100](#)
- gates
 - and [105](#) to ??
 - bidirectional pass [110](#)
 - buf [106](#) to ??
 - bufif [107](#)
 - cmos [112](#)
 - compared to continuous assignments [98](#)
 - connection list [101](#), [102](#)
 - delay [132](#) to [137](#)
 - keywords for types [100](#)
 - MOS [108](#) to ??
 - nand [105](#) to ??
 - nor [105](#) to ??

Verilog-XL Reference

- not [106](#) to ??
- notif [107](#)
- notif0 [107](#)
- notif1 [107](#)
- or [105](#) to ??
- pulldown [113](#)
- pullup [113](#)
- removal of names [140](#) to [141](#)
- syntax [98](#) to [102](#)
- terminal list [101](#), [102](#)
- xnor [105](#) to ??
- xor [105](#) to ??
- ground [45](#)

- H**
- H
 - logic 1 or high impedance state in strength format [338](#)
- h
 - hexadecimal number format [24](#)
 - hexadecimal display format [24](#)
 - and high impedance state [337](#)
 - and unknown logic value [337](#)
- Hi
 - high impedance in strength format [339](#)
- hierarchy
 - display of [358](#)
 - effect of macro modules on path names [231](#)
 - level [228](#)
 - name referencing [228](#) to [236](#)
 - escape sequence for [335](#)
 - of modules
 - assigning [358](#)
 - definition of [209](#)
 - path names for defining abstract data structures [231](#)
 - running backannotation [393](#)
 - scope [228](#)
 - scope rules for naming [234](#) to [236](#)
 - structures [209](#) to [236](#)
 - top level names [228](#)
 - traversal of [358](#)
- high impedance state [114](#)
 - and numbers [25](#)
 - and trireg nets [42](#)
 - display formats [337](#) to [338](#)
 - effect in different bases [25](#)
 - strength display format [339](#)
 - symbolic representation [32](#)
- highz0 [100](#)
- highz1 [100](#)
- history of commands [355](#)

- I**
- identifiers [28](#)
 - definition [28](#)
- if-else statement
 - interactive mode [175](#)
 - omitting else from nested if [174](#)
 - purpose [174](#)
- if-else-if statement
 - compared to case statement [177](#)
 - syntax [175](#)
- implicit
 - declarations
 - for nets [113](#)
 - for variables [39](#)
 - event [183](#)
- incremental pattern files [372](#) to [379](#)
- incremental restart [354](#)
- index
 - of array [46](#)
 - of memory [47](#)
- inertial delay [77](#), [244](#)
- initial [165](#)
 - and activity flow [163](#), [164](#)
 - for specifying waveforms [165](#)
 - syntax [165](#)
- initial statements
 - in UDPs [151](#) to [154](#)
- inout
 - port declaration [220](#)
- input
 - port declaration [220](#)
- input file option
 - and the key file [357](#)
- instance
 - array [101](#)
 - rules [102](#)
 - UDPs [156](#)
- instantiation
 - macro module [217](#)
 - of modules
 - and hierarchy [209](#)
- integers [47](#) to [48](#)
 - division [54](#)
- interactive

Verilog-XL Reference

- mode [347](#)
- source listing [363 to 365](#)
- interconnect delay [421](#)
 - monitoring a signal value [342](#)
- inter-module port connection [225](#)
- intra-assignment timing
 - controls [186 to 189](#)

K

- key file
 - description, syntax, and example [357](#)
- keywords
 - gatetype list [100](#)
 - negedge [291](#)
 - posedge [291](#)
- keywords list [533](#)

L

- L
 - logic 0 or high impedance state in strength format [338](#)
- La
 - large capacitor in strength format [339](#)
- left shift operator [60](#)
- legal module paths
 - one output driver [243](#)
- level-sensitive
 - event control [186](#)
 - sequential UDPs [150](#)
 - versus combinational UDP [150](#)
- level-sensitive UDPs
 - compared to edge-sensitive UDPs [150](#)
- lexical conventions [?? to 30](#)
- lexical token
 - comment [24](#)
 - definition of [23](#)
 - operator [23 to 24](#)
 - white space [24](#)
- limitations
 - saving simulation data [355](#)
- list of formatted output system tasks [345](#)
- list of keywords [533](#)
- loading memories from text
 - files [368 to 369](#)
- log file [356](#)
- logic array personality [407 to 409](#)
 - declaration [407](#)

- formats [407 to 409](#)
- loading [407](#)
- logic gates
 - and [105 to ??](#)
 - bidirectional pass [110](#)
 - buf [106 to ??](#)
 - bufif [107](#)
 - cmos [112](#)
 - compared to continuous assignments [98](#)
 - delay [132 to 137](#)
 - MOS [108 to ??](#)
 - nand [105 to ??](#)
 - nor [105 to ??](#)
 - not [106 to ??](#)
 - notif [107](#)
 - or [105 to ??](#)
 - pulldown [113](#)
 - pullup [113](#)
 - syntax [98](#)
 - xnor [105 to ??](#)
 - xor [105 to ??](#)
- logic one [32](#)
- logic planes [406](#)
- logic strength modeling [114 to 132](#)
- logic zero [31](#)
- logical operators
 - ! [57](#)
 - && [57](#)
 - || [57](#)
 - AND [52](#)
 - and ambiguous results [57](#)
 - and unknown logic value [57](#)
 - compared to bit-wise operators [58](#)
 - equality [52](#)
 - inequality [52](#)
 - negation [52](#)
 - OR [52](#)
 - precedence [57](#)
- looping statements [179 to 182](#)
 - for loop [181](#)
 - forever loop [180](#)
 - repeat loop [180](#)
 - while loop [181](#)
- lsb (least significant bit) [38](#)

M

- macro module [216 to 217](#)
 - and hierarchical names [231](#)

Verilog-XL Reference

- and module parameters [217](#)
 - and specify blocks [239](#)
 - expansion
 - and delay modes [462](#)
 - definition of [216](#)
 - effect of net type combinations
 - on [228](#)
 - instances containing part-selects or concatenations [217](#)
 - instantiation [217](#)
 - macromodule keyword [217](#)
 - port connections in [227](#)
 - syntax [217](#)
- macros
- text substitution [29](#)
- Me
- medium capacitor in strength
 - format [339](#)
- memory [46 to 47](#)
- assigning values to [47](#)
 - declaration syntax [46](#)
 - index [47](#)
 - real number memories [49](#)
 - reducing virtual storage
 - requirements [140](#)
 - using temporary registers for bit- and part-selects [64](#)
- minimum:typical:maximum values
- delay [136 to 137](#)
 - format [66 to 67](#)
- minus sign(-)
- arithmetic subtraction operator [52](#)
- MIPDs [421 to 430](#)
- application [431](#)
 - hierarchical effects [424](#)
 - how they work [423](#)
 - on inputs and outputs only [424](#)
 - specifying with PLI [428](#)
 - unidirectionality [426](#)
- MITD [421](#)
- and pulse handling [436](#)
 - creating [433](#)
 - definition [432](#)
- mnemonic strength notation [130](#)
- modeling
- asynchronous clear/preset on an edge-triggered D flip-flop [94](#)
 - behavioral [163 to 194](#)
 - logic strength [114 to 132](#)
 - sequential circuits with simultaneous input changes [159](#)
- module [210 to 213](#)
- and user-defined primitives [145](#)
 - definition [210](#)
 - hierarchy [209](#)
 - instance parameter value
 - assignment [215](#)
 - instantiation [211](#)
 - keyword [210](#)
 - macro [216 to 217](#)
 - overriding parameter values [213 to 216](#)
 - parameter dependencies [216](#)
 - port [212](#)
 - syntax [210](#)
 - for specifying instantiations [211](#)
 - terminal [212](#)
 - top-level
 - definition of [211](#)
- module input port delays, see MIPDs [421](#)
- module parameter
- as delay [50](#)
 - as width of variables [50](#)
 - compared to specify parameter [238](#)
 - dependencies [216](#)
 - overriding values [213 to 216](#)
 - passing to tasks [199 to 200](#)
 - syntax [50](#)
 - use with macro modules [217](#)
- module path
- definition [239](#)
 - delay [239](#)
 - description syntax [248, 265](#)
 - destination
 - and pulse control [259](#)
 - in declaration of multiple paths [250](#)
 - requirements for [248](#)
 - in behavioral descriptions [255 to 256](#)
 - polarity [254 to 255](#)
 - source
 - and pulse control [259](#)
 - in declaration of multiple paths [250](#)
 - requirements for [248](#)
 - used to calculate delays [240](#)
 - transport delays [244](#)
- module path delays
- pulse handling [260](#)
- module path pulse control
- for specific modules and paths [259 to 260](#)
 - global [257](#)
- modulus operator [52](#)
- definition [55](#)

Verilog-XL Reference

monitor flag [341](#)
monitoring
 continuous [341](#)
 strobed [340](#)
monitoring interconnect delay signal
 values [342](#)
MOS gate [108 to ??](#)
MOS strength handling [131](#)
msb (most significant bit) [38](#)
multi-channel descriptor [343](#)
multiple drivers
 at same strength level [127](#)
 driving the same net [41](#)
 inside a module [242 to 243](#)
 outside a module [243](#)
multiple module path delays
 assigning in one statement [250](#)
multi-source interconnect transport
 delays [421](#)
 and pulse handling [436](#)
 creating [433](#)
 definition [432](#)
multi-way decision statements [175](#)
multi-way decisions
 case statement [176](#)
 if-else-if statement [175](#)

N

named blocks
 and hierarchical names [228](#)
 and scope [235](#)
 purpose [192](#)
named events [184 to 185](#)
 used with event expressions [184](#)
names
 of hierarchical paths [228 to 236](#)
nand gate [105 to ??](#)
negative timing checks
 \$recrem [303](#)
 \$setuphold [309](#)
negedge [291](#)
net and register bit addressing [63](#)
nets [40](#)
 collapsing [225](#)
 delay [132 to 137](#)
 implicit declaration [113](#)
 initialization [40](#)
 names referenced in a hierarchical
 name [141](#)

 not driven by a source [114](#)
 removal of names [140 to 141](#)
 scalar [224](#)
 syntax [35 to 36](#)
 triereg strength [116](#)
 types of [40 to 45](#)
 wired logic [127](#)
newlink displayandwrite [333](#)
nmos [108 to 110](#)
node
 in hierarchical name tree [228](#)
non-blocking procedural
 assignment [167 to 173](#)
 evaluating assignments [168](#)
 multiple assignments [171](#)
 processing assignments [173](#)
 syntax [168](#)
nor gate [105 to ??](#)
not gate [106 to ??](#)
notif gate [107](#)
notifier in edge sensitive UDP [292 to 293](#)
null
 expression [333](#)
 string [66](#)
numbers
 base format [24](#)
 size specification [24](#)
 unsized [25](#)

O

o
 octal number format [24](#)
octal display format [24](#)
on/off control
 of monitoring tasks [341](#)
operands [62 to 66](#)
 bit-select [62](#)
 concatenation [63](#)
 definition [51](#)
 function call [63](#)
 part-select [62](#)
 strings [64 to 66](#)
Operators
 logic table for [59](#)
operators [52 to 62](#)
 - [52](#)
 ! [57](#)
 != [56](#)
 !==

Verilog-XL Reference

- evaluation of [56](#)
 - % [52](#)
 - & [52](#)
 - && [57](#)
 - * [52](#)
 - *> [250](#)
 - + [52](#)
 - / [52](#)
 - < [56](#)
 - << [60](#)
 - <=
 - evaluation of [56](#)
 - used in non-blocking procedural assignment [168](#)
 - = [71](#)
 - == [56](#)
 - ===
 - evaluation of [56](#)
 - => [250](#)
 - > [56](#)
 - >= [56](#)
 - >>
 - evaluation of [60](#)
 - ?: [53](#)
 - ^ [52, 53](#)
 - ^~ [52](#)
 - { } [62](#)
 - | [52](#)
 - || [57](#)
 - ~ [52](#)
 - ~& [52](#)
 - ~^ [52](#)
 - ~| [53](#)
 - and real numbers [49](#)
 - arithmetic [54](#)
 - binary [53](#)
 - conventions for [24](#)
 - bit-wise [58](#)
 - bit-wise AND [52](#)
 - bit-wise equivalence [52](#)
 - bit-wise exclusive OR [52](#)
 - bit-wise inclusive OR [52](#)
 - bit-wise negation [52](#)
 - case equality [52](#)
 - case inequality [52](#)
 - conditional [61](#)
 - definition [23](#)
 - equality [56](#)
 - left shift [60](#)
 - logic table for [58](#)
 - logical AND [52](#)
 - logical equality [52](#)
 - logical inequality [52](#)
 - logical negation [52](#)
 - logical OR [52](#)
 - modulus [52](#)
 - reduction [59](#)
 - reduction AND [52](#)
 - reduction NAND [52](#)
 - reduction NOR [53](#)
 - reduction OR [53](#)
 - reduction XNOR [53](#)
 - reduction XOR [53](#)
 - relational [56](#)
 - right shift [60](#)
 - ternary [24](#)
 - unary [24](#)
 - optimization
 - of processing stimulus
 - patterns [370 to 371](#)
 - or gate [105 to ??](#)
 - output
 - port declaration [220](#)
 - to files [343 to 345](#)
 - overriding global module path pulse control [259 to 260](#)
 - overriding module parameter values [213 to 216](#)
 - assigning values in-line within module instances [215](#)
 - defparam [214 to 216](#)
 - compared to assignments [215](#)
- ## P
- parallel block statement
 - finish time [192](#)
 - fork-join [189](#)
 - start time [192](#)
 - syntax [191](#)
 - parallel connection [249](#)
 - expanding vector bits [400](#)
 - parameter
 - keyword for module parameters [238](#)
 - module type [50](#)
 - syntax [50](#)
 - parentheses
 - and changing operator precedence [54](#)
 - part-select
 - and macro module instances [217](#)
 - and vector ports [220](#)

Verilog-XL Reference

- of vector net or register [63](#)
- references of real numbers [49](#)
- syntax [63](#)
- path delay mode [455](#)
- path delays [239](#)
 - enhancing accuracy [279 to 287](#)
 - expanding vector bits [400](#)
 - expanding vector bits for full connection [400](#)
 - expanding vector bits for parallel connection [400](#)
 - multiple [275](#)
- PATHPULSE\$ [259 to 260](#)
- Pearl command file [403](#)
- personality
 - memory [406](#)
 - of logic array [407 to 409](#)
- PLA devices [405 to 420](#)
 - array logic types [406](#)
 - array types [406](#)
 - list of system tasks [406](#)
 - logic array personality declaration [407](#)
 - logic array personality
 - formats [407 to 409](#)
 - logic array personality loading [407](#)
- plane
 - format [408](#)
 - in programmable logic arrays [406](#)
- plus options
 - +annotate_any_time [393](#)
 - +autonaming [233](#)
 - +listcounts [363](#)
 - +maxdelays [67](#)
 - +mindelays [67](#)
 - +multisource_int_delays [433](#)
 - +neg_tcheck [309](#)
 - +neg_tchk [304](#)
 - +nosdfwarn [395](#)
 - +notimingchecks [289](#)
 - +pathpulse
 - and overriding global pulse control [259](#)
 - +pre_16a_paths [274](#)
 - +pulse_e/n [436](#)
 - +pulse_int_e/n [436](#)
 - +pulse_int_r/m [436](#)
 - +pulse_r/m [436](#)
 - +sdf_error_info [395](#)
 - +sdf_verbose [395](#)
 - +transport_int_delays [433](#)
 - +typdelays [67](#)
 - +x_transport_pessimism [287](#)
- plus sign(+)
 - arithmetic addition operator [52](#)
- pmos [108 to 110](#)
- polarity [254 to 255](#)
 - positive [255](#)
 - unknown [254](#)
- port [219 to 228](#)
 - collapsing [223](#)
 - connecting
 - by name [221 to 223](#)
 - by position with ordered list [220](#)
 - in macro modules [227](#)
 - rules for [224 to 226](#)
 - declaration [220](#)
 - definition [219](#)
 - inter-module connections [225](#)
 - module [212](#)
 - of user-defined primitives [146](#)
 - rules for collapsing [224 to 226, 235](#)
- port value character identifiers [513](#)
- posedge [291](#)
- power supplies
 - modeled by supply nets [34, 45](#)
- precedence
 - equality operators [56](#)
 - logical operators [57](#)
 - relational operators [56](#)
- predefined plus options
 - +maxdelays [67](#)
 - +mindelays [67](#)
 - +typdelays [67](#)
- primitive instance identifier [101](#)
- primitive keyword [145](#)
- printing command history [355](#)
- procedural assignment [166](#)
 - and integers [48](#)
 - and time variables [48](#)
 - blocking [167](#)
 - non-blocking [167 to 173](#)
 - versus continuous assignment [79](#)
- procedural continuous
 - assignments [93 to 96](#)
 - assign [94 to 95](#)
 - deassign [94 to 95](#)
 - definition [93](#)
 - force [95](#)
 - precedence [95](#)
 - release [95](#)
 - syntax [94](#)
- procedural statements

Verilog-XL Reference

- in behavioral models [163](#)
 - procedural timing controls [182](#) to [189](#)
 - delay control [183](#) to [184](#)
 - event control [182](#)
 - fork-join block [192](#)
 - intra-assignment timing controls [186](#) to [189](#)
 - zero-delay control [183](#)
 - procedure
 - always statement [164](#)
 - function [164](#)
 - initial statement [164](#)
 - task [164](#)
 - Profiler [465](#)
 - programmable logic arrays [405](#) to [420](#)
 - list of system tasks [406](#)
 - logic types [406](#)
 - personality
 - declaration [407](#)
 - formats [407](#) to [409](#)
 - loading [407](#)
 - types [406](#)
 - propagation delay
 - for gates and nets [132](#)
 - in logic gate syntax [101](#)
 - protection
 - of data in memory [381](#)
 - Pu
 - pull drive in strength format [339](#)
 - pull0 [100](#)
 - pull1 [100](#)
 - pulldown source [113](#)
 - pullup source [113](#)
 - pulse filtering
 - and cancelled schedules [262](#)
 - on module path delays [264](#)
 - syntax [261](#)
 - pulse handling
 - for SITDs and MITDs [436](#)
- ## R
- race condition
 - and intra-assignment timing control [187](#)
 - random access memory(RAM)
 - modeled by register arrays [46](#)
 - random number generators
 - \$random [347](#)
 - range
 - syntax [37](#) to [38](#)
 - range specification [99](#), [101](#)
 - rcmos [112](#)
 - reading input commands from a file [356](#)
 - read-only memory(ROM)
 - modeled by register arrays [46](#)
 - real numbers [48](#) to [49](#)
 - and operators [49](#)
 - format specifications used with [336](#)
 - functions and tasks that handle [380](#)
 - in port connections [223](#)
 - recursive task calls [201](#)
 - reducing pessimism
 - with a case statement [178](#)
 - with user-defined primitives [158](#)
 - reduction operators [59](#)
 - ~& [59](#)
 - unary NAND [59](#)
 - unary NOR [59](#)
 - XNOR [59](#)
 - XOR [59](#)
 - redundancy
 - in user-defined primitive state tables [157](#)
 - registers [32](#)
 - and level-sensitive sequential UDPs [150](#)
 - declaration syntax [46](#)
 - for modeling memories [46](#)
 - notifier [292](#)
 - syntax [35](#) to [36](#)
 - used in procedural assignments [79](#)
 - relational operators
 - < [56](#)
 - <= [56](#)
 - > [56](#)
 - >= [56](#)
 - and unknown bit values [56](#)
 - precedence [56](#)
 - release keyword [95](#) to [96](#)
 - repeat event control [188](#)
 - repeat loop
 - syntax [180](#)
 - repetition multiplier [62](#)
 - resistive devices
 - modeled with tri0 and tri1 nets [45](#)
 - restart file option [355](#)
 - restarting
 - from command line [355](#)
 - from full save [354](#)
 - from incremental save [354](#)

Verilog-XL Reference

- the simulator [352](#)
- restrictions on data types
 - in continuous assignments
 - for port connections [224](#)
 - right-hand versus left-hand [71](#)
 - in port collapsing [223](#) to [224](#)
 - in procedural assignments
 - right-hand versus left-hand [166](#)
 - when connecting ports [224](#)
- right shift operator [60](#)
- rise delay
 - assigning values for [136](#)
 - definition of [132](#)
- rnmos [108](#) to [110](#)
- rpmos [108](#) to [110](#)
- rtran [110](#)
- rtranif0 [110](#)
- rtranif1 [110](#)

S

- s
 - in string display format [340](#)
- saving simulation data [352](#)
 - limitations [355](#)
- scalared keyword [38](#)
- scalars
 - compared to vectors [37](#)
 - scalar nets and driving strength of continuous assignment [77](#)
- schedule
 - showing or hiding cancelled [265](#)
- schedules
 - cancellation dilemma [266](#)
 - cancelled [262](#), [263](#)
- scientific notation [48](#)
- scope
 - and hierarchical names [228](#)
 - rules [234](#) to [236](#)
- SDF annotation [392](#)
 - configuration file [393](#)
 - creating new delay triplets [396](#)
 - delay values [394](#)
 - examples of running [395](#)
 - for vector bits [400](#)
 - interconnect delay [397](#)
 - log file [394](#)
 - multiple task invocations [396](#)
 - options controlling output [395](#)
 - scaling timing data [394](#)

- SDF Annotator warning suppression [395](#)
- SDPDs [269](#)
 - effects of unknowns [276](#), [277](#)
 - internal logic effects [277](#)
- self-determined expression [68](#)
- sequential block statement [189](#) to [190](#)
 - finish time [192](#)
 - start time [192](#)
 - syntax [190](#)
- sequential UDP initialization [151](#) to [154](#)
- sequential UDPs
 - input and output fields in state table [147](#)
- set of values (0, 1, x, z) [31](#)
- setting a net to a logic value [369](#) to [370](#)
- shift operators
 - << [60](#)
 - >> [60](#)
- signals
 - detecting changes in asynchronous control [299](#), [304](#)
 - multiple conditioning [294](#)
- signed arithmetic
 - shift operators [60](#)
- simulating module path delays
 - one path output driving another [256](#)
 - when driving wired logic [242](#) to [243](#)
- simulation
 - event-driven [160](#)
 - going back with incremental restart [354](#)
 - simulation time and timing controls [182](#)
 - time [346](#)
- single-source interconnect transport
 - delays [421](#)
 - and pulse handling [436](#)
 - creating [433](#)
 - definition [432](#)
- SITD [421](#)
 - and pulse handling [436](#)
 - creating [433](#)
 - definition [432](#)
- size of displayed data [336](#) to [337](#)
- sized numbers [24](#)
- Sm
 - small capacitor in strength format [339](#)
- source
 - pulldown [113](#)
 - pullup [113](#)
- source protection
 - effect on dump file [498](#)
- specify block [237](#) to [293](#)
- specify blocks [237](#)

Verilog-XL Reference

- specify parameters [238 to 239](#)
- specifying transition delays on module
 - paths [251](#)
 - assigning one value [252](#)
 - assigning six values [252](#)
 - assigning three values [252](#)
 - assigning two values [252](#)
 - x transitions [253](#)
- specparam [239](#)
 - syntax [238](#)
 - versus module parameter [239](#)
- St
 - strong drive in strength format [338](#)
- standard output [343](#)
- start time
 - in parallel block statements [192](#)
 - in sequential block statements [192](#)
- state dependent path delays [269](#)
- status
 - of expanded nets [359](#)
 - of module ports [360](#)
 - of variables [358 to 359](#)
- strength [100 to 101](#)
 - ambiguous [117](#)
 - and logic conflicts [40](#)
 - and MOS gates [131](#)
 - and scalar net variables [32](#)
 - charge storage
 - strength levels for [116](#)
 - driving [115](#)
 - gates that accept specifications [100](#)
 - of combined signals [116](#)
 - on trireg nets [42](#)
 - range of possible values [118](#)
 - reduction by non-resistive devices [131](#)
 - reduction by resistive devices [131](#)
 - scale of strengths [116](#)
 - specifying [114 to 116](#)
 - supply net [132](#)
 - trace messages [130](#)
 - tri0 [131](#)
 - tri1 [131](#)
 - trireg [132](#)
- strength display format [338 to 340](#)
 - high impedance [339](#)
 - large capacitor [339](#)
 - logic value 0, 1, H, L, X, Z [338](#)
 - medium capacitor [339](#)
 - pull drive [339](#)
 - small capacitor [339](#)
 - strong drive [338](#)
 - supply drive [338](#)
 - weak drive [339](#)
- strength values for \$dumpports [514](#)
- strings
 - as operands [65](#)
 - definition [26](#)
 - display format [335, 340](#)
 - in vector variables [65](#)
 - manipulation of [26](#)
 - padding [27](#)
 - value padding [65](#)
- strobed monitoring [340](#)
- strong0 [100](#)
- strong1 [100](#)
- structured procedure [164 to 194](#)
 - always statement [164](#)
 - function [164](#)
 - initial statement [164](#)
 - task [164](#)
- Su
 - supply drive in strength format [338](#)
- supply net strength [132](#)
- supply0 [100](#)
 - net [34, 45](#)
- supply1 [100](#)
 - net [34, 45](#)
- switches
 - MOS [108 to 110](#)
- symbolic debugging
 - and hierarchical name referencing [230](#)
- synchronous arrays [406](#)
- syntax
 - \$cleartrace [348](#)
 - \$compare [375](#)
 - \$countdrivers [360](#)
 - \$deposit [369](#)
 - \$disable_warnings [366](#)
 - \$display [333](#)
 - \$enable_warnings [367](#)
 - \$fclose [343](#)
 - \$fdisplay [343](#)
 - \$finish [347](#)
 - \$fmonitor [343](#)
 - \$fopen [343](#)
 - \$fstrobe [343](#)
 - \$fwrite [343](#)
 - \$getpattern [371](#)
 - \$history [355](#)
 - \$incpattern_read [373](#)
 - \$incpattern_write [372](#)
 - \$incsave [352](#)

Verilog-XL Reference

`$keepcommands` [363](#)
`$key` [357](#)
`$list` [363](#)
`$list_forces` [364](#)
`$listcounts` [363](#), [384](#)
`$log` [357](#)
`$monitor` [341](#)
`$monitoroff` [341](#)
`$monitoron` [341](#)
`$nochange` [297](#)
`$nokey` [357](#)
`$nolog` [357](#)
`$random` [347](#)
`$readmemb` [368](#)
`$readmemh` [368](#)
`$recovery` [299](#)
`$recrem` [290](#)
`$removal` [290](#), [304](#)
`$reportprofile` [383](#)
`$reset` [386](#)
`$scope` [358](#)
`$settrace` [348](#)
`$setup` [306](#)
`$setuphold` [307](#)
`$showallinstances` [358](#)
`$showexpandednets` [359](#)
`$showmodes` [362](#)
`$showportsnotcollapsed` [360](#)
`$showscopes` [358](#)
`$showvariables` [358](#)
`$showvars` [358](#)
`$startprofile` [383](#)
`$stime` [346](#)
`$stopprofile` [384](#)
`$strobe` [340](#)
`$strobe_compare` [376](#)
`$time` [346](#)
`$write` [333](#)
`'default_nettype` [114](#)
`always` [165](#)
`array of instances` [99](#)
`assign` [94](#)
`behavioral statements` [524](#) to [525](#)
`conditional operator` [61](#)
`conditional statement` [174](#)
`conditioned event` [293](#)
`continuous assignment` [72](#)
`deassign` [94](#)
`declarations` [521](#) to [523](#)
`declaring events` [185](#)
`delay control` [183](#)
`disable statement` [205](#)
`event control` [184](#)
`event triggering statement` [185](#)
`expressions` [529](#) to [530](#)
`for addressing memory` [64](#)
`for enabling tasks` [199](#)
`for loop` [180](#)
`force` [94](#)
`forever loop` [180](#)
`formal definition` [517](#) to [532](#)
`function` [198](#), [201](#)
`function call` [202](#)
`general` [530](#)
`if-else-if statement` [175](#)
`initial statement` [165](#)
`integer declaration` [47](#)
`logic gates` [98](#)
`macro module` [217](#)
`memory declaration` [46](#)
`module` [210](#)
`module instantiation` [211](#), [523](#)
`module parameter` [50](#)
`module path description` [248](#)
`net declaration` [35](#) to [36](#)
`parallel block statement` [191](#)
`part-select` [63](#)
`PATHPULSES$` [259](#)
`period` [298](#)
`port`
 `declaration` [220](#)
 `definition` [219](#)
`primitive instances` [523](#)
`procedural continuous assignments` [94](#)
`pulse filtering tasks` [261](#)
`range` [38](#)
`register declaration` [35](#) to [36](#)
 `for memories` [46](#)
`release` [94](#)
`repeat loop` [180](#)
`SDPD` [270](#)
`sequential block statement` [190](#)
`source text` [518](#) to [521](#)
`specify block` [238](#)
`specify parameter` [238](#)
`specify section` [526](#) to [528](#)
`specparam` [238](#)
`state dependent path delays` [270](#)
`switch-level modeling` [531](#)
`task` [198](#)
`text macro`
 `definitions` [29](#)

Verilog-XL Reference

- usage [29](#)
- time variable declaration [47](#)
- UDPs [144](#)
- user-defined primitives [144](#) to [145](#)
- wait statement [186](#)
- while loop [180](#)
- system tasks [331](#)
 - file name arguments [332](#)
 - for changing base in formatted output [345](#)
 - for continuous monitoring [341](#)
 - for displaying information [333](#) to [340](#)
 - for displaying the delay mode [362](#)
 - for fetching simulation time [346](#)
 - for generating key files [357](#)
 - for generating random numbers [347](#)
 - for loading memories from text files [368](#) to [369](#)
 - for printing command history [355](#)
 - for processing stimulus patterns faster [370](#) to [371](#)
 - for producing an interactive source listing [363](#) to [365](#)
 - for reading input commands from a file [356](#)
 - for resetting Verilog-XL [384](#) to [392](#)
 - for restarting the simulator [352](#)
 - for running the behavior profiler [382](#) to [384](#)
 - for saving simulation data [352](#)
 - for setting a net to a logic value [369](#)
 - for showing hierarchy [358](#)
 - for showing module port status [360](#)
 - for showing number of drivers [360](#)
 - for showing status of expanded nets [359](#)
 - for showing variable status [358](#) to [359](#)
 - for storing interactive commands [362](#)
 - for writing formatted output to files [343](#) to [345](#)
- generating a checkpoint in the value change dump file [496](#)
- limiting the size of the value change dump file [497](#)
- list of formatted output system tasks [345](#)
- reading the value change dump file during a simulation [497](#)
- resuming the dump into the value change dump file [496](#)
- showing the timescale of a module [446](#)

- specifying how %t reports time information [446](#) to [449](#)
- specifying the name of the value change dump file [494](#)
- specifying the time unit of delays entered interactively [446](#) to [449](#)
- specifying the variables to be dumped in the value change dump file [495](#) to [496](#)
- stopping the dump into the value change dump file [496](#)

T

- t
 - timescale format
 - example of use [448](#)
 - table keyword [146](#)
 - tasks [197](#) to [204](#)
 - and hierarchical names [228](#)
 - and scope [234](#)
 - as structured procedures [164](#)
 - definition [164](#)
 - disabling within a nested chain [206](#)
 - enabling [198](#) to [200](#)
 - when already active [201](#)
 - passing parameters [199](#) to [200](#)
 - purpose [198](#)
 - syntax [198](#)
 - for enabling [199](#)
 - terminal
 - in logic gate syntax [102](#)
 - module [212](#)
 - ternary operators
 - ?: [53](#)
 - text macro substitutions [29](#) to [30](#)
 - definition syntax [29](#)
 - in interactive mode [29](#)
 - redefinition [30](#)
 - usage syntax [29](#)
 - time
 - and incremental restart [354](#)
 - arithmetic operations performed on time variables [48](#)
 - simulation [346](#)
 - and timing controls [182](#)
 - variables [47](#)
 - time precision [440](#)
 - time unit [440](#)
 - timescales

Verilog-XL Reference

- displaying [446](#)
- example [449](#) to [452](#)
- setting [449](#)
- supporting functions and tasks [380](#)
- system tasks for [445](#) to [449](#)
- timing check
 - violations [290](#)
- timing checks [289](#)
 - \$hold system task [295](#)
 - \$nochange [297](#)
 - \$period [298](#)
 - \$recovery [299](#)
 - \$recrem [301](#)
 - \$removal [304](#)
 - \$setup [305](#)
 - \$setuphold [307](#)
 - \$skew [310](#)
 - \$width [311](#)
- and detecting simultaneous input transitions [160](#)
- annotating vector bits [401](#)
- in behavioral descriptions [255](#) to [256](#)
- negative time specifications [295](#)
- system tasks [295](#)
- using edge-control specifiers [291](#)
- using notifiers for timing violations [292](#)
- using with conditioned events [293](#)
- warning messages [290](#)
- with multiple conditioning signals [294](#)
- too few module port connections
 - warning [213](#)
- top-level module
 - definition of [211](#)
- trace
 - \$cleartrace [348](#)
 - \$settrace [348](#)
 - tracing statements inside macro modules [218](#)
 - single step [218](#)
- tran [110](#)
- tranif0 [110](#)
- tranif1 [110](#)
- transistors [110](#)
- transitions
 - 01 [151](#)
 - unspecified [151](#)
- transport delay [244](#)
- tree structure
 - of hierarchical names [228](#)
- tri nets [45](#)
- triereg [41](#) to [45](#)

- and capacitive networks [42](#)
- and charge decay [45](#)
- and charge storage strength [116](#), [132](#)
- and high impedance state [42](#)
- example [42](#)
- vectored keyword inapplicable [38](#)
- turn-off delay [136](#)
- types of nets
 - tri nets [40](#)
 - tri0 [131](#)
 - tri1 [131](#)
 - triand [41](#)
 - trior [41](#)
 - wire [40](#)
 - wired AND [41](#)
 - wired logic [127](#)
 - wired OR [41](#)

U

- UDPs [143](#) to [162](#)
 - and memory considerations [160](#)
 - and performance [144](#)
 - combinational UDPs [148](#) to [150](#)
 - compilation [157](#)
 - definition [145](#) to [147](#)
 - edge-sensitive UDPs [150](#) to [151](#)
 - instances [156](#) to [157](#)
 - level-sensitive dominance [155](#) to [156](#)
 - level-sensitive sequential UDPs [150](#)
 - mixing level- and edge-sensitive descriptions [154](#) to [155](#)
 - ports [146](#)
 - processing simultaneous input changes [159](#)
 - reducing pessimism [158](#)
 - state table [146](#) to [147](#)
 - summary of symbols in state table [148](#)
 - syntax [144](#) to [145](#)
- unary operators
 - ! [57](#)
 - << [60](#)
 - >> [60](#)
- unconnected port [212](#)
- underline character [26](#)
- unit delay mode
 - and timescales and time units [458](#)
 - definition of [454](#)
 - overriding with the parameter attribute mechanism [460](#)

Verilog-XL Reference

unknown logic value
 and numbers [25](#)
 display formats [337](#) to [??](#)
 effect in different bases [25](#)
 in UDP state table [147](#)
 symbolic representation [32](#)
unsized numbers [25](#)
unspecified transitions [151](#)
upwards name referencing [232](#) to [236](#)
user-defined primitives [143](#) to [162](#)
 and performance [144](#)
 combinational [148](#) to [150](#)
 compilation [157](#)
 definition [145](#) to [147](#)
 edge-sensitive [150](#) to [151](#)
 instances [156](#) to [157](#)
 level-sensitive dominance [155](#) to [156](#)
 level-sensitive sequential [150](#)
 mixing level- and edge-sensitive
 descriptions [154](#) to [155](#)
 ports [146](#)
 processing simultaneous input
 changes [159](#)
 reducing pessimism [158](#)
 state table [146](#) to [147](#)
 summary of symbols in state table [148](#)
 syntax [144](#) to [145](#)

V

value change dump file [493](#) to [510](#)
 contents [498](#)
 creating [493](#) to [496](#)
 effect of source protection [498](#)
 format [498](#) to [510](#)
 example [507](#)
 formats of variable values [499](#) to [500](#)
 generating a checkpoint [496](#)
 keyword commands [500](#) to [504](#)
 \$comment [501](#)
 \$date [502](#)
 \$dumpall [502](#)
 \$dumpon [502](#)
 \$dumpvars [503](#)
 \$enddefinitions [503](#)
 \$scope [503](#)
 \$timescale [504](#)
 \$upscope [504](#)
 \$var [505](#)
 \$version [505](#), [506](#)

 limiting the size [497](#)
 reading the value change dump file
 during a simulation [497](#)
 resuming the dump [496](#)
 specifying the name [494](#)
 specifying the variables to be
 dumped [495](#) to [496](#)
 stopping the dump [496](#)
 structure [499](#)
 syntax of VCD file [506](#)
value set (0, 1, x, z) [31](#)
values
 of combined signals [116](#)
Vcc [45](#)
VCD file
 syntax [506](#)
VCD, see value change dump file
Vdd [45](#)
vectored keyword [38](#)
vectors [37](#)
 and timing violations [291](#)
violations
 timing checks [290](#)
Vss [45](#)

W

wait statement
 as level-sensitive event control [186](#)
 syntax [186](#)
 to advance simulation time [183](#)
warning messages
 enabling and disabling [365](#) to [367](#)
 timing checks [290](#)
 too few module port connections [213](#)
warning suppression [259](#)
warning suppression option
 for non-downward path names [236](#)
warnings
 disabling [366](#) to [367](#)
 enabling [367](#)
We
 weak drive in strength format [339](#)
weak0 [100](#)
weak1 [100](#)
while loop
 syntax [180](#)
white space [24](#)
wired logic nets
 wand [127](#)

- wired-AND configurations [41](#)
- wired-OR configurations [41](#)
- wires [40](#)
- word
 - of array [46](#)
- writing formatted output to files [343](#) to [345](#)

X

- X
 - unknown logic value in strength
 - format [338](#)
- x
 - as display format for unknown logic
 - value [337](#)
 - in state table [147](#)
 - unknown logic value [32](#)
- XL algorithm
 - and specify blocks [239](#)
- xnor gate [105](#) to ??
- xor gate [105](#) to ??

Z

- Z
 - as display format for high impedance
 - state [337](#)
 - high impedance state in strength
 - format [338](#)
- z
 - as display format for high impedance
 - state [337](#)
 - high impedance state [32](#)
- zero delay mode
 - definition of [454](#)
 - overriding with the parameter attribute
 - mechanism [460](#)
- zero-delay
 - control [183](#)

Verilog-XL Reference
